CS124 Programming Assignment 2: Matrix Multiplication and Strassen's Algorithm.
HUID: 40907009

I implemented both matrix multiplication algorithms in C. The most important part of my implementation is the function `addMat`, which takes in 9 arguments. These arguments are as follows: 1 is an integer `flag` that determines whether I want to add or subtract the matrices. 2 and 3 are two integers `dim1` and `dim2` that determine the size of the first matrix, which is added to or subtracted from the second matrix in place (so both matrices must be larger than or equal to `dim1 * dim2`). 4-7 are integers `i1`, `j1`, `i2`, and `j2`, which indicate the (i, j) indices to start from in matrices 1 and 2, respectively. Finally, 8 and 9 are int**s that define the two matrices.

For example, `addMat(0, 3, 2, 3, 2, 0, 0, M1, M2)` would *add* the entries from $i = 3$ to $i = 6$ and $j = 2$ to $j = 4$ from matrix M1 to the entries from $i = 0$ to $i = 3$ and $j = 0$ to $j = 2$ in M2, and return the modified M2 (transforming M2 permanently, rather than allocating and copying).

This particular way of implementing addition helped in several ways. First, it does not allocate any new memory to store the result. Second, it allows me to reference submatrices without actually defining them or allocating space for them, by simply defining their dimensions and starting points when adding them. Third, it makes it easy to pad and unpad with zeroes: adding a smaller matrix at position $(0, 0)$ in a larger matrix defined with `calloc()` returns a zero-padded matrix of the larger size. Unpadding is the opposite: define dimensions in M1 that exclude the padding when adding them into the result matrix, and the padding is not carried over.

With this function defined, Strassen's algorithm is straightforward, if a bit tedious. I first find dimensions `div1` and `div2`, which are equal if `dim` is even, and the former is greater by 1 if odd. I then allocate two `div1` * `div1` temporary matrices, to which I add or subtract the appropriate submatrices for each $M1...M7$, then recursively multiply the two tmps. When adding submatrices, I simply set the dimensions and offsets appropriately to grab the right size and location. For example, to set `tmp1` to $-A_{12}$, I would call `matAdd(1, div1, div2, 0, div1, 0, 0, A, tmp1)`, because A_{12} has size `div1` * `div2` and starts at index $(0, div1)$ in A, and i want to place it at the start of `tmp1`, which will result in a column of zeros at the end. After doing this for all $M1...M7$ (resetting the temps to all 0 in between), I add the appropriate matrices to the result matrix C, taking care to insert them starting from the correct index (e.g. $(div1, div1)$ for C_{22}) and with the appropriate dimensions to drop the padding (e.g. when adding $M1$ to C_{22}, the dimensions are `div2` * `div2`). In this way, I avoid padding up to the nearest power of 2, and simply pad up to the next even number and drop padding at each step. This saves time and space.

I tested the correctness of my algorithm by simply writing a function that asserted that all entries of the matrices produced by each algorithm were equal.

To find the crossover point, I defined my `strassen` function with an argument `n0`, such that if the dimension was equal to or below `n0` it would just return the naive matrix product. I then ran `strassen` within a for loop with increasing values of `n0` and timed each one to find the minimum.

I wrote a generator program to create $n * n$ matrices that randomly set each element to 0, 1, or 2. Since these numbers are relatively small, multiplying them is quick relative to very large numbers, so saving a certain number of multiplications via Strassen's algorithm saves less time than if they were very large.

I found (based on 5 trials each over values of n_0 from 40 to 220, with n = [500, 1000, 2000, 3000]) an experimental crossover point of $n_0 = 110$