CS 124 Programming Assignment 2
Owen Hakim

Introduction:
This write-up discusses the theoretical analysis, experimental analysis, and programming implementation of Strassen's Algorithm for matrix multiplication. We will first walk through mathematically solving for optimal cross-over points using recurrence and running times, and distinguishing between the cases for even and odd dimensions. Then I will represent the experimental analysis with tables and graphs and include an analysis of those numbers and the differences found between the two analyses. Finally, I will explain the process undertaken to implement the algorithm, the decisions made to try to optimize the code, and discuss some final conclusions for the assignment.

Theoretical Analysis:

I calculated the cross-over point by considering the runtime for traditional matrix multiplication and the recurrence equation for the modified version of Strassen's algorithm. Let $M(n)$ be the runtime for the traditional matrix multiplication algorithm. The first equation says that:
$M(n) = (2n - 1)n^2$. This is because there are n^2 dot products to compute for matrix multiplication, one dot product for every cell in an n x n matrix, and for each of those dot products there are n multiplications and $n - 1$ additions to compute. Putting this together, we get $n^2(n + n - 1) = (2n - 1)n^2$.

Let $S(n)$ be the recurrence equation for Strassen's algorithm. The second equation gives:
$S(n) = 7S(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil)^2$

This recurrence was derived by counting the number of multiplications, additions, and subtractions used in Strassen's algorithm. When computing the seven products that define Strassen's, there are a total of 7 multiplications, each done on matrices of size $\lceil n/2 \rceil$ based on the recursion in the algorithm, and a total of 18 matrix additions and subtractions, also each on matrices of size $\lceil n/2 \rceil$. Since adding or subtracting matrices takes one computation for each cell in the matrix, which is the dimension squared, you must square the last $\lceil n/2 \rceil$ in the equation. The ceiling function is necessary for this recurrence equation because the algorithm is based on recursing over submatrices that are half the dimension of the original, but splitting matrices up in this way only works for matrices of an even dimension. The way Strassen's algorithm can be implemented to account for this (which was performed in the code) is to pad the matrices with one row and one column of zeros if the dimension is an odd number. Doing this would mean that the recursion would be over matrices of dimension $(n + 1)/2$, which is what the ceiling function would give.

With these two equations, the crossover point can be found by combining both in such a way that will switch over to the traditional multiplication runtime when that becomes optimal. This can be represented in a piecewise function. Let $T(n)$ be the recurrence equation for the modified version of Strassen's:
$T(n) = \{ \ T(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil)^2 \text{ if } n > n_0; \ M(\lceil n/2 \rceil) \text{ otherwise } \}$
This means that the modified Strassen's will work as a normal Strassen's algorithm until some point n_0, at which point it switches to the traditional multiplication.

This switch should occur at the most optimal time, which will be when Strassen's begins to run slower than the traditional method – when $S(n) \geq M(n)$. To find this point, therefore, we can set the two equations

equal together and solve for n:
$$(2n-1)n^2 = 7(2\lceil n/2\rceil - 1)(\lceil n/2\rceil^2) + 18(\lceil n/2\rceil^2)$$

Because this equation requires ceilings, at this point it is necessary to distinguish between even values of n and odd values of n. In the even case, $\lceil n/2\rceil = n/2$, so we can simply take out all the ceiling functions and calculate the equation like that. Plugging this into a calculator, we get $n = 15$. Since we are just looking at even values for n in this case, we round up to the nearest even number and get $n = 16$ as the optimal cross-over point for even n.

For the odd case, we can let $n = 2k+1$, since this will be a valid equation for odd integer n and any integer k. Using the padding method of adding one row and column of zeros on any odd-dimension matrix, we see that all instances of $\lceil n/2\rceil$ will be substituted for $\lceil (2k+1)/2\rceil$, which simplifies to $\lceil k + (1/2)\rceil = k+1$. Plugging this back into the equation and solving with a calculator, we get that $k = 18.085$ and therefore $n = 37.17$. We then want to round this value up to the nearest odd number, since we are only dealing with odd numbers in this case, giving $n = 39$ as the optimal cross-over point for odd n.

Experimental Analysis:

To test the cross-over point experimentally in the code, I ran the algorithm for a variety of cross-over points and recorded the time it took to run each. For every time shown in the data below, it is the average of 5 trials running on the same dimension and cross-over. I set the dimension to be 128 because it was a reasonably medium-sized matrix to test, and I noticed that the times were all about the same for any dimension. In order to create the ASCII files for the matrices, I randomly generated $2 * (dimension)^2$ numbers between 1 and 10.
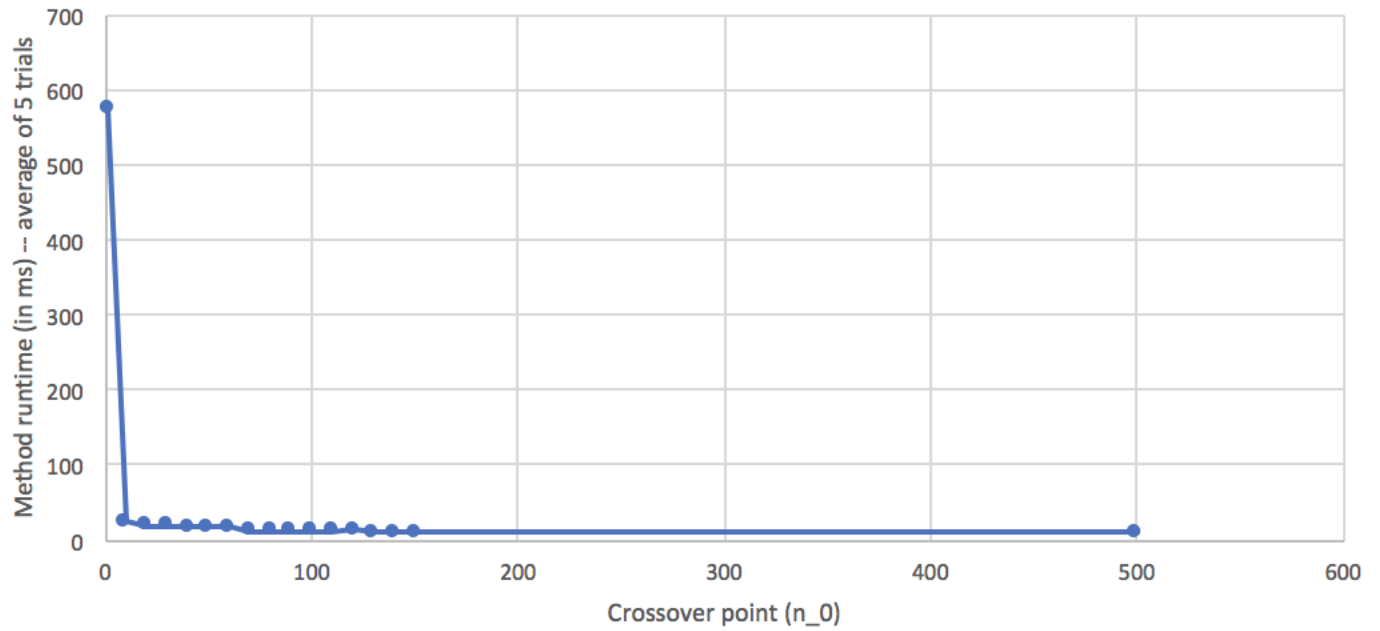
For the cross-over points, I tested every tenth number from 1 to 150 (1, 10, 20, ..., 150). I noticed that at some point the time began to even out and plateau, so I tested the algorithm on dimension 500 to see if it was still the same. The time it took for dimension 500 was the same as it took in the 100-150 range, so at this point I concluded that the time continues to plateau for all dimensions after the cross-over point.

I knew that the time for even dimensions and odd dimensions would differ as a result of the padding and extra code running for that, so I completed the same process to test for cross-over points with the odd dimension 129. I ran the same cross-over points, averaging the time for 5 trials each, and recorded.

The numeric results of the experimental analysis are shown in the following tables and graphs:

FOR FIXED DIMENSION = 128

	Crossover point (n_0)	Method runtime (in ms) -- average of 5 trials
	1	575
	10	25
	20	18.2
	30	18.4
	40	18
	50	17.75
	60	17.833
	70	12
	80	12.6
	90	12
	100	12.5
	110	12.43
	120	13.5
	130	10.5
	140	10.5
	150	10.71
	500	10.87

FOR FIXED DIMENSION = 129

	Crossover point (n_0)	Method runtime (in ms) -- average of 5 trials
	1	3823
	10	31.8
	20	21.4
	30	21.8
	40	16.8
	50	17
	60	17.43
	70	13
	80	13
	90	13.2
	100	13.167
	110	12.8
	120	13
	130	10.86
	140	11
	150	10.4
	500	10.75

For Fixed Dimension = 128

For Fixed Dimension - 129
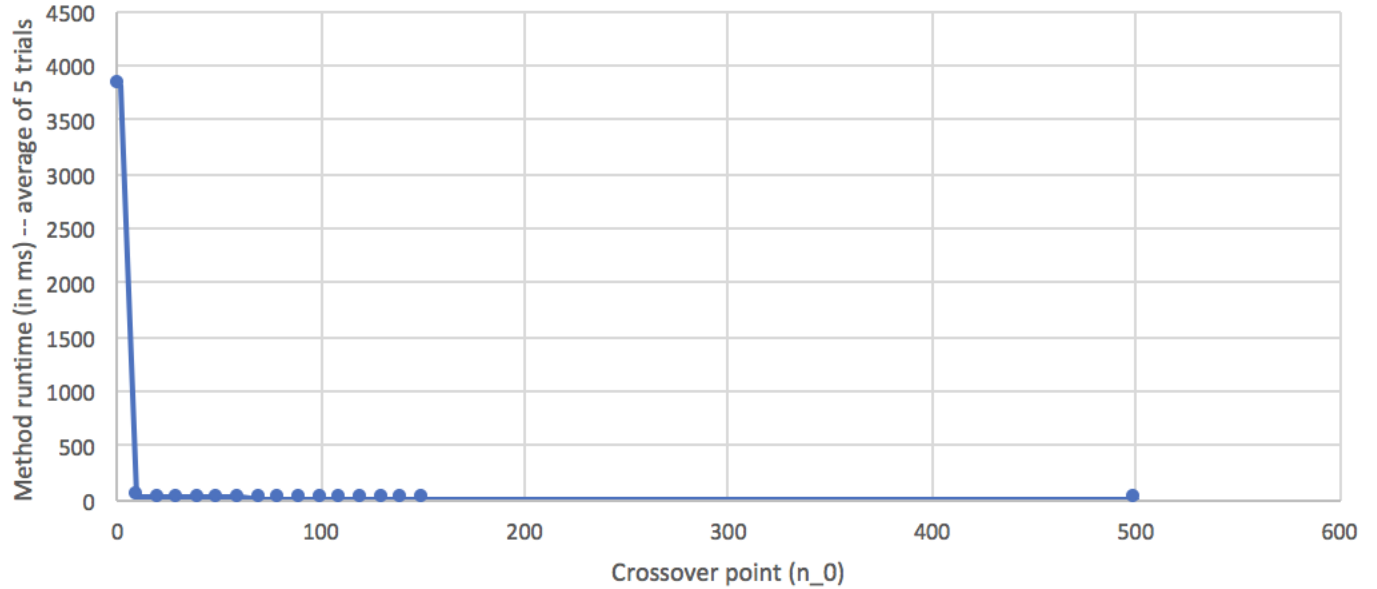
Based on the tables and graphs, I concluded that the code's optimal cross-over point was approximately $n = 130$ for even dimensions and approximately $n = 131$ for odd dimensions. Both times plateaued around 11 milliseconds, the shortest time that was recorded, so the cross-over point was where that trend began.

These results were fairly surprising, given that I had completed and understood the theoretical analy-

sis first, and assumed I would find a similar result around 16 and 39. The odd case ran significantly slower than the even case for smaller cross-over values, which makes sense since this would mean that a layer of padding would need to be added many more times. When the cross-over point was higher, there were fewer and at some point no padding steps necessary. This is when the values align exactly with the even case, which is also reasonable.

Since the lowest time was for cross-over values that were greater than the dimension size, this implis that the fastest case was actually when all the multiplication was done using the traditional multiplication method, not Strassen's. This, of course, is not what should happen in theory, which can be understood from the lecture on Strassen's as well as my specific findings in the theoretical analysis. I understood this as a sign that my implementation of Strassen's algorithm was not fully optimal, and therefore the naive multiplication was still faster. I am not yet totally confident in exactly what part of the code caused the algorithm to run slower than it should, but I considered that it might be a result of making new 2D arrays for every recursion on strassens and for copying in a lot of different information to new arrays. The implementation will be discussed in further detail below.

As for the triangles in random graphs, the results of my code's performance versus the expected number of triangles is displayed in the chart below:

Probability each edge is included	Triangles generated by Strassen's	Expected triangles
0.01	174	178
0.02	1422	1427
0.03	4809	4818
0.04	11425	11420
0.05	22332	22304

So-called "random triangles" can be identified by summing along the diagonal of the cubed adjacency matrix A^3 and dividing that number by six to account for each vertex. This chart compares the number of such triangles in a 1024-vertex graph (where each edge is included with some probability p) generated by my own implementation of Strassen's algorithm to the expected number, calculated by $\binom{1024}{3} * p^3$. For both columns, I rounded the number of triangles to a whole number because decimal numbers of triangles did not appear to be a meaningful statistic in this context. The number of triangles created by Strassen's was consistently quite close to the expected outcome, being within 10 for the first 4 p-values and within 100 for the fifth. The variance in these numbers was quite significant, so I don't think the crossover from less than expected to greater at p = 0.04 is a signifcant data point. One interesting outgrowth of this exercise is that the n'th power of the adjacency matrix, A^n, could be sufficient to calculate the number of n-sided polygons that exist in that matrix's graph, although some accounting would need to be done for repeated edges when getting into higher values of n.

Implementation Details:
(note: "make" will compile the code, but you can only run it with "java strassen <flag> <dimension> <inputfile>" – thank you!)

I used Java to implement the code for this algorithm, taking advantage of its ability to pass arrays between functions, avoiding the need to allocate and deallocate lots of memory, and for the simplicity of dealing with these types in general (all issues that would otherwise have been the case with C or C++). For Strassen's algorithm, I implemented matrices as 2D arrays. In a separate file, I read the ASCII text and write the information into two 2D arrays that are then both put into a size 2 matrix, and passed to the *strassen* file. Dealing with these types was perhaps the most complicated part of the code – the *Matrix* file needed to output a matrix, but the contents of that matrix needed to be a 2D array that *strassen* would then be able to handle. At the top of the *strassen* file, the matrix from *Matrix* is converted into two 2D arrays, which are then immediately run on the *strassens* function.

Implementing the recursion on Strassen's algorithm was relatively easy. Understanding that I would need to pad the matrices with zeros for odd dimensions, I chose to split up the *strassen* function in a few cases: 1) depending on the cross-over point, if the dimension is less than the cross-over, simply run the traditional matrix multiplication function on the matrices and return that; 2) if the dimension is even, then Strassen's can currently recurse on that, so recurse; and 3) if the dimension is odd, pad with one row and column of zeros and recurse over that.

With even dimensions, for the sake of simplicity, I chose to create 8 new 2D arrays, 4 for each matrix A and B, and copied the correct numbers from A and B to the submatrices, then recurse in the definition of the p values for Strassen's 7 products, and, lastly, put that information back into 4 submatrices and then into the final answer matrix, *product*.

In the odd case, I thought that the minimal amount of padding possible to add to make the function work correctly, and therefore the optimal amount of padding, would be to just add one row and one column of zeros any time the dimension is odd. Padding in this way means that every time the dimension is odd, create a new matrix of dimension $n + 1$ and add in the information from the original matrix. The other option I considered was to pad to the next power of two any time the dimension was either odd or any time it was not a power of two. That would have meant, however, that many more zeros would have to be added at once, which could often lead to more padding than necessary. I was also following the analytic conclusion that I had come to, which relied on doing the 1-zero pad method. To get rid of the padding at the very end, when the information gets copied to *product*, the loop will only run until the original dimension size, meaning that the last row and column of zeros will be lost.

This implementation proved to be quite time efficient, taking less than one second to run for most matrix sizes tested, up until about dimension 1024, at which point it took only barely over one second (not including the time to print the result).

Conclusion:

Although this version Strassen's code is perhaps not the most optimal implementation, I concluded that my implementation is correct and fully functional, and fairly close to the running time that it should be. I believe that my choice of padding was optimal and that the treatment of the 7 products is as it should be. The naive multiplication algorithm appears to be more time efficient than Strassen's based on the experimental analysis conducted, and I think that Strassen's algorithm is still running at a reasonable speed. Even for cross-over 1, when Strassen's runs on its own, an even dimension matrix takes less than a second to run and an odd dimension matrix takes about four seconds. Though it is not perfect, I have gained a much deeper understanding of Strassen's algorithm and the ideal cross-over points for matrix

multiplication.