CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Jasmine Zhang, Kaitlyn Zhou

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: Jasmine: 4; Kaitlyn: 3
No. of late days used after including this pset: 0

Homework is due Wednesday 2023-03-29 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).

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.

   - We will first explore the run-time of each algorithm.
     - Starting off, we will be exploring the run-time of the conventional matrix multiplication algorithm. Let $T(n)$ be the number of arithmetic operations required to multiply two matrices of the size $n * n$ using the conventional multiplication algorithm. Thus, we will get $T(n) = 2n^3 - n^2$. This is because the total number of operations required is n^3 multiplication and $n^2(n-1)$ additions (in order to compute the dot product). Therefore, we can approximately say that the total number of operations required is $T(n) = n^3 + n^2(n-1)$ or

       $$T(n) = 2n^3 - n^2$$

     - For the run-time of the original Strassen's algorithm: Let $S(n)$ be the number of arithmetic operations required to multiply two matrices of size $n * n$ using the Strassen's algorithm. For this algorithm, we need to compute 7 matrix multiplication of size $\frac{n}{2} * \frac{n}{2}$ and 18 additions/subtractions of the matrices of the same size. Therefore,

       $$S(n) = 7S(n/2) + 18(n/2)^2$$

     - For the run-time of the our modified Strassen's algorithm: Let $R(n)$ be the number of arithmetic operations required to multiply two matrices of size nxn. Our modified Strassen's algorithm is when we start using Strassen's algorithm and then stop the recursion at some size n_0, and use the conventional algorithm below the point. Therefore, our modified algorithm will be $R(n) = 7(T(n/2)) + 18(n/2)^2$ or

       $$R(n) = 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2$$

   - For finding n_0, we need to split into cases in which n_0 is even or n_0 is odd.

- **When n_0 is an even value:** Therefore, in order to find the cross-over point, we would need to solve this inequality: $2n^3 - n^2 < 7(T(n/2)) + 18(n/2)^2$ or

  $$2n^3 - n^2 < 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2$$

  We need to solve this inequality because it expresses the condition in which at some input size n when n becomes smaller, the Strassen's algorithm will become slower than the conventional algorithm. Therefore, this inequality compares the run-time of the conventional algorithm to the run-time of our modified algorithm and compares when the value of the run-time of the conventional algorithm is less than the run-time of the Strassen's algorithm. Therefore, let's calculate the inequality:

  $$2n^3 - n^2 < 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2$$
  $$2n^3 - n^2 < 14(n/2)^3 - 7(n/2)^2 + 18(n/2)^2$$
  $$2n^3 - n^2 < 14(n^3/8) - 7(n^2/4) + 18(n^2/4)$$
  $$2n^3 - n^2 < 7(n^3/4) - 7(n^2/4) + 9(n^2/2)$$
  $$4(2n^3 - n^2) < 4(7(n^3/4) - 7(n^2/4) + 9(n^2/2))$$
  $$8n^3 - 4n^2 < 28(n^3/4) - 28(n^2/4) + 36(n^2/2)$$
  $$8n^3 - 4n^2 < 7(n^3) - 7(n^2) + 18(n^2)$$
  $$8n^3 - 4n^2 < 7(n^3) + 11(n^2)$$
  $$n^3 - 15n^2 < 0$$
  $$n^2(n - 15) < 0$$
  $$n < 0; n < 15$$

  Therefore, as we calculated through the inequality above, we found that our cross-over point for when n_0 is an even value would be

  $$n_0 = 15$$

- **When n_0 is an odd value:** In order to find the cross-over point when n_0 is an odd value, we need to solve the inequality in which

  $$2(n)^3 - (n)^2 < 7(2((n+1)/2)^3 - ((n+1)/2)^2) + 18((n+1)/2)^2$$

  This is due to the fact that we need to include padding vertically (one vertical row of padding) and horizontally (one horizontal row of padding) to our graph to make the graph even. In order to account for this padding, we need to replace the value of n to $n+1$. Therefore, let's calculate the inequality:

  $$2(n)^3 - (n)^2 < 7(2((n+1)/2)^3 - ((n+1)/2)^2) + 18((n+1)/2)^2$$
  $$2n^3 - n^2 < \frac{7}{4}n^3 + 8n^2 + \frac{43}{4} + \frac{9}{2}$$
  $$0 < -\frac{1}{4}n^3 + 9n^2 + \frac{43}{4} + \frac{9}{2}$$
  $$n < 37.17$$

  Therefore, as we calculated through the inequality above, we found that our cross-over point for when n_0 is an odd value would be

  $$n_0 = 37$$

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.

   - For our code, we made a few optimization choices. One optimization choice that we chose was the use of NumPy, which is a fast and efficient numerical library in Python that we used to optimize the addition of matrices. Therefore, we used NumPy functions to speed up our implementation of our Strassen's algorithm. Along with that, we used array splicing in the Strassen algorithm to help avoid making copies of the input matrices as we made the recursive call, which optimizes memory usage and runtime.
   - Additionally since Strassen's algorithm requires the matrices to have even dimension, if the dimension of the matrices is odd at any point in the recursive call, we coded our algorithm so that we add an additional row and column of zeros so that we can make the dimension even at the step. We did not pad the input matrices to a power of two to avoid extra padding, making the algorithm more efficient.
   - One difficulty arose for us when we were testing are code was that our cross-over points ranged from number that were really low to numbers that were really high in value. Because of that, we had to run many trials, and we saw that after a couple of trials, the numbers leveled off, and we took the average of those trials. In addition to that, we found that Python was also inefficient. We had take a couple of minutes to run larger cases, which made it really inefficient when it came to finding the crossover point.

   - **Methodology:** For any dimension n, since the strassen algorithm recursively divides the matrices into $n/2$ size sub matrices, we observed that there is $log(n)$ possibilities for cross-over points that result in a significant difference in the operations used for calculating the product of the two matrices. For example, for $n = 16$, if the optimal cross-over point is at $n_0 = 8$, so it is optimal to switch to standard multiplication when the sub-matrices have a size less than or equal to 8, then crossover points $9 \leq n_0 \leq 15$ would also be optimal since the algorithm would switch to standard multiplication after it divides the matrix into sub-matrices once. Thus, for a range of dimensions up to $n = 1024$, we ran strassen with n_0 at each of the significant possible $log(n)$ cutoff points, giving us an idea of the ranges for the optimal cross-over point for n and with $n_0 = n$ indicating that it is better to just use standard multiplication for n.
   - **Results:**
   - When n is a power of 2:

n	n_0
2	2
4	4
8	8
16	8-15
32	8-15
64	8-15
128	8-15
256	8-15
1024	8-15
2046	8-15

- When n is a power of 3:

n	n_0
3	3
9	9
27	7-13
81	21-40
243	8-15
729	6-11

- When n is a power of 5:

n	n_0
5	5
25	25
125	8-15
625	10-19

- For other odd/even numbers:

n	n_0
50	25-48
146	19-36
161	21-40
386	25-48
514	17-32
769	25-48

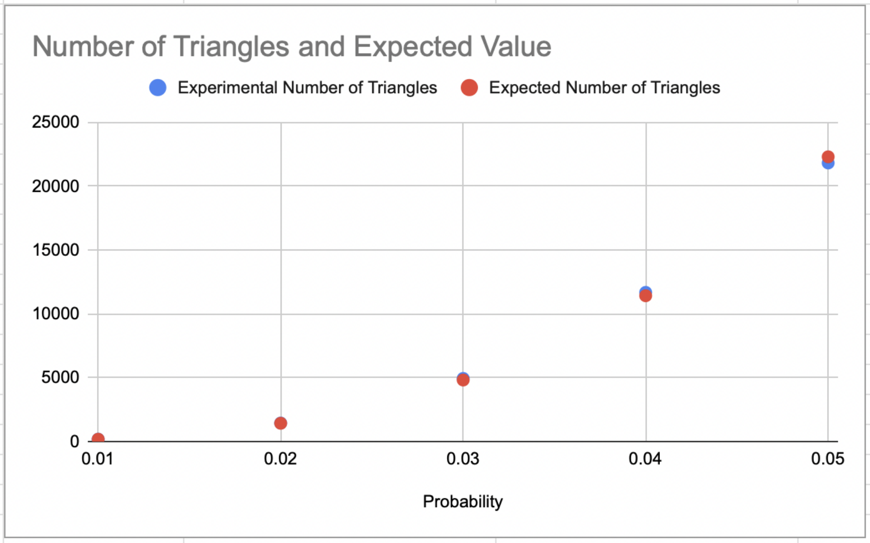
n	Is Strassen's Optimal?
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	True
9	False
10	True
11	False
12	True
13	False
14	True
15	False
16	True
17	False
18	True
19	False
20	True
21	False
22	True
23	True
24	True
25	False
26	True
27	True
28	True
29	False
30	True
31	True
32	True
33	True
34	True
35	True
36	True
37	True
38	True
39	True
40	True
41	True
42	True
43	True
44	True
45	True
46	True
47	True
48	True
49	True

- **Analysis**: Observe from the first four tables that n_0 appears to flatten to be at most in the range 25 to 48 even as n grows. Thus, to narrow on the exact value of n_0, we iterated from dimensions 1 to 50, running strassen at every possible n_0 to see if any of the cross-over points yielded on average a better runtime than just standard multiplication, meaning that it was optimal to run Strassen instead of completely standard multiplication for that dimension. We observed that $n = 29$ was the highest value that standard multiplication was still optimal, so we concluded that $n_0 = 29$ was the general cross-over point for Strassen. However, we observed for n that are powers of two, the optimal crossover point flattens out in the $8 - 15$ range, so we can set $n_0 = 8$ in this case since the matrices can only be sizes that are powers of two. It makes sense that this cutoff is lower than the general cut off, since for powers of two, it will always result in an even sized sub matrix every time we recursively divide the matrix, so we never have to pad the arrays. Thus, we save on the possible $O(n^2)$ cost of padding the matrices, making strassen more efficient so we want to cross over earlier to strassen. For the rest of the numbers, there was much greater variance in ranges optimal values, and this can be also be explained by the padding cost. For example, n = 729 had a lower optimal cross-over point than n = 386 despite being a larger number. Observe that if we decomposed each number into the size of the sub matrices, we get that $729 \to 365 \to 183 \to 92 \to 46 \to 23 \to 12 \to 6 \to 3 \to 2 \to 1$, so we would pad 5 times in total (for n = 729, 365, 183, 23, and 3), while $386 \to 193 \to 97 \to 49 \to 25 \to 13 \to 7 \to 4 \to 3 \to 2 \to 1$ would pad 6 times in total (for n = 193, 49, 25, 13, 7, 3). Thus, n = 386 will result in more padding calls, so it makes sense that it would be more optimal to cross over to strassen later to reduce the extra runtime of padding.
- We note the choices of our matrices does matter, as we ran these experiments on matrices that were randomly filled with either 0 or 1, so single additions and multiplication of matrix elements can be considered mostly $O(1)$ costs and thus it makes sense that our experimental n_0 matches our theoretical calculations. However, if the matrices were populated with large numbers for each element, then we cannot assume that single arithmetic operations take $O(1)$ time.

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.

Probability	Experimental Number of Triangles	Expected Number of Triangles
0.01	185	178.433024
0.02	1447	1427.464192
0.03	4943	4817.691648
0.04	11683	11419.71354
0.05	21822	22304.128



-
- As shown in our table and graph above, our average experimental number of triangles using 5 trials of our Strassen's matrix multiplication code is very close to the expected number of triangles.

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.