CS 124 Programming Assignment 2: Spring 2022

Your name(s) (up to two): Katrina Brown and Shirley Zhu

No. of late days used on previous psets: Katrina (3), Shirley (5)
No. of late days used after including this pset: Katrina (3), Shirley (5)

Task 1:

We will consider the number of arithmetic operations that occur while using optimized Strassen's compared to regular matrix multiplication for an n x n matrix. When the number of operations used in an optimized Strassen's is less than the number of operations used in regular matrix multiplication then the runtime of Strasses's is faster than the runtime of conventional matrix multiplication and we have found the crossover point n_0.

First we will evaluate the number of operations for regular matrix multiplication. In order to get one entry in the final matrix, we multiply across a row of the first matrix and down the column of the second matrix, giving us n multiplication operations between real numbers. Then, we sum the n products which is $n-1$ addition operations. Thus, $n+n-1$ operations are used to calculate one entry in the final matrix. With n^2 entries, we have a total of

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

arithmetic operations for conventional matrix multiplication of two n x n matrices.

Now looking at Strassen's algorithm, we see that there are 18 additions between submatrices of size $\frac{n}{2}$ x $\frac{n}{2}$. Addition between $\frac{n}{2}$ x $\frac{n}{2}$ matrices means $(\frac{n}{2})^2$ arithmetic operations, because there are $(\frac{n}{2})^2$ entries in the final matrix that each require one addition between real numbers. There are also 7 multiplications between submatrices of size $\frac{n}{2}$ x $\frac{n}{2}$ in Strassen's algorithm. Since in optimized Strassen's we will use the conventional matrix multiplication algorithm for submatrix multiplication, then each multiplication between $\frac{n}{2}$ x $\frac{n}{2}$ matrices gives a total of $(\frac{n}{2})^2(2*\frac{n}{2}-1) = (\frac{n}{2})^2(n-1)$ arithmetic operations. Hence, the 18 matrix additions and 7 matrix multiplications give

$$18(\frac{n}{2})^2 + 7(\frac{n}{2})^2(n-1)$$

arithmetic operations for multiplicaton between n x n matrices using Strassen's algorithm.

We set the two equations equal to each other

$$n^2(2n-1) = 18(\frac{n}{2})^2 + 7(\frac{n}{2})^2(n-1)$$

and solve to get

$$n = 15.$$

Even values of n would give integer values for $\frac{n}{2}$, so our expression for Strassen's is valid and the crossover point in the case of even n's is

$$n_0 = 15.$$

Now, we need to consider the second case where there is an odd value for n. For Strassen's, when n is odd, then we will add a padded row of zeroes and a column of zeroes. Then, we will have an $(n+1)$ x $(n+1)$ matrix. The standard matrix will work with odd n so then, we compare

$$n^2(2n-1) = 18(\frac{n+1}{2})^2 + 7(\frac{n+1}{2})^2 n.$$

Solving for n, we get $n^2(2n-1) = (\frac{n+1}{2})^2(18+7n)2n^3 - n^2 = \frac{(n^2+2n+1)(18+7n)}{4}2n^3 - n^2 = \frac{18n^2+36n+18+7n^3+14n^2+7n}{4}8n^3 - 4n^2 = 7n^3 + 32n^2 + 43n + 18n^3 - 36n^2 - 43n - 18 = 0n = 37.17$

The equation is satisfied when $n = 37.17$. Let's examine what occurs at n values close to 37.17 to determine the crossover point.

For $n = 37$,

$$n^2(2n-1) = 37^2(37-1) = 99,937$$

and

$$18(\frac{n+1}{2})^2 + 7(\frac{n+1}{2})^2 n = 18(19^2) + 7(19^2)(37) = 99,997.$$

99,937 operations used in standard matrix multiplication is less than the 99,997 operations used in an optimized Strassen's algorithm, so for $n = 37$, it is still more optimal to use the conventional multiplication algorithm.

For $n = 39$,

$$n^2(2n-1) = 39^2(39-1) = 117,117$$

and

$$18(\frac{n+1}{2})^2 + 7(\frac{n+1}{2})^2 n = 18(20^2) + 7(20^2)(39) = 116,400.$$

117,117 operations used in standard matrix multiplication is more than the 116,400 operations used in an optimized Strassen's algorithm, so for $n = 39$, it is more optimal to use Strassen's algorithm.

For the case of odd n's, when $n < 38$, the number of operations using standard matrix multiplication is less than the number of operations for Strassen's. When $n > 38$, the number of operations using standard matrix multiplication is greater than the number of operations for Strassen's. Thus, the cross-over point for the second case is

$$n_0 = 39.$$

Task 2:

Implementation Overview:

We implemented Strassen's algorithm in C++.

We implement a function runStrassen() that allocates an $n \times n$ result matrix, and then a function strassen() that takes as input pointers to the result matrix, and the two matrices to be multiplied. All subsequent recursive calls occur when strassen() calls itself, and all recursive calls take a pointer to the same $n \times n$ result matrix. In order for future recursive calls to strassen() to edit values directly in the result matrix, and read in values directly from the two input matrices, each call to strassen() also passes the indices of the first row and column in each matrix to be operated upon. Finally, strassen() takes an input parameter for the crossover point that we determined analytically. All together, the function strassen takes the following parameters:

(arr1, arr2, res, int size, int r1, int c1, int r2, int c2, int r3, int c3, int cross_over_point)

We took multiple steps to optimize the time and space complexity of our implementation of Strassen's. As we optimized our Strassen's algorithm to reduce the time required to run on any given input matrices, the experimental crossover point that we measured decreased.

Optimization Steps:

(a) Speed: Our standard/naive matrix multiplication uses ikj index ordering, rather than ijk ordering, as this accesses memory more efficiently and speeds up multiplication.

(b) Space complexity - recursive call allocations:
Each recursive call to strassen() allocates three $\frac{n}{2} \times \frac{n}{2}$ submatrices in order to execute the computations needed to compute the result matrix. We use two matrices as temporary variables to store intermediate calculation steps for subproblems $P_1 - P_7$. After calculating each subproblem P_i, we add/subtract P_i to the result quadrants that use P_i, and then can discard the value of P_i and use the matrix previously storing P_i to compute P_{i+1}, etc.

(c) Handling $n \times n$ matrices where n is not a power of 2:
We initially handled calls to strassen for $n \times n$ matrices where n is not a power of 2 by resizing the input matrix to the next highest power of 2, and padding the new matrix values with zeros. This obtained the correct result, but was unnecessarily space-intensive, because it required us to allocate and operate upon matrices with $n^2 <$ number of elements $< 2n^2$, rather than matrices of size n^2. In our final implementation of strassens, we do not allocate any additional zeros for our execution of strassen(), regardless of the size of the input matrix. Let n' be the next highest power of 2 for n. Our call to strassen() divides up the matrix in recursive calls as if our matrix were an $n' \times n'$ matrix. When the functions add_matrices(), subtract_matrices(), and standard_matrix_multiplication() are called, they perform calculations on the given matrices using the allocated known size of the input vectors, and for any index values beyond the bounds of the matrix, the value is simply not computed.
E.g. given an input 17×17 matrix A, strassen would be called recursively on what strassen treats as four 16×16 matrices. However, the recursive call on the fourth quadrant of A would only execute calculations on 1 value. Similarly, the recursive calls on the second and third quandrants of A would only execute calculations on 16 values.
We accomplish this using careful indexing in the for-loops used for addition/subtraction/multiplication. To demonstrate this indexing, the functions for addition (subtraction is similar) and multiplication are shown for reference below. Each call to the standard multiplication/addition/subtraction algorithms must also temporarily store the size of the two input vectors and the result vector in order to achieve this indexing, but this does not affect the asymptotic space complexity of our strassen implementation. This optimization reduces the time and space required by our algorithm by avoiding unnecessary computations involving zeros.

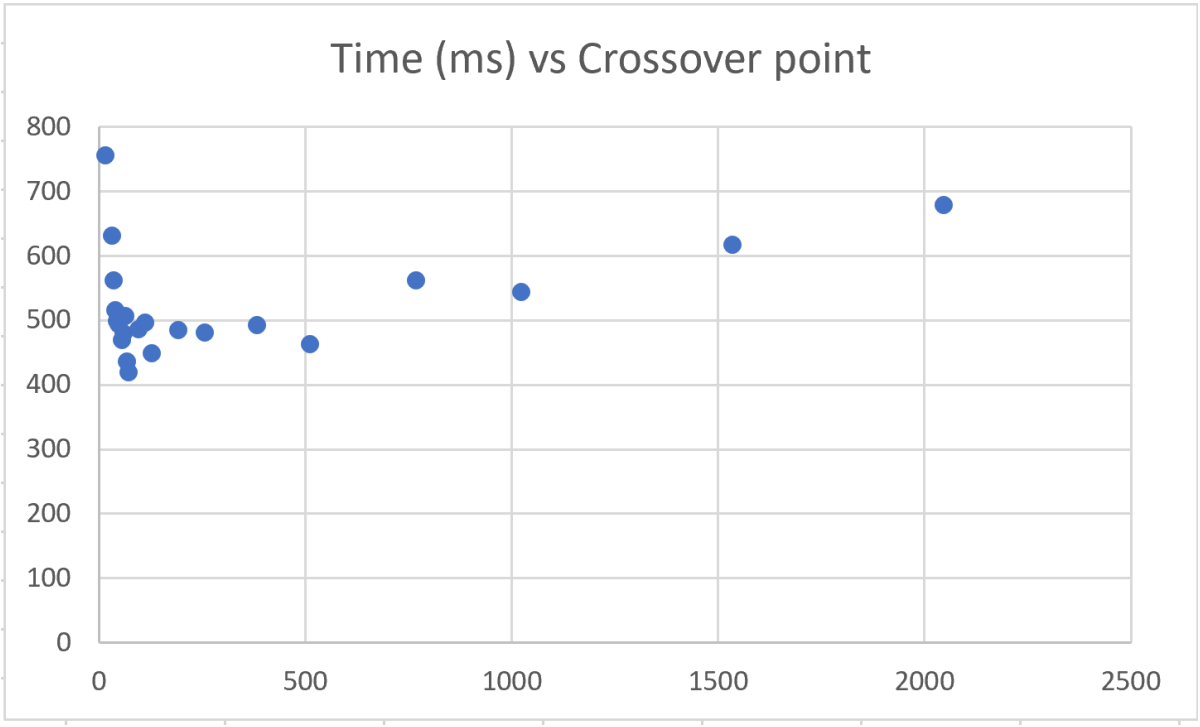
```
void standard_matrix_multiplication(vector< vector<long long int> > &arr1,
vector< vector<long long int> > &arr2, vector< vector<long long int> > &res, int n,
int r1, int c1, int r2, int c2, int r3, int c3) {
// Multiply arr1 and arr2 and store result
// Use i k j ordering to speedup multiplication
int size1 = arr1.size();
int size2 = arr2.size();
for(int i = 0; i < min(size1-r1,n); ++i)
    for (int k = 0; k < min(size1-c1,min(size2-r2,n)); ++k)
        for(int j = 0; j < min(size2-c2,n); ++j)
            res[i+r3][j+c3] += arr1[i+r1][k+c1] * arr2[k+r2][j+c2];

void add_matrices(vector< vector<long long int> >
&arr1, vector< vector<long long int> > &arr2, vector< vector<long long int> >
```

```
&res, int s, int r1, int c1, int r2, int c2, int r3, int c3) {
    // Add arr1 and arr2 and store result
    int size1 = arr1.size();
    int size2 = arr2.size();
    int sizeRes = res.size();
    for(int i = 0; i < min(sizeRes-r3,s); ++i) {
        for(int j = 0; j < min(sizeRes-c3,s); ++j) {
            res[i+r3][j+c3] = ((i+r1<size1 && j+c1<size1) ? arr1[i+r1][j+c1] : 0) +
            ((i+r2<size2 && j+c2<size2) ? arr2[i+r2][j+c2] : 0);
        }
    }
}
```

Experimentally Optimizing for the Crossover Point

We measured the crossover point by timing, for each potential crossover point k, the time (in milliseconds) required to run strassen's algorithm using that crossover point. We tested for $k \in \{16, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 96, 112, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048\}$. For each k, we timed k when run on 1024×1024 matrices, and we averaged the times over 10 trials for each value of k. We graphed the average time vs crossover point below. We obtained a crossover value $n_0 = 68$, such that strassen() ran the fastest on average when using a crossover value of 68.

We also ran the same tests for 2048×2048 matrices, but obtained similar results for the crossover point.



Difference Between Experimental and Theoretical Crossover Point

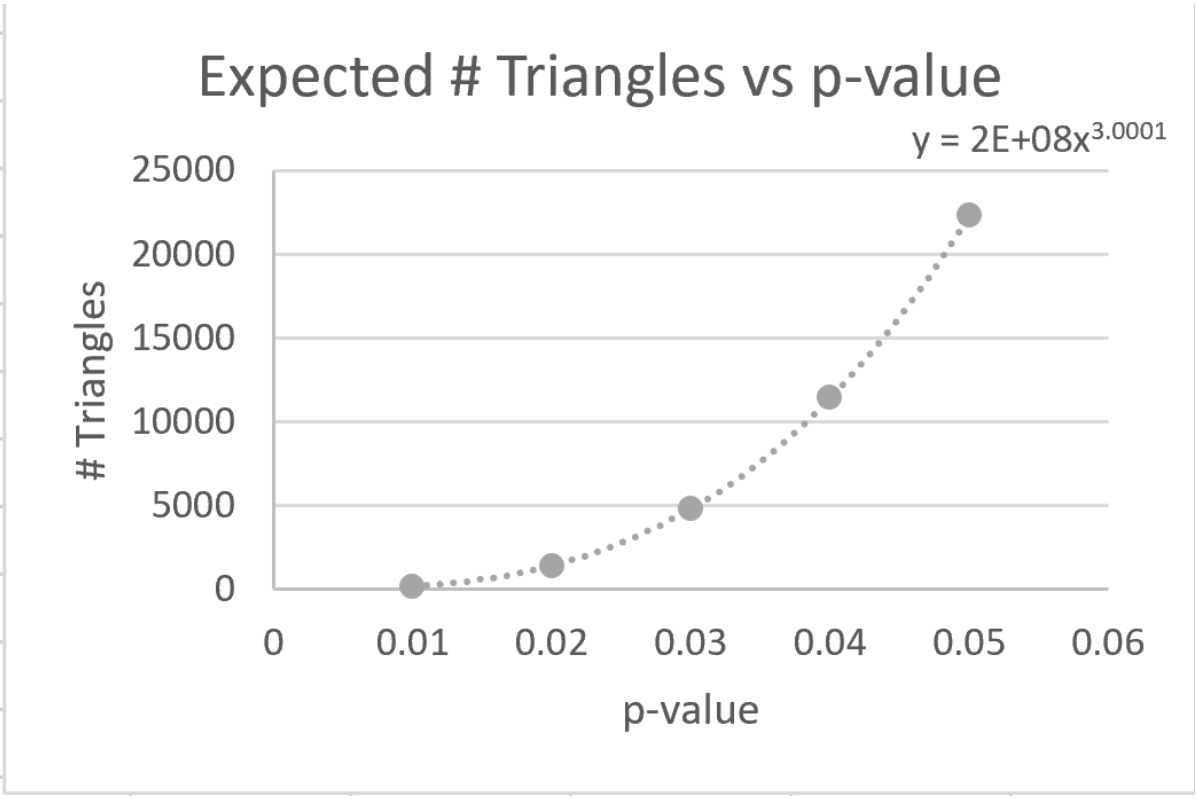
Our experimental crossover point of 68 is greater than the analytical crossover point of 38 (for the odd case that we found in task one).

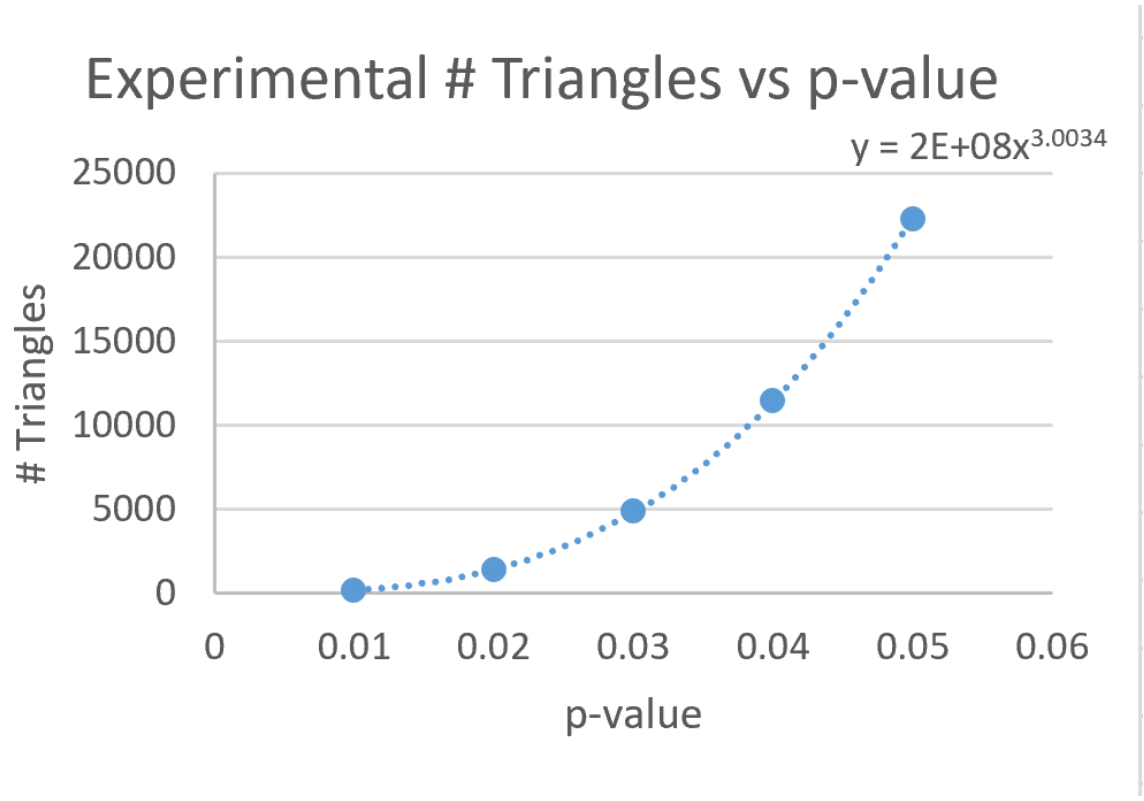
In our theoretical analysis, we assume that only addition/subtraction/multiplication/division between real numbers take time. Our theoretical analysis does not account for memory allocation. In our experimental approach, we use three submatrices for Strassen's algorithm and the space allocation takes time which is not accounted for in the theoretical Strassen's algorithm. Thus, the experimental Strassen's takes longer, contributing to a larger crossover point.

Another assumption in our theoretical analysis is that the arithmetic operations take constant time, however actual implementation of multiplication between large numbers is not constant time. This also contributes to the experimental Strassen's having a larger runtime. When Strassen's takes longer, then the conventional multiplication algorithm will be more time effient at larger values of n and the crossover point is then larger than the one found with theoretical analysis.

Task 3:

p	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial Avg	Expected
0.01	180	183	195	169	167	178.8	178.4
0.02	1462	1320	1359	1405	1479	1405	1427.5
0.03	4726	5087	4961	4955	4791	4904	4817
0.04	11403	11191	11672	11384	11697	11469.4	11419
0.05	21830	22372	22200	22332	22605	22267.8	22304





The expected number of triangles, given p, is given by $E(p) = \binom{1024}{3}p^3$. For the values of p that we tested (0.01,0.02,0.03,0.04,0.05), based on the results of the experimental values and the expected values, we can approximate

$$E(p) = 2 \cdot 10^8 x^3$$

Our experimental values closely match the expected values when we average our results over 5 trials.