Steven Lee

Lab 2 - Towers of Hanoi

Analysis

In order to solve this assignment, I used stacks in the iterative solution, and no manually created data structures in the recursive solution. In my stack implementation, I decided to use a singly linked list. In the code for the project, there is a Stack interface, which ensures that basic stack methods would be written, and on which I could depend to produce reliable behavior for other parts of the program. The stack interface is implemented by the ListStack class, which in turn, uses a custom Node class to store data and pointers to the next object.

I decided to use a list implementation because the requirements for the project were variable, which means that declaring an array of static size would always waste space. I knew that the maximum possible size would be 50, since that is the maximum size of the problem which was requested to be solved. I would never need random access, which is a major benefit of arrays, and I would only ever access the head of the array. At any given point of stack manipulation, I would need to only manipulate the head of the array, but this part needed to be done quickly, as the solver algorithm starts with two pop operations from stacks. An array may have required resize operations, which would have been costly in terms of time required to complete. Finally, a singly linked list could provide the same functionality as an array, but an array would waste space. For this reason, I chose a singly linked list to implement the stack.

Not wasting space is important, because I was attempting to speed up my application through memoization in the iterative problem. The solution for the 4 disk Tower of Hanoi problem starts with the complete solution for the 2 disk Tower of Hanoi problem. For this reason, it was good to save the previous solutions for later use. However, this requires a total of five stacks. Three stacks represent the towers in which the program moves disks, and two stacks store the results of the latest solution for later. Wasting space in all five of my stacks would increase space complexity significantly.

The design of the program is as follows: Main is used for initializing the RecursVSIterTester Class, starting the test, and writing the results to a text file. Main also takes in two arguments from the command line as the output file for the metrics, and the output file for the steps. Finally, Main logs any IOExceptions which are thrown during execution. RecursVSIterTester is called from Main, and it is in charge of testing the execution speed of the iterative and recursive algorithms. It stores the execution speed in a Metrics class object, and returns the metrics and steps to Main once the test is finished. HanoiSolver is an abstract superclass of both IterativeHanoi and RecursiveHanoi. The purpose of HanoiSolver is to contain methods which are common to both IterativeHanoi and RecursiveHanoi, and needed method bodies. Since I was defining default behavior which both the iterative and recursive classes had in common, a superclass was better than an interface. IterativeHanoi contains the algorithm for solving the towers of Hanoi problem in an iterative manner, along with helper methods for setting and getting the results, and resetting stack variables on completion so the class can be reused. RecursiveHanoi solves the Towers of Hanoi recursively.

I chose to implement my classes in order to attempt to make individual parts of the program work earlier rather than later in the design and building process. I created the two recursive and iterative solver classes first, and I was able to test and debug them using trivial examples, before moving onto the program requirements. This made sense for the project because I would need to make sure that I had identical and optimal solutions from each algorithm, so having both algorithms in the same class would have been more difficult to debug. Both solver classes implement a superclass because they had many writing methods in common which were important for setting and retrieving data, but lied outside of the scope of the algorithm itself. Finally, RecursVSIterTester was necessary because the testing process still required additional setup, and I needed to be able to write helper methods in order to make the tests function correctly. Finally, RecursVSIterTester could be called from Main, and kept my main class modular by allowing the focus of main to be primarily on getting arguments and writing results.

With respect to time, this algorithm executes according to O(2^N) notation. Initially, I was only able to execute the problem up to n = 14, but a few changes made the algorithm execute up to n = 20. First, I changed the algorithm so that it only saved the characters necessary to imply the steps, and then I could actually write the step instructions later. Initially, I appended each step to a string, which was slowing my algorithm down significantly. The iterative hanoi algorithm optimally executes in (2^N - 1) steps with no speedups. I added one piece of memoization, where, if possible, the algorithm reuses the result from two disks ago to start. This reduces the number of steps necessary, but it does not change the O(2^N) notation and growth rate.

With respect to space, the recursive algorithm will take space proportional to the depth of recursion, which will at most be N. Additionally, the algorithm needs a character array large enough to store at most 4 characters for each step in the process, and this grows at a rate of 2^N. Since this is a primitive type array, it will be constant, and the number of bits required to maintain that array is smaller than the number of bits required to perform a recursive call. This is because a char is 16 bits, while a recursive call requires an operation stack to be stored while the method is called again. The iterative algorithm will grow at the rate of 2^N as well. This is because the algorithm requires 5 stacks in order to work, and each of these stacks is proportional to N in terms of 2^N.

For what I learned, I found out that speeding up algorithms which have bad time complexity like 2^N is very difficult, and ultimately meaningless for large N. Initially, I ran both algorithms, and I was able to get to N = 14 before timing out. However, I used smaller variable sizes, and I thought that would help. Unfortunately, it was only trivially faster. Next, I thought that memoization would help. Looking at the solution for each algorithm, I proved by induction that for all n > 2, the solution contained (n - 2). For examples, note n $\in\{1,2,3,4,5\}$. In fact, the solution for n - 2 appears twice, although this was not in my proof. However, even with optimal memoization, the steps to solve the equation go from (2^N - 1) to (2^{N-2} + 4). This is better, but still goes to infinity at the same rate. I also learned how unintuitively quickly equations like this go up in time required to complete. The algorithm solves fairly quickly up to n = 13, then slows down for n = 14, and is almost completely stopped by n = 20.

In terms of what I would do differently next time, I would check first to see if there was any way to change the governing behavior of the O(N) notation for the algorithm. Speeding up a very slow algorithm is only minorly helpful, but changing the governing behavior from O(2^N) to O(N) or O(log(N)) is much more important. This also taught me the importance of selecting the correct data structures going forward. If I select a bad implementation, then that could actually be the difference between the application not running at all, and running quickly enough to be useful. I think I have not realized the implications of big O notation until now because I had only seen small N examples, where a computer could handle even very inefficient algorithms. For design decisions, I would have modeled the whole application on a piece of paper before writing pseudocode. I made trivial mistakes while writing this project, which happened because I converted my pseudocode to Java too quickly and without proofreading.

Finally, this assignment completes the lab requirements and contains printouts for n = 20. Past that, the algorithm ceases to function usefully. As an enhancement, I added memoization to the iterative solver algorithm, which results in minor speedups, allowing the algorithm to work for larger N. I also included summary statistics with average runtime, total runtime, and the runtime of each execution in a separate text file. I also print the number of disks which are being used to solve the problem to the solution output. Finally, I print the number of iterations which are being done to the console, indicating to the user how quickly each iteration is being done. Note, that no output in terms of steps or metrics are printed to the console. This is purely an indication of progress.