CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Alex Wilentz, Andrew Lu

Collaborators: N/A (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: 2, 6
No. of late days used after including this pset: 3, 7

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).

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 theoretical, 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.

   Let us analyze the number of operations in the conventional matrix multiplication algorithm. Consider a matrix of size $n_0 \times n_0$. For a given dot product between vectors of size n_0, we make n_0 multiplications and $n_0 - 1$ additions. As we compute each of the n_0^2 elements in the resulting product

with such a dot product, we thus perform $n_0^2(n_0 + n_0 - 1) = 2n_0^3 - n_0^2$ total operations.

For Strassen's algorithm, we divide the matrix into four parts, each of size $n_0/2 \times n_0/2$. Element-wise addition of matrices of this size thus takes $(n_0/2)^2$ operations, as we perform one addition for each of the $(n_0/2)^2$ resulting elements. If we suppose we switch to conventional matrix multiplication for values below n_0, we see that multiplying these matrices takes $2(n_0/2)^3 - (n_0/2)^2$ operations. Strassen's algorithm requires 7 multiplications and 18 additions of these matrices, and thus requires a total of $7 \cdot (2(n_0/2)^3 - (n_0/2)^2) + 18 \cdot (n_0/2)^2 = 14(n_0/2)^3 + 11(n_0/2)^2$ operations.

We thus can solve for the value of n_0 such that $2n_0^3 - n_0^2 = 14(n_0/2)^3 + 11(n_0/2)^2$. This occurs at $n_0 = 15$, and we see that below this point the conventional matrix algorithm requires less operations, whereas above this point Strassen's algorithm requires less operations. Thus, $n_0 = 15$ is a theoretical estimate of the cross-over point between Strassen's algorithm and the standard matrix multiplication algorithm.

However, for our implementation of Strassen's algorithm, we additionally pad matrices with odd dimensions, increasing their dimension by one if it is odd on any given iteration. Thus, if n_0 is odd, Strassen's algorithm now takes $14((n_0 + 1)/2)^3 + 11((n_0 + 1)/2)^2$ operations. Thus, we can solve $2n_0^3 - n_0^2 = 14((n_0+1)/2)^3 + 11((n_0+1)/2)^2$ for n_0 to find the cross-over point when n_0 is odd, which occurs at $n_0 \approx 37.17$.

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.

   To find the optimal cross-over point, we implemented the standard matrix multiplication algorithm and Strassen's algorithm in Python. We made several modifications to Strassen's algorithm, though. Firstly, in order to make Strassen's algorithm work for arrays of all sizes, not just simply powers of 2, at each level or recursion, if the length of the matrix was odd, we padded the matrix with an additional row and column of 0s, in order to make the matrix even. Then, after the completion of all of the multiplications, the padding was removed such that the array would be the original size, and then returned the array. Performing this padding operation leads to a correct outcome because each element in the matrix is a dot product of a row and column, and the dot product of the row and column, with and without padding, is equivalent because 0 is added to the end of the dot product. Since each element is a dot product of the respective rows and columns, the padding has no effect on the total matrix multiplication. This method of padding has the advantage in that though we have to pad at multiple depths, as opposed to once if we simply padded the array to the next largest power of 2, we avoid multiplying matrices that are much larger than our original input. The next change we made was that upon recursion of Strassen's to at most a certain matrix length n_0, we

would then switch to standard matrix multiplication. As such, we could find the optimal cross-over point by testing different values of n_0.

For our testing, we timed Strassen's algorithm on random arrays of lengths 8, 16, 32, 64, 128, 256, and 512, which are simple powers of 2, and then on lengths 9, 17, 33, 65, 129, 257 to test the time cost of padding. For the powers of 2, we tested possible values of n_0: 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512, up to the dimension of the matrix. For the numbers to be padded, we tested possible of n_0: 1, 2, 3, 5, 9, 17, 33, 65, 129, and 257, as these are the values that result from adding one and then halving the size of the matrix. The results can be seen in the below tables:

Dimension	n_0	time (s)
8	1	0.002659797668
8	2	0.0009980201721
8	4	0.0004689693451
8	8	0.0001049041748
16	1	0.02244925499
16	2	0.00500202179
16	4	0.002499103546
16	8	0.001051902771
16	16	0.0007920265198
32	1	0.1465630531
32	2	0.04219579697
32	4	0.01637101173
32	8	0.009093284607
32	16	0.006721973419
32	32	0.005661964417
64	1	0.962389946
64	2	0.2788002491
64	4	0.1169042587
64	8	0.0661239624
64	16	0.04920864105
64	32	0.04403805733
64	64	0.04359793663
128	1	6.682222128
128	2	1.924031258
128	4	0.7930388451
128	8	0.4639072418
128	16	0.3465719223
128	32	0.3056519032
128	64	0.317636013
128	128	0.3447318077
256	1	50.0056138
256	2	14.59077191
256	4	6.003774881
256	8	3.543484926
256	16	2.656284094
256	32	2.298361063
256	64	2.588552952
256	128	2.657334089
256	256	4.569479942
512	1	370.3361506
512	2	99.59448481
512	4	40.13186789
512	8	23.072088
512	16	17.46000361
512	32	15.55762005
512	64	16.75935173
512	128	18.11493897
512	256	22.11420608
512	512	26.1159308

Dimension	n_0	time (s)
9	1	0.0202858448
9	2	0.005891799927
9	3	0.002016067505
9	5	0.000540971756
9	9	0.0001437664032
17	1	0.1529028416
17	2	0.04905128479
17	3	0.01228690147
17	5	0.00354719162
17	9	0.00163936615
17	17	0.0009627342224
33	1	1.059470892
33	2	0.3053939342
33	3	0.0829308033
33	5	0.03048682213
33	9	0.01331400871
33	17	0.008904933929
33	33	0.006870031357
65	1	7.543219805
65	2	2.513921022
65	3	0.5948479176
65	5	0.2011108398
65	9	0.1052350998
65	17	0.0651679039
65	33	0.05140709877
65	65	0.05004501343
129	1	55.26625896
129	2	15.1324079
129	3	3.989215851
129	5	1.457595825
129	9	0.7247359753
129	17	0.4675016403
129	33	0.3845930099
129	65	0.3459641933
129	129	0.3838508129
257	1	371.578815
257	2	108.527364
257	3	28.29938579
257	5	10.23206401
257	9	5.068638086
257	17	3.318114996
257	33	2.75971508
257	65	2.618654966
257	129	3.217173815
257	257	3.195759058

As such, we can find the optimal cross-over point by selecting the value of n_0 that minimizes the time for each dimension. For the powers of 2, the approximate cross-over points are: $n_0 = 8$ for dimension 8, $n_0 = 16$ for dimension 16, $n_0 = 32$ for dimension 32, $n_0 = 64$ for dimension 64, $n_0 = 32$ for dimension 128, $n_0 = 32$ for dimension 256, and $n_0 = 32$ for dimension 512. We see that for arrays of a small enough dimension, the overhead cost of Strassen's is large enough such that we would prefer to simply compute the product via standard matrix multiplication. For the larger matrices, we generally find $n_0 = 32$, which is higher than the expected $n_0 = 15$, which would correspond approximately to our test at $n_0 = 16$.

This is for several reasons. One is that it is likely our Strassen's algorithm has time costs in our allocation of memory and other maintains operations that aren't accounted for in our theoretical approach. Every time a submatrix is created, it requires the allocation and initialization in order to perform the submultiplications. Given $n_0 = 32$ is the next dimension of matrix above our predicted value, it makes sense that these additional time costs would prompt us to switch to conventional matrix multiplication earlier.

For the padded dimensions, the cross-over points are: $n_0 = 9$ for dimension 9, $n_0 = 17$ for dimension 17, $n_0 = 33$ for dimension 33, $n_0 = 65$ for dimension 65, $n_0 = 65$ for dimension 129, and $n_0 = 65$ for dimension 257. Like above, for the smaller matrices, it is better to simply do conventional matrix multiplication due to the overhead costs of Strassen's. However, we do see for the larger matrices, $n_0 = 65$ becomes the optimal cross-over value. Once again, this is higher than our predicted value of $n_0 \approx 37.17$. While this is once again partially due to the additional overhead of Strassen's, a large factor is the time cost of padding our matrices, which is not accounted for in our theoretical approach. As such, given we must incur the time it takes to pad the matrices for Strassen's, the cross-over point is more likely to favor switching to conventional matrix multiplication earlier.

In implementing our Strassen's algorithm and naive algorithm, we performed significant optimizations. For Strassen's, instead of continually initializing and allocating new memory, our algorithm attempts to initialize and define each matrix at most once. For the naive algorithm, we experimented with vectorization by using dot products, but found that the best optimization would be to use python's built in matrix multiplication function.

We also tested on 0/1 matrices vs 0/1/2 matrices, and found that on the whole, it seemed to matter very little whether the inputs were 0/1 or 0/1/2. There was a slight runtime advantage to 0/1 matrices, but it was marginal in that the difference was less than 10%, not anything near an order of magnitude or a doubling.

Ultimately, unfortunately, due to the large computational cost and overhead, as well as the memory usage, Strassen's algorithm in practice does not prove very useful. For example, for dimensions below 32x32, the optimized naive implementation outperforms. Beyond these sizes, matrix multiplications typically occur in large matrices primarily in machine learning, where there are substantially faster optimizations for sparse matrices. Strassen's could be useful to some degree with a few iterations of submatrix multiplication, but certainly not the full recursive algorithm. As a fun fact, DeepMind's AlphaTensor has developed an algorithm faster than Strassen's, so it may be true that Strassen's is literally going to be discarded and useless. Just kidding, but the point stands that even in practice, there may be more optimized algorithms for matrix multiplication that emerge.

One observation and important note from the results is while the number of matrices for multiplication increases by a factor of 4 for every halving of n_0, as we observe, such as $n_0 = 1$ vs $n_0 = 2$ with dimension 512, that the time it takes is smaller than a factor of 4, and further, that looking toward larger values of n_0, the runtimes decrease by smaller factors. This is likely caused by two phenomena, one being that naive matrix multiplications on larger matrices correspond to longer runtimes and another being that optimized naive multiplications scale better than the generic $O(n^2)$ runtime. This phenomenon occurs across the board and is interesting to explore as part of the outcomes and results of our analysis.

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 Strassen's algorithm, we calculated and averaged the number of triangles on a random graph of 1024 vertices over 5 trials. Our results, in which we see that our calculated amounts of triangles are similar to the expected amounts (and only differ by approximately 2%), are summarized in a table below.

p	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Expected Edges
0.01	146	170	190	195	172	174.6	178.433024
0.02	1434	1471	1499	1466	1460	1466	1427.464192
0.03	4984	4901	4704	4921	4878	4877.6	4817.691648
0.04	11507	11336	11515	11794	11099	11450.2	11419.71353
0.05	22404	22067	22533	23301	22376	22536.2	22304.128