CS 124 Programming Assignment 2

HUIDs: 10977193 and 40983120

March 24, 2017

1 Introduction

In programming assignment 2, we implemented Strassen's algorithm in C and tested for the optimal crossover point to switch to conventional matrix multiplication. We tested on various-sized matrices and experimented with various ways to adapt and optimize our basic implementation of Strassen's algorithm. We implemented two different methods of padding zeros to the edges of matrices.

2 Procedure

We implemented our matrices in C using int pointers, storing all elements of a matrix in a single long block of memory. We chose C because we were most comfortable with C, C was fast, and we wanted control over the memory management. However, we soon realized that choosing C presented certain difficulties for this assignment, because we couldn't dynamically resize arrays when padding zeros, and we couldn't take advantage of data structures like C++'s vectors or Java's ArrayList.
For our conventional matrix multiplication algorithm, we used an optimized matrix multiplication function we wrote in CS61, which involved switching the order of the i, j, k for-loops to optimize our cache access pattern.
To implement Strassen's, we wrote a function that takes in three matrices and a dimension. In each recursive step, the function allocates memory for each of the matrices P_1 through P_7 as well as each of the 14 factors that go into these P's. However, the function doesn't allocate additional space for additions and subtractions, because we provide addition/subtraction functions that modify submatrices.

2.1 One_pad Method

Strassen's algorithm presents difficulties for matrices with odd dimension n because the indices of the submatrices do not match up. Our first strategy to handle such matrices is to pad zeros to the right and at the bottom of the matrix until the dimension reaches the next largest power of 2. This is the most straightforward method because we only have to reallocate the matrix once (hence the name "one_pad method"), and it's guaranteed to work even if our crossover point n_0 is 1. However,

there are clear drawbacks to this method. For any matrix of size $2^k + 1$ for instance, we would have to pad with tons of zeros and perform many useless additions/multiplications of zeros. Consider a 1025×1025 matrix for example; we'd have to pad with $2048^2 - 1025^2 = 3145728$ extra zeros, which is obviously inefficient.

We came up with one minor optimization: allocating a matrix of dimension $2^{\lceil \log_2 n \rceil}$ right off the bat. This avoids a situation where we'd allocate a matrix of dimension 1025, fill it in with values from the text file, and immediately reallocate a matrix of dimension 2048.

We realized that we could reduce the number of zeros allocated if, instead of padding up to the next smallest power of 2, we padded up to the next smallest multiple of $a2^b$ for some a and b, where a is odd. Fine-tuning a and b could reduce the size of the matrices. However, this optimization requires that $n_0 > a$ (or else we would have to perform a recursive step on an odd-dimensional matrix).

2.2 Many_pad Method

An alternate strategy is to pad up only to the next multiple of 2, and do more pads later in future recursive steps. For example, if we start with a 1025×1025 matrix, we pad one row/column of zeros to get to dimension 1026 (2051 zeros padded), then split into 513×513 matrices and pad one row/column of zeroes each again to get up to dimension 514 (1027 additional zeros padded). We can continue like this until we reach the crossover point. This is clearly much more memory-efficient, because we are allocating space for far fewer useless zeros. However, this comes at the price of more allocations, as opposed to the single allocation in the "one_pad" method. In addition, we also need to strip away the extraneous zeros after multiplying, which requires more reallocations and copying. Determining whether this multi-padding strategy is better than our original "one_pad" algorithm depends on how close n is to the next power of 2.

We realized that we could reduce the number of times we re-padded and stripped by, instead of padding up to the next multiple of 2, we padded up to the next multiple of 2^b. We call this number 2^b the "padding parameter" p_0. Appropriate values of p_0 could cut down on the number of allocations. However, this optimization requires that $n_0 > p_0$ (or else we could imagine a nightmare scenario in which every tiny matrix gets padded into a 32×32 matrix, and we'd never reach the base case to start running the regular algorithm).

We implemented both the one_pad and many_pad methods, and tinkered with the parameters to determine whether the optimizations would actually affect our runtimes.

2.3 Testing

To test the correctness of our algorithms, we generated small (ex: 17×17) matrices of random integers and checked our results against a Python script that we wrote. We also tested on larger dimensional matrices (ex: 1024×1024) that contained only ones. The output, as expected, was a matrix of 1024's. Thus, we verified that our implementations of both conventional matrix multiplication and Strassen's algorithm on all matrix sizes were correct.

3 Theoretical Runtime

We assume that all multiplications and additions cost 1 unit, and all allocations and other operations cost 0 units.

Let's consider matrix multiplication on a matrix of order n, where n is a power of 2. We first identify our recurrence equations.

$R(n) = n^2(2n-1) = 2n^3 - n^2$, because for each of the n^2 elements in the matrix, we perform n multiplications and $n-1$ additions.

In Strassen's algorithm, we perform 7 multiplications and 18 addition/subtraction operations on submatrices of dimension $\frac{n}{2}$. This gives $S(n) = 7 \cdot \min(R(n/2), S(n/2)) + \frac{9}{2}n^2$.

Here, we define $S(n)$ as: the minimum number of operations required for matrix multiplication where we use at least one iteration of divide-and-conquer. For instance, the result for $S(16)$ depends on which of $S(8)$ and $R(8)$ is faster. Defining $S(n)$ like this allows us to identify the optimal crossover point to switch from Strassen's to the standard algorithm.

n	$R(n)$	$S(n)$
1	1	1
2	12	25
4	112	156
8	960	1072
16	7936	7872
32	64512	59712
64	520192	436416
128	4177920	3128640

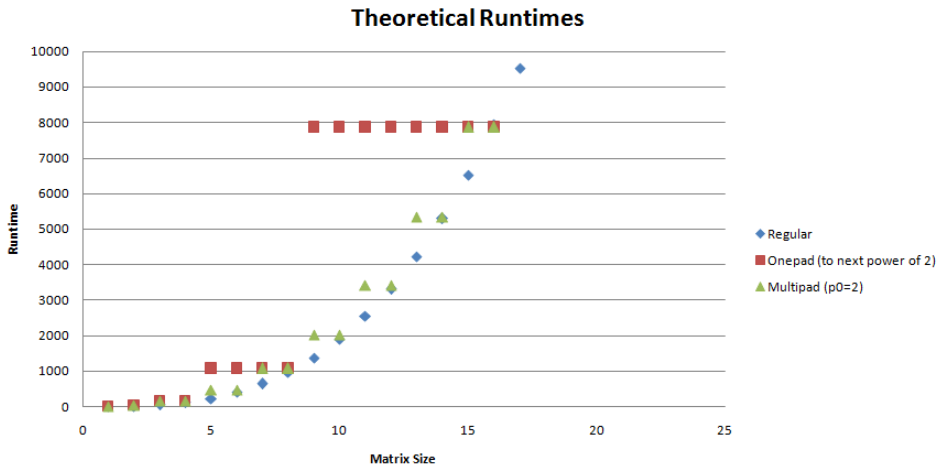
Note that we are only concerned with powers of 2 for the moment, because $S(n)$ is more difficult to calculate when n is not a power of 2. $S(n)$ is dependent on the implementation in this case. For the moment, we assume that the size of the initial input matrix is at least twice as big as the optimal n_0, so that the implementation details do not affect our choice of the optimal n_0. Given all of this, we observe that the theoretical crossover point occurs at $n = 16$. We interpret this as follows: Given an 8×8 matrix, it is optimal should perform conventional matrix multiplication. Given a 16×16 matrix, it is optimal to perform one iteration of divide-and-conquer to get down to 8 by 8 matrices, at which point you should switch to the standard algorithm.

Now we consider runtimes for n that are not powers of 2. These runtimes depend on the implementation.

For instance, if we choose a method that pads a $n = 9$ matrix into a $n = 16$ matrix, then $S(9) = S(16)$. For this method, which pads n into $2^{\lceil \log_2(n) \rceil}$, we have the recurrence equation $S(n) = 7 \cdot \min(R(x), S(x)) + 18x^2$, where $x = 2^{\lceil \log_2(n) \rceil - 1}$.

By contrast, a recurrence equation for the many_pad method with padding parameter p_0 (assuming $p_0 = 2^t$ with $t \geq 1$) is $S(n) = 7 \cdot \min(R(x), S(x)) + 18x^2$ where $x = \frac{p_0}{2} \lceil \frac{n}{p_0} \rceil$.

We expect that this second recurrence equation gives a better runtime for more values of n, as indicated in the graph below. The blue points represent standard matrix multiplication. The red points represent the one_pad method (padding up to the nearest power of 2). The green points represent the many_pad method with padding parameter p_0.



4 Analysis

4.1 Basic Testing

We tested our most basic implementation of Strassen's algorithm, the one_pad method with padding from n to $2^{\lceil \log_2 n \rceil}$. We varied the crossover point and dimension of the initial input matrix. Since all our input matrix dimensions were powers of 2, we expected that different matrix dimensions would exhibit the same crossover point.

Dimension \ Crossover	1	2	4	8	16	32	64
128	0.7411s	0.1612s	0.0666s	0.0412s	0.0360s	0.0346s	0.0302s
256	5.1136s	1.1606s	0.4852s	0.3500s	0.2503s	0.2412s	0.2489s
512	35.3667s	8.2421s	3.4245s	2.2460s	1.9263s	1.7444s	1.7543s
1024	258.5283s	58.4654s	24.0741s	15.4092s	13.1493s	12.6321s	12.9219s
2048	2012.0145s	447.7091s	190.3718s	123.8375s	98.4698s	92.5041s	96.7118s

Below, the first table shows the results when we tested the same set of crossover values and initial input-matrix dimensions, but using the one_pad method with padding parameter 64. The second table shows the results using the many_pad method with padding parameter $p_0 = 2$.
One_pad:

Dimension \ Crossover	1	2	4	8	16	32	64
128	0.6894s	0.1522s	0.0552s	0.0494s	0.0300s	0.0274s	0.0286s
256	4.8945s	1.1354s	0.4763s	0.3549s	0.2658s	0.2393s	0.2310s
512	36.8922s	7.9007s	3.2107s	2.0565s	1.7931s	1.6578s	1.6583s
1024	243.3897s	54.5271s	22.8673s	15.635s	12.7359s	11.601s	12.3255s
2048	4263.202s	360.745s	150.4463s	104.2093s	89.745s	78.6640s	79.0632s

Many_pad:

Dimension \ Crossover	4	8	16	32	64
128	0.0583s	0.0392s	0.0346s	0.0275s	0.0289s
256	0.4250s	0.2692s	0.2324s	0.2194s	0.2113s
512	3.3154s	2.1006s	1.8390s	1.6689s	1.6750s
1024	21.3253s	13.1290s	11.3273s	10.6051s	10.8331s
2048	153.0599s	101.3582s	87.5428s	77.0498s	79.6136s

4.2 Other Tables

Now, we move on to testing our algorithms on matrices of sizes that are not powers of 2. We choose to test three different types of matrices, with dimensions $2^n - 1$, $2^n + 1$, and $2^n + 2^{n-1}$. For matrices of dimension $2^n - 1$, we expect that the three optimized algorithms should produce relatively similar results. This is because each of the algorithms will only need to pad one row of zeroes for these, so we do not include the tables here and instead include the results in an attached txt file. For large values of n, we also see that $2^n - 2^{n-1}$ is even, so we should not have too much difficulty applying Strassen's to these either. We again include these results in the txt file attached.
What we are most interested in is the behavior of multiplication in a matrix of dimension $2^n + 1$, because this will result in the greatest variation in how we pad zeroes. For our basic Strassen's algorithm implementation, which involves padding up to the next largest power of 2, we expect this to be the most efficient. This is confirmed by the following data.

Dimension \ Crossover	1	2	4	8	16	32	64
129	4.3491s	0.9850s	0.4003s	0.2567s	0.2330s	0.2488s	0.2908s
257	32.3632s	7.0634s	2.8725s	1.7900s	1.5272s	1.4583s	1.4374s
513	213.3792s	50.9706s	20.1071s	12.5970s	10.6670s	9.9070s	10.1555s
1025	1506.5414s	343.6648s	141.4910s	90.1217s	74.4083s	69.7943s	71.4577s

Below are results from the one_pad method with padding parameter $p_0 = 64$.

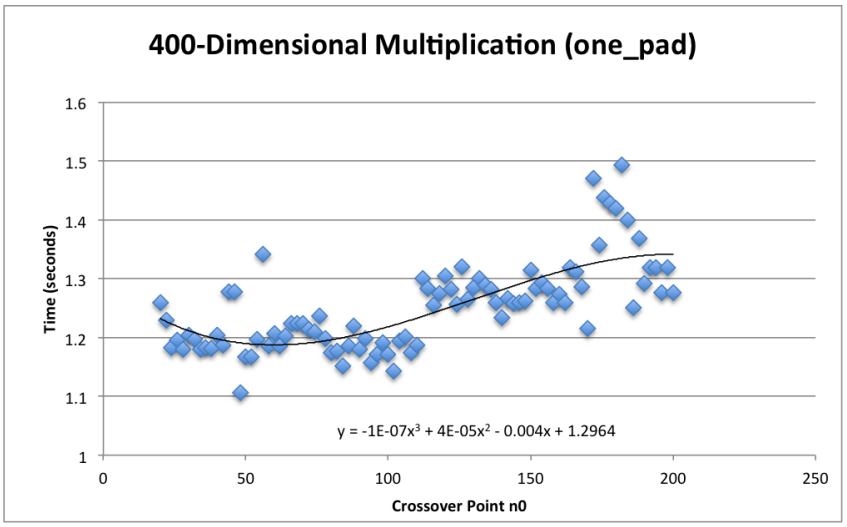
Dimension \ Crossover	1	2	4	8	16	32	64
129	0.7585s	0.1684s	0.0679s	0.0399s	0.0371s	0.0336s	0.0320s
257	4.9050s	1.2547s	0.6796s	0.3729s	0.2394s	0.2220s	0.2508s
513	35.3669s	7.8394s	3.2392s	1.9866s	1.7225s	1.6626s	1.6847s
1025	241.065s	53.013s	21.5859s	13.2972s	12.0169s	11.425s	12.0847s
2049	2053.595s	372.886s	150.3210s	91.728s	79.8672s	73.136s	73.6889s

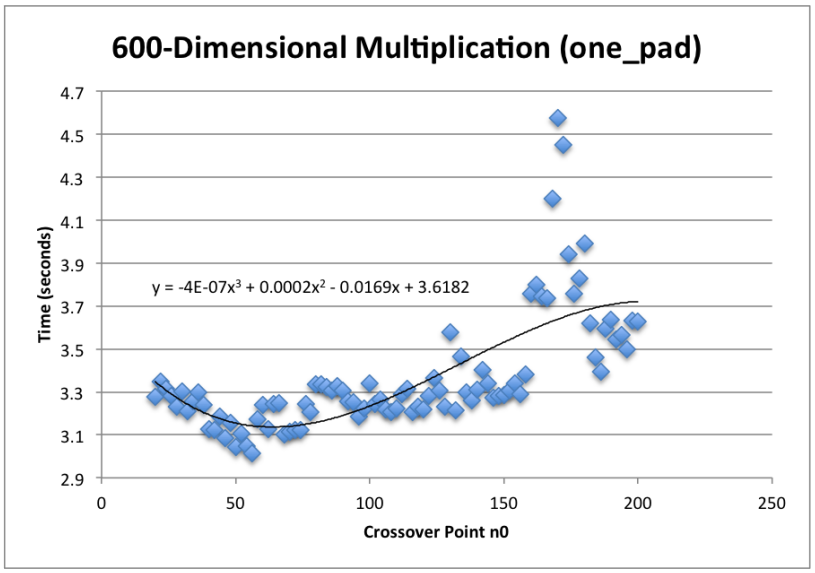
Below are results from the many_pad with padding parameter $p_0 = 4$.

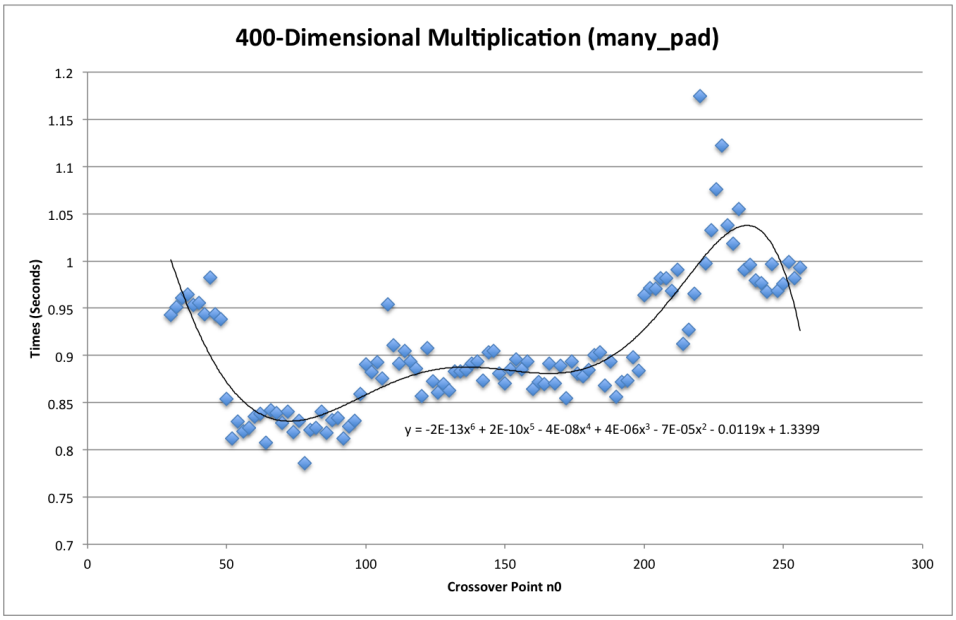
Dimension \ Crossover	4	8	16	32	64	128
129	0.4709s	0.1607s	0.0839s	0.0564s	0.0361s	0.0383s
257	3.1036s	1.0697s	0.5415s	0.3523s	0.2811s	0.2686s
513	21.1236s	7.5849s	3.7806s	2.5074s	2.0056s	1.8934s
1025	1082.495s	59.537s	29.644s	18.908s	14.753s	13.5926s
2049	1112.910s	400.5871s	196.7085s	128.3462s	104.5737s	97.4528s

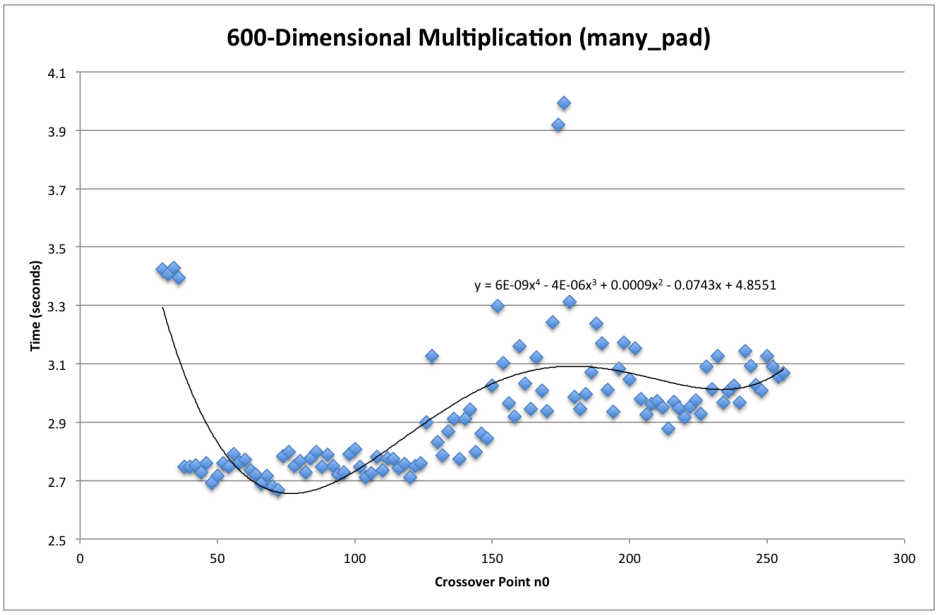
4.3 Experimental Graphs

For one_pad and many_pad, we seek to experimentally determine the best crossover point by graphing the time it takes to multiply medium-sized matrices, and fitting a function to find an approximate optimal n_0 point. Below, we see the graphs generated from multiplying a 400×400 and a 600×600 matrix together, and an approximate polynomial fit that allows us to determine a crossover point that gives us the approximate minimum. We disregard the fit of the trendline outside of the region between 32 and 128, because we know the minimum exists approximately within this region. For the one_pad tests, we run with a padding parameter of $p_0 = 64$ and for the many_pad tests, we run with a padding parameter of $p_0 = 4$.


400-Dimensional Multiplication (one_pad)
Time (seconds)
1.6
1.5
1.4
1.3
1.2
1.1
1
y = -1E-07x³ + 4E-05x² - 0.004x + 1.2964
0
50
100
150
200
250
Crossover Point n0


600-Dimensional Multiplication (one_pad)
Time (seconds)
4.7
4.5
4.3
4.1
3.9
3.7
3.5
3.3
3.1
2.9
y = -4E-07x³ + 0.0002x² - 0.0169x + 3.6182
0
50
100
150
200
250
Crossover Point n0





5 Discussion

5.1 Basic Table Conclusions

Looking at the first three tables, although the timings vary widely with very small crossover points across our different algorithm optimizations, we can see that at the crossover point of $n_0 = 32$, the

times for the three algorithms are pretty similar. This makes sense because when we are dealing with matrices whose dimensions are a power of 2, we don't need to worry about how our algorithms pad zeros. The only thing that could cause a slight difference in the timings is our memory management. In the basic Strassen implementation, we realloc a very large matrix at the beginning whereas in many_pad, we realloc many smaller matrices. However, since the times remain relatively close, it is hard for us to tell which algorithm is better in these situations. We can see that for all sizes of matrices from crossover points 4 and upwards, the times are nearly identical, which is what we hypothesized to occur. We also see from the basic tables that best experimental crossover point should be around the 2^5 to 2^6 range.

5.2 $2^n + 1$ Optimization Discussion

Looking at the tables for matrices with dimensions of the form 2^n+1, we see very clearly how our zero-padding strategy affects our matrix multiplication times. In the first table, where we pad up to the next largest power of 2, we don't even test a matrix of dimension 2049 because it is unbearably slow. For both the one_pad and many_pad methods, we are able to complete multiplication on matrices of dimension 2049 with relative ease. We can see that with a crossover point of 32, we are about 6-7 times as fast running one_pad or many_pad as we are the basic Strassen's (next largest power of 2). This clearly indicates that despite a larger number of allocations in one_pad and many_pad, the greatly reduced number of wasteful multiplications of zeros makes the two optimizations much more efficient than the first.

5.3 Theoretical vs. Experimental Crossover

Theoretically, as stated above, we found a crossover point of $n_0 = 16$, meaning we run Strassen's until we hit a 16×16 matrix and run conventional matrix multiplication for everything smaller than that. Experimentally, we look at our graphs for one_pad and many_pad to estimate the exact crossover point by fitting a trendline and finding the n_0 that minimizes runtime.

For both one_pad and many_pad, we tested our matrix multiplication with dimensions 400 and 600 incrementing our crossover point n_0 in intervals of 2 from 30 to 256 and looking for where the minimum time appears to occur. After using Excel to produce the graphs, we fit a relatively decent trendline for the domain that we care about, between 32 and 150, which we approximately knew n_0 should be in based on our tables, and then used WolframAlpha to compute the minimums of the regression equations on these domains. For one_pad, for the 400-dimensional matrix, we find a $n_0 \approx 50$ and for the 600-dimensional matrix, we find a $n_0 \approx 66$, based on where the minimum time occurs. For many_pad, for the 400-dimensional matrix, we find a $n_0 \approx 60$ and for the 600-dimensional matrix, we find a $n_0 \approx 67$. We can see in general that the crossover point appears to be hovering around an average of $n_0 = 58$ for one_pad, which may be slightly low due to the difficulty of fitting a good trendline for the domain we wanted for these graphs. For many_pad, we have an approximate average of $n_0 = 64$ based on the graphs. Since these are experimental, there may be some discrepancies in the best crossover point between different matrix sizes due to

system inconsistencies when running tests or the fact that we struggled to fit really good trendlines to accurately find the minimum times where crossover should occur on Excel.

6 Error Analysis

This has the illusion of a 4-fold error, but it's actually not as bad as it looks. How bad is our Strassen implementation? In ipython we fit a function of the form $f(x) = \frac{1}{b}(x^j + cx^2)$ to our Strassen's empirical runtime results (with $n_0 = 1$). According to pure theory, j should be $\log_2 7 \approx 2.708$ and c should be $\frac{9}{2}$ as demonstrated earlier. A simple least-squares regression gives optimal parameters as $b = 3400000, c = 2.25, j = 2.97$ It makes sense that j is close to our predicted value but slightly larger, because we are doing additional work (allocs and other memory management) per recursive step, which is not accounted for in our theoretical analysis.

In my sensitivity analysis, I realized that the constant on the x^2 term doesn't really have any significance in the long run. We can attribute our low value of c to mainly noise. Using the values $b = 1000000$, $j = \log_2 7$ and $c = 4.5$ gives a strong fit as well, with error within range of a few seconds. We predicted long-run growth of $f(x) = x^{\log_2 7} \approx x^{2.807}$ and our experiments gave us an answer of about $x^{2.95}$, which is relatively close, meaning our results are not actually that large of a deviation from what we would expect theoretically.

There are some possible sources of error that would result in our implementation of Strassen's algorithm being slower:

In our implementation, we require a lot of memory allocations as we break down our matrices. More allocations (calling 21 additional allocations per layer of recursion) will increase our runtime. In addition, we are also passing many arguments back and forth to the add/sub functions. Since there are 12 functions, it takes these functions more time to parse through these and complete its task, also adding to the run time. In addition, indexing into various parts of arrays to add/sub may be less cache-efficient than the highly efficient regular multiplication. Conventional multiplication directly follows the cache line as much as possible, whereas the add/sub functions require relatively complex indexing schemata and result in us potentially having to access many different cache lines at once, which is more inefficient. Each of these is unaccounted for in the theoretical analysis because we assume the memory allocation is a zero-time operation. Lastly, the theoretical analysis may have been somewhat flawed as well because it doesn't account for the size of the numbers we are multiplying. We assume that every addition and multiplication is worth the same amount of time, and in reality, it is likely that multiplication of numbers is a bit slower.