CS 124 Programming Assignment 2: Report

Names: Jared Ni and Aditi Raju

No. of late days used on previous psets: 6 (Jared), 8 (Aditi)

No. of late days used after including this pset: 6 (Jared), 8 (Aditi)

1. Analytical Estimate of Crossover Point

Let $C(n)$ be a cost function for matrix multiplication, where n is the number of rows or columns in the matrix, and the cost of each arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is 1.

The standard matrix multiplication algorithm has the following cost function:

$$C_{std}(n) = n^2(2n-1)$$

Strassen's algorithm has the following cost function:

$$C_{str}(n) = 7C_{str}(n/2) + 18n^2$$

We want to find the point at which Strassen's algorithm should use standard matrix multiplication. In other words, we want to determine the optimal base case for Strassen's algorithm, which is when the cost of matrix multiplication becomes less than that of Strassen's algorithm. In the case where n is even, we substitute $n/2$ into $C_{std}(N)$ and use this in our recurrence relation for Strassen's algorithm:

$$n^2(2n-1) \leq 7(n/2)^2(2(n/2)-1) + 18(n/2)^2$$

$$2n^3 - n^2 \leq (7/4)n^3 + (11/4)n^2$$

$$(1/4)n^3 \leq (15/4)n^2$$

$$n^2(n-15) \leq 0$$

$$n \leq 15$$

In the case where n is odd, dividing by 2 would result in a non-integer value, so we turn n into an even number by adding 1. Thus, we substitute $(n+1)/2$ into $C_{std}(n)$ and use this in our recurrence relation for Strassen's algorithm:

$$n^2(2n-1) \leq 7((n+1)/2)^2(2((n+1)/2)-1) + 18((n+1)/2)^2$$

$$2n^3 - n^2 \leq (7/4)(n^2+2n+1)n + (18/4)(n^2+2n+1)$$

$$2n^3 - n^2 \leq (7/4)n^3 + 8n^2 + (43/4)n + 18/4$$

$$n^3 - 36n^2 - 43n - 18 \leq 0$$

$$n \leq 37.17$$

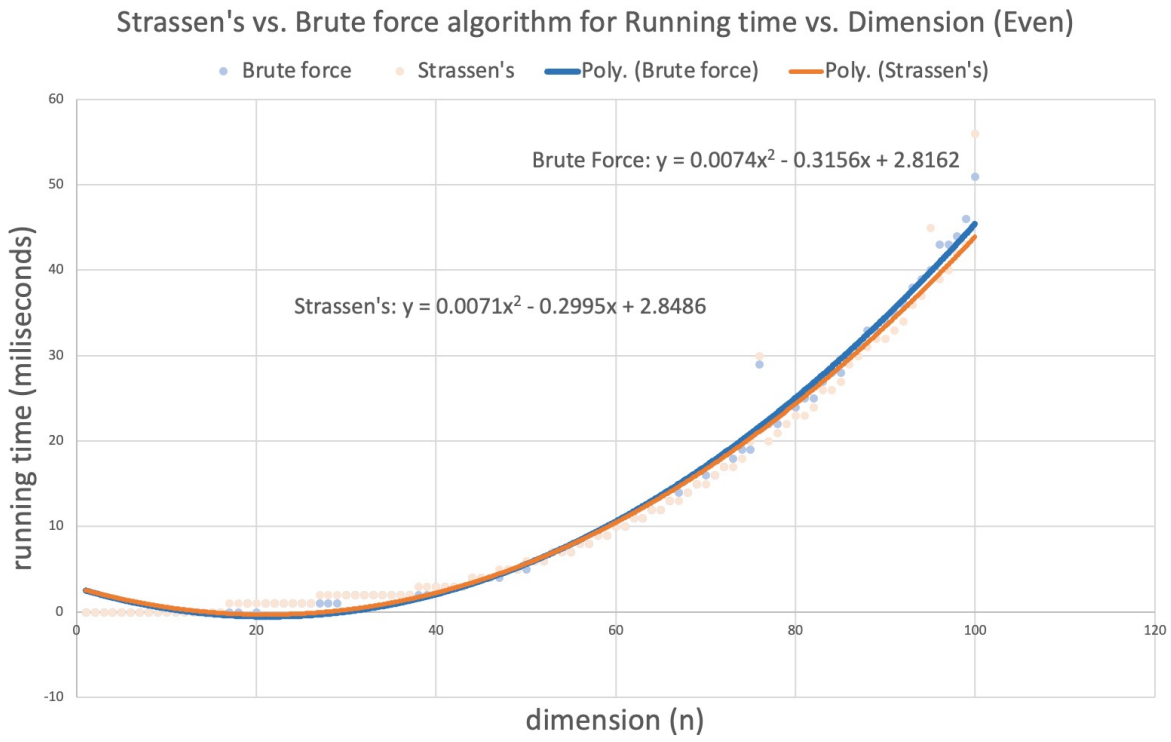
Thus, the crossover point between Strassen's algorithm and the standard matrix multiplication algorithm is **15** when n is even and **37** when n is odd. In other words, Strassen's algorithm should be used when $n > 37$ and standard matrix multiplication should be used when $n \leq 15$.

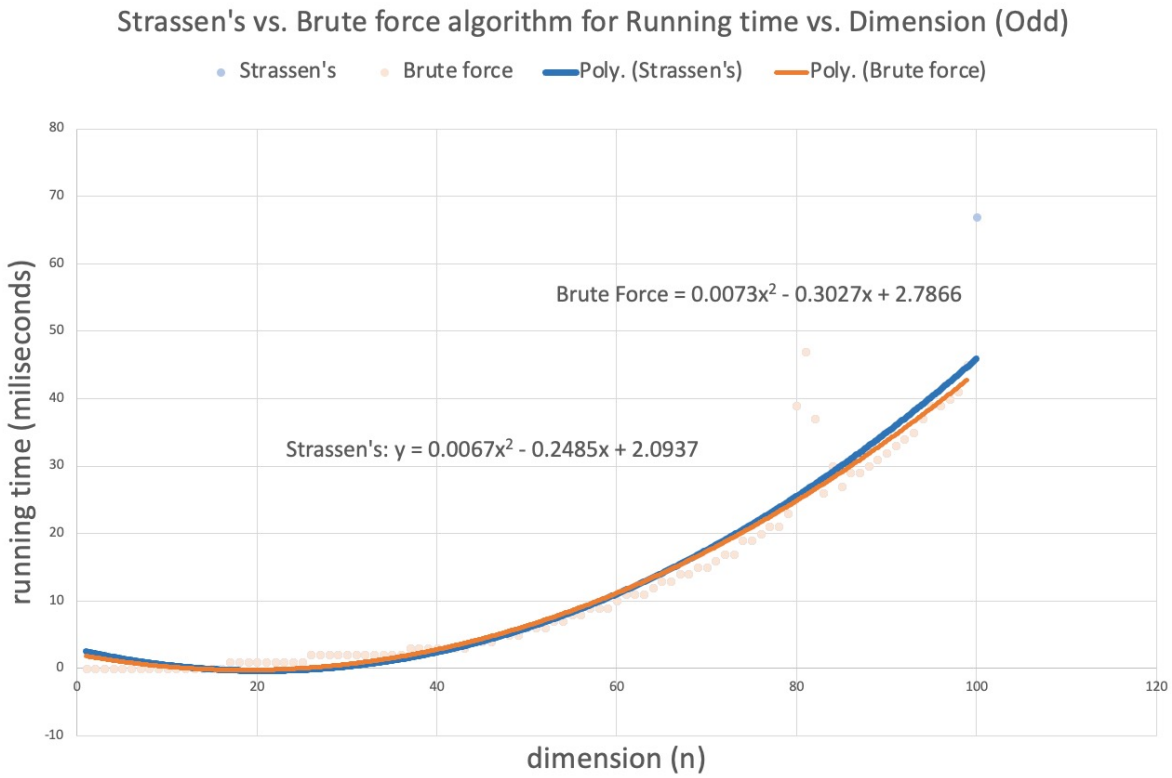
2. Empirical Estimate of Crossover Point

Experimental Setup and Results:

We have written a program in C++ to help us test the ideal crossover point. We have two different experimental setups: in the first experimental set up, we follow the equation for n_0 as it is listed in 1), where $n_0 = n/2$ for even dimensions and $n_0 = (n+1)/2$ for odd dimensions, and we run experiments of dimensions n ranging from $1, ..., n$ separately for even and odd dimensions, incrementing the dimension by 2 for every trial. This way, the crosspoint n_0 we test would range from 1 to 100. We plot both Strassen's Algorithm's runtime to the runtime of the brute force method, and then we compare the best fit line of both algorithms' runtime to see where they cross each other (when Strassen's become lower in runtime than brute force): this would be the ideal crosspoint value we are finding. We ran our experiment using two methods: in the first, we tested different dimensions and calculated n_0 to be $n/2$ for even dimensions and $(n+1)/2$ for odd dimensions. In the second, we tested different values of n_0 on constant dimensions.

Using the first experimental method, we get the following results:





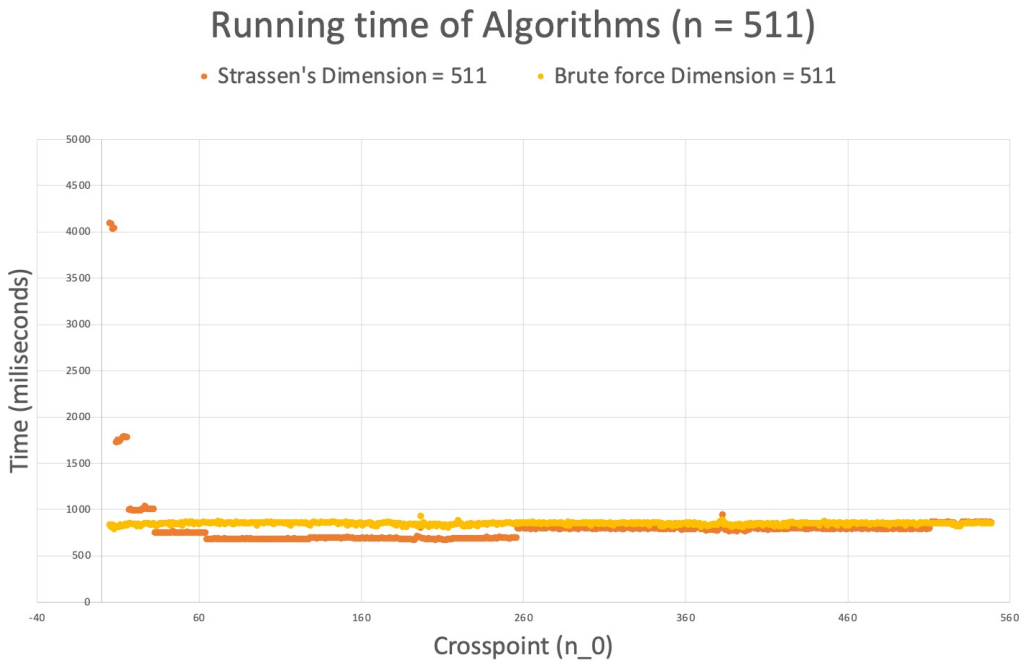
Given the results that we graphed, excel estimated a best fit line for each algorithm's runtime (y variable) vs. crosspoint n_0 (x variable). The cross point we got for odd dimensions is $n_0 = \mathbf{54}$, and the cross point we got for even dimensions is $n_0 = \mathbf{74}$, based on the best fit line approximations. n_0 is the dimension value where Strassen's algorithm becomes faster in runtime than the brute force method, and this was reflected in our Strassen's algorithm in this experiment, where it is the point we switch from divide and combine using Strassen's to the brute force matrix multiplication algorithm. This minimizes the running time of Strassen's algorithm.

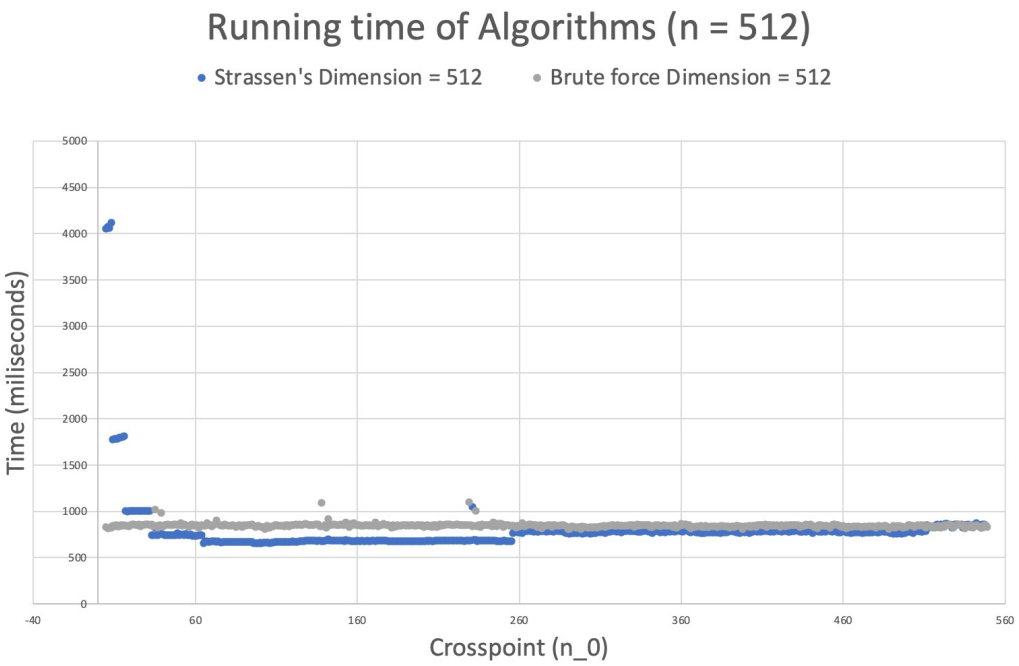
In the second experiment that we ran, we incrementally assigned a higher crossover point in each iteration, where the Strassen's Algorithm function calls the brute force matrix multiplication function instead of further dividing and combining. Each time, we time the execution speed of the Strassen's algorithm in multiplying a matrix of a chosen dimension. We chose 4 dimensions of matrices to run our experiment on: 1024x1024, 1023x1023, 512x512, and 511x511. Theses matrix dimensions are just the right size: they are small enough where we can run 5 trials for each increment of the crossover point in a reasonable amount of time, and large enough where we begin to see performance differences between the brute force algorithm and Strassen's algorithm in multiplying two matrices, because it is much larger than the theoretical crossover point of $n_0 = 16$ for even and $n_0 = 37$ for odd, across which, as we calculated in number 1), is the point where Strassen's should start to be faster than the Brute Force algorithm. For each combination of dimension and cross point, we timed the running time of both the brute force matrix multiplication algorithm and Strassen's algorithm for 5 trials. In our graphs, we averaged between the 5 trials and graphed the running time (y-axis) vs. cross point (x-axis) of the different dimensions; as we plotted the changing running time of Strassen's algorithm, we also plotted the running time of the brute force algorithm as a benchmark, though its running time is not affected by the crossover point. Though these dimensions are close to a power of 2, our Brute Force and Strassen's algorithms both works for multiplying dimensions of any positive integer

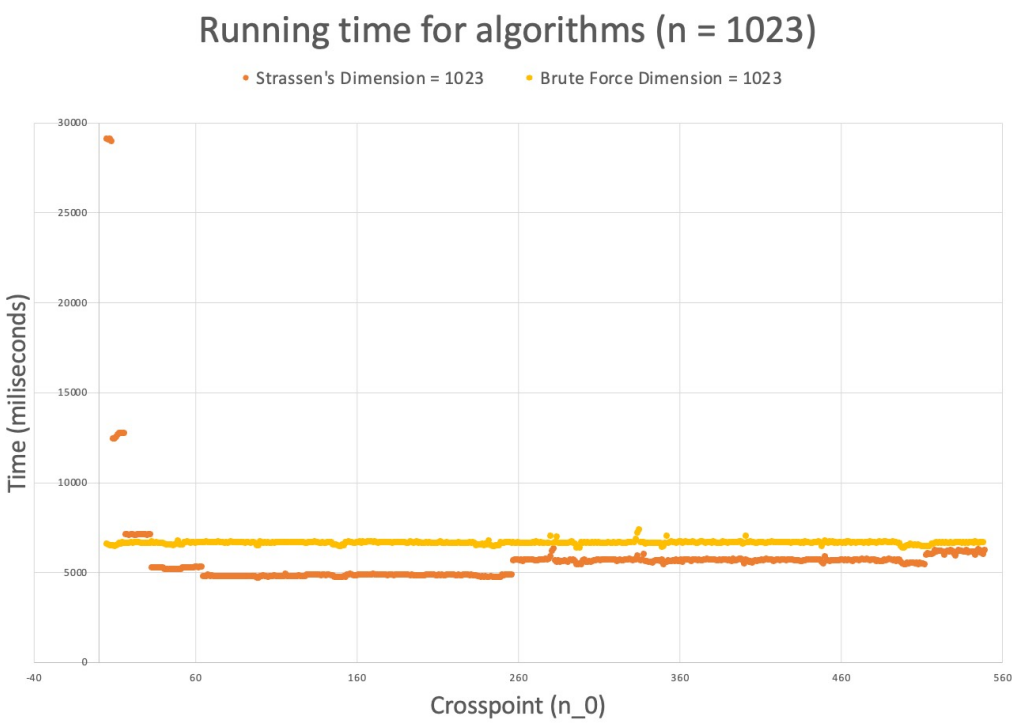
dimensions, as long as the number of columns of the first matrix is the same as the number of rows of the second matrix that we are multiplying. We have both even and odd dimensions because we have calculated two different theoretical crossover point in number 1, so we expect different results for each and odd when running the experiment.

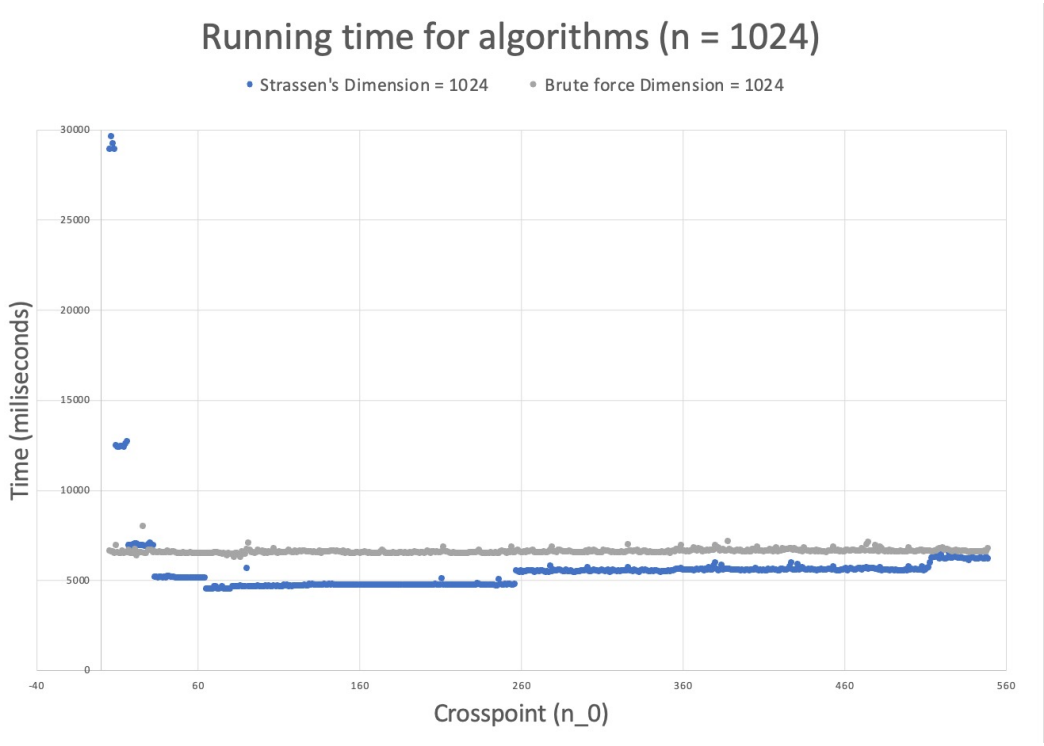
For every dimension of matrix we multiply, we initially set the crossover point to be 5. Then, after every 5 trials of performing brute force and Strassen's and on the current crossover point, we increment the crossover point, until the crossover point reaches 550. We gathered data for this huge range of crossover points in order to see which crossover point would produce the lowest runtime on Strassen's algorithm, after we graph the crossover point (x-axis) to the running time of Strassen's on multiplying two matrices of our selected dimension.

Below are the graphs we constructed from our experimental data of crossover point vs. time (measured in milliseconds), one for each dimension of the two matrices we are multiplying. In our Strassen's Algorithm, we change from Strassen's to brute force once the dimension of the matrix is less than the crossover point (so the cross point we produced in our experiment is 1 + the actual cross point).




Running time of Algorithms (n = 512)
Strassen's Dimension = 512
Brute force Dimension = 512
Time (miliseconds)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
-40
60
160
260
360
460
560
Crosspoint (n_0)


Running time for algorithms (n = 1023)
Strassen's Dimension = 1023
Brute Force Dimension = 1023
Time (miliseconds)
30000
25000
20000
15000
10000
5000
0
-40
60
160
260
360
460
560
Crosspoint (n_0)



We conclude the following results from our second experiment:

Dimension (n)	**Crossover point** (n_0)
511	78
512	64
1023	98
1024	64

Average crosspoint of the even dimensions is **64**.
Average crosspoint of the odd dimensions is (78 + 98) / 2 = **88**.

Thus, looking at results from both experimental methods, we observed experimental crosspoints of **54-64** for even dimension matrices, and crosspoints of **78-88** for odd dimension matrices.

Range of Crossover Points:

Considering the results from both our first and second experiments, the crossover point (n_0) is $54 - 64$ for even dimensions and $74 - 88$ for odd dimensions.

As expected, the crossover point range we got from odd dimensions are higher than the crossover point range we got from even dimensions, since this is consistent to the theoretical findings we concluded in 1). Although the data is consistent for even dimensions, it is unclear whether these crossover points are statistically significant for odd dimensions, given that the nature of our experiments includes large uncertainties that are dependent upon our algorithm and our machines. In the first experiment, the best fit line is approximated and could change due to the conditions we ran our algorithm in. In the second experiment, the Strassen's Algorithm run-times n_0 are often similar to the running time of other iterations of Strassen's Algorithm with cross points within the

same power of 2. As we can see from the graphs, it is often the case where Strassen's Algorithm plateaus in running time as its crossover point becomes greater than or equal to 64 (2^6), and then the running time stays relatively constant throughout the current power of 2, before it goes back up slightly and enters another running time plateau at crosspoint = 256 (2^8), another power of 2. Due to this statistical uncertainty, our crossover points range should be seen as an approximation for what the actual crosspoint in Strassen's could be, given the context of our algorithm and machines.

Theoretical vs. Experimental Crossover Points:

As noted, the optimal crossover points in our experiments are higher than the optimal crossover points we calculated in theory (54-64 and 74-88 vs. 15 and 37). This is because our theoretical calculation of the crossover point does not account for certain expensive operations in Strassen's algorithm.

One expensive task is the extra memory required for Strassen's algorithm: we are calculating 7 preliminary results p_1 through p_7 and storing these values to then be used in the formation of the 4 submatrices C_1 through C_4. C_1 through C_4 also needs to be stored in order to obtain the final result matrix. Thus, Strassen's algorithm requires more memory than naive matrix multiplication.

Another expensive task is the padding of matrices that Strassen's algorithm requires. Strassen's only works on square matrices of even dimension, so before doing any calculations, each matrix and submatrix had to be padded with zeros to meet these conditions. Naive matrix multiplication does not require any padding and works on matrices of any dimensions. So the extra time required to pad matrices with zeros increases the runtime of Strassen's algorithm.

Finally, Strassen's algorithm has an expensive task of merging the submatrices back together to output a result matrix. After calculating C_1 through C_4, the algorith must combine these submatrices to form a result matrix, which adds to the runtime.

Thus, the extra tasks in Strassen's algorithm of requiring more memory, padding with zeros, and merging submatrices makes the experimental crossover points greater than the theoretical points.

Practicality of Crossover Points:

Several observations in our data indicated that the framework of a "crossover point" is appropriate in practice. Our graphs show that naive matrix multiplication tends to have faster runtimes for smaller matrix dimensions, whereas Strassen's algorithm has faster runtimes for larger dimensions. Since naive matrix multiplication is faster than Strassen's algorithm for smaller matrices, it is useful to have a larger base case that uses naive matrix multiplication instead of using Strassen's algorithm all the way down to a base case of matrix dimension 1. Although this experimental method of finding a crossover point does not return an exact value, working with a range of crossover points is still helpful. This just means that if the matrix dimension happens to be within the range of crossover points, it is ambiguous as to which algorithm is faster.

Thus, the framework of a "crossover point" is very useful in practice, because given any matrix, we are able to determine which algorithm (naive matrix multiplication or Strassen's algorithm) is more optimal. For matrices of even dimension, the crossover point is 54-64. For matrices of odd dimension, the crossover point is 74-88. Thus, given a matrix of even dimension n, use naive matrix multiplication if $n \leq 54$, either method if $55 < n < 64$, and use Strassen's algorithm if $n \geq 65$.

Otherwise, if the matrix is of odd dimension m, use naive matrix multiplication if $m \leq 74$, either method if $75 < m < 88$, and use Strassen's algorithm is $m \geq 89$.

Optimizations:

For our brute force algorithm, we optimized it such that it doesn't return a vector of vectors because dynamic memory allocation within the function would be very time costly. Instead, we preconstructed the two matrices we are multiplying and the resulting matrix outside of the function each as a vector of vectors, and then we passed those matrix objects into the function as references. This way, we can simply use the vectors as arrays with pre-allocated memory, and it would be very efficient.

For our Strassen's algorithm, we have the same optimization where we are passing matrices as references instead as object we dynamically construct within the function. This includes all the submatrices we constructed in the divide and combine process of Strassen's, so no memory and runtime is wasted when dynamically allocating memory within a recursive call.
Another major optimization we did in Strassen's was padding the matrix up to an even dimension, instead of up to a power of 2. This is a major optimization because as dimension gets larger, the amount of zeros one needs to pad up to a power of 2 increases drastically, so if we pad up to a power of 2 on a large dimension, it will pad many unncessary zeros into our matrix. Instead, if we pad up to an even number, we only have to pad up to one column and row at each recursive call. Since the number of recursive calls don't grow as fast as the numbers within a single power of 2, this drastically decreases the number of zeros we need to pad into the matrices and resulting matrix we are multiplying, saving both runtime and space.

Types of Matrices:

In terms of different types of matrices, we tested our algorithm on matrices of even and odd dimension. However, there was no significant difference in runtime for even and odd matrices (comparing the runtimes for a n-dimension matrix with a $n + 1$-dimension matrix).

Additionally, we tested the algorithm on matrices composed of numbers of different ranges. We randomly initialized matrices with numbers 0 and 1, as well matrices with numbers 0, 1, and 2. However, there was no discernible runtime difference between these different input ranges. We also tested the algorithm on matrices with negative numbers, so matrices with inputs ranging from -1 to 1, and matrices with inputs ranging from -2 to 2. Once again, there was no discernible difference in the runtime of matrix multiplication for these different inputs.

One interesting observation we did make is that at each crossover point that is a power of 2, the runtime of Strassen's algorithm would either significantly increase or decrease. For example, while testing the crossover points, when the crossover point increased from 31 to 32, the runtime of Strassen's algorithm significantly decreased. On the other hand, when the crossover point increased from 63 to 654 the runtime of Strassen's algorithm significantly increased. This can be explained by the fact that every time a crossover point reaches the next power of 2, Strassen's algorithm has to divide the matrices into half one more time. This extra recursion decreases the runtime of the algorithm if the crossover point we're using is less than the optimal crossover point (since this means standard matrix multiplication is still more efficient and the crossover point should increase so less of Strassen's algorithm is utilized). Similarly, this extra recursion increases the runtime of the algorithm if the crossover point we're testing is greater than the optimal crossover point (since this means standard

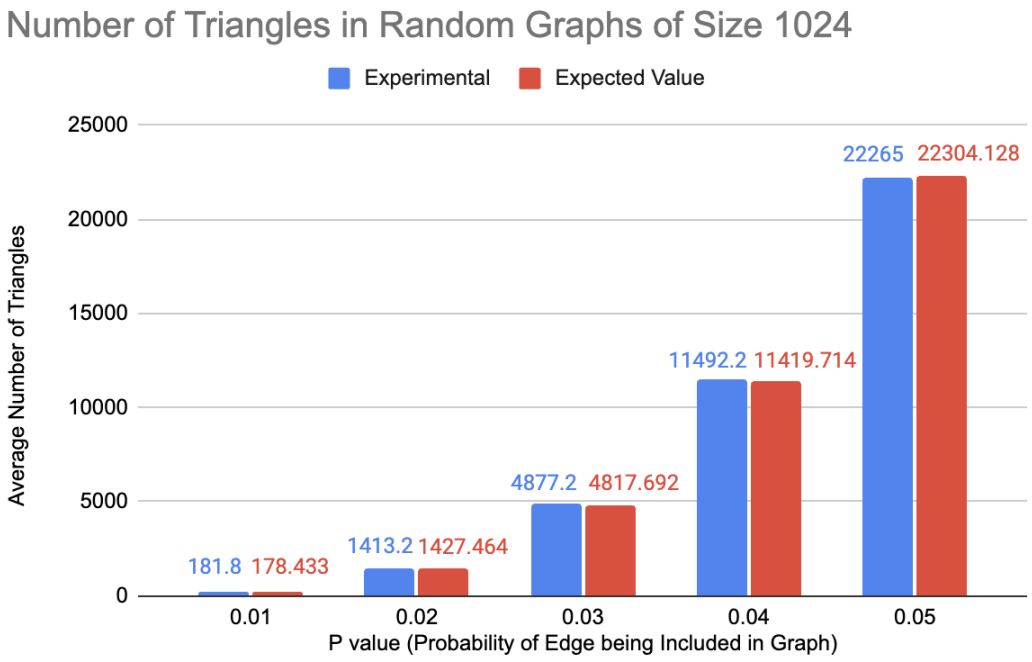
matrix multiplication is less efficient and the crossover point should decrease so more of Strassen's algorithm is utilized).

3. Triangles in Random Graphs

The following table displays the number of triangles found in random graphs with 1024 vertices across 5 trials, for values of p (the probability of an edge being included in the graph) including 0.1, 0.2, 0.3, 0.4, and 0.5. The expected number of triangles in the random graph is calculated using the expression $\binom{1024}{3}p^3$.

	p=0.01	**p=0.02**	**p=0.03**	**p=0.04**	**p=0.05**
Number of Triangles (Trial 1)	169	1462	4877	11348	22053
Number of Triangles (Trial 2)	172	1424	4703	11496	22243
Number of Triangles (Trial 3)	186	1382	4950	11651	22655
Number of Triangles (Trial 4)	189	189	4793	11139	22143
Number of Triangles (Trial 5)	193	1409	5063	11827	22231
Avgerage Number of Triangles	181.8	1413.2	4877.2	11492.2	22265
Expected Number of Triangles	178.433	1427.46	4817.69	11419.71	22304.13

Comparing the average number of triangles in the experimental method to the expected value of triangles, we get the following bar graph:



We observe that the experimental values align closely with the expected value for the number of triangles in the random graphs. However, the experimental values do not exactly equal the expected value because the randomized graphs include edges with a certain probability. So the proportion of edges in the randomized graphs is not exactly p. Beyond this slight variation, the experimental values are very similar to the expected value for the number of triangles in random graphs.