Rebecca Lee
CS3310.01
October 15, 2021

Project 1

PROBLEM 1

My approach to the merge sort algorithm was similar to the pseudocode shown in class, but with some inputs and method parameters changed, and a while loop instead of an if loop at the end of the merge function when adding the remaining values to the "main" array. The merge sort algorithm recursively splits the given array in half until it is an array of size 1, then runs a merge method to iterate through both lists and add the smaller of the two values to the main array. I encountered difficulties splitting the array (my first attempts resulted in the second to last number entered twice and overwriting the last number) and failing to trigger the if loop. The first problem was linked to the way I was cutting the arrays; I was using Python's slice function incorrectly. I was unable to resolve the if loop, and opted to use a while loop instead. To be more accurate, I ran the program 3 times and used their average for the data point. 20,000,000 values takes 6 minutes for the mergesort algorithm to run, not including the time it takes to populate the array. Values over 20,000,000 have been omitted because in total they take more than 10 minutes to run.

Merge Sort				
Amount of Values Sorted	Trial 1	Trail 2	Trial 3	Average Time Taken
10,000	0.054	0.057	0.060	0.057
20,000	0.085	0.083	0.103	0.090
50,000	0.213	0.231	0.221	0.221
80,000	0.492	0.373	0.401	0.422
90,000	0.557	0.465	0.429	0.484
100,000	0.515	0.453	0.460	0.476
200,000	1.138	0.941	0.976	1.018
500,000	3.573	3.031	3.156	3.253
1,000,000	12.619	12.188	11.881	12.229
2,000,000	28.228	29.172	29.711	29.037
5,000,000	84.470	84.340	83.695	84.168
10,000,000	173.609	161.254	176.834	170.565

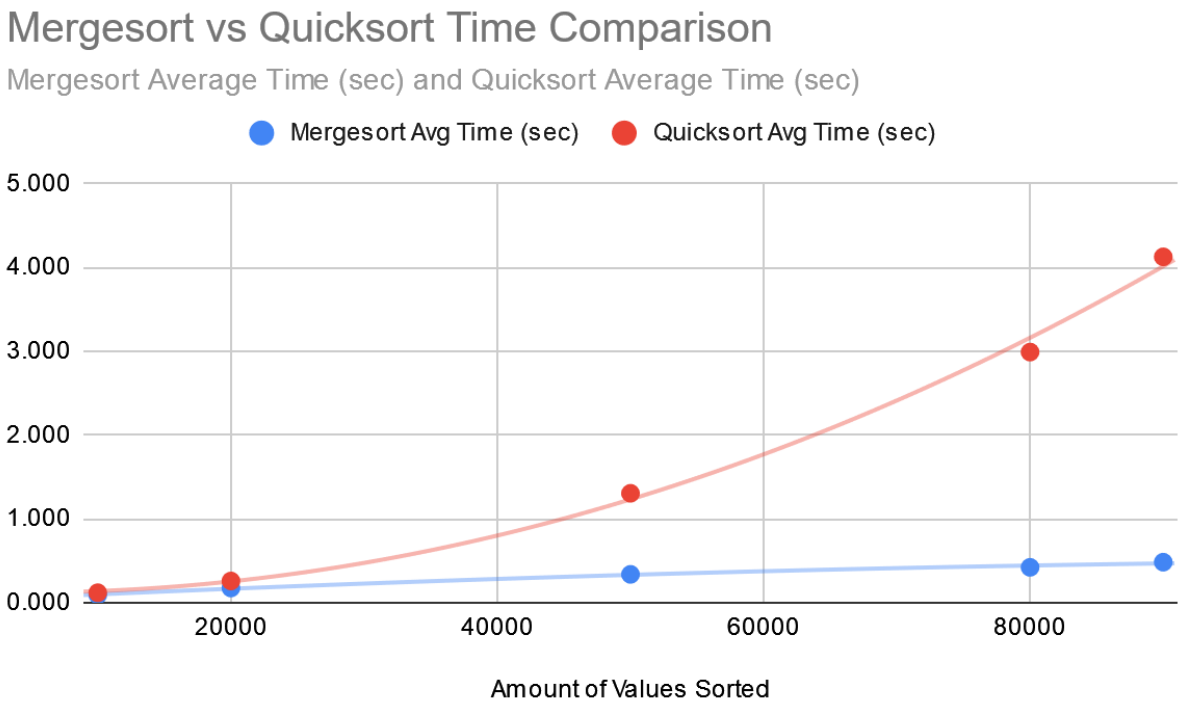
I employed a similar strategy for quick sort, but ran into an issue with using the variable pivotpoint as a global variable. I had the partition function return a value, j, with the location of the pivot point, and used that as the variable ppoint in my quicksort calls. I had an issue with middle values not sorting that was quickly resolved (I forgot to include the partition in the sort). To match the accuracy of merge sort, I also ran the quicksort program 3 times and used their average for a data point. Unfortunately, my computer couldn't handle 100,000 values ("maximum recursion depth exceeded"), so I added data points 80,000 and 90,000 for more points.

Quick Sort				
Amount of Values Sorted	Trial 1	Trail 2	Trial 3	Average Time Taken
10,000	0.084	0.122	0.153	0.119
20,000	0.219	0.349	0.210	0.259
50,000	1.188	1.381	1.348	1.306
80,000	2.902	3.101	2.969	2.990
90,000	3.902	3.373	5.104	4.126

Both algorithms had their timers start after the creation of the arrays they would organize. I set up a random number generator in a for loop that generated numbers 0 to 100. The algorithms were given different sets of numbers to work with.

Based on my implementation, the merge sort algorithm seems to be the better of the two. In the cases I was able to run for both algorithms, mergesort consistently had shorter times, and appears to take up less space. The difference in times is expected, as the time complexity for mergesort is always O(n lgn) (for best, average, worst case) while quicksort has time complexities of O(n lgn) for best and average and $O(n^2)$ for worst-case.

Amount of Values Sorted	Mergesort Average Time (sec)	Quicksort Average Time (sec)
10000	0.092	0.119
20000	0.176	0.259
50000	0.338	1.306
80000	0.422	2.990
90000	0.484	4.126



PROBLEM 2

I had a difficult time implementing the Tower of Hanoi algorithm because I kept trying to implement some form of scoring or quantification. I had to do some research to figure out what exactly was going on in the problem. The recursion moves the disks out of the way, allowing the current disk to move. For the data collection, I only ran the program once and recorded the results. Since there is no randomness/random numbers associated with the Towers of Hanoi, the resulting time should always be the same.

The recurrence relation for Tower of Hanoi is **T(n) = 2(n-1) + 1**. The 2(n-1) is from the two recursive calls made in the function itself, while the + 1 is from moving the current disk.

The time complexity can be found using the Expand-Guess-Verify method:

$T(1) = 1$; $T(n) = 2(n-1) + 1$

Expand

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

$= 2[2(n-2) + 1] + 1$

$= 2^2[2(n-3) + 1] + 1 + 1$

$= 2^3[2(n-4) + 1] + 1 + 1 + 1$

Guess

After k such expansions, the equation has the form

$T(n) = 2^k T(n - k) + k$

It stops when $n - k = 1$, that is when $k = n - 1$

$T(n) = 2^{n-1} T(n - (n - 1)) + n - 1 = 2^{n-1} T(n - n + 1) + n - 1$

$= 2^{n-1} T(1) + n - 1 = 2^{n-1} - 1 + n$

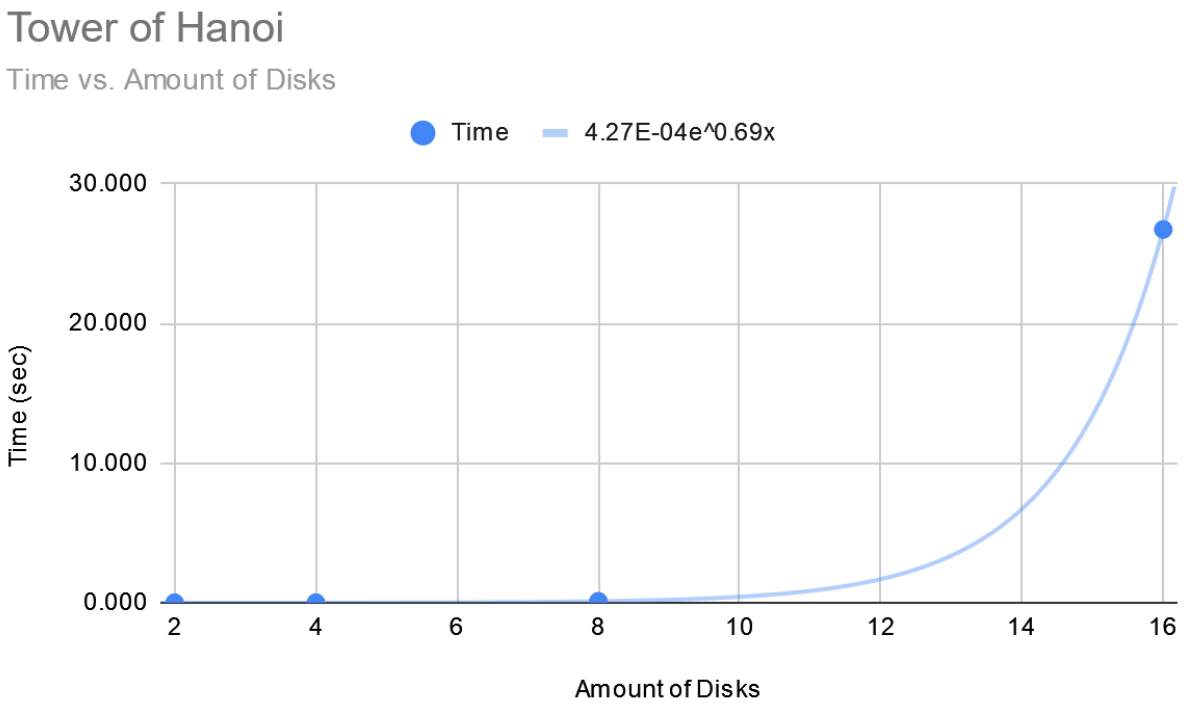
Verify: proof by induction

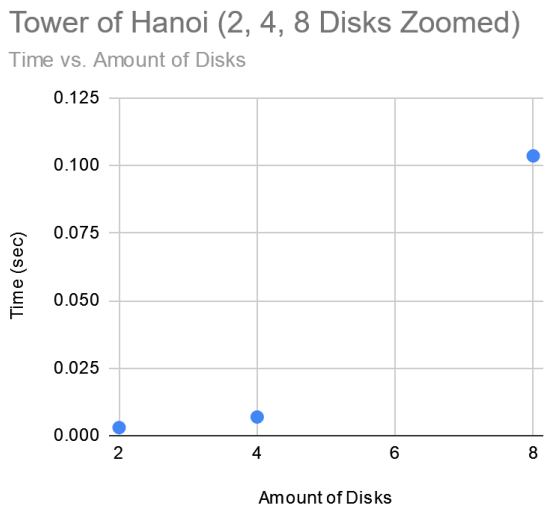
$T(1) = 2^{1-1} - 1 + 1 = 1 - 1 + 1 = 1$ True

$T(k + 1) = 2^{(k+1)-1} - 1 + k + 1 = 2^k + k$

$T(k + 1) = 2T(k) + 1 = 2(2^{k-1} - 1 + k) + 1 = 2^k - 2 + k + 1 = 2^k + k - 1$

Amount of Values Sorted	Time
2	0.003
4	0.007
8	0.104
16	26.719
32	Over 10 minutes





PROBLEM 3

Similar to merge sort, my approach to programming classical matrix multiplication was to use the pseudocode provided in class as a base and change it to fit Python. I encountered a bit of difficulty creating the matrix, as this was my first time doing so on Python. I solved this by creating lists a1, b1, and c1 to represent a row, then appending these rows to the list a, b, and c, since a list of lists is a matrix. Generating the matrices took time, as the computer had to iterate through every value and assign values to 3 matrices (generate random values for a and b, and assign 0 for c), so I started the timer after their creation. The largest matrix my computer could handle in 10 minutes was 1024x1024. Due to time constraints, I only tested this algorithm once and assumed the time I got was an accurate representation for the size.

In the Strassen method of matrix multiplication, I had trouble with the matrix operations. I made separate functions for the operations so I could more easily subtract, add, etc. I reused the same code from the classical matrix multiplication to generate the random matrices. The Strassen method for 1024x1024 matrices took more than 10 minutes and had to be terminated.

Both matrices are made up of random numbers from 0 to 10 because smaller numbers are easier to calculate.

I found the Strassen method harder to implement, and appeared to take more time to execute than the classical method. Theoretically, Strassen's method is O(n log7), which is faster than classical's $O(n^3)$, but Strassen's constant creation of matrices inflates the time complexity to be larger.

Matrix Size (one side)	Classical Matrix Multiplication	Strassen Matrix Multiplication
2	0.000	0.000

4	0.000	0.001
8	0.000	0.004
16	0.000	0.024
32	0.008	0.220
64	0.068	1.440
128	0.354	5.945
256	4.219	45.014
512	28.615	179.729
1024	505.276	more than 10 minutes