Name: Gary Huang

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	
(100 points)	**Total**	

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

Total: 100 Points

100

Data Sets, Test Strategies, and Results

Data

Operating System: macOS
Processor: 2.8Ghz Quad-Core
Memory: 16GB DDR3
IDE: IntelliJ

Naïve method:

N = 4, $runtime_{avg}$ = 0.000005s
N = 16, $runtime_{avg}$ = 0.000152s
N = 32, $runtime_{avg}$ = 0.000481s
N = 64, $runtime_{avg}$ = 0.001403s
N = 128, $runtime_{avg}$ = 0.003839s
N = 256, $runtime_{avg}$ = 0.019816s
N = 512, $runtime_{avg}$ = 0.288399s
N = 1024, $runtime_{avg}$ = 2.703397s
N = 2048, $runtime_{avg}$ = 61.346637s
N = 3000, $runtime_{avg}$ = 242.196490s

Divide and Conquer:

Implementation 1 (Bad):

N = 4, $runtime_{avg}$ = 0.000121s
N = 16, $runtime_{avg}$ = 0.002563s
N = 32, $runtime_{avg}$ = 0.017437s
N = 64, $runtime_{avg}$ = 0.129520s
N = 128, $runtime_{avg}$ = 0.810628s
N = 256, $runtime_{avg}$ = 6.328628s
N = 512, $runtime_{avg}$= 50.367202s
N = 1024, $runtime_{avg}$ = 408.770645s

Implementation 2 (Good):

N = 4, $runtime_{avg}$ = 0.000050s
N = 16, $runtime_{avg}$ = 0.002032s
N = 32, $runtime_{avg}$ = 0.007873s
N = 64, $runtime_{avg}$ = 0.060710s
N = 128, $runtime_{avg}$ = 0.563504s
N = 256, $runtime_{avg}$ = 2.838304s
N = 512, $runtime_{avg}$= 22.382253s
N = 1024, $runtime_{avg}$ = 175.405727s

Strassen's Method:

N = 4, $runtime_{avg}$ = 0.000138s
N = 16, $runtime_{avg}$ = 0.003991s
N = 32, $runtime_{avg}$ = 0.020121s
N = 64, $runtime_{avg}$ = 0.125625s
N = 128, $runtime_{avg}$ = 0.745805s
N = 256, $runtime_{avg}$ = 4.563777s
N = 512, $runtime_{avg}$ = 32.920223s
N = 1024, $runtime_{avg}$ = 217.722262s

Test Strategies

Matrices increased in size among all three programs. Each matrix size input was the same for all programs to maintain consistency all throughout. I also tested to see if integer values made a difference. My functions allowed for me to edit values fairly easily. I tried all values of 1, random integers, and a predefined values scaling from 1-25 and resetting to 1 at 25. Each run of the matrix was meant to calculate an average of 5 runs, outputting the result in seconds up to the 6^{th} decimal.

In the naïve approach, a simple 3-way nested loop was created. No recursion was needed here, but I made sure to keep in mind the theoretical runtime is O(n^3).

Since theory of each runtime is being tested, I implemented the divide and conquer algorithm in two different formats. Implementation 1 used a lot more space due to the partitioning of each subarray. This test was basically the precursor to applying Strassen's method, however, instead of applying the formulas he derived, it did a simple partitioning call, then the program uses a merge call of all arrays to combine the previously separated sub-arrays. Implementation 2 was a more streamlined approach. This implementation used less function calls. In total, the main method calls two functions whereas the previous implementation utilizes three function calls. Implementation 2 does not use a partitioning method call, instead relies specifically on recursion to apply partitioning itself and then applies and iterative approach to merge, rather than calling a separate merge function.

For Strassen's algorithm, this was used primarily to compare with the other two methods of matrix multiplication. I did create a minor implementation change to observe just how badly the would run. The adjustment I made involved taking the 4 subarrays and simply call functions that would, otherwise, normally be allocated to a portion in memory as an array. implementation, while terrible in practice, this really shows how expensive recursive calls can be on a standard computer, absent of some sort of parallel processing system.

Results

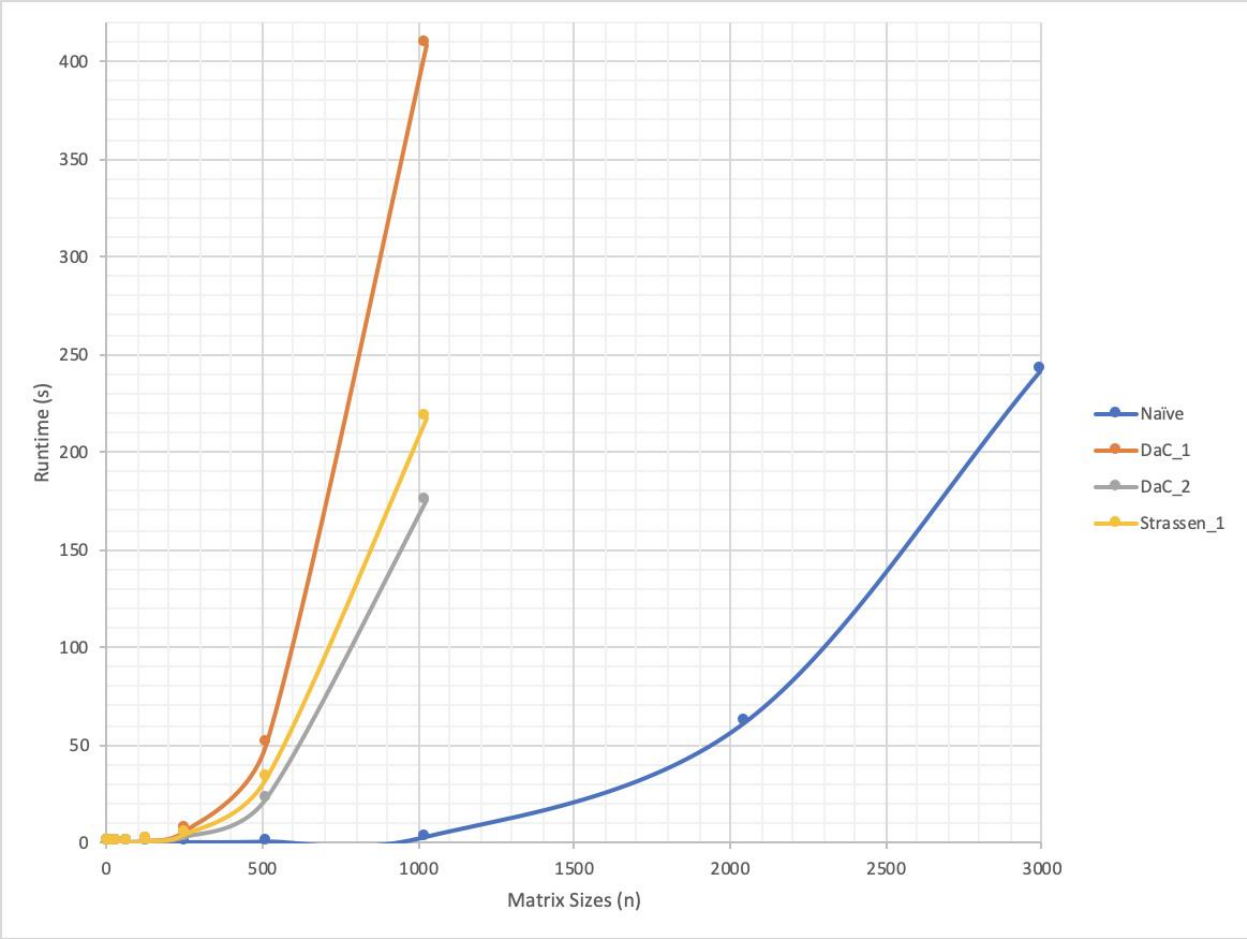

Figure 1 - Graph of Runtimes, Relative to 'n' matrix sizes

The graph above indicates the runtimes relative to the matrix sizes. Strassen implementation 2 was excluded because it was something that no programmer would ever do and is worse than all other implementations. Clearly, the most efficient program was the naïve approach. I was surprised because the theory behind our divide and conquer approach was indicating faster run times, but after running my programs multiple times I found that the classical matrix multiplication algorithm was definitely superior. The naïve algorithm only really begins to have problems around n=3000 whereas the more efficient Divide and Conquer algorithm had issues at n=1024. Similarly, Strassen's proved to be no better, if anything performing worse than the standard Divide and Conquer implementation 2. It is indicated that implementation 1 is much worse than implementation 2 of the standard Divide and Conquer algorithms as well. The explanations of all of these concerns will be addressed in after the theoretical complexity analysis.

Theoretical Complexity Analysis

Naïve

Proof of concept:

$$T(n) = \sum_{k=1}^{n} (A_{ik} * B_{kj})$$

$$= \sum_{k=1}^{n} (A_{ik} * B_{kj}) \text{ each matrix A and B n times gives us:}$$

$$= \sum_{k=1}^{n} (n * n) = \sum_{k=1}^{n} n^2 \text{ and sum our multiplication n times gives us:}$$

$$= \text{n} * n^2$$

$$= n^3$$

Divide and Conquer

Proof of concept:

$$T(n) = aT\left(\frac{n}{b}\right) + bn^2$$

$$T(1) = 1$$

Since we are working with 8 sub arrays, each of size n/2 we have the following:

$T(n) = 8T\left(\frac{n}{2}\right) + bn^2$ Solving this recurrence relation using substitution we get:

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

$$= 8\left[8T\left(\frac{n}{2^2}\right) + \left(\frac{n^2}{2^2}\right)\right] + bn^2 \qquad T\left(\frac{n}{2^2}\right) = 8T\left(\frac{n}{2^3}\right) + b(\frac{n}{2^2})^2$$

$$= 8^2T\left(\frac{n}{2^2}\right) + 2bn^2 + bn^2 \qquad T\left(\frac{n}{2^3}\right) = 8T\left(\frac{n}{2^4}\right) + b(\frac{n}{2^3})^2$$

$$= 8^3T\left(\frac{n}{2^3}\right) + 4bn^2 + 2bn^2 + bn^2 \qquad T\left(\frac{n}{2^4}\right) = 8T\left(\frac{n}{2^5}\right) + b(\frac{n}{2^4})^2$$

$$= 8^4T\left(\frac{n}{2^4}\right) + 8bn^2 + 4bn^2 + 2bn^2 + bn^2$$

$$8^iT(1) + bn^2(\frac{2^{3i-3}}{2^{2i-2}})$$

$$8^i = n \qquad \frac{2^{3i-3}}{2^{2i-2}} = n$$

$$i = \lg(n-8) \qquad \frac{2^{3i}*2^{-3}}{2^{2i}*2^{-2}} = n \rightarrow n = \frac{2^{3i}}{2^{2i}*2} \rightarrow 2^{3i} = n\left(2^{2i+1}\right) \rightarrow 3i = 2in + lgn$$

Substituting back into our equation… $8^iT(1) + bn^2(\frac{2^{3i-3}}{2^{2i-2}})$

$$= 2^{3^i} + bn^2(\frac{2^{3\log(n-8)-3}}{2^{2\log(n-8)-2}}),\ 3i = 2in + logn$$

$$= 2^{2in+lgn} + (\frac{n-8}{2})bn^2$$

$= [(n-8)n^2 + n] + \left[\frac{bn^3 - 8bn^2}{2}\right]$

$= \left(n^3 - 8n^2 + n + \frac{bn^3 - 8bn^2}{2}\right) \rightarrow bn^3 + n^3 = n^3(b+1)$

Therefore, this is still a O(n^3) at the end of the day.

We can also apply recurrence relations using the tree method:

$$C_n \rightarrow bn^2$$

$$C_{\frac{n}{2}} + \cdots (2^3\ times) \cdots +\ C_{\frac{n}{2}} \rightarrow 2^3 b\left(\frac{n}{2}\right)^2 = 2bn^2$$

$$C_{\frac{n}{4}} + \cdots\ (2^3)^2\ times \cdots +\ C_{\frac{n}{4}} \rightarrow\ 2^6 b\left(\frac{n}{2^2}\right)^2 = 4bn^2$$

$$\vdots\ continues\ until\ base\ case$$

$$1 \rightarrow 2^{logn} bn^2 = O(n^3)$$

I must note that if implementation 2 were introduced, we get a runtime faster than Strassen's on my particular computer. However, the theoretical analysis would still find this particular implementation continues to run at $O(n^3)$. The reason for this explanation will be explained in the next section.

Strassen's Algorithm

Proof of concept using recurrence relations:

$T(n) = aT\left(\frac{n}{b}\right) + cn^2$

$T(1) = 1$

We are now working with 7 subarrays, each of size n/2:

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

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

$= 7\left[7T\left(\frac{n}{2^2}\right) + \left(\frac{n^2}{2^2}\right)\right] + cn^2$ $\qquad$ $T\left(\frac{n}{2^2}\right) = 7T\left(\frac{n}{2^3}\right) + c(\frac{n}{2^2})^2$

$= 7^2T\left(\frac{n}{2^2}\right) + 2cn^2 + cn^2$ $\qquad$ $T\left(\frac{n}{2^3}\right) = 7T\left(\frac{n}{2^4}\right) + c(\frac{n}{2^3})^2$

$= 7^3T\left(\frac{n}{2^3}\right) + 4cn^2 + 2cn^2 + cn^2$ $\qquad$ $T\left(\frac{n}{2^4}\right) = 7T\left(\frac{n}{2^5}\right) + c(\frac{n}{2^4})^2$

$= 7^4T\left(\frac{n}{2^4}\right) + 8cn^2 + 4cn^2 + 2cn^2 + cn^2$

$7^iT(1) + 2^{i-1}cn^2$

$7^i = n$ $\qquad$ $2^{i-1} = n$

$i = \lg(n-7)$ $\qquad$ $i = \lg(n-2) + 1$

Substitute and do some work…

$= 7^{\lg(n-2)+1} + 2^{\lg(n-7)}cn^2$

$= (\frac{7}{4})^{\log\, n} cn^2$
$= n^{0.8074} * bn^2$
$= O(2^{2.8074})$

We can also apply recurrence relations using the tree method:

$$C_n \to bn^2$$
$$C_{\frac{n}{2}} + \cdots (7\ times) \cdots +\ C_{\frac{n}{2}} \to 7b\left(\frac{n}{2}\right)^2 = \frac{7}{4}bn^2$$
$$C_{\frac{n}{4}} + \cdots\ (7)^2\ times \cdots +\ C_{\frac{n}{4}} \to\ 7^2 b\left(\frac{n}{2^2}\right)^2 = (\frac{7}{4})^2 bn^2$$
$$\vdots\ continues\ until\ base\ case$$
$$1 \to (\frac{7}{4})^{logn} bn^2 = O(n^{2.8074})$$

The theoretical analysis of each function: naïve, Divide and Conquer, and Strassen's indicate that the following in terms of runtime (from fastest to slowest):

Strassen's < naïve = Divide and Conquer

However, we know this is not true in our experiment. We must keep in mind that there are more powerful systems out there and this test alone does not verify that Strassen's Algorithm is worse than the naïve approach in all scenarios.

Classical V.S. Divide and Conquer

Comparing the theoretical values of all 3 algorithms, the one that may seem to be the obvious winner is actually the worst runtime of the 3 (if we exclude implementation 2 of the DaC algorithm); however, given the 2 different implementations of the divide and conquer method, there is a discrepancy the ranking of worst algorithm. Lets take a look at Implementation 1 for the standard Divide and Conquer algorithm.

With implementation 1, we are simply just using a simplified version of Strassen's algorithm. Without any of the optimizations he provided, it is obvious that the algorithm runs slower than Strassen's method. Although, when we implementation 2, this method is clearly faster than Strassen's algorithm. Because we are no longer requiring to explicitly call partitioning functions, each recursive call to the main function has less work. For instance, implementation 1 calls 8 partitioning functions. This alone means every time a recursive call is made, it explicitly runs an additional 8 function calls, per partition. Furthermore, we are also calling 4 final merge functions. What implementation 2 does differently, is simply require a larger parameter to be input, and does the partitioning and merging through recursion alone without explicit partitioning and merging functions. Even though implementation 2's method is fantastic, it still leaves the main issue unanswered. Why is the classical/naïve method of matrix multiplication better than our attempts at using a theoretically superior algorithm, Strassen's Algorithm?

The main issue I see comparing the two algorithms is based around how a computer utilizes its resources when dealing with allocation of memory. Both divide and conquer algorithms require an upfront cost in resource when they split arrays. Moreover, the expensive nature of utilizing recursion only adds to the issues. Recursion must make an additional n loops per call which makes it no better than our naïve algorithm. For example, in our divide and conquer implementation 2, the code calls to add(mult(a,b), mult(a,b)). This occurs 4 times on top of having to recursively call n/2 times which basically ends in a n(n(n))) loop. We basically have a 3-way nested for-loop with the added resource hungry allocation of memory.

The naïve algorithm, while simple in nature, is effective because of how easy it is for a computer to iterate over values. We aren't asking the computer to allocate space for sub arrays with an order of (n/2) magnitudes. The arrays have already been created, and the algorithm's job is simply to make 3 arithmetic operations and a store call into the register. This makes it extremely easy for the computer to do, not to mention quick as well. Perhaps the better increase divide and conquer performance would be to use a hybrid algorithm, similar to what was done in implementation 2.

Strengths and Constraints

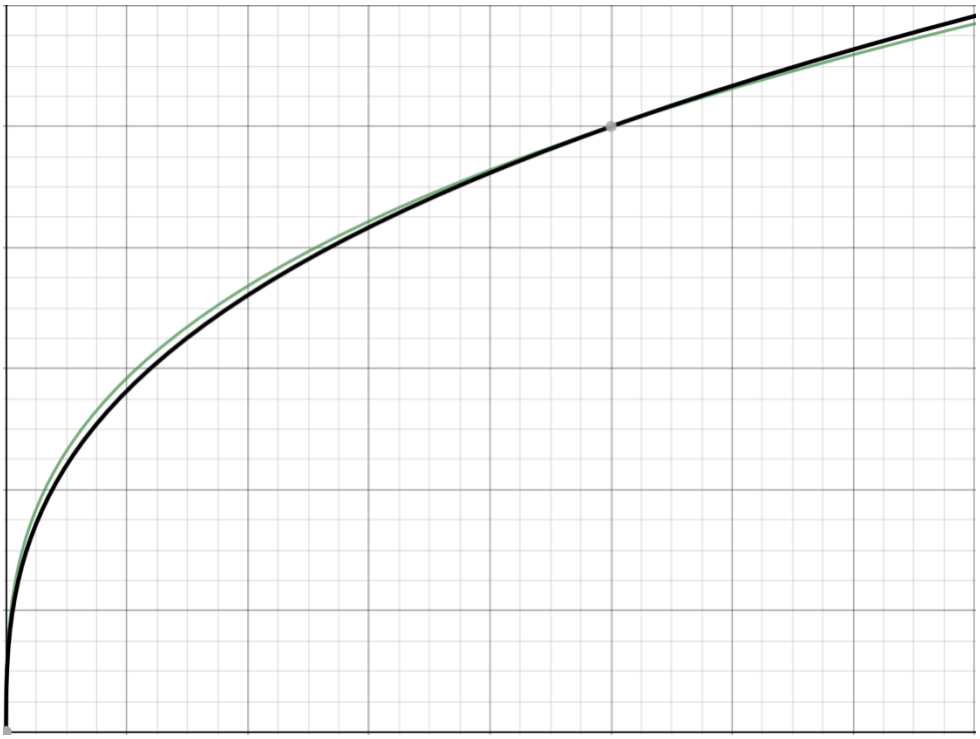

Figure 2 - Naive (Green) vs Strassen's (Black)

Looking at the graph above, its clear that Strassen's algorithm will become better than naïve method at a certain point. However, the constraints require Strassen's algorithm to have a square matrix of size 10,000 or above in order for the effectiveness to shine. Until that point, the naïve algorithm is generally better for dealing with smaller sets of matrices.

The main issue seems that Strassen's algorithm would work most efficiently when paired with a parallel processing system, such that there exists enough resources and computation power, such that the system would not be affected extremely taxing recursive calls.

Since all divide and conquer method utilize separation of tasks into sub tasks, this would prove extremely useful if the computer could handle and manage such resources (in parallel). Most standard computers don't seem to be able to handle this issue well; therefore, it is much more convenient for standard computers to apply iterative solutions to these kinds of problems.

While our standard computer works best with the naïve approach, this does not discount the fact that Strassen's/Divide and Conquer methods will hold their value when given a proper system. For example, if a software engineer was provided a computer that could handle parallel processing, it would be extremely waste to apply the naïve approach, given that the computer can handle multiple method calls asynchronously.