CS 124 Programming Assignment 2: Spring 2021

Your name(s) (up to two): Isha Sangani, Bilal Hussain

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

No. of late days used on previous psets:
4 **No. of late days used after including this pset:** 4

Homework is due Wednesday 2021-03-24 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Overview:

Strassen's divide and conquer matrix multiplication algorithm for n by n matrices is asymptotically faster than the conventional $O(n^3)$ algorithm. This means that for sufficiently large values of n, Strassen's algorithm will run faster than the conventional algorithm. For small values of n, however, the conventional algorithm may be faster. Indeed, the textbook Algorithms in C (1990 edition) suggests that n would have to be in the thousands before offering an improvement to standard multiplication, and "Thus the algorithm is a thoeretical, not practical, contribution." Here we test this armchair analysis.

Here is a key point, though (for any recursive algorithm!). Since Strassen's algorithm is a recursive algorithm, at some point in the recursion, once the matrices are small enough, we may want to switch from recursively calling Strassen's algorithm and just do a conventional matrix multiplication. That is, the proper way to do Strassen's algorithm is to not recurse all the way down to a "base case" of a 1 by 1 matrix, but to switch earlier and use conventional matrix multiplication. That is, there's no reason to do a "base case" of a 1 by 1 matrix; it might be faster to use a larger-sized base case, as conventional matrix multiplication might be faster up to some reasonable size. Let us define the *cross-over point* between the two algorithms to be the value of n for which we want to stop using Strassen's algorithm and switch to conventional matrix multiplication. The goal of this assignment is to implement the conventional algorithm and Strassen's algorithm and to determine their cross-over point, both analytically and experimentally. One important factor our simple analysis will not take into account is memory management, which may significantly affect the speed of your implementation.

Tasks:

1. Assume that the cost of any single arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is 1, and that all other operations are free. Consider the following variant of Strassen's algorithm: to multiply two n by n matrices, start using Strassen's algorithm, but stop the recursion at some size n_0, and use the conventional algorithm below that point. You have to find a suitable value for n_0 – the cross-over point. Analytically determine the value of n_0 that optimizes the running time of this algorithm in this model. (That is, solve the appropriate equations, somehow, numerically.) This gives a crude estimate for the cross-over point between Strassen's algorithm and the standard matrix multiplication algorithm.

2. Implement your variant of Strassen's algorithm and the standard matrix multiplication algorithm to find the cross-over point experimentally. Experimentally optimize for n_0 and compare the experimental results with your estimate from above. Make both implementations as efficient as possible. The actual cross-over point, which you would like to make as small as possible, will depend on how effi-

ciently you implement Strassen's algorithm. Your implementation should work for any size matrices, not just those whose dimensions are a power of 2.

To test your algorithm, you might try matrices where each entry is randomly selected to be 0 or 1; similarly, you might try matrices where each entry is randomly selected to be 0, 1 or 2, or instead 0, 1, or -1. We will test on integer matrices, possibly of this form. (You may assume integer inputs.) You need not try all of these, but do test your algorithm adequately.

3. Triangle in random graphs: Recall that you can represent the adjacency matrix of a graph by a matrix A. Consider an undirected graph. It turns out that A^3 can be used to determine the number of triangles in a graph. To see this, you can see that (ij)th entry in the matrix A^2 counts the paths from i to j of length two, and the (ij)th entry in the matrix A^3 counts the path from i to j of length 3. To count the number of triangles in in graph, we can simply add the entries in the diagonal, and divide by 6. This is because the jth diagonal entry counts the number of paths of length 3 from j to j. Each such path is a triangle, and each triangle is counted 6 times (for each of the vertices in the triangle, it is counted once in each direction).

   Create a random graph on 1024 vertices where each edge is included with probability p for each of the following values of p: $p = 0.01, 0.02, 0.03, 0.04$, and 0.05. Use your (Strassen's) matrix multiplication code to count the number of triangles in each of these graphs, and compare it to the expected number of triangles, which is $\binom{1024}{3}p^3$. Create a chart showing your results compared to the expectation.

Code setup:

So that we may test your code ourselves as necessary, please make sure your code compiles and runs as follows:

```
$ make

$ ./strassen 0 dimension inputfile
```

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. The dimension, which we refer to henceforth as d, is the dimension of the matrix you are multiplying, so that 32 means you are multiplying two 32 by 32 matrices together. The inputfile is an ASCII file with $2d^2$ integer numbers, one per line, representing two matrices A and B; you are to find the product $AB = C$. The first integer number is matrix entry $a_{0,0}$, followed by $a_{0,1}, a_{0,2}, \ldots, a_{0,d-1}$; next comes $a_{1,0}, a_{1,1}$, and so on, for the first d^2 numbers. The next d^2 numbers are similar for matrix B.

Your program should put on standard output (in C: printf, cout, System.out, etc.) a list of the values of the *diagonal entries* $c_{0,0}, c_{1,1}, \ldots, c_{d-1,d-1}$, one per line, including a trailing newline. The output will be checked by a script – add no clutter. (You should not output the whole matrix, although of course all entries should be computed.)

The inputs we present will be small integers, but you should make sure your matrix multiplication can deal with results that are up to 32 bits.

Your program will be compiled, run, and tested by a script, so make sure it actually works as specified: ideally, you should be able to type "make", then "./strassen 0 <dimension> <inputfile>" for some dimension and input file, and have it work.

Do not turn in an executable.

What to hand in:

As before, you may work in pairs, or by yourself. Hand in a project report (on paper) describing your analytical and experimental work (for example, carefully describe optimizations you made in your implementations). Be sure to discuss the results you obtain, and try to give explanations for what you observe. How low was your cross-over point? What difficulties arose? What types of matrices did you multiply, and does this choice matter?

Your grade will be based primarily on the correctness of your program, the crossover point you find, your interpretation of the data, and your discussion of the experiment.

Hints:

It is hard to make the conventional algorithm inefficient; however, you may get better caching performance by looping through the variables in the right order (really, try it!). For Strassen's algorithm:

- Avoid excessive memory allocation and deallocation. This requires some thinking.
- Avoid copying large blocks of data unnecessarily. This requires some thinking.
- Your implementation of Strassen's algorithm should work even when n is odd! This requires some additional work, and thinking. (One option is to pad with 0's; how can this be done most effectively?) However, you may want to first get it to work when n is a power of 2 – this will get you most of the credit – and then refine it to work for more general values of n.

Abstract

Strassen's matrix multiplication algorithm reduces the number of recursive calculations that need to be done from 8 to 7. However, when the sub-matrices reach a size that is small enough, it becomes advantageous to use conventional matrix multiplication (using dot products of rows and columns) instead of using Strassen's all the way to a base case of a 1x1 matrix. In this assignment, we found the crossover point both experimentally and theoretically. For dimension 300, our crossover point appeared to be roughly between 20 and 35. For dimension 600, our crossover point appeared to be between 20 and 40. We also used Strassen's matrix multiplication algorithm to count the number of triangles in a randomly generated graph and compared it with the expected number of triangles.

Theoretical Results

First, we find the theoretical crossover point at which the runtime of Strassen's algorithm is equal to the runtime of the conventional matrix multiplication algorithm. We represent the runtime of Strassens' algorithm, $S(n)$, with the following recurrence:

$S(n) = 7S(\frac{n}{2}) + \frac{9}{2}n^2$

This is because Strassens' algorithm needs to be recursively applied 7 times on submatrices of dimension $n/2$, and there are 18 additions and subtractions of submatrices of size $n/2$. Each addition or substraction of 2 matrices of dimension $n/2$ takes $(\frac{n}{4})^2$ time. The time to multiply 2 matrices of size n using the conventional algorithm can be defined as:

$C(n) = 2n(n-1)$

This is because there are $n^2(n-1)$ additions ($n-1$ additions for each entry of the product matrix, and there are n^2 entries) and n^3 multiplications (each entry in the first matrix is multiplied with n entries in the second matrix, and there are n^2 entries). Thus, in total, the runtime for the conventional algorithm can be represented as $2n^3 - n^2$.

Then, we find the two functions' crossover point.

$C(n) = S(n)$

$2n^3 - n^2 = 7S(\frac{n}{2}) + \frac{9}{2}n^2$

$S(\frac{n}{2}) = 2(\frac{n}{2})^3 - (\frac{n}{2})^2$

$\frac{n^3}{4} - \frac{n^2}{4} = \frac{2n^3}{7} - \frac{11n^2}{14}$

$n = 15$

The theoretical crossover point for even n is 15.

However, this does not work for odd n. For odd n, we modify our recursion to reflect padding. It becomes

$$2n^3 - n^2 = 7S(\tfrac{n+1}{2}) + \tfrac{9}{2}(n+1)^2 + 2n + 1$$

We add 1 to n because our matrix dimension increases by 1. We add the $2n + 1$ term because this is the number of new 0's we have to add. The solution to this recursion is 37.

Implementation of Strassen's Algorithm

We implemented conventional matrix multiplication using three for loops. The outer for loop iterates over the rows in the first matrix; the middle for loop iterates over the columns in the second matrix; and the inner for loop iterates over each entry pair in the corresponding row/column pair.

We found that iterating over the first matrix's rows in the outer for loop took less time than doing things in the opposite order: iterating over the second matrix's columns. We weren't sure why this happened, but we guess it is something to do with the cache and accessing memory.

Our implementation of Strassen's was as follows.

- Read in the file and populate our two matrices.
- Check the dimension of the matrix and see if it is even or odd. If it is odd, we pad the matrix with an extra row and an extra column of 0's.
- Divide the matrix into quarters.
- Calculate the products recursively using Strassen's algorithm until we hit a base case of our given threshold.
- Calculate the quarters of the resulting product matrix.
- Merge the quarters to form the product matrix.
- Unpad the product matrix by removing the last row and last column if we padded it previously.

Initially, we used a Matrix object. Its attributes included a 2D array and an integer to keep track of this dimension. We quickly realized this implementation was inefficient since it introduced unnecessary complications and took up unnecessary space. Instead, we simply used a 2D array and did not store its dimensions separately.

Our intention with the original implementation had been to store the dimension separately so that the length of a matrix would not need to get recalculated in linear time whenever we needed that quantity.

To achieve this objective without using a Matrix object, we simply stored the length of whatever matrices we were working with at the beginning of every function. We referenced that variable instead of calling *len*() every time we needed the length.

One nice perk of Python is that dynamic memory allocation and deallocation is taken care of. Therefore, we never needed to explicitly free matrix variables from the stack. This meant that we did not run into memory leakage errors.

Python has its quirks, however. One issue that we encountered when declaring and initializing matrices was that when we used a certain type of syntax, "$[[0]*d]*d$", where d is the dimension of the matrix, our matrices did not get populated correctly–columns would repeat themselves. We switched to different syntax, "[0 for i in $range(d)$] for i in $range[d]$", which for some reason worked correctly where the other syntax hadn't.

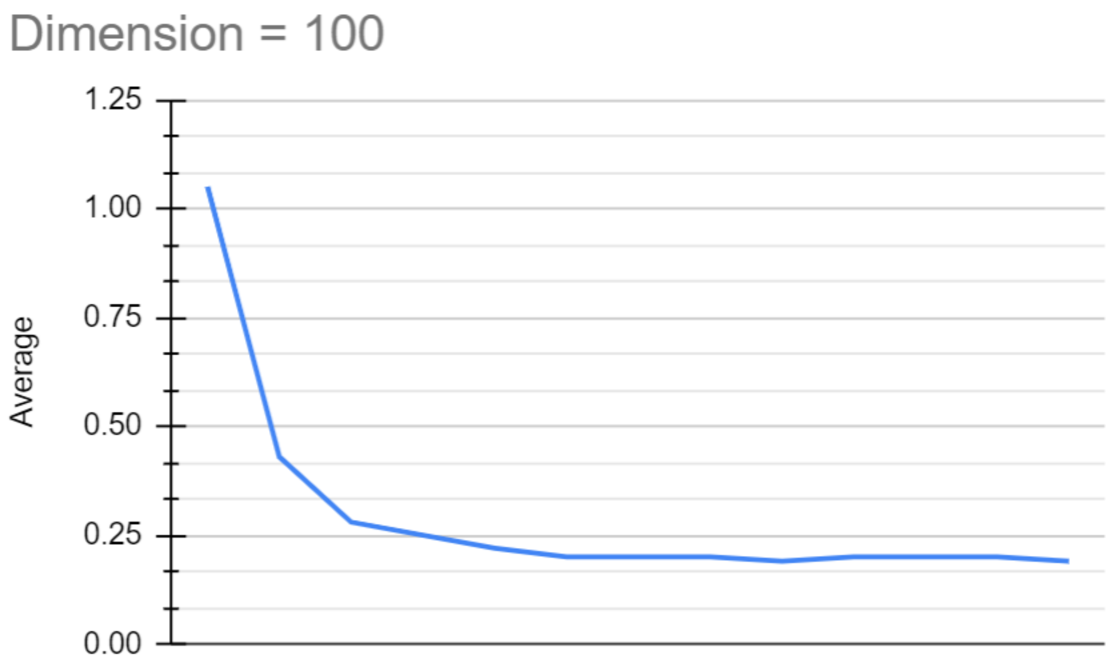
Experimental Results

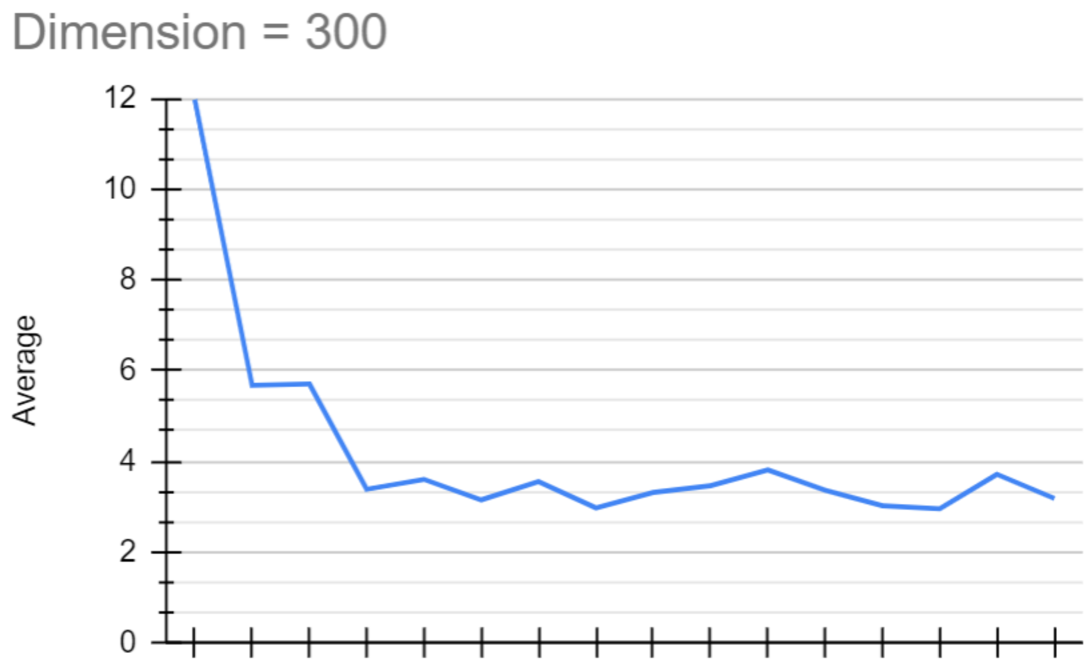
In our test file which to test Strassen's algorithm, we randomly generated $2d^2$ numbers between -1 and 1. The matrices would get populated with these numbers in *strassen.py*.

We tested Strassen's algorithm for dimensions 100, 300, and 600. We chose to multiply large matrices to more clearly tease out differences in runtime.

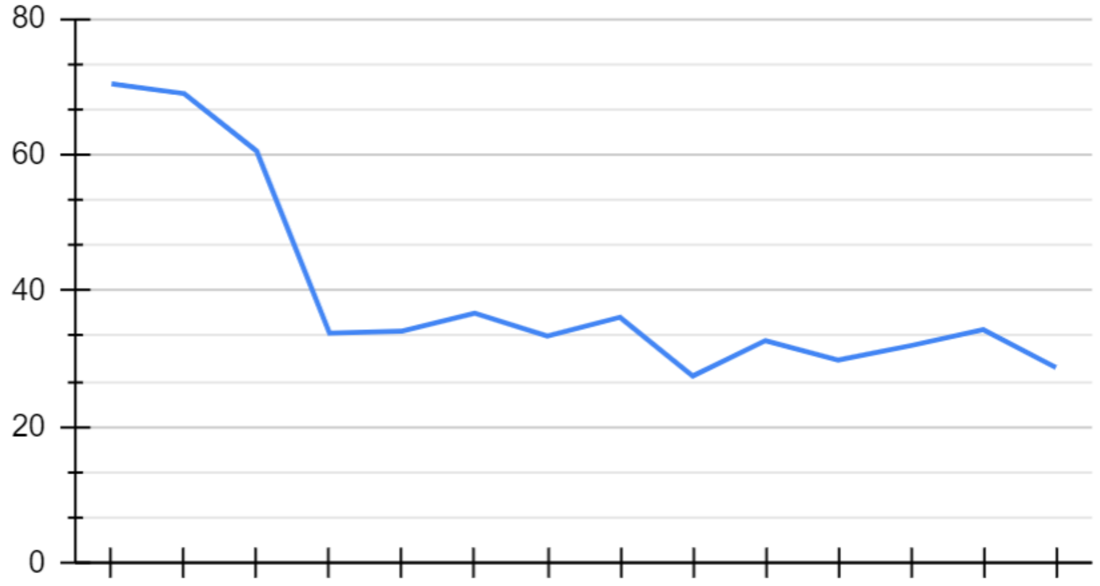
Our data tables and plots are visible below. We tested all over 5 trials. To find the crossover point, we compared the runtime of Strassen's algorithm to the runtime of the conventional matrix multiplication algorithm.

For dimension 100, our crossover point appeared to be between 25 and 40. For dimension 300, our crossover point appeared to be roughly between 20 and 35. For dimension 600, our crossover point appeared to be between 20 and 40. We could not provide more specific data due to high variability.

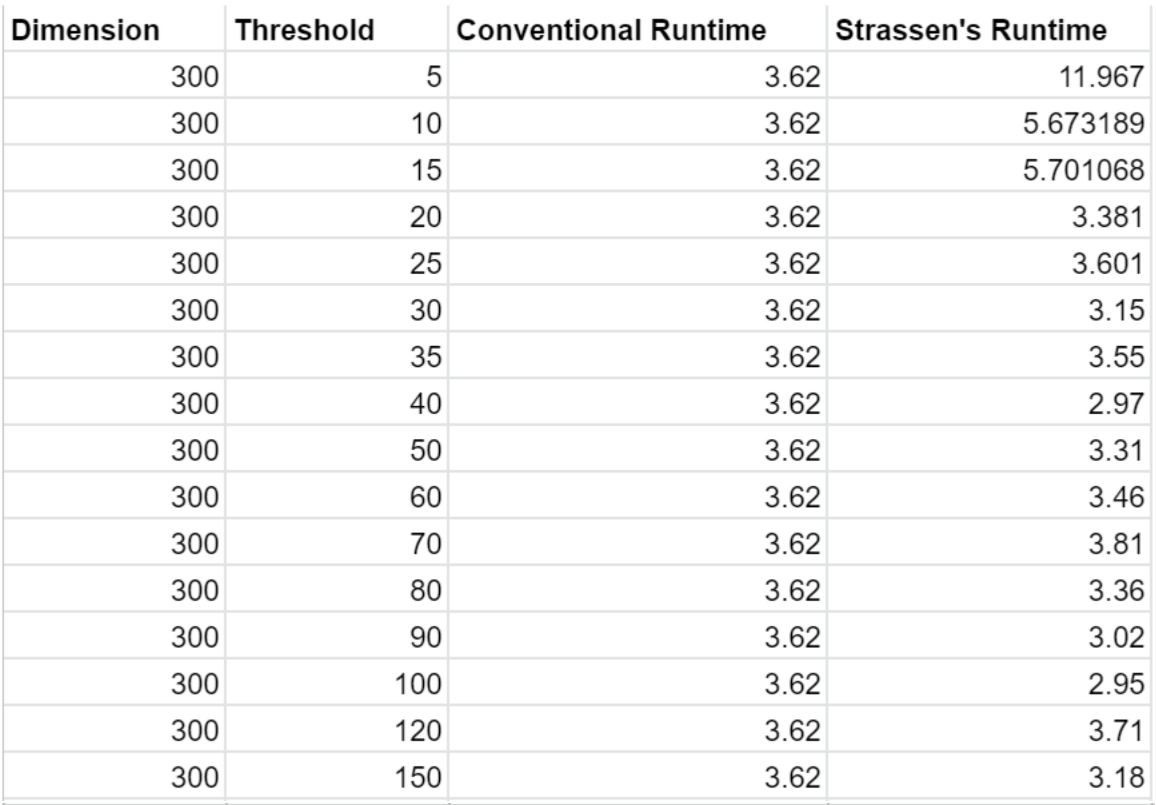

Dimension = 100
1.25
1.00
0.75
0.50
0.25
0.00
Average


Dimension = 300
12
10
8
6
4
2
0
Average

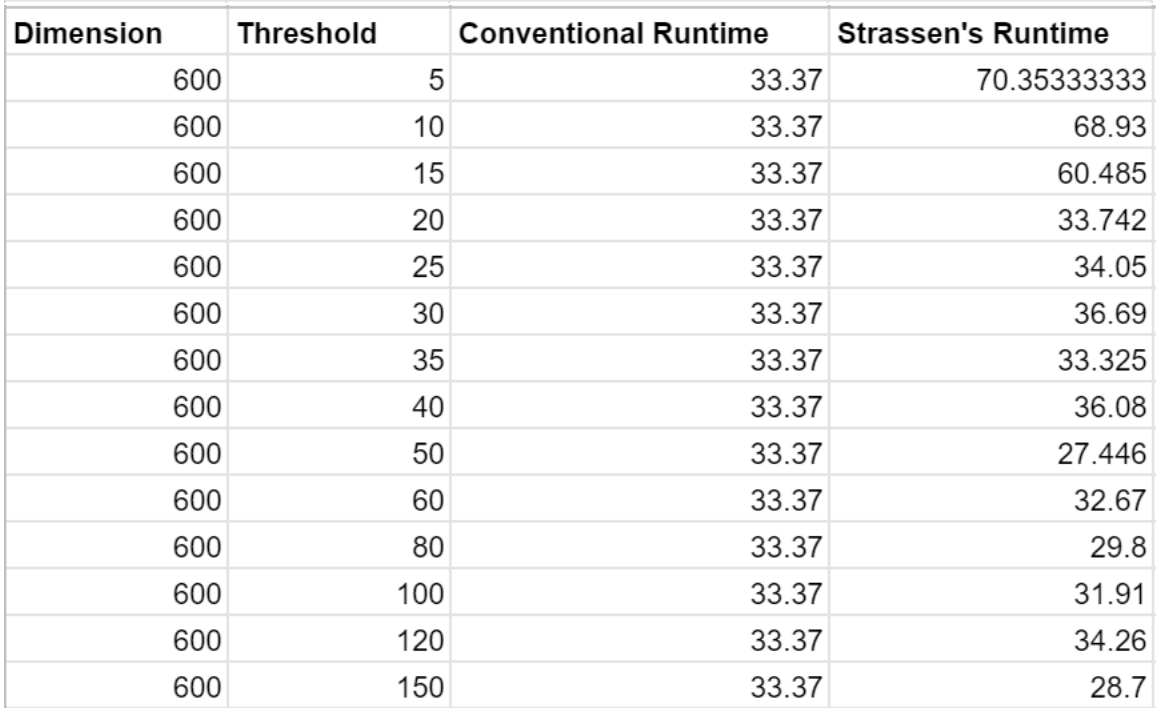
Dimension = 600



Dimension	Threshold	Conventional Runtime	Strassen's Runtime
100	5	0.21	1.05
100	10	0.21	0.43
100	15	0.21	0.28
100	20	0.21	0.25
100	25	0.21	0.22
100	30	0.21	0.2
100	35	0.21	0.2
100	40	0.21	0.2
100	50	0.21	0.19
100	60	0.21	0.2
100	70	0.21	0.2
100	80	0.21	0.2
100	90	0.21	0.19



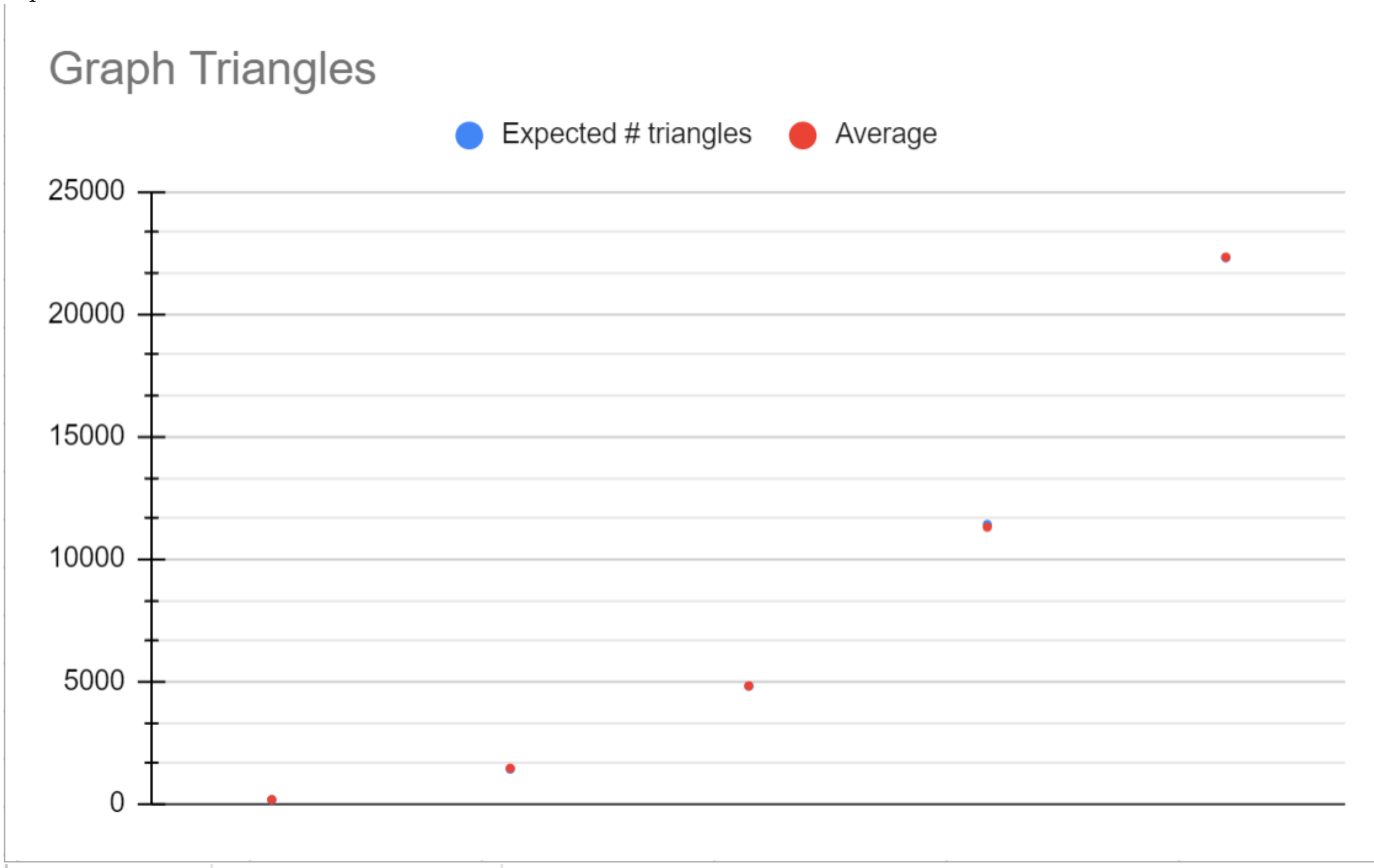
Dimension	Threshold	Conventional Runtime	Strassen's Runtime
300	5	3.62	11.967
300	10	3.62	5.673189
300	15	3.62	5.701068
300	20	3.62	3.381
300	25	3.62	3.601
300	30	3.62	3.15
300	35	3.62	3.55
300	40	3.62	2.97
300	50	3.62	3.31
300	60	3.62	3.46
300	70	3.62	3.81
300	80	3.62	3.36
300	90	3.62	3.02
300	100	3.62	2.95
300	120	3.62	3.71
300	150	3.62	3.18



Dimension	Threshold	Conventional Runtime	Strassen's Runtime
600	5	33.37	70.35333333
600	10	33.37	68.93
600	15	33.37	60.485
600	20	33.37	33.742
600	25	33.37	34.05
600	30	33.37	36.69
600	35	33.37	33.325
600	40	33.37	36.08
600	50	33.37	27.446
600	60	33.37	32.67
600	80	33.37	29.8
600	100	33.37	31.91
600	120	33.37	34.26
600	150	33.37	28.7

Graph Triangles

Overall, our results for the graph triangle problem were very similar to the expected values. We did 7 trials for each p because there was significant variation in the data. Overall, the data conformed to our expectations.



p-value	Expected # triangles	Average # triangles
0.01	178	182.6666667
0.02	1427	1465.5
0.03	4817	4826
0.04	11419	11337
0.05	22304	22333.5

Discussion

Our implementation did not conform to our expectations for crossover points. We will address key questions one by one.

Why was there a difference between theoretical and experimental crossover points?

Our theoretical analysis of the runtime did not include considerations of memory allocation. Memory allocation when creating matrices can be a time-consuming process and can vary significantly, as we will discuss more later. This is why our actual crossover point was slightly higher than the theoretical values.

Why was there a difference in crossover point across dimensions?

It's possible that for certain numbers, division into 4 sub-matrices may have ultimately resulted in more odd numbers throughout the process. That is, a sub-matrix of dimension $n/2$ might itself have a dimension that is an odd number, and then its own submatrix of dimension $n/4$ might be a dimension that is an odd number, and so on such that odd numbers appear more frequently. In these cases, more padding is necessary, which means the runtime would become longer and the crossover point would become higher.

Moreover, it's possible that when dividing the matrices, a submatrix of size $n/2$ could have a dimension significantly below the threshold t, while the "parent" matrix of dimension n could have a dimension significantly higher than t. In this case, the recursion would continue until hitting the $n/2$ case, when it would begin conventional matrix multiplication. This would result in a different runtime than if the division had actually resulted in a submatrix whose dimension was close to t.

Why was there so much noise in the runtime data?

We could not give narrower ranges for the data due to the large variability. Why was the data so variable? After doing some research, we learned that the time to allocate memory can vary significantly. One example of a factor affecting this time include the other tasks being processed by the CPU. Moreover, whether a variable is stored in the cache or in memory can make a different to the runtime. These are just a small number of factors that could potentially explain the variation in the data.

There appeared to be more noise in the data for higher dimensions, which makes sense, since the matrices are larger, there is more information to be stored, and thus more variation with times for memory allocation.