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

Student Name: Noah Reef

Date: 10/22

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.

$$i \in [0, 1]$$

Algorithm	theoretical worst-case complexity	describe the worst-case input
Classical matrix multiplication	O(n^3)	Same for each input size n x n matrix. Input independent.
Divide-and-conquer matrix multiplication	O(n^3)	Same for each input size n x n matrix. Input independent.
Strassen's matrix multiplication	O(n^2.81)	Same for each input size n x n matrix. Input independent.

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

Classical Matrix Multiplication:

func matmul(A,B):
    n := size(A)
    for $i \in \{1,2,\ldots,n\}$:
        for $j \in \{1,2,\ldots,n\}$:
            $c_{ij} := \sum_{k=1}^{n} a_{ik} + b_{jk}$
        end-for
    end-for

return C
end-func

Divide-and-Conquer Matrix Multiplication:

func divmatmul(A,B):
n:= size(A)
if n equals 2:
return A*B

else:
$C_{11} = \text{divmatmul}(A_{11}, B_{11}) + \text{divmatmul}(A_{12}, B_{21})$
$C_{12} = \text{divmatmul}(A_{11}, B_{21}) + .\text{divmatmul}(A_{12}, B_{22})$
$C_{21} = \text{divmatmul}(A_{21}, B_{11}) + \text{divmatmul}(A_{22}, B_{21})$
$C_{22} = \text{divmatmul}(A_{21}, B_{12}) + \text{divmatmul}(A_{22}, B_{22})$

return C
end-if
end-func

Strassen's Algorithm:

func Strassen(A,B):
n:= size(A)

if n equals 2:
return A*B
else:
$P = \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$
$Q = \text{Strassen}(A_{21} + A_{22}, B_{11})$
$R = \text{Strassen}(A_{11}, B_{12} - B_{22})$
$S = \text{Strassen}(A_{22}, B_{21} - B_{11})$
$T = \text{Strassen}(A_{11} + A_{12}, B_{22})$
$U = \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$
$V = \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$
$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

return C
end-if
end-func

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	A = [[1,1],[1,1]] B = A	C = [[2,2],[2,2]]	√	√	√
2	A = [[1,2],[3,4]] B = [[5,6],[7,8]]	C = [[19,22], [43,50]]	√	√	√
3	A = [[1,2,3,4], [4,5,6,7], [7,8,9,10], [11,12,13,14]] B = [[15,16,17,18], [19,20,21,22], [23,24,25,26],	C = [[230 240 250 260] [482 504 526 548] [734 768 802 836] [1070 1120 1170 1220]]	√	√	√
4	A = [[4,-3],[2,8]] B = [[4/19,3/28], [-1/19,2/19]]	C = [[1. 0.11278195] [0. 1.05639098]]	√	√	√
5	A =[[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]] B =A	C = [[90 100 110 120] [202 228 254 280] [314 356 398 440] [426 484 542 600]]	√	√	√
6	A = [[4,-3],[2,8]] B =[[0,1],[1,0]]	C =[[-3 4] [8 2]]	√	√	√
7	A = [[1,3],[7,5]] B = [[1,7],[3,5]]	C = [[10 22] [22 74]]	√	√	√
8	A = [[2,2],[2,2]] B =A	C = [[8 8] [8 8]]	√	√	√
9	A = [[0,0],[0,0]] B = [[10,20], [30,40]]	C = [[0 0] [0 0]]	√	√	√

10	A = [[0,1],[1,0]] B =A	C = [[1 0] [0 1]]	√	√	√

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 $X = \{x_1, x_2, x_3 \ldots x_m\}$ 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.

1. I plan on using the *numpy* library to generate random n x n *numpy matrices* with **np.random.randn()**. All algorithms will run over the same matrix before generating a new matrix.

2. I plan to use m = 10 as the number of iterations per trial.

Part 2: Implementation (35 points)

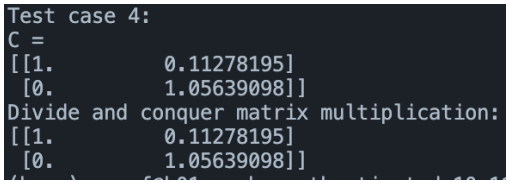
a. Code each program based on the design (pseudocode or flow chart) given in Part 1(b).

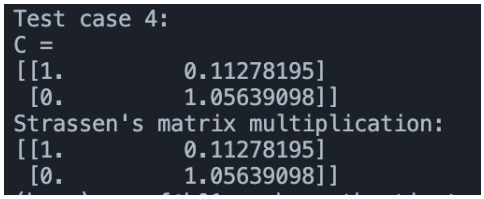
<Generate three programs with proper documentation and store them in three files. Note: They are required to be submitted to Canvas as described in the submission instructions>
<No insert here>

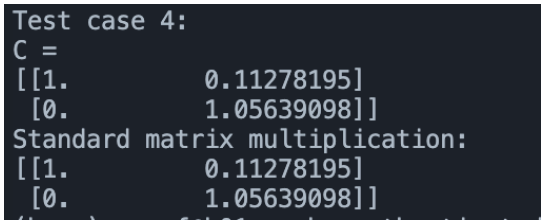
b. Test you program using the designed testing input data given in the table in Part 1(c), Make sure each program generates the correct answer by marking a "√" if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

<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







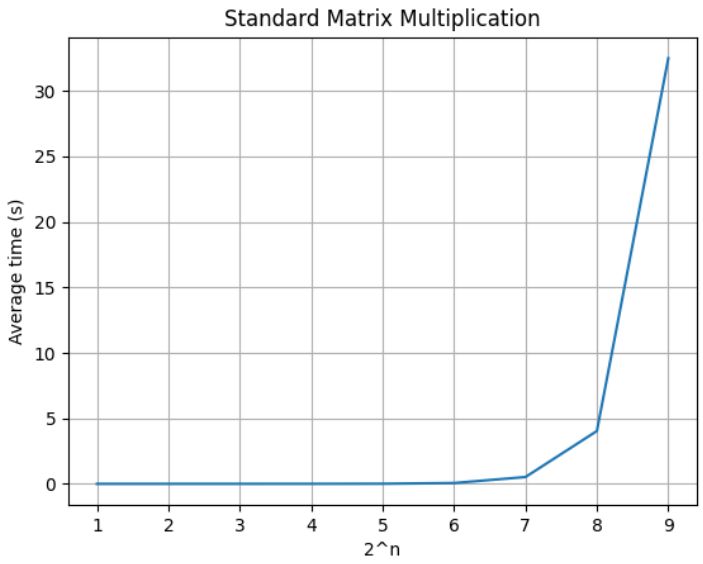
By now, three working programs for the three algorithms are created and ready for experimental study in the next part, Part 3.

Part 3: Comparative Analysis (35 points)

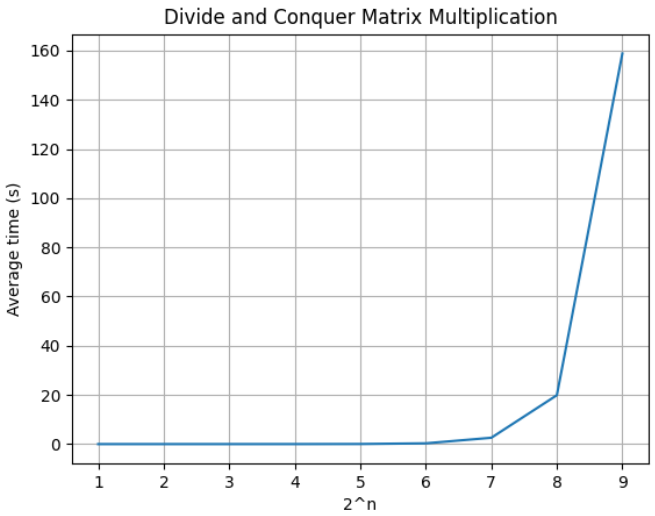
a. Run each program with the designed randomly generated input data given in Part 1(d). Generate a table for all the experimental results as follows.

Input Size n	Average time (Classical matrix multiplication)	Average Time (Divide-and-conquer matrix multiplication)	Average Time (Strassen's matrix multiplication)
2	3.63 x 10^-6 s	1.00 x 10^-5 s	4.17 x 10^-6 s
4	1.91 x 10^-5 s	8.01 x 10^-5 s	4.11 x 10^-5 s
8	1.38 x 10^-4 s	6.45 x 10^-4 s	3.00 x 10^-4 s
16	1.03 x 10^-3 s	4.96 x 10^-3s	2.00 x 10^-3 s
32	7.94 x 10^-3 s	3.88 x 10^-2 s	1.38 x 10^-2 s
64	6.33 x 10^-2 s	3.13 x 10^-1 s	9.94 x 10^-2 s
128	5.19 x 10^-1 s	2.58 x 10^0 s	7.19 x 10^-1 s
256	4.03 x 10^0 s	1.98 x 10^1 s	4.77 x 10^0 s
512	3.25 x 10^1 s	1.58 x 10^2 s	3.34 x 10^1 s

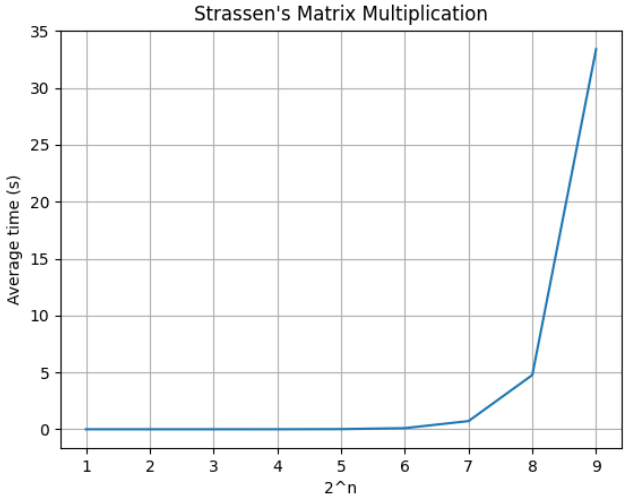
b. Plot a graph of each algorithm and summarize the performance of each algorithm based on its own graph.



Here we see that Standard Matrix Multiplication grows exponentially with the input size, which roughly starts to grow at around n=7. It reaches a maximum of 30s on average at n=9.

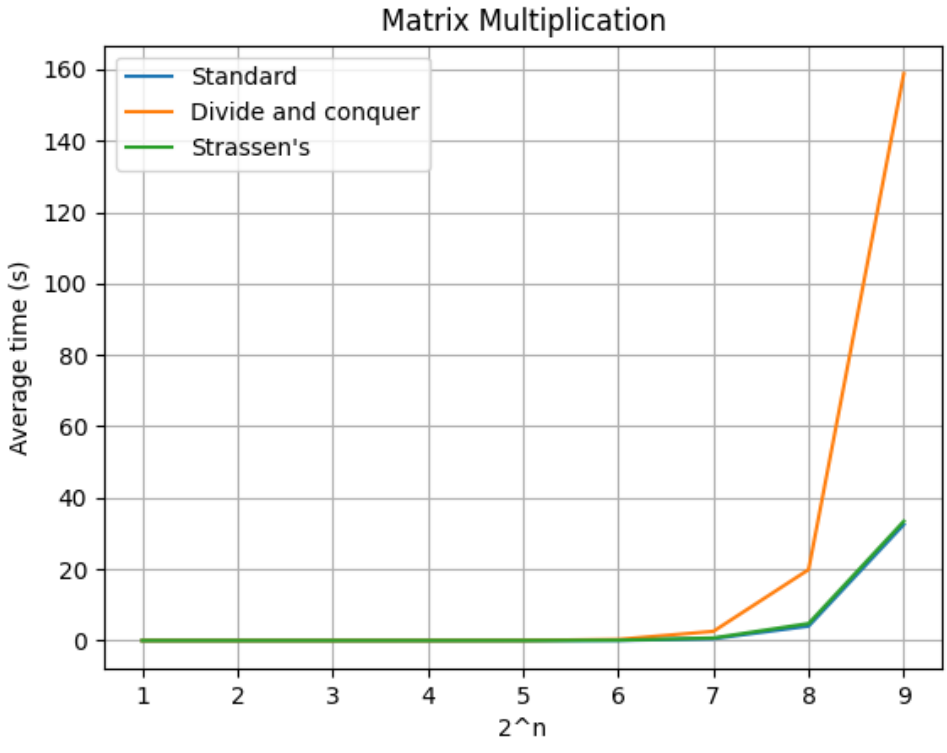


Here we see that Divide-and-Conquer Multiplication grows exponentially with the input size, which roughly starts to grow at around n=7. It reaches a maximum of 160s on average at n=9.



Here we see that Strassen's Matrix Multiplication grows exponentially with the input size, which roughly starts to grow at around n=7. It reaches a maximum of 33s on average at n=9.

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.



Here we see that Strassen's and Standard Matrix multiplication have similar rates of growth from the data, however we see that a Strassen's is slightly worse for smaller matrix sizes. We also note that Divide and Conquer has the worst (fastest) growth rate in time with the increase in matrix size.

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

From the theoretical results we should have seen that both Divide-and-Conquer and Standard Matrix multiplication have similar results in rate of growth, while Strassen should have the best (slowest) growth rate. Our empirical results are quite different, which may be due to the fact that the input matrix sizes were not sizable enough to yield the trade-off that Strassen and Divide-and-Conquer provide. For the matrix sizes ran for the experiment we note that Standard matrix multiplication is still best in practice and for much larger sizes we should see Strassen and Divide-and-Conquer beat out Standard. There may have also been possible error in the recursive implementation of both Strassen and Divide-and-Conquer that may have contributed to the slower runtimes in practice given the runtime environment.

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

Model: Macbook Air
Chip: Apple M2
Memory: 8GB
OS: MacOS Sonoma
IDE: VSCode

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.

I believe that my experiment outlined some of the limitations of the divide-and-conquer algorithms discussed in-class in terms of implementation. As we saw from above, for too small of problems it turns out that divide and conquer strategies can have worse performance than standard procedures, such as Standard matrix out performing Strassen's. If I could run on a different environment with longer runtimes, I would re-run the experiment on a larger matrix sizes in order to see the trade-off benefit of both the Divide-and-Conquer and Strassen's matrix multiplication algorithms. I would still keep the same programming language and data set generation, since I think in practice they are the most convenient and most reliable in getting meaningful results out of the experiment. Though running the algorithm on faster languages, such as C/C++ may yield faster runtimes and hence more time for running on larger matrix sizes. I think running on my macbook laptop set some constraints on the size of matrices I could run, given the limited

computing power, plus lack of fan cooling meant that heavier loads caused throttling on performance way too earlier in the experiment. If I could do the experiment again, I would run on a device with more computing power in order to run larger matrices.