CS 124 Programming Assignment 2: Spring 2022

Your name(s) (up to two): Rei Yatsuhashi, Evan Jiang

No. of late days used on previous psets: Rei: 5, Evan: 5
No. of late days used after including this pset: Rei: 7, Evan: 7

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).

Optimizations Implemented:

Our matrix multiplication algorithm (a hybrid between Strassen's recursive algorithm and the conventional algorithm) implemented the following optimizations:

1. Generally, when implementing conventional matrix multiplication for two N by N matrices, we run three for loops with indices $0 \leq i, j, k < N$ (in this order), calculating $\mathrm{C}[i][j] = \mathrm{A}[i][k] \cdot \mathrm{B}[k][j]$. However, we made the following modification, running through the same N^3 cases, but nesting them in the order i, k, j. This is faster if you consider the number of cache hits during our loops. Because our array memory is stored linearly in the order $\mathrm{A}[0][0], \mathrm{A}[0][1], \ldots, \mathrm{A}[0][N-1], \mathrm{A}[1][0], \ldots, \mathrm{A}[1][N-1], \ldots, \mathrm{A}[N-1][N-1]$. By iterating through k first, we are able to traverse through $\mathrm{A}[i][k]$ in the order it is stored in the memory, implying that there are more cache hits (the alternative i, j, k ordering misses the cache on virtually every iteration for large N), and benefits from cache prefetching.

2. In implementing our padding, we had two main options. Either pad at every step of the recursion where the subproblem had odd dimension (M by M matrix where M is odd), or pad once at the beginning until a power 2^k. We chose to implement the former. In evaluating why this is the better method, we consider memory in the worst case. If our matrix is of size $2^k + 1$ by $2^k + 1$, given the method of padding to the nearest power of 2, we must pad with $(2^{k+1})^2 - (2^k + 1)^2 = 2^{2k+2} - 2^{2k} - 2^{k+1} + 1 = O(N^2)$ zeroes. In contrast, if we pad by one row and column at every step where the subproblem has odd dimension, in the worst case where our initial matrix is of size $2^k + 1$ by $2^k + 1$, we will only use $\leq \sum_{i=0}^{k} (2^i + 2)^2 - (2^i + 1)^2 = O(N)$ extra memory.

The multiplication algorithm as optimized above (in Python) was sufficiently efficient for this assignment. If this were insufficient, we considered the following optimization. Instead of allocating new memory to solve each subproblem which would require $\sim O(N \log N)$ space (requires $\log N$ copies of the initial matrix), we considered keeping track of the pointer addresses and computing using those, so that we could run Strassen's using only one copy of the matrix, or $\sim O(N)$ space.

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 consider the number of arithmetic operations necessary for the multiplication of two n by n matrices A and B, with a product $C = AB$.

   (a) In the conventional algorithm, there are n^3 multiplications ($a_{i,j}b_{j,k}$ where $1 \leq i, j, k \leq n$). We know that $c_{i,k} = \sum a_{i,j}b_{j,k}$, of which there are n summands, so there are $n-1$ additions per entry for the n^2 entries in our product matrix. Thus, the conventional algorithm for two n by n matrices has cost $n^3 + (n-1) \cdot n^2 = 2n^3 - n^2$.

   (b) With our implementation of Strassen's algorithm, for odd values of n, we pad our matrix with an extra column and row of 0's to yield an $n+1$ by $n+1$ matrix with even dimension. Thus, we have the following recurrence where $T(n)$ represents the number of single arithmetic operations necessary to multiply two n by n matrices: $T(n) = 7T\left(\lceil \frac{n}{2} \rceil\right) + 18\lceil \frac{n}{2} \rceil^2$ as there are 10 additions to make the seven $\lceil \frac{n}{2} \rceil$ by $\lceil \frac{n}{2} \rceil$ matrices used in Strassen's algorithm, and 8 additions to create the four $\lceil \frac{n}{2} \rceil$ by $\lceil \frac{n}{2} \rceil$ matrices that will be combined to yield our product matrix C.

   In finding n_0, the cross-over point, we find the values n such that the **conventional algorithm** costs less operations than running **one iteration of Strassen then the conventional algorithm for the subproblems of size** $\lceil \frac{n}{2} \rceil$. We break our problem down into the even and odd cases, i.e. when $n = 2k$ or $n = 2k+1$ for some $k \in \mathbb{Z}_+$:

   (a) **If n is even**: The conventional algorithm for the multiplication of two $2k$ by $2k$ matrices takes: $2(2k)^3 - (2k)^2 = 16k^3 - 4k^2$ single arithmetic operations.
   Running Strassen for one iteration, then using the conventional algorithm for the smaller subproblems takes: $T(n) = 7 \cdot (2k^3 - k^2) + 18k^2 = 14k^3 + 11k^2$.
   We see that our Strassen approach is faster, i.e. $14k^3 + 11k^2 \leq 16k^3 - 4k^2$ when $n = 2k \geq 15$. Thus, for matrices with even dimension, the cross-over point is for dimension 15 by 15 (given a matrix 15 by 15, pad then Strassen's).

   (b) **If n is odd**: The conventional algorithm for the multiplication of two $2k+1$ by $2k+1$ matrices takes: $2(2k+1)^3 - (2k+1)^2 = 16k^3 + 20k^2 + 8k + 1$ single arithmetic operations.
   Running Strassen for one iteration, then using the conventional algorithm for the smaller subproblems takes: $T(n) = 7 \cdot (2(k+1)^3 - (k+1)^2) + 18(k+1)^2 = 14k^3 + 53k^2 + 64k + 25$.
   We see that our Strassen approach is faster, i.e. $14k^3 + 53k^2 + 64k + 25 \leq 16k^3 + 20k^2 + 8k + 1$ when $n = 2k + 1 \geq 38$. Thus, for matrices with odd dimension, the cross-over point is for dimension 37 by 37 (given a matrix 37 by 37, pad then Strassen's).

   Thus, our crude estimates for n_0 are:

   (a) $n_0 = 15$ when dealing with an n by n matrix where n is even

   (b) $n_0 = 37$ when dealing with a n by n matrix where n is odd

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.

   We determined our cross-over point n_0 experimentally as follows. We run the method described later for values $N \in \{2^7+1,\ 2^8,\ 2^8+1,\ 2^9,\ 2^9+1,\ 2^{10},\ 2^{10}+1\}$ and our initial candidates for $n_0 \in \{2^k \mid 2 \leq k \leq 8\}$. We then attempt to find a more precise value for n_0. The rationale for the candidate values for N and n_0 can be explained as follows. Given $N = 2^k$, until the cross-over point, Strassen will recurse through matrices of size 2^ℓ by 2^ℓ for $\ell < k$, thus implying that all subproblems are of identical structure. This also makes testing values of n_0 convenient as we only need to check values of n_0 of the form 2^m. To test how our algorithm works with odd outputs, we test values of the form $N = 2^k + 1$. Given our algorithm's padding method, our algorithm will add a row and column of zeroes to yield a matrix of size $N + 1 = 2^k + 2$, so Strassen's will break our matrix down into subproblems of size $2^{\ell-1}+1$ by $2^{\ell-1}+1$ for $\ell < k$, thus implying that all subproblems are of identical structure. This also makes testing values of n_0 convenient as we only need to check values of n_0 of the form $2^m + 1$, but for our initial experiment, cross-over values 2^m are a sufficient estimate. We propose the following experimental method for candidate values (N, n_0):

   (a) Generate matrices of dimension N by N with entries randomly chosen $\in \{0, 1, 2\}$.

   (b) Run our version of Strassen's algorithm (for odd dimension output, pad with zeroes) until reaching a matrix of size M by M where $M \leq n_0$. Then switch to the standard matrix multiplication method.

   (c) Time and record the running time of the algorithm.

   (d) Repeat (a) through (c) for three trials, and record the average time across these trials as our final result to reduce error.

   The table for the time per trial [s] for $N = 2^k$ is below:

Crossover point (n_0)	$N = 256$	$N = 512$	$N = 1024$
4	4.67	33.33	236.96
8	2.73	19.53	140.51
16	2.03	14.56	104.51
32	1.84	12.75	92.46
64	**1.74**	**12.68**	**91.87**
128	1.81	13.34	97.56
256	1.90	14.62	105.59

   We can reasonably restrict our range for the experimental cross-over point to $[50, 80]$. To test the consistency of this cross-over value with odd values in a non-redundant manner, we record the time per trial [s] for $N = 2^k + 1$ for values in this range:

Crossover point (n_0)	$N = 129$	$N = 257$	$N = 513$	$N = 1025$
50	0.298	2.20	15.9	114.9
55	0.297	2.18	15.9	113.6
60	0.297	2.18	15.9	113.0
65	**0.244**	**1.83**	**13.5**	**95.9**
70	0.260	1.94	14.6	101.7
75	0.260	1.94	14.4	101.5
80	0.260	1.94	14.3	101.4

We observe a clear local minimum at $n_0 = 65$, which is consistent with our findings for the power of 2 case above. Finally, to isolate a more accurate value of n_0, testing different subproblem sizes, we observe runtimes for our algorithm on $250 \leq N \leq 254$ and $60 \leq n_0 \leq 70$:

n_0	$N = 250$	$N = 251$	$N = 252$	$N = 253$	$N = 254$
60	1.85	1.83	1.83	1.83	1.82
61	1.85	1.83	1.83	1.82	1.82
62	1.84	1.83	1.83	1.82	1.82
63	**1.68**	**1.67**	**1.66**	1.83	1.82
64	1.79	1.78	1.77	**1.77**	**1.76**
65	1.79	1.78	1.77	1.78	1.77
66	1.79	1.78	1.77	1.78	1.77
67	1.79	1.78	1.78	1.83	1.77
68	1.79	1.78	1.77	1.81	1.77
69	1.78	1.78	1.78	1.78	1.77
70	1.78	1.78	1.80	1.78	1.77

Thus, we have experimentally determined our value of n_0 to be ≈ 63.

From Part 1, we make two assumptions that contribute to the variance in the experimental versus analytical crossover point n_0.

(a) The cost of any single arithmetic operation is 1.

(b) The cost of all other operations is 0.

However, the cost of single arithmetic operations is not the same — for example, the cost of an addition of smaller-bit numbers is going to be less expensive than the cost of adding two larger-bit numbers. Moreover, we do many comparisons and creating of submatrices, which contribute a non-zero amount to the overall runtime. These factors result in our Strassen's being slower than analytical expectation, which is why our resulting $n_0 \approx 63$ instead of matching the estimates from Part 1.

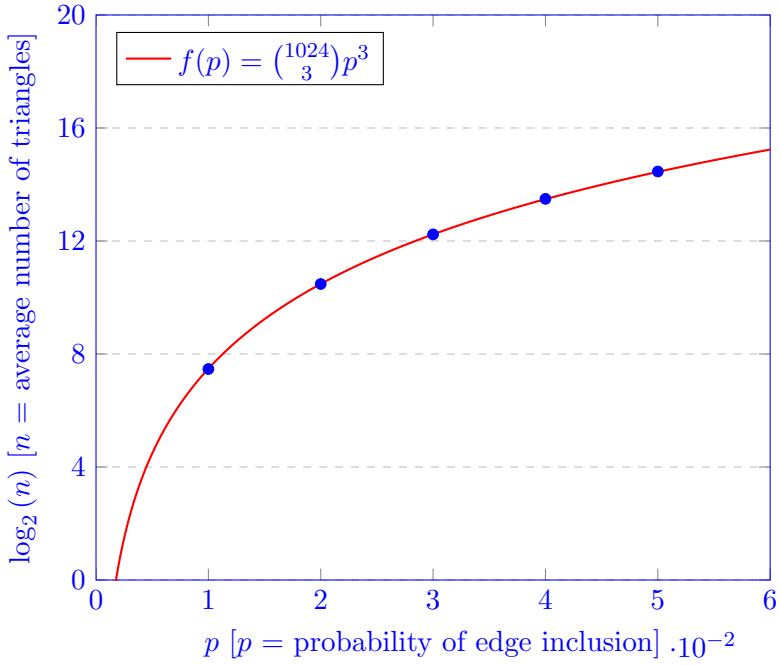
3. Triangle in random graphs: Recall that you can represent the adjacency matrix of a graph by a matrix A. Consider an undirected graph. It turns out that A^3 can be used to determine the number of triangles in a graph: the (ij)th entry in the matrix A^2 counts the paths from i to j of lenth two, and 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.
   Using the above procedure and our matrix multiplication algorithm, we recorded the following results:

p	Trial 1	Trial 2	Trial 3	Average	$\binom{1024}{3}p^3$
0.01	187.33	171.17	166.83	177.11	178.43
0.02	1456.17	1419.83	1402.50	1426.17	1427.46
0.03	4810.00	4808.17	4834.17	4817.44	4817.69
0.04	11576.50	11504.67	11483.17	11521.44	11419.71
0.05	22217.33	22424.33	22964.00	22535.22	22304.13

   Plotting the average number of triangles on a random graph of 1024 vertices by p-value yields:

   Average Number of Triangles n on a Random Graph of 1024 vertices with Edge Inclusion Probability p



   as desired.

The variance from the expected and the experimental values in Part 3 comes from the random generation of edges for the graph, as we generate edges whenever the randomly generated number $\in [0, 1]$ is less than or equal to the p value at hand. Since the number of edges created is not going to match the expected number of edges, the number of triangles from the experiments will be slightly different than what is predicted by $\binom{1024}{3}p^3$.