CS 124 Programming Assignment 2: Spring 2024

Your name(s) (up to two): Joshua Zhang, Pedro Garcia

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: 8
No. of late days used after including this pset: 8

Homework is due Wednesday 2024-03-27 at 11:59pm ET. You are allowed up to **twelve** late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two**.

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 theoretical, 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.

   Solution: Strassen's algorithm for matrix multiplication has a faster asymptotic runtime than the conventional matrix multiplication algorithm, namely $O(n^{\log_2 7})$ compared to $O(n^3)$. However, it does have higher startup costs, meaning that for small values of n, we should still default to the conventional algorithm. Now, we will try to analytically derive the threshold at which Strassen's algorithm becomes more efficient than the conventional algorithm.

First, let's try to get a more precise expression for how many arithmetic (addition, subtraction, multiplication) operations the conventional algorithm has. Given two n by n matrices, we are essentially taking the dot product between the first matrix's rows and the second matrix's columns. Each dot product includes n many multiplication operations, and then $n-1$ many addition operations to combine the n products. Thus, each dot product takes $n+(n-1)=2n-1$ operations

Now let's think about how many dot products we take. For each of the n rows of the first matrix, we take the dot product between this and the n other columns of the second matrix. Thus, there are n^2 dot products. Therefore, conventional matrix multiplication takes $(2n-1)n^2 = 2n^3 - n^2$ arithmetic operations, hence why we express this in big-Oh notation as $O(n^3)$.

Let's try to find $T_{\text{Strassen}}(n)$. Here, we will actually have to do a little bit of case work, distinguishing whether n is a power of 2 or not. This is because when n is not a power of, we have to pad our matrix blocks, thus changing our computational and runtime complexity. These cases are exhaustive of all possible cases, because a matrix either has even or odd dimensions.

<u>Case 1: n is even.</u> Suppose we have two matrices of size n by n where n is an even number. Strassen's algorithm will break each matrix up into 4 blocks without padding. Let A, B, C, D and E, F, G, H be the upper-left, upper-right, lower-left, lower-right blocks respectively for matrix 1 and matrix 2 respectively. Each of these blocks is $\frac{n}{2}$ by $\frac{n}{2}$. We then calculate 7 more variables in terms of these blocks, and we should find the number of arithmetic operations required in terms of n.

$$P_1 = A(F-H), P_2 = (A+B)H, P_3 = (C+D)E, P_4 = D(G-E)$$

The calculation of these 4 variables involves adding or subtracting two matrices of size $\frac{n}{2}$ by $\frac{n}{2}$. After the adding and subtracting, we multiply two matrices of size $\frac{n}{2}$ by $\frac{n}{2}$, which is an identical subproblem, but of half the size, denoted $T_{\text{Strassen}}(\frac{n}{2})$. Therefore, we can write P_1, P_2, P_3, P_4 all have the runtime and computational complexity of

$$T_{\text{Strassen}}(\frac{n}{2}) + \frac{n^2}{4}$$

Now, let's look at the other three variables.

$$P_5 = (A+D)(E+H), P_6 = (B-D)(G+H), P_7 = (C-A)(E+F)$$

Following the same logic as our previous analysis, we can express the runtime and computational and runtime complexities for $P_5, P_6, P7$ as $T_{\text{Strassen}}(\frac{n}{2}) + \frac{n^2}{2}$. Each of these also have a subproblem of multiplying two $\frac{n}{2}$ by $\frac{n}{2}$ matrices. However, we have two addition or subtraction operations between matrices of this same size, hence the term $2(\frac{n}{2})^2 = \frac{n^2}{2}$.

We also have to look at the number of operations for the following calculations in our algorithm. $AE + BG = -P_2 + P_4 + P_5 + P_6$ has $\frac{3n^2}{4}$ operations. $AF + BH = P_1 + P_2$ has $\frac{n^2}{4}$ operations. $CE + DG = P_3 + P_4$ has $\frac{n^2}{4}$ operations. $CF + DH = P_1 - P_3 + P_5 + P_7$ has $\frac{3n^2}{4}$ operations.

Thus, we can combine all these terms of runtime and complexities. Now, we can combine the runtime and computational complexities of these operations to express $T_{\text{Strassen}}(n) = 4(T_{\text{Strassen}}(\frac{n}{2}) + \frac{n^2}{4}) + 3(T_{\text{Strassen}}(\frac{n}{2}) + \frac{n^2}{2}) + 2n^2 = 7T_{\text{Strassen}}(\frac{n}{2}) + \frac{9}{2}n^2$.

Now, we want to find the value n at which $T_{\text{Conventional}}(n) = T_{\text{Strassen}}(n)$. Given the recurrence relation we have described above, we want to find the point at which we can switch to our conventional algorithm rather than a recursive call to Strassen's. Thus, inside the runtime of the subproblem, we will substitute $2n^3 - n^2$, the complexity of the conventional algorithm. Then, we set this equal to $2n^3 - n^2$.

$$7(2(\frac{n}{2})^3 - \frac{n}{2}) + \frac{9}{2}n^2 = 2n^3 - n^2 \Rightarrow 7(\frac{n^3}{4} - \frac{n^2}{4}) + \frac{9}{2}n^2 = 2n^3 - n^2 \Longrightarrow \frac{7n^3}{4} - \frac{7n^2}{4} + \frac{18n^2}{4} = 2n^3 - n^2$$

$$\Rightarrow \frac{7n^3}{4} + \frac{11n^2}{4} = 2n^3 - n^2 \Rightarrow -\frac{n^3}{4} + \frac{15n^2}{4}n^2 = 0 \Rightarrow n^2(-\frac{1}{4}n + \frac{15}{4}) = 0$$

The last form above has two solutions. One is a trivial solution, $n = 0$, since of course for a matrix with 0 by 0 matrix, both algorithms take the same time, because there's nothing to compute. The other solution can be solved $-\frac{1}{4}n + \frac{15}{4} = 0 \Rightarrow n = 15$. Thus, $n = 15$ is theoretically the threshold for which Strassen's algorithm becomes more efficient than the conventional algorithm.

Case 2: n is odd Most of the logic holds for this case. The only difference is the size of the subproblems, because we pad matrices with odd dimensions. We are assuming that padding is a constant time operation. Then, we express the recurrence relation as $T_{\text{Strassen}}(n) = 7T_{\text{Strassen}}(\frac{n+1}{2}) + \frac{9}{2}(n+1)^2$. Then, we solve in the same manner by plugging in the computational complexity for the conventional algorithm with this new input of $n+1$.

$$T_{\text{Strassen}}(n) = 7T_{\text{Strassen}}(\frac{n+1}{2}) + \frac{9}{2}(n+1)^2 \Rightarrow 7(2(\frac{n+1}{2})^3 - (\frac{n+1}{2})^2) + \frac{9}{2}(n+1)^2 = 2n^3 - n^2$$

$$\Rightarrow \frac{7}{4}(n+1)^3 - \frac{7}{4}(n+1)^2 + \frac{9}{2}(n+1)^2 = 2n^3 - n^2$$

Notice that the left hand side above stays the same, because the way we describe the number of arithmetic operations in the conventional algorithm doesn't change depending on even or odd dimensions. At this point, we could solve this out through some algebraic manipulation, but instead we can just plot this curve on Desmos and look at the solution. Rounding to the nearest whole number, this yields the solution $n = 37$. Thus, we have found our theoretical thresholds for both cases.

For even n the threshold is $n = 15$, and for odd n the threshold is $n = 37$. The whole derivation process and reasoning is given above.

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 efficiently 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.

Submitted on Gradescope. Our experimental testing shows that our threshold is significantly higher than that of the theoretical threshold, calculated in the previous problem. Our algorithm implementation has many helper functions where that create new arrays for Strassen's algorithm. Some of these functions carry out more operations than is theoretically needed.

For example, we have functions to split a matrix into 4 blocks, as well as a function to merge 4 blocks into one. These are done with the dynamic list functionality in Python. We also initialize many new lists for these. The way that we did it included making new lists entirely. For each matrix, we initialized a new list.

Then, for each row, we made another new list, and then appended entries into that list, and then appended that list for the row to the larger list. Thus, we'd end up with 2 dimensional lists, or a matrix in other words. This is not the most efficient implementation, hence the higher crossing over threshold.

Another thing to note is that we assumed that padding is a constant time operation. However, in practice, how many zeros we pad with is a function of the matrix dimension.

Room for Optimization: Of course, Python is not a language known for its speed. A way to optimize runtime a little more would to simply code in C++ or Go. For example, because Python is an interpreted language, the runtime includes reading, parsing, and then finally executing code. A language like C++ is already compiled at runtime and is thus faster. We have other ideas though.

For the conventional algorithm, at each loop, we calculate Instead of using the append method, we could simply initialize arrays. List comprehension is often faster than appending over and over again, trying to use a dynamic array type of functionality. Even better, with a lower level of thinking, we could simply assign the points in memory to certain blocks of a matrix at any recursive iteration.

For Strassen's algorithm, we use the NumPy library to pad our matrix. This is likely not the most efficient way to pad our algorithm, although this library abstracts away a lot of matrix functionality. However, NumPy works with it's own np.arrays. Thus, when we use this functionality, this requirest that we convert between different datatypes, which is more operations.

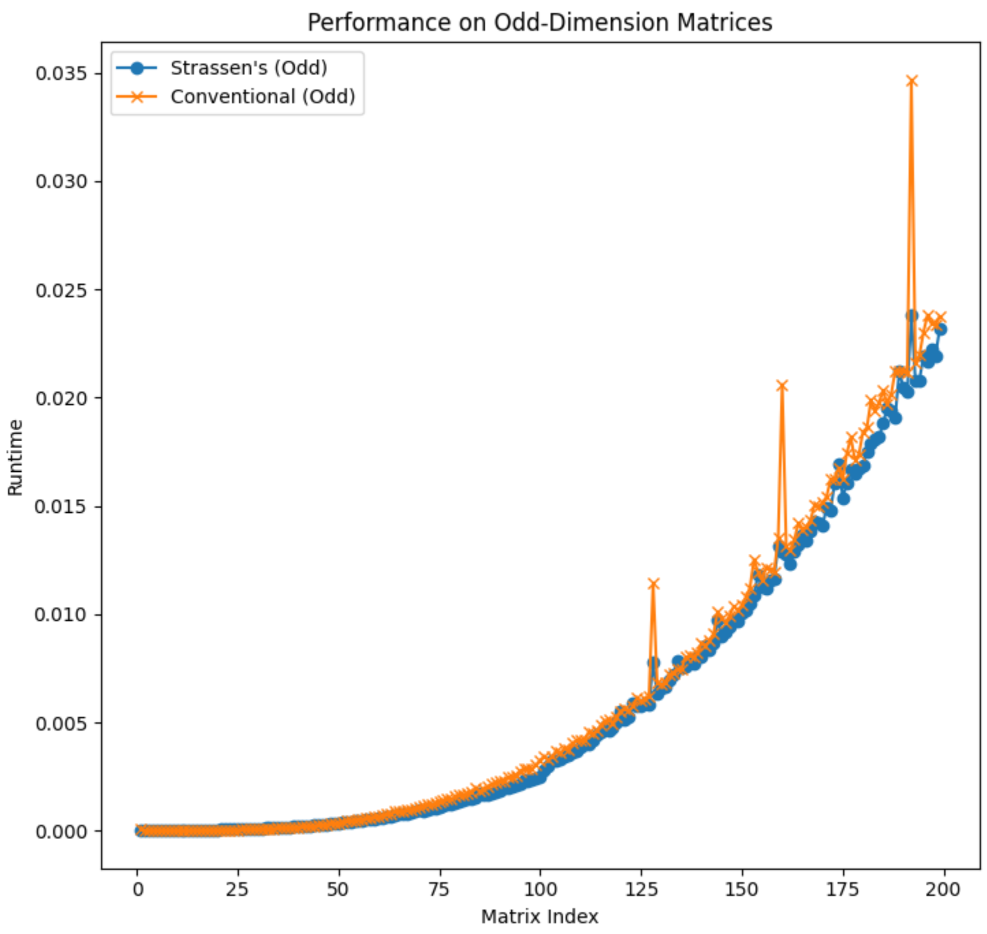


Figure 1: Enter Caption

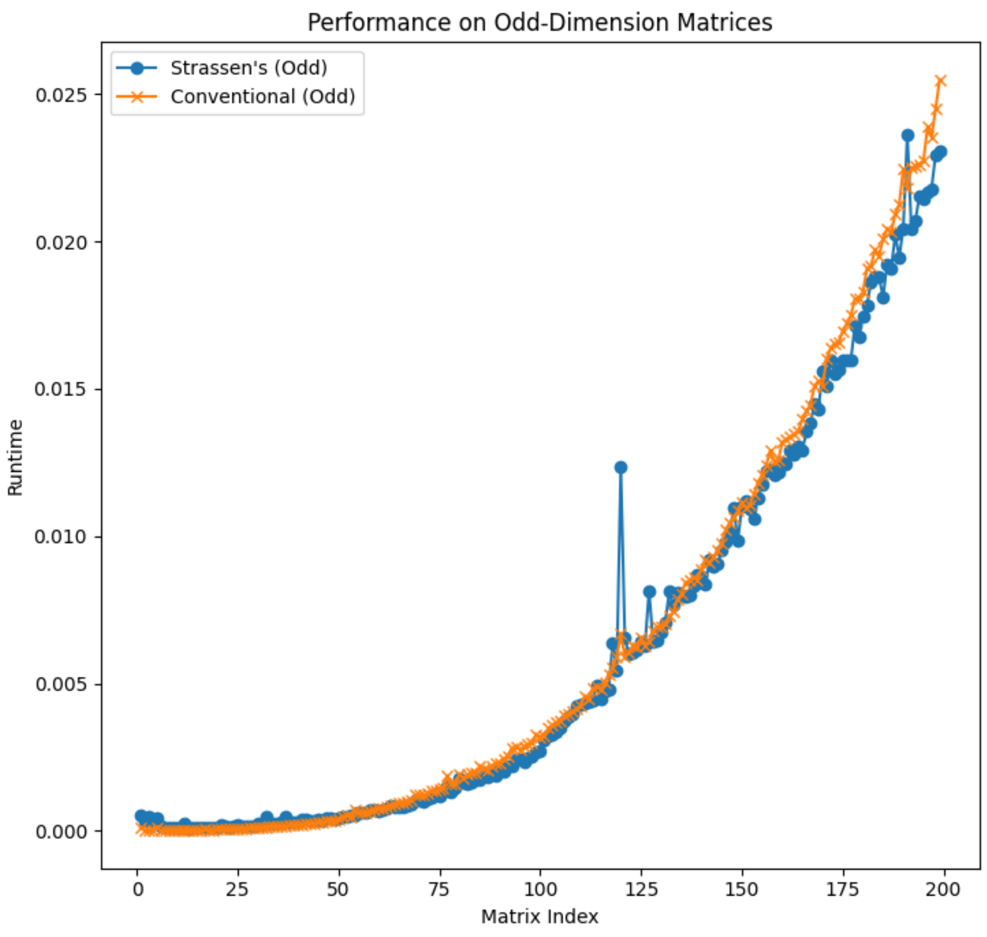


Figure 2: EVEN THIS IS EVEN

THE SECOND GRAPH IS FOR EVEN I JUST MESSED UP
We ran Strassen's and the conventional algorithm in the odd and even cases separately since the expected switch we found was different for the two cases. In these findings, we see that both the odd and the even dimension matrices are faster in matrix multiplication at first using the conventional algorithm. Slowly we see Strasen slowly become faster than Conventional by the gap we see. For Odd matrices, we see the curves differ around 75. OF course the experimental trial might lead to some deviations from the true value since we did actually expect the value that one should switch from conventional to strassens to be 37. Then in the even case Strassens see out perform around 50 instead of 16. We already discussed reasons but another reason why this might be worse than expected is because python has to cache and recurse.

Note Didn't seem to matter what the different entries were 0/1 or 0/1/2 or even random integers from 0 to 100.

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: the (ij)th entry in the matrix A^2 counts the paths from i to j of lenth 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.

Solution: We get around the same results for expected and observed, we took the average over 10

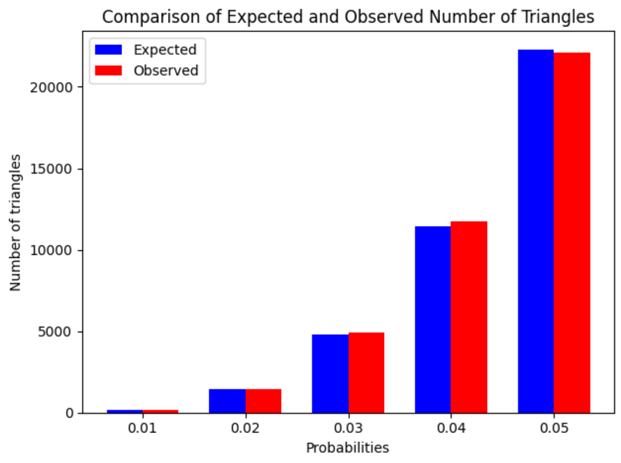
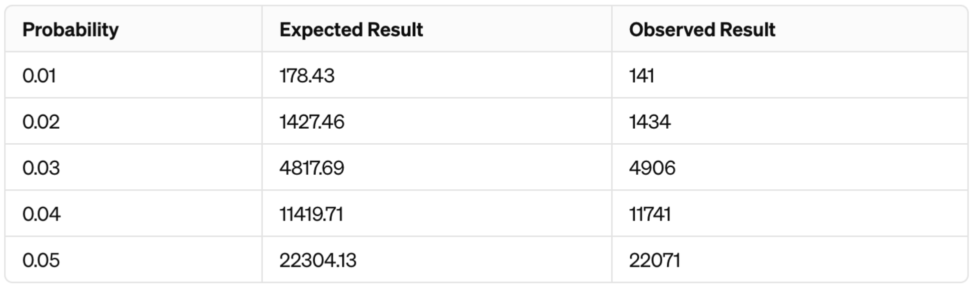


Figure 3: Results



Probability	Expected Result	Observed Result
0.01	178.43	141
0.02	1427.46	1434
0.03	4817.69	4906
0.04	11419.71	11741
0.05	22304.13	22071

Figure 4: Results

trials for each. This should make sense since we are experimentally determining the results. With more and more trials the observed results will approach into expected values.

Code setup:

60% of the score for problem set 2 is determined by an autograder. You can submit code to the autograder on Gradescope repeatedly; only your latest submission will determine your final grade. We support the following programming languages: Python3, C++, C, Java, Go; if you want to use another language, please contact us about it.

Option 1: Single-source file:

In this option, you can submit a single source file. Please make sure to NOT submit a makefile/Makefile if you elect to use this option, as it confuses the autograder. Please ensure that you have exactly one of the following files in your directory if you choose to use this option:

1. strassen.py - for python. In this case, we will run

   ```
   python3 strassen.py <args>
   ```

2. strassen.c - for C. In this case, we will run

   ```
   gcc -std=c11 -O2 -Wall -Wextra strassen.c -o strassen -lm -lpthread
   ./strassen <args>
   ```

3. strassen.cpp - for C++. In this case, we will run

   ```
   g++ -std=c++17 -O2 -Wall -Wextra strassen.cpp -o strassen -lm -lpthread
   ./strassen <args>
   ```

4. strassen.java / Strassen.java - for Java. In this case, we will run

   ```
   javac strassen.java (javac Strassen.java)
   java -ea strassen <args> (java -ea Strassen <args>)
   ```

5. strassen.go - for Go. In this case, we will run

   ```
   go build strassen.go
   go run strassen.go <args>
   ```

Option 2: Makefile:

In this option, you submit a makefile (either Makefile or makefile). In this case, the autograder first runs make. Then, it identifies the language and runs the corresponding command above.

Your code should take three arguments: a flag, a dimension, and an input file:

```
$ ./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.

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.