Name: Jorge Aranda

(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

Test Strategies

My program works by first asking the user to input a number ‘n’ for a nxn matrix. My program then creates two different nxn matrices with a random set of numbers from 1 – 100. The numbers are always random and thus each test case should be different. Once the two matrices are created, my program then runs it through a method that would conduct each algorithm. Right before and after each method is called, I set two timers to record the time taken for the code to be executed. One is recorded in nanoseconds and the other is recorded in milliseconds. Each algorithm is executed with the same exact matrix so that it ensures each algorithm is being tested with the same results.

My program can continue running until the user enters -1. Once the user ent ers this sentinel value, the program finishes running. To ensure that each test case runs with the maximum amount of processing power, I closed every other program except excel. I kept excel open in order to input my values after each test case. Each test case in the following data tables was conducted as follows: First, I would run the program. After the program starts to run, I enter 2 so that my program creates two 2x2 matrices and then executes each algorithm on them. Once it finishes executing, I enter 4, 8, 16, and so on until I got to 512. My computer was not able to execute the code at 1024x1024, it would take extremely long to execute. After I reached 512, I stopped the program and recorded my data. I would then restart the program and repeat the process for the other test cases. I conducted a total of 20 test cases and recorded my data in excel. I have attached the excel document along with this report. The excel document contains all the test cases as well charts for all the different nxn test cases.

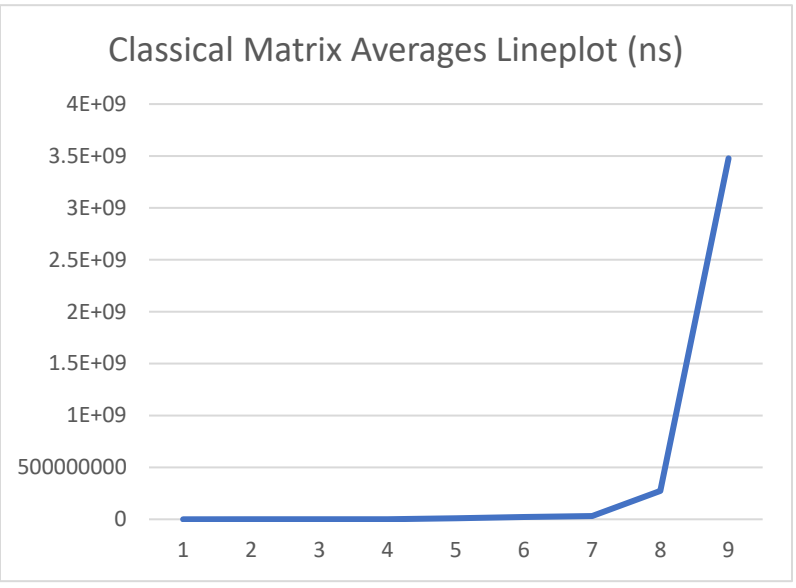
Unfortunately, the algorithm I used to get my Divide and Conquer algorithm working was only able to make it to 16x16. Once it attempted to do a matrix at 32x32, it would cause a stack overflow error. So, I was only able to conduct test cases up to 16x16 for my Divide and Conquer algorithm. I first commented out the other two algorithms in order to conduct the Divide & Conquer algorithm test cases first. After I finished collecting data for the Divide & Conquer algorithm I commented out the Divide & Conquer algorithm to conduct the other two algorithms without running into the stack overflow error.

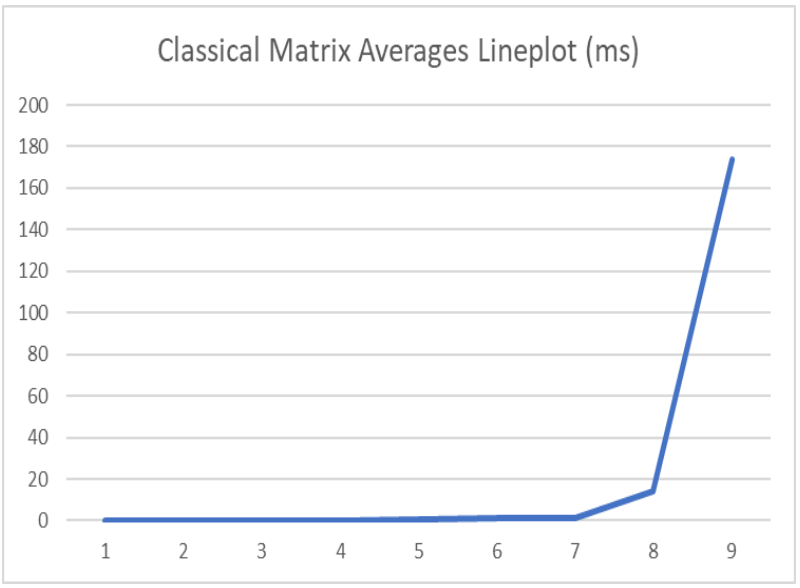
The following three pages contains my data sets for each algorithm, charts for the averages of the nxn test cases, as well as averages for nanoseconds and milliseconds on each nxn test case.

Classical Matrix Multiplication Algorithm Data Set

	2x2		4x4		8x8		16x16		32x32		64x64		128x128		256x256		512x512	
	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds
1	2500	0	2700	0	10100	0	68300	0	503600	1	1145900	1	1276200	1	13810800	14	187210300	187
2	2600	0	2700	0	10200	0	69600	0	504500	1	1096600	1	1289100	1	13660200	14	158695800	158
3	2500	0	2700	0	10300	0	68200	0	500900	0	1165500	1	1284500	2	13279200	13	167531800	168
4	2600	0	3500	0	10200	0	68100	0	502900	1	1115900	1	1279400	1	13222800	14	225244700	224
5	2600	0	2600	0	10300	0	68400	0	504200	1	1111900	1	1285400	1	13322800	14	166144200	165
6	2700	0	2700	0	10400	0	68400	0	501600	0	1232300	1	1307300	1	13706000	13	153586900	153
7	2500	0	2700	0	10300	0	68400	0	509000	1	1131100	1	1278600	1	13858400	14	156763900	158
8	2900	0	2700	0	10300	0	68700	0	501800	0	1117200	1	1273800	1	13679500	13	165595000	165
9	2500	0	2800	0	10100	0	70000	0	502500	0	1129400	1	1314400	1	13532800	14	167219200	167
10	2800	0	2600	0	10200	0	68200	0	501200	0	1147100	1	1291800	1	13562200	14	155467500	156
11	2500	0	2700	0	10700	0	70000	0	500500	0	1161800	1	1280600	2	13667600	14	167853000	168
12	3100	0	2600	0	10300	0	68700	0	503700	0	1200200	1	1367500	1	13346400	14	158837500	160
13	2700	0	2700	0	10300	0	68300	0	501700	0	1209000	1	1291700	2	13381600	13	175899500	177
14	2600	0	2600	0	10200	0	68100	0	501600	0	1116200	1	1307000	2	13536000	14	179599600	180
15	2600	0	2700	0	10100	0	68600	0	503500	1	1130600	1	1276700	1	13406500	14	172895100	172
16	2300	0	2800	0	10200	0	69900	0	500700	0	1145900	1	1289600	1	13713700	14	246816300	246
17	2500	0	2700	0	10300	0	70600	0	506900	1	1104300	1	1272700	1	13945100	14	196196400	196
18	2700	0	2700	0	10300	0	68400	0	515600	1	1095700	1	1283400	2	13308600	14	158384600	159
19	2500	0	8600	0	10300	0	68100	0	502000	0	1104300	1	5432000	5	16001400	16	156097800	151
20	2400	0	2700	0	10300	0	68400	0	502000	0	1161800	1	2323800	2	13935600	14	161001600	161

nxn	2x2	4x4	8x8	16x16	32x32	64x64	128x128	256x256	512x512
Nanosecond Averages	52100	60500	205400	1375400	10070400	22822700	31005500	273877200	3477040700
Millisecond Averages	0	0	0	0	0.4	1	1.5	13.9	173.55

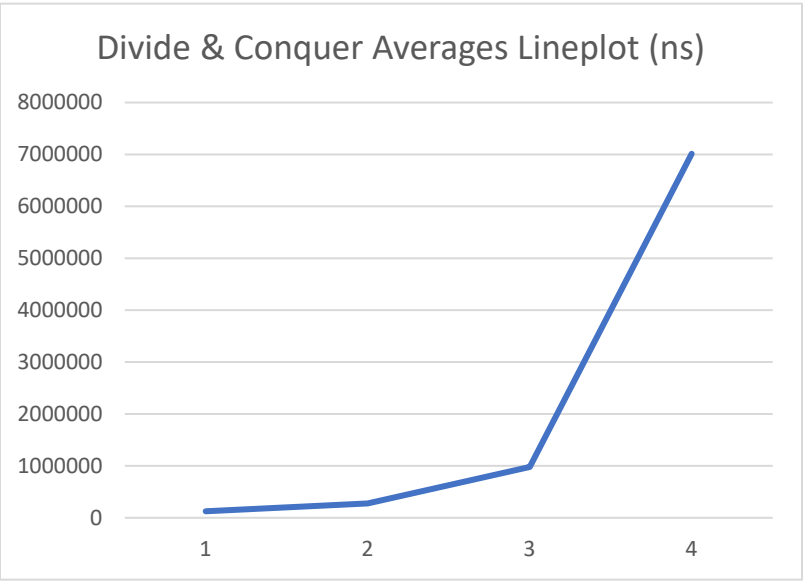


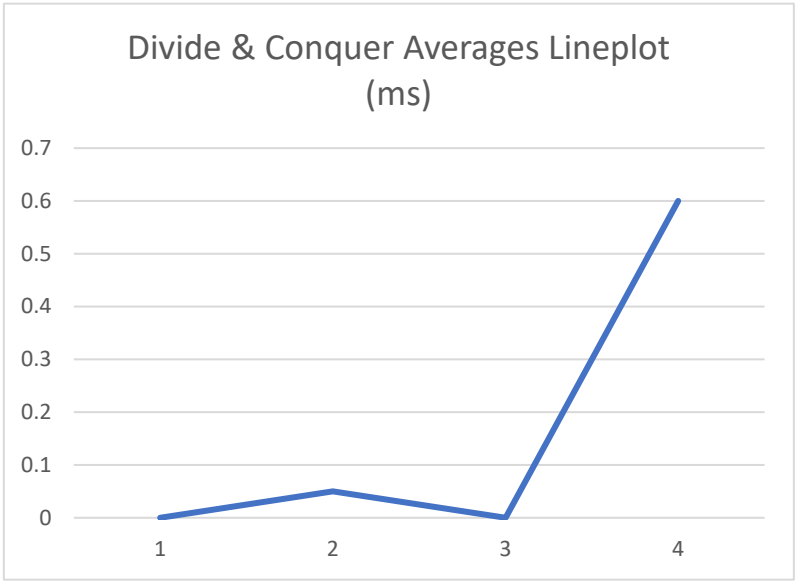


Divide & Conquer Algorithm Data Set

nxn	2x2		4x4		8x8		16x16		32x32
Test Case	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	
1	6000	0	15000	0	54900	0	381100	1	Stack Overflow
2	6300	0	13000	0	4970	0	380400	1	Stack Overflow
3	6200	0	14200	1	51500	0	370900	1	Stack Overflow
4	6100	0	12800	0	50700	0	385700	1	Stack Overflow
5	6100	0	12700	0	52300	0	400700	0	Stack Overflow
6	6100	0	14900	0	54400	0	370900	1	Stack Overflow
7	8700	0	2300	0	6900	0	54300	1	Stack Overflow
8	5900	0	12300	0	67200	0	377400	0	Stack Overflow
9	6000	0	13700	0	57200	0	458100	0	Stack Overflow
10	6200	0	14000	0	52900	0	389300	1	Stack Overflow
11	6100	0	12900	0	54000	0	385500	1	Stack Overflow
12	6200	0	13000	0	50700	0	376500	1	Stack Overflow
13	6300	0	13900	0	53000	0	384200	1	Stack Overflow
14	6200	0	14200	0	53500	0	374100	0	Stack Overflow
15	6200	0	13800	0	51100	0	383700	0	Stack Overflow
16	6000	0	13800	0	52800	0	383400	1	Stack Overflow
17	6300	0	13600	0	50400	0	372500	0	Stack Overflow
18	6000	0	30000	0	50900	0	371200	0	Stack Overflow
19	6400	0	12400	0	56100	0	37600	1	Stack Overflow
20	5900	0	14100	0	52800	0	374900	0	Stack Overflow

nxn	2x2	4x4	8x8	16x16
Nanosecond Averages	125200	276600	978270	7012400
Millisecond Averages	0	0.05	0	0.6

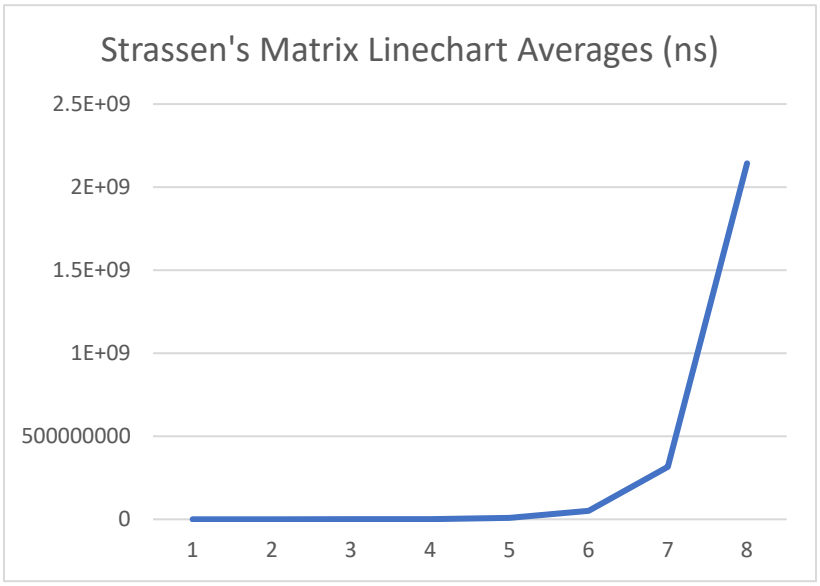


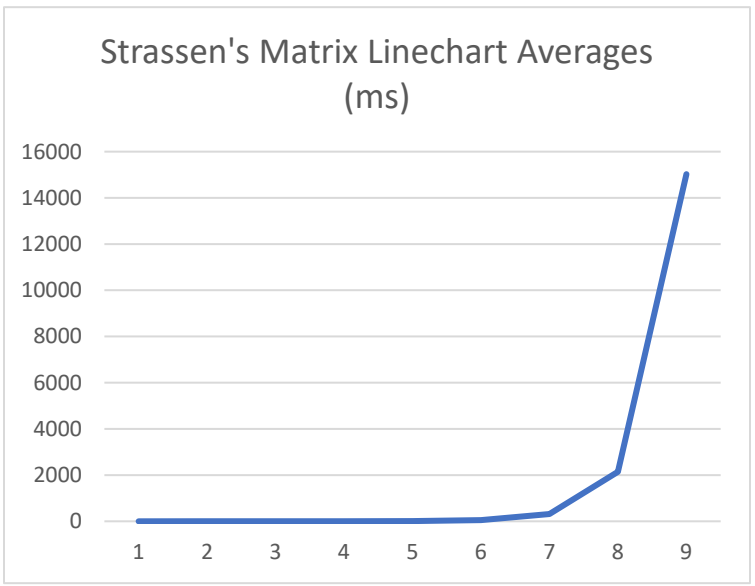


Strassen's Matrix Multiplication Data Set

	2x2		4x4		8x8		16x16		32x32		64x64		128x128		256x256		512x512	
	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	Nanoseconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds	NanoSeconds	Milliseconds
1	14500	0	68700	0	500300	0	1306400	2	8420700	8	53123300	53	309400900	310	2131842700	2133	14897376300	14897
2	14400	0	68200	0	468300	0	1316100	1	8157500	8	52475200	52	309629100	310	2136743000	2137	14959988000	14961
3	14600	0	67400	0	484100	1	1321900	1	8509900	9	52989100	53	312930000	312	2143692800	2144	15045504900	15045
4	14800	0	68300	0	455000	0	1408100	2	8324500	8	55343800	55	311531600	312	2137060000	2137	15025063200	15026
5	14400	0	67300	0	524100	0	1322400	2	14484400	14	52791100	53	310492300	311	2133144300	2133	14906748500	14907
6	14700	0	68800	1	571200	1	1330800	1	8201600	7	53670800	54	314725100	315	2151148200	2150	15129939600	15130
7	14300	0	67900	0	483800	0	1325700	1	8305500	8	55077700	55	313693500	313	2149853800	2149	15131470900	15131
8	14700	0	67500	0	467000	0	1343700	1	8284300	8	53822100	54	314606600	315	2146428400	2147	15099410700	15100
9	14300	0	68500	0	465900	0	1352400	2	8227000	9	52864400	53	315357100	315	2139192600	2139	15028590500	15029
10	14600	0	68400	0	459200	1	1323200	2	8184500	9	52411500	51	311486700	311	2130397900	2130	14945784300	14946
11	15800	0	68100	0	671700	1	1293900	2	8191900	9	52936100	53	312322300	312	2141167000	2140	14909240300	14909
12	14900	0	68000	0	506200	1	1323200	1	8187500	8	54915000	55	313265500	313	2151509200	2141	15044100800	15043
13	14900	0	68000	0	476800	1	1325700	1	8255300	9	52941300	54	312643800	312	2158386400	2160	15014450300	15013
14	15200	0	67400	0	487800	0	1329100	1	8199800	9	52831700	52	313749200	313	2148686600	2148	14136337900	15137
15	14600	0	67200	0	458200	0	1383300	2	8209500	8	52648900	53	310013700	311	2126636200	2127	14949806600	14950
16	15200	0	68400	0	469500	0	1316700	2	8316300	9	53005000	53	311142200	310	2129512300	2129	14915720100	14917
17	14600	0	69000	0	474000	1	1334800	2	8218700	8	52952100	52	314786600	315	2147433800	2148	15116573900	15116
18	14400	0	67700	0	455100	0	1322300	2	8350800	8	53363500	53	313836700	313	2142415700	2142	15027944300	15027
19	14800	0	153500	0	10300	0	1317900	1	8154200	8	52936100	53	322670300	323	2150030700	2150	15104404300	15105
20	14600	0	67600	0	470300	1	1326400	1	8204700	9	53519000	54	363900500	364	2166445100	2167	15119350500	15119

nxn	2x2	4x4	8x8	16x16	32x32	64x64	128x128	256x256	512x512
Nanosecond Averages	14715	72295	467940	1331200	8569430	50922530	315609185	2143086335	1.50E+10
Milliseconds Averages	0	0.05	0.4	1.5	8.65	53.25	315.5	2142.53	15025.4





Classical Matrix Multiplication Data Set Explanation

Classical Matrix Multiplication is said to have a time complexity of $O(n^3)$ since we use a total of 3 for loops. Through the data I collected, classical matrix multiplication performed well with all of the matrices sizes I tested it on. Classical Matrix Multiplication had the best performance, surprisingly, compared to the other two algorithms. As the sizes of the matrices grew, the averages for the time taken began to increase exponentially. It is much easier to tell by the recorded nanoseconds, since it wasn't able to begin clocking in milliseconds until the 32x32 matrices. Even then, it was not consistently recording milliseconds. You are able to tell much easier by the graphs of its exponential growth.

Divide & Conquer Algorithm Data Set Explanation

Divide and Conquer, like Classical Matrix Multiplication, is said to have a time complexity of $O(n^3)$. You can somewhat notice the time complexity with the average time in nanoseconds more so than the milliseconds. Unfortunately, the way that I implemented it did not allow me to try any matrices bigger than 16x16 as I ran into stack overflow errors once I reached 32x32. Nevertheless, you can still somewhat see the time complexity in the graphs provided with both the nanoseconds as well as the milliseconds. I have tried to implement it another way but I still ran into stack overflow errors. I think this is just common when trying the Divide and Conquer algorithm and it is best used with smaller matrices.

Strassen's Matrix Multiplication Algorithm Data Set Explanation

Strassen's Matrix Multiplication is supposed to be an improvement of both Classical Matrix Multiplication and Divide & Conquer Algorithms at a time complexity of $O(n^2)$. However, with my test cases, Classical Matrix Multiplication performed better than Strassen's Matrix Multiplication. This was only up until 512x512 however. Unfortunately, my computer was not able to handle anything more. From the graphs of the average times you are able to see this exponential growth in both nanoseconds and milliseconds. Surprisingly, Strassen's Matrix Multiplication took much longer than Classical Matrix Multiplication in most instances.

Theoretical Complexity Comparisons and Conclusion

As mentioned previously, both Classical Matrix Multiplication and Divide & Conquer both have a time complexity of $O(n^3)$ and Strassen's Matrix Multiplication has a time complexity of $O(n^2)$. Theoretically, the Strassen's Matrix Multiplication should perform better than both other algorithms. However, through my test cases, I was not able to come to that conclusion. Classical Matrix Multiplication performed better than both other algorithms. Nevertheless, theoretically, I am sure that once we get to much bigger matrices such as 4096, Strassen's Algorithm should start to outperform both Classical Matrix Multiplication and Divide and Conquer.

Through the test cases that I was able to get from my Divide and Conquer algorithm, it never did perform better than Classical Matrix Multiplication. So, from that, I will conclude that classical matrix multiplication should be faster than Divide and Conquer all the time, despite the fact that they have the same time complexity. I believe this might be the case because Classical Matrix Multiplication is much simpler, being only 3 for loops, while Divide and Conquer is done through a series of recursive calls. However, I was unfortunately unable to 100% confirm this as the way I implemented Divide and Conquer, I ran into stack overflow errors. Nevertheless, I still do believe that Classical Matrix Multiplication might perform slightly better than Divide and Conquer.

The strengths of my work are that I did a lot of comparisons. With a total of 20 test cases, I was able to bring a fairly accurate average for all the nxn cases for each algorithm. With this many cases I was able to give decent graphs for the data. On the excel sheet, I was also able to create many graphs for each individual nxn case for both nanoseconds and milliseconds for a more accurate representation of each case. The constraints of my work have to do with both my computer as well as my Divide and Conquer algorithm. My computer was not able to handle much test cases past 512. This is definitely a big constraint as I was not able to confirm if Strassen's performs better on bigger matrices. My Divide and Conquer algorithm were also another big constraint. The way I implemented it I was not able to do more than 16x16 matrices and thus unable to confirm if there are any cases where it outperformed Classic Matrix Multiplication.