CS 124 Programming Assignment 2: Spring 2022

Your name(s) (up to two): Vanessa Hu, Daniela Shuman

Collaborators: N/A

No. of late days used on previous psets: Daniela: 5, Vanessa: 4
No. of late days used after including this pset: Daniela: 6, Vanessa: 5

Homework is due Wednesday 2022-03-30 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.

   1. Answer.

   Here, operations refers to non matrix operations unless otherwise specified.

   $$n_{0even} = 16$$

.

$$n_{0odd} = 37$$

. For even vs. odd-sized matrices.

The cross-over point should be when Strassen's runtime $T_s(n)$ no longer has advantage over the standard matrix multiplication's (SMM) runtime $T_m(n)$ (for matrix size n by n): **so when the runtimes are equal and we calculate all further smaller subproblems with SMM and not Strassen's.**

In SMM, where X, Y are two n by n matrices, to calculate each of the n^2 entries $A[i][j]$ for answer $A = XY$, we need to do n multiplications: for row i in $X = [x_1, x_2, ...x_n]$, column j in $Y = [y_1, y_2, ...y_n]$, we must find $x_1y_1, x_2y_2, ...x_ny_n$. Then, we do $n-1$ additions to add up those products to find $A[i][j]$, or $2n-1$ operations per entry: i.e. $n^2(n-1)$ operations to calculate A, so

$$T_m(n) = n^2(2n-1)$$

Now, because Strassen's recursively takes $n/2$-sized subproblems, we consider two cases:

Case 1: even n. Next, Strassen's recurrence is $T_s(n) = 7 \cdot T(\frac{n}{2}) + O(n^2)$, with 7 subproblems of size $n/2$ and $O(n^2)$ work for matrix operations.

To find more exact values for the $O(n^2)$ term, based on our subproblem definitions of $P_1...P_7$, we do 10 overall matrix add/sub operations of matrices size $\frac{n}{2}$ when we calculate all those subproblems (e.g. 1 to find $F-H$ for P_1, 1 to find $A+B$ for P_2, 2 to find $A+D, E+H$ for P_5, etc.) So, $10 \cdot \frac{n^2}{4}$ individual operations to add each matrix entry. To find the each "quarter" of the final answer matrix, we do 3 operations on matrices of size $n/2$ (i.e. involving our P_i's) to find $AE + BG$ ($P_4 + P_5 + P_6 - P_2$), 1 to find $AF + BH$, 1 for $CE + DG$, 3 for $CF + DH$: i.e. 8 matrix operations on matrices of size $n^2/4$, so $8 \cdot \frac{n^2}{4} = 2n^2$ individual operations. $10 \cdot \frac{n^2}{4} + 2n^2 = \frac{9n^2}{2}$ operations total.

Moreover, because the crossover point is when we begin to use SMM for smaller subproblems, instead of $T_s(n/2)$, we instead plug in runtime via SMM: $T_m(n/2) = (\frac{n}{2})^2(2\frac{n}{2} - 1)$, so $T_s(n)$ at crossover point is

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

.

Now, we set the runtimes for SMM, Strassen's at cutoff, to be equal, and solve for n, which will give us our ideal cutoff:

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

$$n = 15$$

However, since this case solely deals with even-sized matrix multiplication, we can make our cutoff point switch at $n_0 = 16$, so all problems of size below the cutoff are better runtime (or theoretically the same at 15) using SMM compared to Strassen.

Case 2: odd n. In order to execute $n/2$-sized subproblems, our n must be even: so if our n is odd, we can "pad" an extra column and row of 0's onto our matrix and multiply two even-sized matrices of size $n+1$. $T_m(n)$ would be the same, since SMM works the same regardless of even/odd size.

Our $T_s(n)$ at crossover point would be modified from Case 1 to replace n with $n+1$ because of our padding method, so it would be

$$7 \cdot \frac{(n+1)^2}{4}(n) + \frac{9(n+1)^2}{2}$$

.

Now, we set the runtimes for SMM, Strassen's at cutoff, to be equal, and solve for n, which will give us our ideal cutoff:

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

$$n \approx 37.170 \approx 37$$

So our cutoff point switch here would be $n_{0odd} = 37$.

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.

   2. Answer.

   Strassen Implementation.

   Our code is in Main_Opt2.java.

   Our Strassen's follows the original algorithm with some space optimizations described below; it relies on *strassen*, a method that writes into a *Goal* matrix (via pointer) with the answer rather than returning a new matrix recursively each time (we wrap it in $finalStrassen$ to create a final matrix to write into, which also calls *padZeros* to optionally pad X, Y up to its next highest power of 2 if it isn't one already; this addresses our odd case for now – see further reflections.).

   To multiply XY, we split each matrix into quadrants. For each quadrant, we rearrange and multiply to reduce the number of multiplications that need to happen. We can describe this algorithm as follows:

   (a) *Cutoff:* if our dimension is less than/equal to the cutoff, we call our $standardMM$ method that writes its answer into the *Goal* and return.

   (b) *Base case:* If we're multiplying 1x1 matrices, we write the product of the two values in X, Y respectively into the 1x1 *Goal* matrix and return.

   (c) Otherwise, we split each matrix into quadrants such that in clockwise order, the matrix X: A, B, C, D and matrix Y: E, F, G, H. (We don't actually generate 8 new matrices, however, refer to each of these subparts using indices).

(d) In order, calculate 7 subparts $P_1, ...P_7$. **To save space,** we allocate fixed memory in 3 matrices of size $n/2$: $left, right, subAns$, to calculate the P_i values starting from $P_1...P_7$. For instance, since $P_1 = A(F-H)$, we write values from X that correspond to A into $left$, values (with subtraction) indexed from Y corresponding to F, H into $right$, and running $strassen$ to multiply $A(F-H)$, writing into $subAns$. And, instead of allocating more space to do matrix additions/subtractions to calculate each "quarter" of the final answer matrix, we add it directly from the current P_i in $subAns$ to where it's used; for instance, $AF + BH = P_1 + P_2$ (top right quarter), $CF + DH = P1 - P3 + P_5 + P_7$ (bottom right), so our *populateAns* when doing P_1 adds the P_1 values to the top right, bottom right respectively (the final argument to *populateAns* when false, subtracts in the case of subtraction). We proceed to find all 7 subproblems, writing them into our final answer matrix.

Optimization Notes. As noted above, we optimized space by avoiding memory copying and creating new matrices per recursive step. In the original Strassen's algorithm, per recursive level, there are an additional eight $n/2$-size matrix allocations for all subcomponents of each matrix (dividing into ABCD, EFGH), seven $n/2$-size matrix allocations per p matrix (and 10 to account for needing to subtract/add matrices to calculate the P subproblems) four $n/2$-size matrix allocations to find the final "quarters" of the answer matrix (with 8 operations as mentioned in Part 1, so 4 additional intermediate matrices when adding/subtracting.)

However, we are reducing these matrix allocations per recursion to only 3 matrix allocations, with $left, right, subAns$.

Further Reflections.

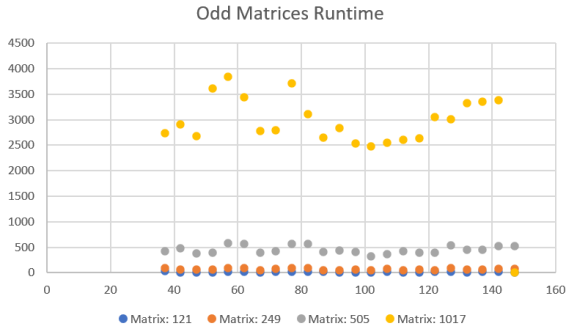
- With more time, we would make our Strassen more space-efficient by creating an odd case (like our theoretical analysis) that just pads the matrix with one extra row/column, instead of padding matrices to the next highest power of 2 to avoid odd dimensions entirely. (You can see remnants of these modifications with $hSize = \lceil n/2 \rceil$ slightly different then $half = \lfloor n/2 \rfloor$ to refer to half the size of a matrix; they're indistinguishable for even matrices). For instance, when we encounter an odd matrix, we could pad it with one row/column, recursively calculate it/ write it into the intended matrix while avoiding the indices with 0 padding (or may need intermediate matrix before transferring the results).
- We also attempted a telescoping method. In order to save space, instead of storing each p matrix per level of Strassens, which would typically be size n/2, we store each recursive value of p within an existing matrix to avoid copying into more memory. We implement this by creating 7 global p matrices for each value of p, each of size n entered into memory once. For every recursive level of strassens, we store the corresponding matrix p in the next $n/2^j$ slots of matrix p, where j is the level in the recursion of strassens. Concretely, the leftmost corner of the new matrix p will be stored at [0, n - $n/2^j$]. This way, we avoid generating each new matrix p for all seven ps per recursion in strassens.

Experimental Setup. For our first two experimental setups, we test an array of input size that we test (we tested matrices of sizes ranging from 100 to 4096), and we tested cutoff points from around 15 up to 245. We ran each test 10 times to reduce the variability, and we took the average of the results. We also ran the algorithm on a few odd-size matrices and a few non-power of 2 even size matrices to see if the pattern held.

Experimental Results.

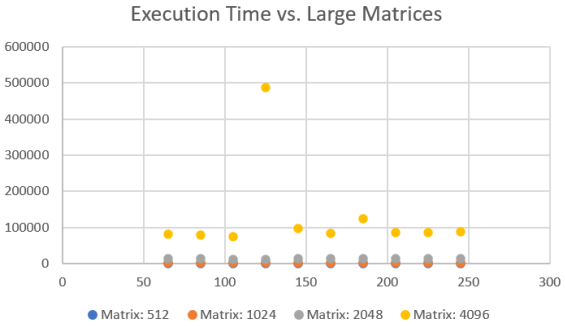
For low (< 1024) matrix size n, we see that there is quite a lot of variation in our final crossover points. We expect there to be some difference in our even matrix sizes and odd matrix sizes. Based on

our implementation, we expect some variability, and there to be some consistent patterns across the matrix sizes. We found that there is a steady increase with some variability until around a crossover point of 100 (which we narrowed to around 105 in the following experiments). Then there is a steady increase with less variability as the crossover point increases.

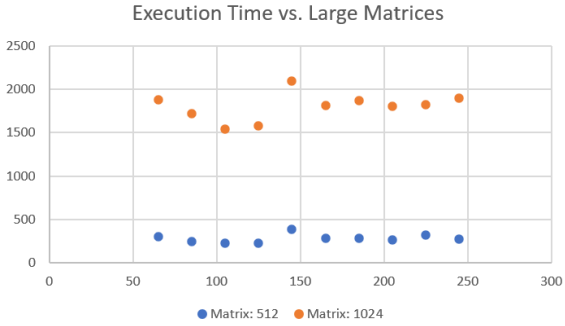


We ran the algorithm on more matrix sizes which are in the appendix.

We ran the algorithm on larger size matrices to get a clearer picture for the best crossover point.



We can look at only matrices of size 512 and 1024 to get a clearer picture.



It looks like our best crossover point, i.e the point that reduces that time of execution is around **105**. This is consistent across matrices that are odd or even.

Our full results are:

The best execution time was found for crossover point at around 105. This means that matrices of size 105 or less are run on the standard matrix multiplication. Matrices larger than this size are more quickly multiplied using our Strassens multiplication algorithm. This crossover time is consistent across our matrix sizes, including even, odd, and powers of 2.

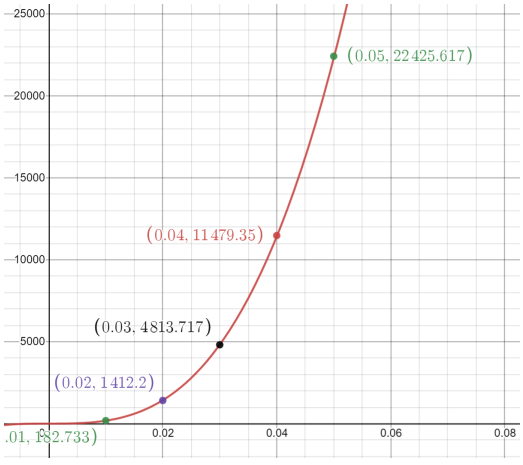
Table 1: Execution Times

Crossover	**Matrix: 512**	**Matrix: 1024**	**Matrix: 2048**	**Matrix: 4096**
65	295.75	1877.25	13213.5	81691
85	240.5	1720.25	12905.75	78870.5
105	227	1542.25	11320.25	74100.25
125	221.25	1583.75	10736	486997.5
145	387.75	2094.75	13972.5	95662.25
165	277.25	1817.75	12225	83904.75
185	277.25	1872.25	12458.25	124052
205	263.25	1806.5	12142	84833
225	321.75	1824.5	12209.75	86437.75
245	273.5	1902.75	12331.5	88574

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.

   3. Answer. *See Triangle.java for our setup.*



   For each value of p, we took the average number of triangles over 10 trials and found results quite consistent with the expected values:

probability	*expected*	*experimental*
0.01	179.433024	182.733333
0.02	1427.464192	1412.2
0.03	4817.691648	4813.71666667
0.04	11419.713536	11479.35
0.05	22304.12800	22425.61666667

Appendix: