CS 3310 Design and Analysis of Algorithms
Project #1
Due: 4/6
(Total: 100 points)

Student Name: Roy Afaryan

Date: 4/10/2022

Important:
- Please read this document completely before you start coding.
- Also, please read the submission instructions (provided at the end of the project description) carefully before submitting the project.

Project #1 Description:

Program the following algorithms that we covered in the class:

a. Classical matrix multiplication
b. Divide-and-conquer matrix multiplication
c. Strassen's matrix multiplication

You can use either Java, or C++ for the implementation. The objective of this project is to help student understand how above three algorithms operates and their difference in run-time complexity (average-case scenario). The project will be divided into three phases to help you to accomplish above tasks. They are Part 1: Design and Theoretical analysis, Part 2: Implementation, and Part 3: Comparative Analysis.

This project will ask student to conduct the matrix multiplication on the numeric data type. You can implement the above three algorithms in your choice of data structures based on the program language of your choice. Note that you always try your best to give the most efficient program for each problem. Let the matrix size be n x n. Carry out a complete test of your algorithms with n = 2, 4, 8, 16, 32, 64, 128, 256, … (up to the largest size of n that your computer can handle). The size of the input will be limited in the power of 2. They are 2, 4, 8, 16, 32 up to 2^k where 2^k is the largest size of the input that your computer can handle or your program might quit at size of 2^k due to running out of memory. The reason for the power of 2 is to ease the programming component of the assignment.

Submission Instructions:

After the completion of all three parts, Part 1, Part 2 and Part 3, submit (upload) the following files to Canvas:

- ***three program files of three algorithms or one program file including all three algorithms (your choice of programming language with proper documentation)***
- ***this document in pdf format (complete it with all the*** <Insert > ***answers)***

Part 1: Design & Theoretical Analysis (30 points)

a. Complete the following table for theoretical worst-case complexity of each algorithm. Also need to describe how the worst-case input of each algorithm should be.

Algorithm	theoretical worst-case complexity	describe the worst-case input
Classical matrix multiplication	O(n^3)	The worst-case input is 2 n x n matrices with non-zero integers
Divide-and-conquer matrix multiplication	O(n^3)	The worst-case input is when the dimensions of the two matrices are n x m where $n \neq m$. This makes the recursive calls very expensive.
Strassen's matrix multiplication	$O(n^{\log_2(7)})$	The worst-case input is when the dimensions of the two matrices are n x m where $n \neq m$. This makes the recursive calls very expensive.

b. Design the program by providing pseudocode or flowchart for each algorithm.

Classical Matrix Multiplication Pseudocode:

```
function classical(a_matrix, b_matrix)

    n = size of matrix
    i, j, k = 0

    for i to n:

        for j to n:
            c_matrix[i,j] = 0

            for k to n:
                c_matrix[i,j] += a_matrix[i,k] * b_matrix[k,j]
```

Divide and Conquer Matrix Multiplication Pseudocode:

```
function divide_and_conquer(a_matrix, b_matrix)

    n = size of matrix

    # base case
    if n == 1:
        c_matrix[0,0] = a_matrix[0,0] * b_matrix[0,0]

    else:
        # divide matrices into 4 equal parts
        a11, a12, a21, a22 = a_matrix.split
        b11, b12, b21, b22 = b_matrix.split

        # recursive call
        c11 = divide_and_conquer(a11, b11) + divide_and_conquer(a12, b21)
        c12 = divide_and_conquer(a11, b21) + divide_and_conquer(a12, b22)
        c21 = divide_and_conquer(a21, b11) + divide_and_conquer(a22, b21)
        c22 = divide_and_conquer(a21, b12) + divide_and_conquer(a22, b22)


        # combine step
        c_matrix = c11, c12, c21, c222
```

Strassen's Matrix Multiplication Pseudocode:

```
function strassen(a_matrix, b_matrix)

    n = size of matrix

    # base case
    if n == 1:
        c_matrix[0,0] = a_matrix[0,0] * b_matrix[0,0]

    else:
        # divide matrices into 4 equal parts
        a11, a12, a21, a22 = a_matrix.split
        b11, b12, b21, b22 = b_matrix.split

        # recursive call
        P = (a11 + a22)(b11 + b22)
        Q = (a21 + a22)(b11)
        R = (a11)(b12 - b22)
        S = (a22)(b21 - b11)
        T = (a11 + a12)(b22)
        U = (a21 - a11)(b11 + b12)
        V = (a12 - a22, b21 + b22)

        # combine step
        c11 = P + S - T + V
        c12 = R + T
        c21 = Q + S
        c22 = P + R - Q + U

        c_matrix = c11, c12, c21, c222
```

c. Design the program correctness testing cases. Design at least 10 testing cases to test your program, and give the expected output of the program for each case. We prepare for correctness testing of each of the three programs later generated in Part 2.

Testing case #	Input	Expected output	Classical matrix multiplication (√ if CORRECT output from your program)	Divide-and-conquer matrix multiplication (√ if CORRECT output from your program)	Strassen's matrix multiplication (√ if CORRECT output from your program)
1	Matrix A: [[2 8] [6 1]] Matrix B: [[9 1] [3 6]]	[42 50] [57 12]	√	√	√
2	Matrix A: [[0 9] [3 6]] Matrix B: [[1 3] [3 10]]	[27 90] [21 69]	√	√	√
3	Matrix A: [[2 9] [2 0]] Matrix B: [[7 3] [5 8]]	[59 78] [14 6]	√	√	√
4	Matrix A: [[2 7 8 5] [2 4 1 3] [2 1 3 0] [8 4 9 7]] Matrix B: [[7 9 5 9] [3 0 8 3] [10 2 5 1] [5 5 1 8]]	[[140 59 111 87] [51 35 50 55] [47 24 33 24] [193 125 124 149]]	√	√	√

5	Matrix A: [[0 2 4 0] [0 5 8 10] [7 7 7 9] [3 6 6 1]] Matrix B: [[9 7 7 5] [10 8 7 8] [6 8 1 5] [4 0 7 4]]	[44 48 18 36] [138 104 113 120] [211 161 168 162] [127 117 76 97]	√	√	√
6	Matrix A: [[10 6 8 8] [5 6 0 9] [0 4 10 4] [3 7 10 0]] Matrix B: [[7 8 3 3] [7 10 3 7] [1 9 8 9] [0 3 5 8]]	[120 236 152 208] [77 127 78 129] [38 142 112 150] [80 184 110 148]	√	√	√
7	Matrix A: [[5 1 2 0] [10 2 2 7] [5 10 4 1] [2 10 0 3]] Matrix B: [[3 0 1 9] [8 10 6 10] [5 0 10 7] [0 10 3 7]]	[33 10 31 69] [56 90 63 173] [115 110 108 180] [86 130 71 139]	√	√	√
8	Matrix A: [[9 10] [0 2]] Matrix B: [[6 1] [5 3]]	[110 45] [33 54]	√	√	√
9	Matrix A: [[2 2] [10 4]] Matrix B: [[9 9] [5 4]]	[28 26] [110 106]	√	√	√

10	Matrix A: [[2 3] [7 8]] Matrix B: [[5 3] [5 10]]	[25 36] [75 101]	√	√	√

d. Design testing strategy for the programs. Discuss about how to generate and structure the randomly generated inputs for experimental study in Part 3.

Hint 1: The project will stop at the largest input size n (which is in the form of 2^k) that your computer environment can handle. It is the easiest to use a random generator to help generate numbers for the input data sets. However, student should store all data sets and use the same data sets to compare the performance of all three Matric Multiplication algorithms.

Hint 2: Note that even when running the same data set for the same Matrix Multiplication program multiple times, it's common that they have different run times as workloads of your computer could be very different at different moments. So it is desirable to run each data set multiple times and get the average run time to reflect its performance. The average run time of each input data set can be calculated after an experiment is conducted in m trails; but the result should exclude the best and worst run. Let X denotes the set which contains the m run times of the m trails, where X = {x_1, x_2, x_3 … x_m} and each x_i is the run time of the i^{th} trial. Let x_w be the largest time (worst case) and x_b be the smallest time (best case). Then we have

$$\textit{Average Run Time} = \frac{\sum_{i=1}^{m} Xi \; - Xw - Xb}{m-2}$$

The student should think about and decide how many trials (the value of m) you will use in your experiment. Note that the common choice of the m value is at least 10.

<Insert answers here – on

1. How you generate and structure the randomly generated inputs?

   I generated the random inputs by using list comprehension as well as python's "random" library and is shown in the following code:

```
a_matrix_1 = np.array([[random.randint(0, 10) for i in range(size)] for j in range(size)])
b_matrix_1 = np.array([[random.randint(0, 10) for i in range(size)] for j in range(size)])
```

2. What value of *m* you plan to use? >

   I plan to use an *m* value of 0 because I think 10 tests are adequate for assessing accuracy.

Part 2: Implementation (35 points)

a. Code each program based on the design (pseudocode or flow chart) given in Part 1(b).

<Generate three programs with proper documentation and store them in three files. Note: They are required to be submitted to Canvas as described in the submission instructions>
<No insert here>

b. Test you program using the designed testing input data given in the table in Part 1(c), Make sure each program generates the correct answer by marking a "√" if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

<Complete the testing with testing cases in the table @Part 1(c)>
<No insert here>

c. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

```
Matrix A:
[[ 1  9  5  5  8  9  4  0]
 [ 4  3  7  7  6  0  6  5]
 [ 0 10  5  2  3  4  8 10]
 [ 4  9  4  6  4 10 10  4]
 [ 2 10  2  7  0  7  3  7]
 [ 5  8  7  9  3  1 10  8]
 [ 6  8  7  2 10  9  5  5]
 [ 8  6  1  5  3  5  4  6]]
Matrix B:
[[10  0  6  0  1 10  9  2]
 [ 2  6  3  8  2 10  9  3]
 [ 9  3  7  6  5  1  8  3]
 [ 7  8 10  1  3  9  6  3]
 [ 6  1  5  9  7  1  8 10]
 [ 5  6  9  9  9  9  2  6]
 [ 6  0  3 10  9  5  0 10]
 [ 8  0  0  2  8  6  2  0]]

Classical MM:
[[225 171 251 300 232 259 242 233]
 [270 101 200 197 202 206 219 179]
 [245 118 160 275 260 262 194 185]
 [302 178 279 336 300 356 245 265]
 [216 164 198 206 199 305 194 133]
 [339 150 247 267 267 328 269 218]
 [328 149 275 339 299 311 302 267]
 [251 112 195 183 190 290 210 152]]
--- 0.00397038459777832 seconds ---

Divide and Conquer MM:
[[225 171 251 300 232 259 242 233]
 [270 101 200 197 202 206 219 179]
 [245 118 160 275 260 262 194 185]
 [302 178 279 336 300 356 245 265]
 [216 164 198 206 199 305 194 133]
 [339 150 247 267 267 328 269 218]
 [328 149 275 339 299 311 302 267]
 [251 112 195 183 190 290 210 152]]
--- 0.003999471664428711 seconds ---

Strassen MM:
[[225 171 251 300 232 259 242 233]
 [270 101 200 197 202 206 219 179]
 [245 118 160 275 260 262 194 185]
 [302 178 279 336 300 356 245 265]
 [216 164 198 206 199 305 194 133]
 [339 150 247 267 267 328 269 218]
 [328 149 275 339 299 311 302 267]
 [251 112 195 183 190 290 210 152]]
--- 0.007998228073120117 seconds ---
```

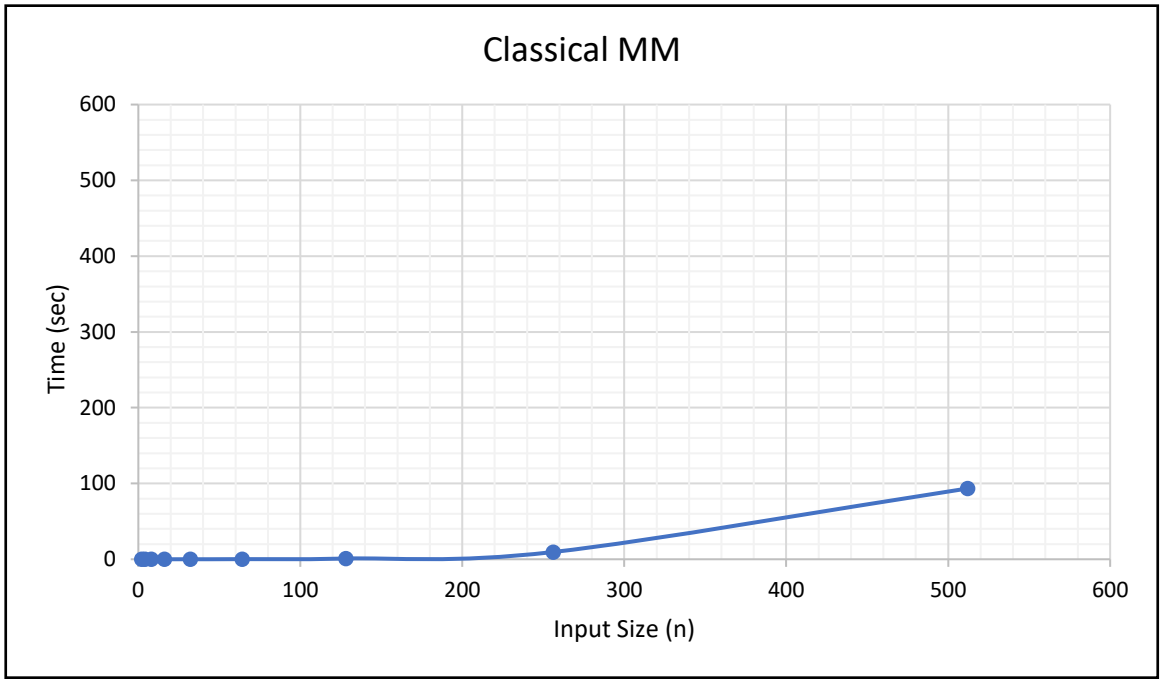
By now, three working programs for the three algorithms are created and ready for experimental study in the next part, Part 3.

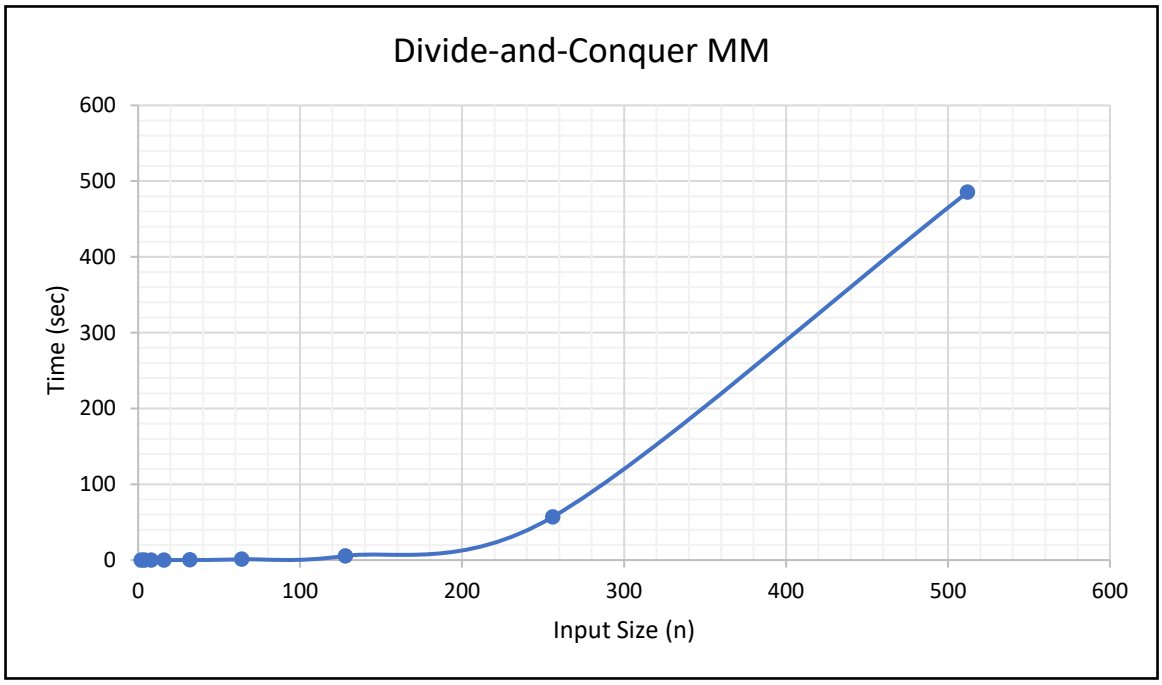
Part 3: Comparative Analysis (35 points)

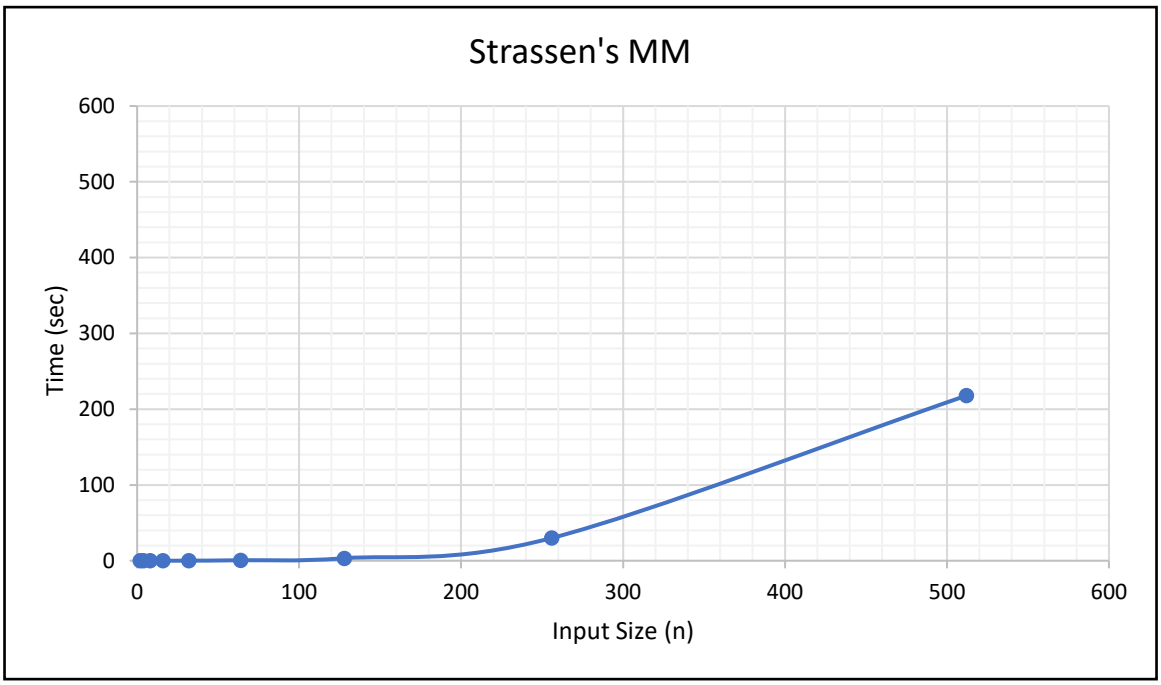
a. Run each program with the designed randomly generated input data given in Part 1(d). Generate a table for all the experimental results as follows.

Input Size n	Average time (Classical matrix multiplication) (in seconds)	Average Time (Divide-and-conquer matrix multiplication) (in seconds)	Average Time (Strassen's matrix multiplication) (in seconds)
2	0.0006	0.0004	0.0006
4	0.0006	0.0021	0.0016
8	0.0013	0.0056	0.0023
16	0.0048	0.0173	0.0133
32	0.0192	0.0952	0.0816
64	0.1575	0.9360	0.6506
128	0.9365	5.4831	3.2825
256	9.5217	56.9511	29.9558
512	93.388	485.216	217.850

b. Plot a graph of each algorithm and summarize the performance of each algorithm based on its own graph.

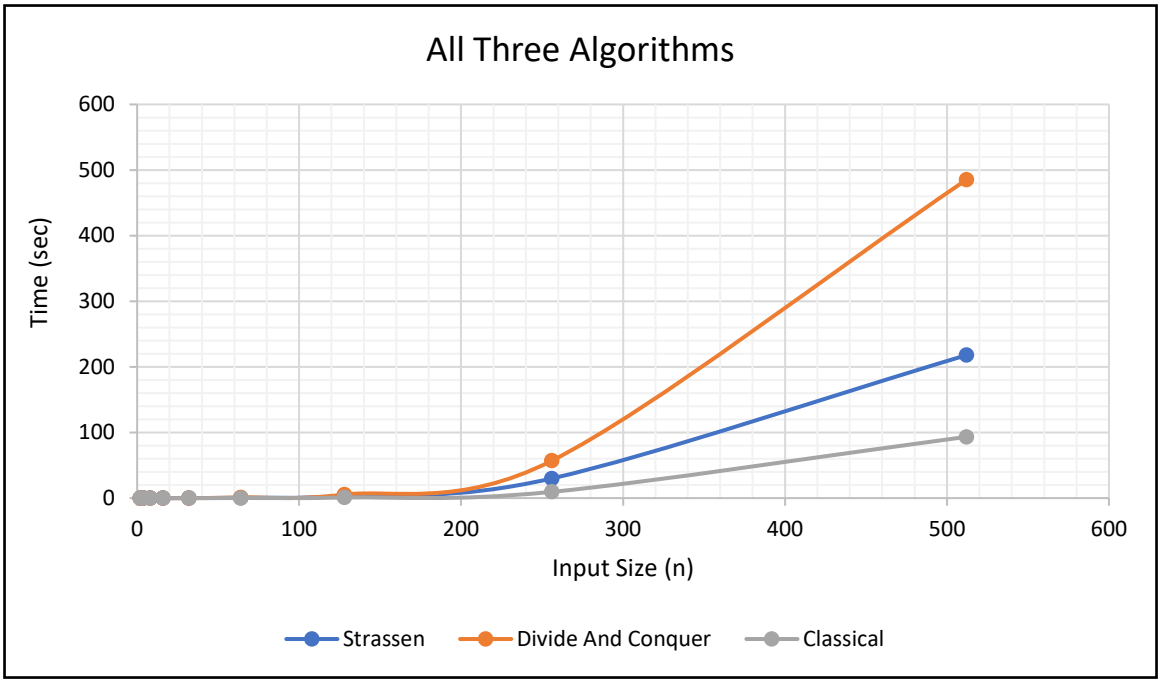






The Classical Matrix Multiplication algorithm ran quite efficiently for matrices that were of size n = 512 and under. The divide-and-conquer ran relatively slow and was by far the least efficient algorithm of the three at least when it comes to smaller matrices. Strassen, despite having a slightly lower theoretical time complexity, ran much slower than I originally expected.

Plot all three graphs on the same graph and compare the performance of all three algorithms. Explain the reasons for which algorithm is faster than others.



Plotting all 3 algorithms in one graph makes it very evident that classical matrix multiplication runs the most efficient at least for matrices of size n <= 512. It should also be mentioned that divide-and-conquer grows significantly faster than Strassen and Classical despite having very similar theoretical time complexity.

c. Compare the theoretical results in Part 1(a) and empirical results here. Explain the possible factors that cause the difference.

According to the theoretical time complexity, classical and divide-and-conquer matrix multiplication were supposed to be near equivalent in efficiency while Strassen's matrix multiplication was supposed to be slightly more efficient. However, that is not the case for matrices n <= 512. The growth order from best to worst goes as follows: classical, Strassen, divide-and-conquer. The reason for this significant difference is due to small testing sizes. Furthermore, extremely large values for n would most probably show that Strassen runs most efficiently; however, for these smaller values, classical matrix multiplication is clearly the best.

d. Give a spec of your computing environment, e.g. computer model, OS, hardware/software info, processor model and speed, memory size, ...

Item	Value
OS Name	Microsoft Windows 10 Home
Version	10.0.19044 Build 19044
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	LAPTOP-5TEAOQV3
System Manufacturer	Razer
System Model	Book 13 - RZ09-0357
System Type	x64-based PC
System SKU	RZ09-03571E92
Processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mh...
BIOS Version/Date	Razer 2.02, 3/29/2021
SMBIOS Version	3.3
Embedded Controller V...	1.00
BIOS Mode	UEFI
BaseBoard Manufacturer	Razer
BaseBoard Product	MA310
BaseBoard Version	4
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32

e. Conclude your report with the strength and constraints of your work. At least 200 words. Note: It is reflection of this project. Ask yourself if you have a chance to re-do this project again, what you will do differently (e.g. your computing environment, programing language, data structure, data set generation, …) in order to design a better performance evaluation experiment.

Upon completion of this project, I feel like I have learned a significant amount when it comes to the analysis of algorithms. I also feel that I am much more confident in my understanding of the divide-and-conquer approach, which, in my opinion, is my greatest takeaway because the concept of recursion was always my weakest link. I believe my greatest strength when tackling this project was my implementation. By using python and several imported libraries, I was able to make concise and easy-to-follow code. A constraint of my work was implementing efficient and simple testing parameters; however, I feel that I tackled this weak point of mine throughout this project. If I had the opportunity to redo this project, one thing I would do is find access to a better computing environment. Due to the constraints of my system, I was only able to test matrices of size n <= 512. Although this gives significant insight as to how the algorithms run, it obviously does not give the best insight. If I could have my way, I would love to see how each of the three algorithms compare when n = 2048; however, that is something I could not test in my case. Ultimately, this project has further excelled my analysis of algorithms.