CS 124 — Programming Assignment #2
60977623, 50983552

In this experiment we examined the optimal crossover point (the point at which one would switch during matrix multiplication) from Strassen's algorithm to the standard matrix multiplication algorithm. First we found a theoretical estimate for this crossover point using the time recurrences for the algorithms. Since this estimate did not take into account memory management, we also found a crossover point experimentally, by implementing both Strassen's algorithm and the standard algorithm, then comparing runtimes for various matrix sizes and crossover points candidates. As expected, our experimental crossover point was higher than our analytical prediction.

After implementing our first working version of Strassen's for matrix dimensions in powers of 2 we expanded the function to work for other (i.e. odd) dimensions using padding, and then improved upon that function by minimizing memory allocation and deallocation, and avoiding copying large blocks of data. In implementing the standard matrix multiplication algorithm we started out with a function that mimicked the steps one might take when multiplying two matrices by hand, and then made it faster by minimizing costly row switches (they are more often cache misses) within our for loops.

PART 1: ESTIMATING OPTIMAL CROSSOVER ANALYTICALLY

Here we assume that integer adding, subtracting, multiplying, and dividing each have time cost 1. For the standard matrix multiplication algorithm, then, we have runtime D(n) = 2n^3 - n^2, where n is the dimension of the matrices being multiplied, as D(n) is the sum of the n^3 multiplications (n multiplications for each of the n^2 entries in the product matrix) and (n-1)*n^2 additions (n-1 additions for each of the n^2 entries in the product matrix) involved in the algorithm.

For Strassen's algorithm we get runtime E(n) = 7*E(⌈n/2⌉) + 18(⌈n/2⌉)^2 since a multiplication of two n*n matrices requires 7 multiplications of ⌈n/2⌉*⌈n/2⌉ matrices (the ceiling function accounts for padding in the event of odd-dimensioned matrices) and 18 additions of ⌈n/2⌉*⌈n/2⌉ matrices, which gives 18(⌈n/2⌉)^2 integer matrices.

To estimate the optimal crossover point from Strassen's algorithm to standard matrix multiplication, we must find n' such that the runtime D(n') of standard matrix multiplication used alone is equal to the runtime F(n') of a hybrid of Strassen's and the standard algorithm that starts by running Strassen's (just once) but does all of its sub-multiplications using the standard algorithm. This n', when it is the first value of n for which we start using the standard algorithm after using Strassen's from the given dimension down, will give us an optimal runtime because for any higher n', it will have been faster to keep using Strassen's a bit longer (for *that* n', D(n') will be greater than F(n'), in which we use Strassen's one more time), and for any lower n', it will have been faster to start using the standard algorithm a bit earlier (for *that* n', D(n') will be less than F(n'), in which we use Strassen's one too many times). Thus we look for the point at which the two runtimes are equal.

We can define F(n) in terms of D(n) and E(n) since it runs Strassen's on the outside to wrap up all its sub-multiplications, which are done using the standard algorithm. That gives us F(n) = 7*D(⌈n/2⌉) + 18(⌈n/2⌉)^2.

Solving for our n' by setting D(n') = F(n'), we get D(n') = 7*D(⌈n'/2⌉) + 18(⌈n'/2⌉)^2, which gives us n' = 15 in the case of even n's and n' = about 37 in the case of odd n's.

PART 2: OPTIMAL CROSSOVER IN OUR EXPERIMENT

To determine optimal crossover experimentally, we looked at a range of 8 dimensions (from 720 down to 713) as a way to account for all of the different possible occurrences of padding for odd dimensions. For each dimension n in this range, we knew that the optimal crossover point would be at some division of n by two. For example, we knew that the optimal crossover point for dimension 720 would be 360, 180, 90, 45, or 23 and so on. In the case of odd numbers, we rounded up to reflect dimension added by padding, e.g. from 45/2 = 22.5 to 23. For another example, we tried crossover points 357, 179, 90, 45, and 23 for dimension 713. Here we came up with these crossover candidates by dividing 713 by 2, rounding up, dividing the result by 2, rounding up, and so on, rounding up whenever necessary. Here we were able to exclude other crossover point candidates and look only at divisions by two of n because we knew that

recursive calls to the modified Strassen's function would only be made on matrices of size ⌈n/2⌉*⌈n/2⌉ or ⌈n/4⌉*⌈n/4⌉ or ⌈n/8⌉*⌈n/8⌉ and so on.

For each dimension (e.g. 720) and each crossover point candidate (e.g. 360) we ran ten trials -- in each of these trials we randomly generated a matrix A and matrix B by randomly picking each entry from somewhere within an input text file. After recording each trial's runtime for multiplying A and B using Strassen's down to the crossover point and using standard multiplication from the crossover point on, we averaged the ten runtimes for that crossover point candidate. The crossover point candidate with the minimum average runtime over its ten trials among all crossover point runtimes was considered the optimal crossover point for that particular dimension.

To find one optimal crossover point from all the dimensions we tried, we took the average of the optimal crossover points across all dimensions.

See the table below for the average runtimes in seconds over ten trials for each dimension n (rows) and cutoff candidate (columns). Minimum runtimes for each dimension are highlighted.

n / cutoff	12	23	45	90	180	179	360	359	358	357
720	2.57091	1.497103	1.192206	1.134244	1.945624		2.128576			
719	2.562879	1.475109	1.259583	1.970571	1.944379		2.080157			
718	2.582076	1.488203	1.595019	1.572404	1.990327			2.058305		
717	2.538005	1.498626	2.335556	2.064268	2.195399			2.057431		
716	2.496996	1.497052	2.046714	2.065752		1.92964			2.04545	
715	2.545483	1.482683	1.222612	2.087514		1.922049			2.039	
714	2.513959	1.455231	1.858667	2.02585		1.93936				2.025495
713	2.58726	1.45785	1.388255	2.011092		1.930443				2.029098

The weighted average of optimal crossover points over dimensions 720 down through 713 was 39.625, which gave us a crossover point of about 40.

PART 3: OUR IMPLEMENTATIONS AND EXPERIMENT DESIGN

Our final implementation of Strassen's algorithm initialized as few matrices (represented as arrays of arrays) as possible. This version was the result of multiple iterations, decisions we made as we tried to speed our process up. We outline those choices starting from section 3.1 below.

3.1 On a Cache-Efficient Conventional Matrix Multiplication Algorithm

To make runtimes faster overall regardless of cross-over point, we optimized the standard matrix multiplication algorithm to make as much use of caching as possible. Since we had implemented our matrices as arrays (representing rows) of arrays (representing columns), it would cost less to switch between entries in different columns of a single row (these entries were closer in memory, being in the same array, and thus more likely to be cached together) than to switch between entries in different rows of a single column. Knowing that we needed to minimize row switches, we modified our original standard multiplication function so that the innermost for loops (the ones that would run most often) involved column switches and only the outermost for loops (the ones that would run fewer times) involved row switches.

Usually when we multiply matrices by hand in math class, we find each entry of the product matrix one by one, going from left to right and top to bottom. The problem with this approach is that every time we sum up the appropriate products of entries of some A and B to find one entry of their product matrix C, while we traverse a single row of A, and a single column of B, we end up switching between all of the rows of B. An alternative approach would be to use all of the entries in a given row of A *and* all of the entries in a given row of B that will be multiplied with entries of this particular row of A first, before moving on to the next row of B, even if this involves starting on more entries of C before we have finished computing the first one. This potential problem of having multiple entries of C being computed at once is neutralized by the fact that C is in this case a blank array full of variables that are being modified anyway as a part of the multiplication and can be used for temporary storage. Any additional traversals of C involved in this second approach are less costly column switches, as we are only ever computing a single row of C at once.

Both of these approaches require (n-1) row switches within A, but while our grade school approach involves (n^3 - 1) row switches within B (about one switch every time we compute another product of entries), our new one involves only (n^2 - 1), or about one row switch for each entry of A that has to be multiplied by the entries in a new row of B.

Our implementation of the second approach brought times down across the board.

3.2 On Avoiding Excessive Memory Allocation and Deallocation

One of the major design decisions we made to speed up our implementation of Strassen's algorithm was to minimize the amount of memory we dynamically allocated and deallocated. We took steps at every level of our program to achieve this minimization. Much of this effort rested on using pointers (we used C) to repeatedly update existing blocks of memory instead of always generating, returning, and freeing new blocks of memory. For example, our functions to add and subtract matrices takes in three matrix arguments: two matrices to add/subtract together, and one matrix to store the result. Most of the times that the matrix_add and matrix_subtract functions are called, we recycle a pre-existing matrix as the third argument and we simply write over the old data. The functions simply update the third argument matrix so we avoid generating (allocating more memory for) and returning a new matrix each time they are called. We use this same method in our conventional matrix multiplication function - simply updating the recycled third argument matrix instead of generating (allocating more memory for) new matrices each time.

Another design decision that sped up our implementation of Strassen's algorithm significantly was recycling and updating the same set of matrices (p1, p2, p3, p4, p5, p6, p7) to hold the 7 products for each level of recursion. Initially, we allocated and deallocated memory for 7 new matrices of dimensions (n/2) x (n/2) at each level of recursion for our Strassen function. However, once we realized this mistake, we decided to create a set of 15 matrices of dimensions N x N (the size of the original two matrices to be multiplied) upon initiation of our program and then pass them through to our Strassen function at each level of recursion so that

they could be repeatedly recycled and written over. This saved a lot of unnecessary memory allocations and deallocations.

A significant part of our memory design decisions were concerning how to break up a matrix into "quadrants" and then join those "quadrants" back together in our implementation of Strassen's algorithm. In our initial design, at each level of recursion for our Strassen's multiplication function, we allocated and deallocated memory for 4 new submatrices to simultaneously store each quadrant of the larger result matrix is returned. Once we realized that this was a waste of time and memory, we decided to just generate one "helper" matrix the size of the original matrix at the start of the program to be repeatedly passed into our Strassen's multiplication function each time it was called. We recycle this matrix, rewriting over it again and again with data from different quadrants of the different (larger and larger) matrices. We also use the same "helper" matrix to store each of the 4 submatrices, one at a time. We do this by updating the larger matrix one quadrant at a time instead of storing all 4 quadrants in 4 different helper matrices simultaneously and joining them all together at once. Also, our functions to split up a matrix into quadrants and then merge quadrants into a larger matrix uses pointers to update existing matrices instead of allocating memory for and returning new matrices each time they are called.

We also saved unnecessary memory allocation and deallocation through our padding scheme for matrices of odd-dimensions in our implementation of Strassen's algorithm. We explain this design further in depth in *Section 3.4.1: On Padding for Odd-Dimension Matrices*.

3.3 On Avoiding Copying Large Blocks of Data Unnecessarily

As described in *Section 3.2*, we use pointers in all of our "helper" functions (add, subtract, multiply, split, join) on matrices to directly populate or update existing matrices instead of generating and returning new matrices. This allows us to avoid copying the data from the new matrices returned by a "helper" function into the matrices we are actually using in the Strassen multiplication function.

We also avoided a lot of data copying through our padding scheme for matrices of odd-dimensions in our implementation of Strassen's algorithm. We explain this design further in depth in *Section 3.4.1: On Padding for Odd-Dimension Matrices*.

3.4.1 On Padding for Odd-Dimension Matrices

At each level of recursion, Strassen's algorithm splits up the two matrices to be multiplied into 4 submatrices of equal dimensions - "quadrants". So for Strassen's algorithm divides an NxN matrix into 4 submatrices of dimensions (N/2)x(N/2). In the case where a matrix has odd dimensions, we must manipulate the matrix so that it can be divided evenly. We decided to conceptually pad odd matrices with an extra (bottom) row and (right) column of zeros, so the dimensions of our theoretically augmented NxN matrix becomes (N+1)x(N+1), which can then be evenly divided into quadrants with dimensions (N+1)/2 x (N+1)/2.

However, allocating and deallocating memory for whole new augmented (N+1)x(N+1) matrix every time we come across an odd-dimension matrix uses unnecessary time and space. Moreover, copying the odd-dimension NxN matrix into a new (N+1)x(N+1) matrix wastes even more time. Therefore, we decided to implement our "padding" through our "split" functions (dividing the larger matrix into four quadrants). We had a specific split function for each quadrant (1,1) (1,2), (2,1), (2,2) of a matrix, and the process for populating each quadrant was unique and dependent on whether or dimension of the larger matrix was odd or even. We directly padded each quadrant individually when we populated it with values in order to avoid any unnecessary copying of data.

For each iteration of our Strassen's function, we checked initially if the matrices to be multiplied were of odd or even dimensions. If they were of even dimensions, we didn't do anything special and simply allocated memory for 4 quadrants of size (N/2)x(N/2). If they were of odd dimensions, however, we allocated memory for 4 submatrices of dimensions (N+1)/2 x (N+1)/2. For quadrant (1,1), we populated it with the data in the first (N+1)/2 rows and (N+1)/2 columns of the larger matrix - filling it entirely with values from the larger matrix. For quadrant (1,2), we populated the first (N+1)/2 rows and ((N+1)/2 - 1) columns with the data from rows 1 to (N+1)/2 and columns (N+1)/2 to N of the larger matrix. Then we filled the last row of quadrant (1,2) entirely with 0s. For quadrant (2,1), we populated the first ((N+1)/2 - 1) rows and (N+1)/2 columns with data from rows (N+1)/2 to N and columns 1 to (N+1)/2 of the larger matrix. Then we filled the last row of quadrant (2,1) entirely with 0s. For quadrant (2,2), we populated the first ((N+1)/2 - 1) rows and ((N+1)/2 - 1) columns with data from rows (N+1)/2 to N and columns (N+1)/2 to N of the larger matrix. Then we filled the last row and last column of quadrant (2,2) entirely with 0s. The figure below is a visual of our design.

Example of a padded 7x7 matrix

X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
X	X	X	X	X	X	X	0
0	0	0	0	0	0	0	0

Our function to merge the padded quadrants into the larger parent matrix, was, in effect, a reversal of the split function. Our merge function used if-statements so that the method of copying the quadrant into the larger parent matrix was specific to the quadrant ((1,1) (1,2), (2,1), or (2,2)). Again, this allowed us to avoid unnecessary memory allocation and deallocation or copying large blocks of data.

3.4.2 On Padding for Matrices whose Dimension is not a Power of 2

Not every pair of even-dimensional matrices to be multiplied by Strassen's algorithm will be of a dimension NxN where N power of 2, so that it can be evenly divided into N/2 x N/2 - dimensional "quadrants" at every level of recursion. For example, a 6x6 matrix is an even-dimensional matrix and will work smoothly on the first level of recursion. However, N/2 is 6/2 = 3, which is odd-dimensional, so when our Strassen's is called on each N/2xN/2 quadrant, we will have to pad the quadrants. Therefore, we had to check at each level of recursion of our implementation of Strassen's algorithm whether or not the matrices to be multiplied were of even or odd-dimension. If they were of odd-dimensions, we padded the two matrices as described in *Section 3.4.1*.

IN SUM: One broad lesson that we learned in implementing and optimizing both of our multiplication algorithms is that memory management is crucial in runtime. In CS50 we had always heard that some solutions would take less time but more space, or more time but less space -- that there was a tradeoff -- but this time around, solutions that took less space often took less time.