CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Michael Xiang, Marcos Johnson-Noya

Collaborators: None.
No. of late days used on previous psets: 0 for both.
No. of late days used after including this pset: 0 for both.

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.

Notation: Let $S(n)$ denote Strassen's algorithm and let $T(n)$ represent the traditional algorithm.

Strassen: From lecture we know that the recurrence of Strassen's algorithm is

$$S(n) = 7S(\lceil \frac{n}{2} \rceil) + O(n^2)$$

But we require a more rigorous analysis. From our implementation of Strassen's algorithm, we see that we have 18 matrix additions and subtractions, so $O(n^2)$ is really $18(\lceil \frac{n}{2} \rceil)^2$.
Thus, we have that

$$S(n) = 7S(\lceil \frac{n}{2} \rceil) + 18(\lceil \frac{n}{2} \rceil)^2$$

with the following base case

$$S(1) = 1$$

Traditional: From the lecture we know that the runtime of the traditional algorithm is $O(n^3)$. On closer analysis of our implementation of the traditional algorithm, though, we see that we have n^3 multiplications. We also have $n^2(n-1)$ additions and subtractions, because, for each of the n^2 entries in our matrix, we add n terms, which requires $n-1$ additions. Thus, our total runtime of $O(n^3)$ can really be expressed as

$$T(n) = n^3 + n^2(n-1) = n^2(2n-1)$$

Modified: Our modified algorithm does Strassen's until it reaches some n_0, where n_0 is the point at which running the traditional algorithm is equal to Strassen's. This is because, for smaller values, the traditional algorithm runs faster because the constants in Strassen's runtime are much bigger than the traditional algorithm. Thus, we want to find the point n_0 where Strassen's algorithm will begin to be more efficient than the traditional algorithm, which starts for all $n > n_0$, so to find n_0, we will set the respective runtimes equal to each other. Our modified algorithm will recurse and choose the algorithm with the fastest runtime, so it will choose either the traditional or Strassen's algorithm when it recurses. Thus, our modified algorithm can be expressed as follows.

$$M(n) = 7\min(S(\lceil \frac{n}{2} \rceil), T(\lceil \frac{n}{2} \rceil)) + 18(\lceil \frac{n}{2} \rceil)^2$$

At the point n_0, $T(n_0) = S(n_0)$, and for any $n < n_0$, the traditional algorithm is faster. Thus we can find n_0 through the following equation:

$$S(n) = T(n) \Rightarrow n_0^2(2n_0 - 1) = 7(\lceil \frac{n_0}{2} \rceil^2 (2\lceil \frac{n_0}{2} \rceil - 1) + 18(\lceil \frac{n_0}{2} \rceil)^2$$

Since we want to account for all $n \times n$ matrices, we have to account for both odd-sized and even-sized matrices. Since n_0 is also representing the size of the matrix such that $T(n_0) = S(n_0)$, we must account for two options for n_0. Either n_0 is even or n_0 is odd, to represent even and odd-sized matrices respectively.

Case 1: $\mathbf{n_0}$ is even.

If n_0 is even, then $n_0 = 2k$ for some $k \in Z$.

Then, by substituting $n_0 = 2k$, our equation for equating the runtimes for the traditional and Strassen's algorithm turns from

$$n_0^2(2n_0 - 1) = 7(\lceil \frac{n_0}{2} \rceil^2 (2\lceil \frac{n_0}{2} \rceil - 1) + 18(\lceil \frac{n_0}{2} \rceil)^2$$

into

$$(2k)^2(4k - 1) = 7(k^2(2k - 1)) + 18k^2$$

Solving for k With Wolfram-Alpha, we get that $k = \frac{15}{2}$, and since $n_0 = 2k$, we get that

$$n_0 = 2k = 2(\frac{15}{2}) = 15$$

Since we want n_0 to be even, we round down since we want to find the point at which it is definitely more optimal to run the traditional algorithm. If we round up to 16, then running Strassen's would be more optimal since $16 > 15$. Thus, we want to round down to 14, so we get

$$n_0 \approx 14$$

Case 2: $\mathbf{n_0}$ is odd.

If n_0 is odd, then $n_0 = 2k + 1$ for some $k \in Z$.

By substituting $n_0 = 2k + 1$ in our equation that equates the runtimes of the traditional algorithm and Strassen's algorithm, we get

$$(2k + 1)^2(4k + 1) = 7((k + 1)^2(2k + 1)) + 18(k + 1)^2$$

Solving for k with Wolfram-Alpha, we get that $k \approx 18.085$, so $k > 18$. Since we define $n_0 = 2k + 1$, we get that

$$n_0 > 2(18) + 1 = 37$$

Thus, we get that $n_0 \approx 37$ when n_0 is odd.

So, for matrices of **even size**, the crossover point is ≈ 14, while for matrices of **odd size**, the crossover point is ≈ 37.

Considerations:

There are some machine-dependent factors that we should consider. For instance, it may take longer to experimentally reach the crossover point because we have to consider the time it takes to store matrices in memory. Also, depending on the machine, it may take longer to access data within memory, which may affect the time it takes to conduct certain operations like accessing matrix entries and adding them. Thus, the crossover points that we have calculated may differ from our experimental results.

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.

   Implementation:
   In order to determine n_0, We randomly generated two matrices A and B of size dimension. We did this in C++ using the following code:

```
std::vector<int> values = {-1, 0, 1};

// Seed the random number generator
std::srand(static_cast<unsigned>(std::time(0)));
dimension = 512;
// Create a 2D vector of size (rows, cols) with random elements from the 'val
std::vector<std::vector<int>> matrix_a(dimension, std::vector<int>(dimension)
std::vector<std::vector<int>> matrix_b(dimension, std::vector<int>(dimension)

for (int i = 0; i < dimension; ++i) {
    for (int j = 0; j < dimension; ++j) {
        int random_index = std::rand() % values.size();
        matrix_a[i][j] = values[random_index];
    }
}
for (int i = 0; i < dimension; ++i) {
    for (int j = 0; j < dimension; ++j) {
        int random_index = std::rand() % values.size();
        matrix_b[i][j] = values[random_index];
    }
}
```

   Afterwards, we then iterated through all possible values of n_0, and compared whether Strassen was faster than the traditional method. We tested with dimension as 512 strassen was faster than traditional averaged over 10 trials.
   Results: We get the following results when dimension is 512:

n_0	**Average Traditional Time**	**Average Strassen Time**
16	99.4683	102.546
17	99.506	102.326
18	99.4549	102.282
19	100.733	102.561
20	100.295	105.004
21	100.688	102.856
22	99.9878	102.509
23	99.8058	102.197
24	99.3932	101.517
25	99.3913	106.215
26	100.426	101.828
27	100.059	102.542
28	100.378	102.53
29	100.217	106.139
30	100.36	102.394
31	100.386	102.915
32	100.351	56.7021
33	99.9483	57.0991
34	100.338	56.7396
35	100.416	56.5622
36	99.8791	56.999
37	99.8041	57.2325
38	100.309	57.1661
39	100.301	56.5747
40	100.331	56.6498
41	100.33	56.6851
42	100.069	56.6366
43	99.8854	56.2016
44	100.147	56.2366
45	100.071	56.9867
46	100.347	56.6879
47	99.9261	56.9854
48	100.529	56.6954
49	99.9157	56.6512
50	100.363	56.5636
51	99.6987	56.6807
52	99.7452	56.8605
53	99.6364	56.5703
54	99.9013	56.4636
55	99.8887	56.5134
56	99.8211	56.5132
57	99.4955	56.2637
58	99.5219	56.2587
59	99.4961	56.2391
60	99.4381	56.2112
61	99.4596	56.2493
62	99.3636	56.7463
63	99.3397	57.0416
64	99.4641	46.6952

Analysis: Given this data we may claim that the experimental cross over point is $n_0 \approx 32$. Using this crossover point, the time to perform the Strassen algorithm dramatically decreases. We know since from how we've implemented our algorithm we pad the matrix to the next power of two with zeros, so we know that the runtime of multiplying matrices of size 2^k will be the same as $2^{k+1} - 1$. For this reason it is not necessary to analyze n_0 for matrices which are not of the form $n = 2^k$.

Interestingly, our experimental crossover points are higher than the theoretical ones. This discrepancy can be attributed to various factors that were not considered during the analytical calculations. One significant factor is the time spent on memory allocation, which becomes more pronounced as new matrices of size n/2 are created recursively. Additionally, our efficient implementation of the traditional multiplication algorithm may have reduced its run time, making it harder for Strassen's algorithm to outperform it for smaller matrices.

Our experiments involved multiplying integer matrices due to the requirements of our implementation. We chose integers to avoid potential issues that may arise with floating-point values, such as approximation errors and the possibility of constant time operations no longer being constant.

In terms of practical application, the concept of a "crossover point" provides a useful framework for determining when Strassen's algorithm performs better than the recursive approach. However, it is essential to consider the specific implementation details and the nature of the input matrices to obtain accurate results.

We also implemented substantial optimizations to both the naive and Strassen's algorithms, which could have contributed to the differences between experimental and theoretical crossover points. Comparing different types of multiplicand matrices could provide further insights into the performance of these algorithms under various scenarios.

In conclusion, our findings emphasize the importance of considering implementation details and potential optimizations when determining the optimal crossover point between the traditional multiplication algorithm and Strassen's algorithm. Additionally, a thorough examination of different matrix types could provide valuable information about the algorithms' behavior in diverse situations.

To optimize the traditional matrix multiplication method we added loop tiling to optimize cache utilization. The blockSize parameter can be adjusted based on the system's cache size for better performance. Loop tiling, also known as loop blocking, is an optimization technique that helps improve the cache utilization and memory access pattern of algorithms, such as matrix multiplication.

Matrix multiplication involves accessing elements from two input matrices and accumulating the results in a third output matrix. In the standard matrix multiplication method, the elements are accessed in a row-major order for one matrix and column-major order for the other. This access pattern can lead to cache misses, as the elements are not accessed sequentially in memory.

Loop tiling aims to improve cache efficiency by dividing the matrices into smaller submatrices or blocks. This way, a block of elements can be loaded into the cache, and the computation can be performed on this block. The idea is to perform calculations on smaller blocks of data that fit into the cache, reducing cache misses and improving the overall memory access pattern.

In addition we get that for random multiplicand matrices containing {-1,0,1} took 94654084 nanoseconds to be multiplied, random matrices containing {0,1} took 94569209 nanoseconds to be multiplied, and last matrices containing {0,1,2} took 94628209 nanoseconds to complete. Overall, we can say that the multiplicands have no significant effect on performance of the algorithm.

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.

   Implementation:
   We created an adjacency matrix A and stored it as a 1024×1024 size matrix, where $A[i][j]$ represents whether or not there is an edge from vertex i to vertex j. To simulate probability, we used C++'s uniform real distribution method from its standard library to generate a number p_0 in the interval of $[0, 1)$. During a double for loop through our 1024 vertices, for all pairs of vertices (i, j), we would generate an independent, new random number p_0. If $p_0 < p$ for the following values of p listed above, then we would include an undirected edge from i to j and from j to i. We know that this method of generating edges ensures that an edge is included with probability p, because the uniform real distribution method, by construction, helps generate a uniform number p_0. By rules of uniform numbers, the probability that $p_0 < p$ is equal to p, so including the condition that an edge is only included if $p_0 < p$ will ensure that an edge is added with probability p, since our randomly generated p_0 is less than p with probability p. We then used our modified Strassen's algorithm to calculate A^3 by calculating $A \times A = A^2$, and then $A^2 \times A = A^3$, and we added along the diagonals of A^3 and divided the total number we got after additions by 6 to report our result, which lines up with the problem statement.
   For each value of p, we ran 100 trials and averaged them out, so that we could report a number that converged closer to the expected value by the law of large numbers. We also did this to shield our data against any outliers.
   Results:
   For each p, we calculated the expected number of triangles via the equation given, $E = \binom{1024}{3}p^3$. We also reported the average number of triangles per 10 trials ran. Our results are below.

p	**Calculated Number of Triangles**	**Expected Number of Triangles**
0.01	179.71	178.43
0.02	1434.74	1427.46
0.03	4815.12	4817.69
0.04	11383.82	11419.71
0.05	22315.23	22304.13

   As we can see, the calculated number of triangles for all p is very close to the expected number, differing at most by ≈ 36, which is an insignificant number considering the large magnitude of the expected number of triangles for all p. We see here that, by using our modified Strassen's algorithm, we are able to compute A^3 and count the number of triangles within an arbitrary graph efficiently and correctly.

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.