Mathematical Analysis of Value of n_0

If we assume that the cost of any single arithmetic operation like adding, subtracting, multiplying or dividing two real numbers is 1 and that all other operations are free such as accessing the element of an array or allocating $O(n^2)$ memory for an array, then we can form relatively simple run-time expressions for the conventional ("naive") calculation of matrix multiplication and for the modified Strassen's algorithm outlined in the problem set-up.

For the conventional ("naive") calculation, with two square n x n matrices, the product matrix will have n^2 elements, each of which needs to be computed. To compute an arbitrary element located at row i and column j in the product matrix, we take the dot product of the n elements in row i of the first factor matrix and the n elements in the column j of the second factor matrix. To compute this dot product, we compute n multiplications and then perform $(n-1)$ additions as adding n numbers requires $(n-1)$ additions. Performing $(n+n-1)$ operations for each of n^2 entries in the product matrix yields the runtime expression of $(n+n-1)\cdot n^2$, which for some square matrix of dimension n x n, is

$$T(n)_{conventional} = 2n^3 - n^2 \tag{1}$$

For Strassen's algorithm from lecture notes (not modified yet), we generate the following recurrence for the run-time,

$$T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 \tag{2}$$

In class we covered the 7 recursive calls to Strassen on $\frac{n}{2}$ by $\frac{n}{2}$ matrices. There are 18 additional operations, each on the order of $(\frac{n}{2})^2$ because we perform 18 additions and subtractions on $\frac{n}{2}$ by $\frac{n}{2}$ matrices.

For the modified Strassen's algorithm, we seek the input size n_0 at which point the modified Strassen's performs better than the conventional algorithm. Given this "crossover" in algorithms, we now have an expression for $T(\frac{n}{2})$ in Strassen's algorithm, because we crossover to the conventional algorithm. We can use the run-time expression calculated in the first paragraph. The recurrence above becomes the following

$$T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$$

$$T(n)_{modified\_strassen} = 7\left(2(\frac{n}{2})^2 - (\frac{n}{2})^2\right) + 18(\frac{n}{2})^2 \tag{3}$$

Availing ourselves of computational intelligence, we plug equations (1) and (3) into a calculator to solve for the value of n at which point $T(n)$ in equation 3 is lower than $T(n)$ in equation 1, which yields the correct crossover point. We find n to be 15; however, given that we are dealing with even matrices, we round up to **16** to guarantee a correct optimal cross-over point when the first equation overtakes the third in magnitude.

The computation is slightly different for matrices with **odd** dimensions. In these cases, given padding optimizations we discuss later in this document, we increment the odd dimensions by 1 such that an n x n matrix becomes a $(n+1)$ x $(n+1)$ matrix. Then, each time we divide the matrix by 2, we are guaranteed an

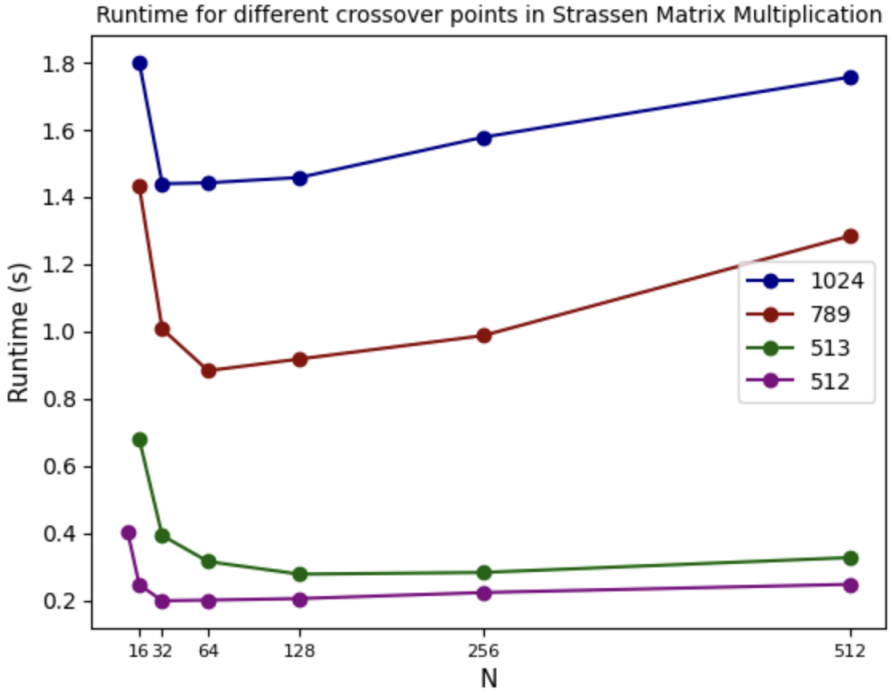
integer dimension. So while the conventional algorithm equation does not change, the Strassen's recurrence indeed does to:

$$T(n)_{modified\_strassen} = 7\left(2(\frac{n+1}{2})^2 - (\frac{n+1}{2})^2\right) + 18(\frac{n+1}{2})^2 \tag{4}$$

Solving for the theoretical crossover point as we did for even numbers, but now using equation (4) instead of equation (3) yields a crossover point of 37.17, which for correctness purposes, we can round down to 37 or round up to **38**, where we round up once again in preference of an even number.

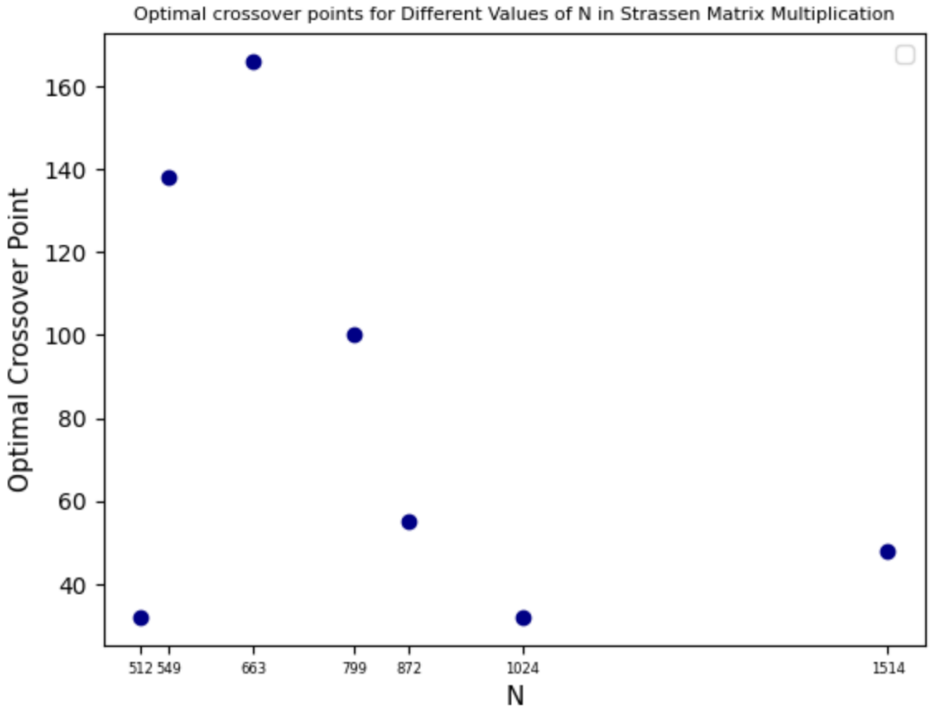
Experimental Analysis of Value of n_0

We came up with multiple methods for generating a range of experimental cross-over points, which we discuss below.



The above image captures one way of depicting results, with a particular framework for computing these results. Our framework was to multiply two n x n matrices and generate the run-time for each potential cross-over point. We then seek the minimum run-time and this gives us the optimal cross-over point. For matrices of size 512 and 1024, we observe that the optimal crossover point is 32. For odd matrices, the result is not so clear-cut, which actually comports with our theoretical analysis. For matrices of size 513, we found a minimum around 128 (if you are concerned about our specificity for n dimensions that are not powers of 2, we handle this in our subsequent method!). For matrices of size 789, we found the minimum at 64. Observe that both of these crossover points are greater than 32, which was the crossover point for the two even numbered dimension matrices pictured on the graph. Given this framework, our experimental value was either 32 (for even dimensions) and between 64 and 140 for odd dimensions. However, we were concerned this method was not meticulous enough for generating experimental values, which is why escalated the rigor of discovering our results.

To escalate the rigor of discovering our results, we generated 7 random matrices of randomly selected dimensions between 128 and 2048: 512, 549, 663, 799, 872, 1024, 1514. For **each** of these vertices, we found the optimal crossover point. For example, for a matrix of dimension 663×663, the potential cross-over points are [1, 2, 3, 6, 11, 21, 42, 83, 166, 332, 664]. We found the optimal cross-over point to be 166. We then created a scatter plot of the optimal cross over points vs. the dimension value n of the factor matrices. The plot is provided below:



At first glance, the plot may seem bizarre. But, these results actually make sense given our implementation. Firstly, for 663, the optimal crossover point was 166 and for 799, it was 100. That's not to say that larger odd numbers have lower crossover points. This is because the crossover points are calculated by repeatedly dividing by 2. For 799, these points are [799, 400, 200, 100, 50, 25, etc] and for 663, these points are [663, 332, 166, 83, 42, 21, etc]. A crossover point of 166 translates to a crossover point of 100 in the 799 matrix since $200 > 166$ but $100 < 166$. One way to think about this is that the crossover points are similar. Additionally, a look at this scatter plot reveals that the odd dimensions tend to have higher cross over points than the even dimensions. Matrices of size 549, 663, and 799 had crossover points 138, 166, and 100 respectively whereas matrices of size 872, 1024, and 1514 had crossover points 55, 32, and 48 respectively.

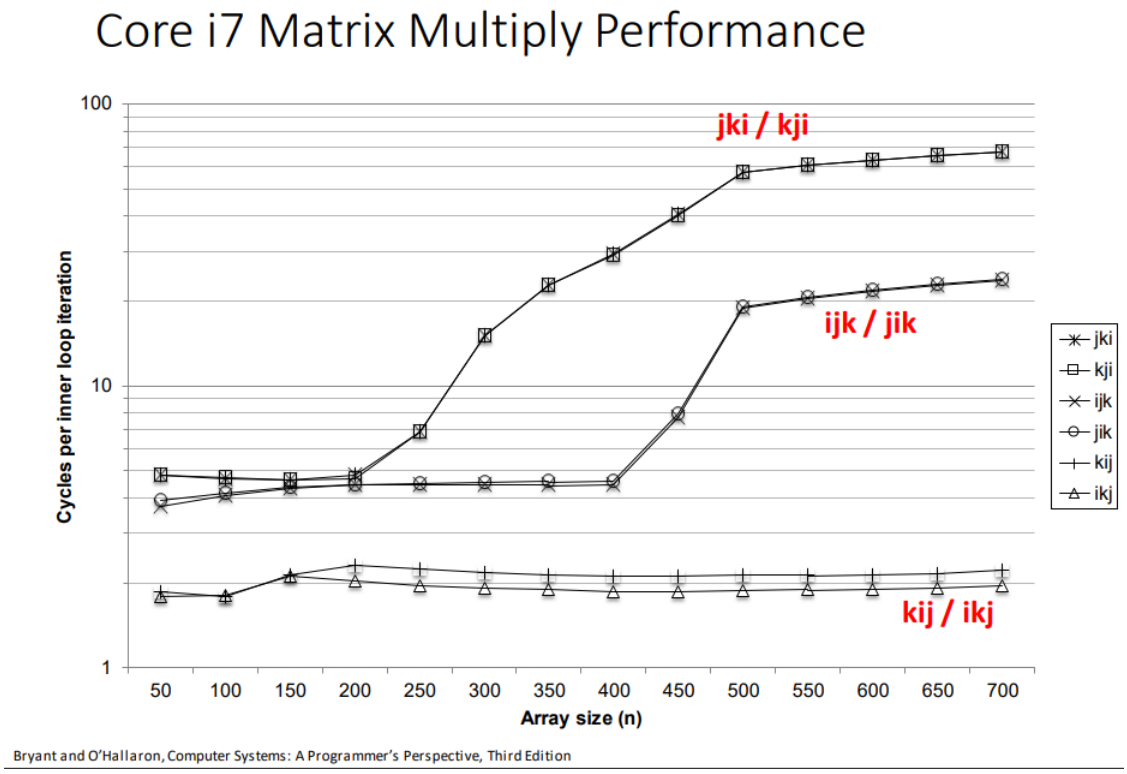
What we conclude is a reasonable range for crossover points for even numbered dimensions is 32 to 64, whereas for odd numbered dimensions, a reasonable range for crossover points is 90 to 170. From our analysis, we conclude that the framework of a single "crossover point" from Strassen's to the standard algorithm isn't appropriate, given that the optimal crossover point varies depending on the dimensions of the input matrices. A more appropriate approach may be to have a function that computes crossover points given a matrix dimension as input, and then using this function to calculate a crossover point given two initial $n \times n$ matrices to multiply, and passing the computed crossover point between recursive calls to Strassen's.

Affect of Matrix Values on Performance
We tested our final implementation of Strassen's algorithm on matrices containing values in $\{0, 1\}, \{0, 1, 2\}$ and $\{-1, 0, 1\}$. Across each of these test cases, we did not find a significant difference in the average runtime of our implementation. This is likely due to the fact that the values within all of these sets are of length 1, and thus require the same amount of computation time to add, subtract, and multiply. If we were to multiply matrices containing values in $\{0, 1, 124124124, 999999999\}$, we would likely notice a decrease in performance.

Discrepancy Between Theoretical and Experimental Crossover Points
While we found our theoretical crossover point to be 16 for even matrices and 38 for odd matrices, our experimentally-obtained crossover points were consistently larger than our theoretical values. This difference can likely be attributed to simplifying assumptions made in our theoretical analysis, and to inefficiencies in our implementation of Strassen's algorithm. While conducting our theoretical analysis, we assumed that all arithmetic operations had cost 1, and all other operations were free. In reality, arithmetic operations have computation runtime that grows with the size of their operands, meaning that only in a best case scenario would they be completed in time $O(1)$. Additionally, our implementation of Strassen's algorithm includes frequent memory allocations and accesses, all of which take time to compute. Because we assumed these operations were free in our theoretical estimate, our experimental results were greater. Lastly, it is unlikely that our implementation of Strassen's algorithm is as efficient as is possible to create in C++, whereas our implementation of the standard algorithm is likely close to as efficient as possible due to its relative simplicity. As a result of this likely disparity in efficiency, our crossover point is likely skewed upwards.

Discussion of Optimizations in Standard Algorithm Implementation

To optimize our implementation of the standard matrix multiplication algorithm, we experimented with looping through our variables in different orders, as was suggested by the hint given in the assignment specification. After doing outside research, we found that when multiplying matrices A and B using the standard algorithm, looping in the order ikj is fastest, where i indexes into rows, j indexes into columns, and k is used to iterate through the ith row of A and jth column of B. Included below is a figure from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition, that shows the efficiencies of different loop orders, and finds ikj to be the most efficient. This is because arrays in C++ (the language that we used) are stored in row-major order, so because of spatial locality in the cache, ikj looping is more optimized than ijk looping.



To verify this, we tested our Strassen algorithm with a version of the standard algorithm iterating in ijk order, and a version iterating in ikj order. Testing on 1024×1024 matrix with $n_0 = 8$, the former of the two implementations terminated in 3.137 seconds, while the latter completed in 2.929 seconds, confirming that ikj was the faster of the two methods.

Discussion of Optimizations in Strassen's Algorithm Implementation

In optimizing the runtime of our implementation of Strassen's algorithm, we primarily focused on whether to decrease the total number of memory allocations, and on how best to pad odd numbered matrices.

Motivated by the hint "Avoid excessive memory allocation and deallocation" provided in the assignment specification, we sought to decrease the total number of memory allocations necessary to compute the intermediate products necessary for Strassen's algorithm. While our algorithm initially allocated 8 $(n/2)\times(n/2)$ matrices a through e for the four quadrants of each of the $n \times n$ matrices being multiplied, as well as 10 additional $(n/2) \times (n/2)$ matrices for the 10 additions and subtractions required to compute P_1 through P_7, we were able to decrease this total by allocating 3 $(n/2) \times (n/2)$ matrices to store temporary values as we computed P_1 through P_7. While this approach led to allocating 15 fewer $(n/2) \times (n/2)$ matrices, it required recomputing matrices a through e as their values were overwritten in our temporary matrices, resulting in an increase in the total number of order $O(n^2)$ loops required for our Strassen's implementation. After implementing both versions of the algorithm and testing them on 1024×1024 matrices, we found the version with more allocations and less redundancy to be less efficient than the implementation with fewer allocations and greater redundancy, performing multiplications on average in time XX seconds and XX seconds, respectively. As a result, we decided to include the latter of the two implementations for our final version.

Another optimization we explored was how best to pad odd numbered matrices. Given that Strassen's algorithm works by dividing its factor matrices into quadrants, we needed to modify our implementation to ensure the factor matrices were always divisible into quadrants, even with odd-numbered dimensions. Given two $n \times n$ matrices to multiply, our first attempt at doing so was to pad our matrix with additional rows and columns of zeroes, such that the padded matrix had dimensions $n' \times n'$, where n' is the nearest power of 2 that is greater than or equal to n. This implementation ensures that all recursive calls to Strassen's will also multiply matrices with even dimensions, since recursive calls multiply matrices of size $n/2$, and n is a power of two. While this implementation of padding has low cost when n is already close to a power of 2, it can in the worst case increase matrix dimensions by $(n-1)$, such as with a matrix with dimensions like 513×513, which would have dimensions 1024×1024 after padding. To combat this, we opted for an alternative padding strategy that adds one row and column of padding prior to calling Strassen's if a matrix has odd dimensions, and then also adds one row and column of padding to the intermediate matrices in Strassen's if $(n/2)$ is an odd number. This method of padding an $n \times n$ matrix creates at most $\log n$ additional rows and columns across all recursive calls to Strassen's, since only $\log n$ levels of recursion can occur before a 1×1 matrix is reached, and at most one row and column of padding is added per level. Testing both implementations on 513×513 matrices with crossover point $n = 32$, our first method for padding completed in 1.445039 seconds while the latter method performed in 0.334982 seconds. With a crossover point of $n = 64$, the former method finished in 1.445039 seconds while the latter finished in .260707 seconds. As a result, we opted to include the latter of the two in our final implementation.

Counting Triangles Results & Discussion

Experimental vs. Exact values for the Number of Triangles for Each Value of p

Probability p	Average Experimental Value	Theoretical Value	Percent Difference $\frac{\text{Experimental}-\text{Theoretical}}{\text{Theoretical}}$
0.01	175	178.433024	-1.9618%
0.02	1401	1427.464192	-1.8890%
0.03	4879	4817.691648	1.2566%
0.04	11561	11419.71354	1.2221%
0.05	22437	22304.128	0.5922%

(Experimental values are averaged over 10 trials)

Included above is a table with our results from computing the number of triangles in five random graphs with 1024 vertices that have edges between vertices i and j with probabilities $0.01, 0.02, 0.03, 0.04$ and 0.05. Compared to the expected number of triangles given for each graph by $\binom{1024}{3}p^3$, our experimental values were within 2% of the exact values for each value of p. Given that in some cases our experimental values were greater than the exact values, and in other cases they were less than the exact values, the error present is likely random error that can be attributed to the generation of the random graphs.