Recursive Programming

Dr. John Matta

Ryan Bender

CS 340 – 002

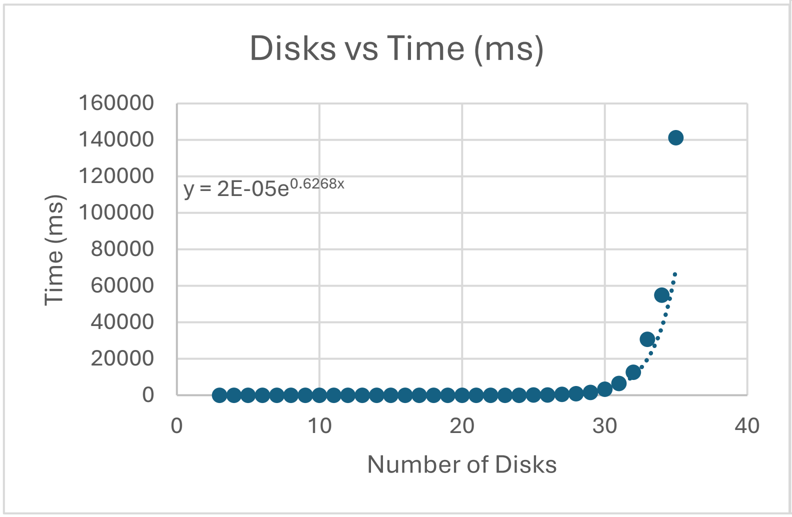
Preview:

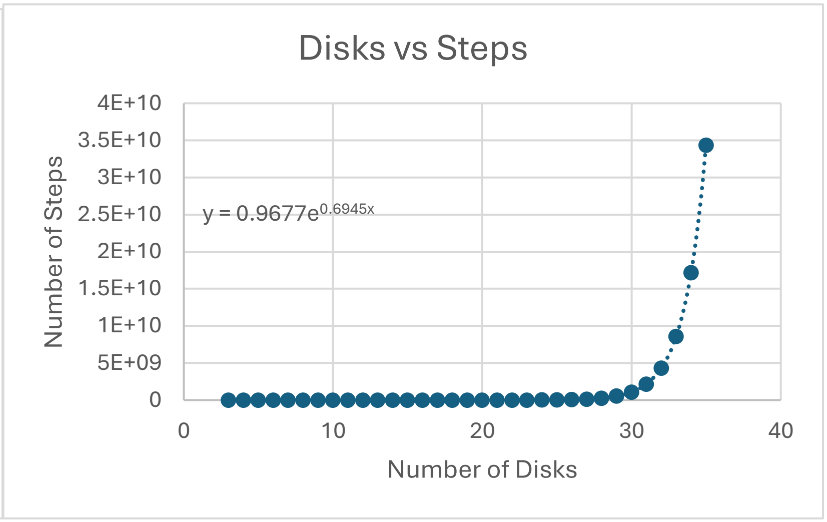
The purpose of this project is to practice recursive programming and understand the runtime of different programs. Recursion is the process of breaking a task into smaller tasks of the same design to compute the answer to the overall task at hand. In recursive programming, the function calls smaller instances of itself until it hits a base case. For this project, I programmed three separate tasks and analyzed the results of each one.

Data and Analysis:

Tower of Hanoi:

The Tower of Hanoi is a puzzle where the goal is to move all the disks from one post to another without stacking a larger disk atop a smaller one. For this project, I computed the puzzle starting at $n = 3$ disks to $n = 35$ disks. I calculated how many steps each number of disks took and how long the computer took to complete moving all the disks for the value n. I graphed the number of disks vs runtime of each step and the number of disks vs number of steps it took to compute.

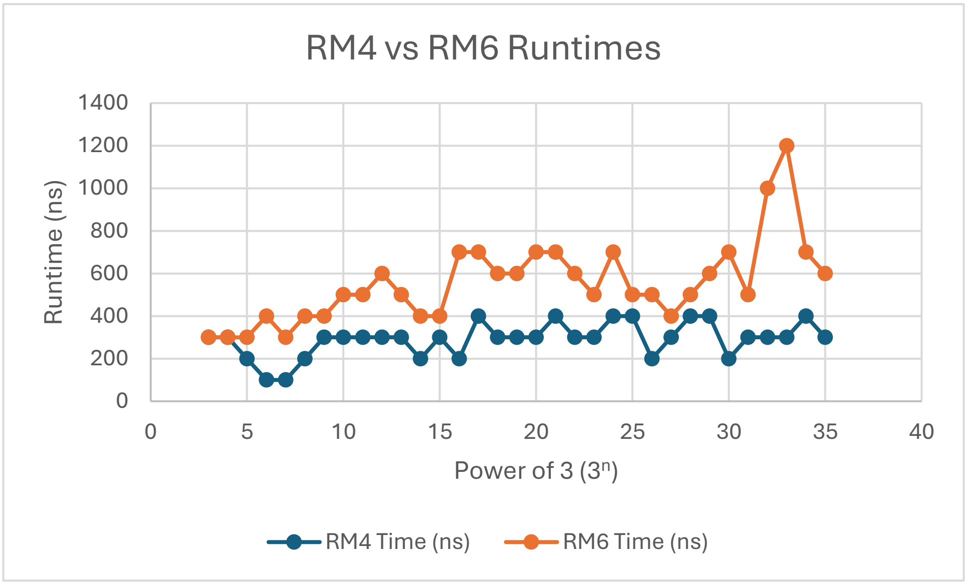




The two graphs have an extremely similar pattern. As the number of steps increase exponentially, the time to complete each step also increases exponentially. Looking at the graphs, the largest Tower of Hanoi that could be reasonably solved would be 40 disks. Using the equation of the exponential trend line in Excel, $y = 0.00002e^{0.6268x}$, I found that 40 disks would take about 25 hours. Since the time increases exponentially, the next few disks would take days to solve. Due to this nature of the graph, by the time the program takes a full day to execute, I would say that is the last disk with a reasonable runtime. A well-known legend claims the world will end when a large Tower of Hanoi is solved, and the accuracy of this claim could be reviewed with the Disk vs Time graph above. When a computer that can compute that many steps in a reasonable time exists, the computer's potential to complete anything exists. This legend is likely viewing the Tower of Hanoi as a benchmark for how powerful computers have become and comparing their power to what else they can compute. Having a computer that powerful could be used for the wrong reasons and danger the world, so I believe that this legend has some merit to it.

Power of 3:

In the second program, I examined the runtime of two different recursive programs that both calculate 3 to the n power. Although both functions computed the same final answer, their method of finding the final answer is different. I ran each program from n = 3 to n = 35 and graphed both functions' runtimes.



Examining the above graph, the RM6 function took a slightly longer time to run on average. These results were expected because it makes twice the number of recursive calls than RM4 does. The two lines follow the same general trend except for a spike at the end of RM6's program. Despite this graph showing RM6 consistently taking a longer time to compute, this was not always the case. Some executions of the program led to both functions taking the same time, around 200 – 300 nanoseconds per step. This variation in runtime could be due to the execution of the program by the IDE. The mathematical computations are very simple in both programs, so they do not contribute much to everything the computer is running in the background when executing this program. Therefore, the runtime of each power calculated was most likely drowned out by other tasks the computer was completing, so this data will fluctuate. Additionally, the runtime per calculation did not increase much from the start of the program to the final execution. This lack of change is due to the program's simple calculations. Each n value just must complete that number of multiplication operations, so at the last iteration the program is only completing 35 multiplications. These are simple operations that do not need a much power to compute.

Subset Sum:

The final program is Subset Sum, in which the program calculates the largest sum of a subset that is less than the target. In this example, our target was 1000. We tried two separate methods to find the subset sum, brute force and a greedy algorithm. The base set had 20 integers that were randomly generated with values between 1 and 150. The greedy algorithm sorted the set and descended the set, starting at the largest, adding the values until it could not add any more without going over the limit.

Iterations	Brute Force	Greedy	Approximation Factor	Nanosec BF	Nanosec Greedy
1	1000	962	96.2%	1.31E+08	200
2	1000	885	88.5%	1.35E+08	200
3	1000	904	90.4%	1.37E+08	300
4	1000	963	96.3%	1.40E+08	200
5	1000	959	95.9%	1.30E+08	300
6	1000	925	92.5%	1.47E+08	100
7	1000	952	95.2%	1.41E+08	200
8	1000	976	97.6%	1.53E+08	200
9	1000	998	99.8%	1.48E+08	200
10	1000	885	88.5%	1.25E+08	200
Average	1000	940.9	94.1%	138627600	210.00

The chart above shows the data from the Subset Sum program. I ran the file 10 times and recorded the outputs and times for each function and its accuracy for each trial. The brute force algorithm always found the correct answer in my trials, as it examined every subset of the original set and found the closest. This is not a guarantee that the brute force will always find the target, but it has a high likelihood of occurring with this setup. Using brute force, the answer always was exactly 1000. I was originally surprised by these results until I examined the math behind the function. With a base size of 20 integers, 2^{20} different subsets exist. With that many sets, there is a high chance that one of them would add up to the exact target.

Overall, the approximation algorithm performed extremely well. With a target of 1000, it averaged 940.9, with the closest value being 998. This greedy algorithm was quite effective. It was the simplest algorithm that could be compared to the brute force algorithm, and it would work for any size array. Additionally, it outperformed the Brute Force algorithm in runtime by a large margin. The greedy algorithm took an average of 210 nanoseconds to compute its final answer, while brute force took an average of $1.38\ x\ 10^8$ nanoseconds. With a small set like the one in this project, the speed of the brute force algorithm is minimal, but this runtime will grow exponentially with the growth of the original set is.

I believe that the greedy algorithm is worth using over the brute force algorithm. Since it only must sort and then iterate through the original set, it is quick with its longest procedure being the sorting of the set. On the contrary, the brute force algorithm is much slower, increasing exponentially since it must create more and more subsets as the size of the base set increases. The greedy algorithm could be slightly improved without interfering too much during the runtime. If the greedy algorithm keeps adding values through the whole duration of the set instead of stopping once the next number cannot be added, it could become more accurate with only adding less than *n* to its runtime. For example, if the algorithm is adding values in descending order and it hits index 15 which makes it go over the limit, it would skip that index and examine index 14 until it hits index 0. This way the program would add smaller numbers at the end of the set to help get closer to the target. The updated greedy algorithm is worth using due to its quick runtime and high accuracy that is minimally affected with increasing size of the base set.