Jingyu Yang- Lab 2 Analysis

Lab-Specific Question (Compare and Contrast Recursive vs Iterative):

In comparing the recursive and the iterative implementations, I knew coming in that the recursive would be significantly faster than the iterative, but the magnitude of difference did surprise me. With 2^N runtime complexity, the Tower of Hanoi problem can get very slow with increasing input sizes (an input size increase by one would double its runtime). In terms of ease of writing, the iterative solution was easier, since it made more “logical sense” in the straightforward way that it ran, while the recursive solution was not as intuitive. Overall, this was exactly what I had expected in terms of runtime, since the Tower of Hanoi is such an recursive problem, it makes sense that the iterative solution would be significantly slower.

I was also surprised by how the runtime of the size 1 towers were so high, even though I had tried to cut out all the instantiation code and only time the actual function itself. It would be interesting to learn why that is the case, or if my timing was incorrect somehow.

Timing table:

Size	**Recursive (nanoseconds)**	**Iterative (nanoseconds)**	**Size**	**Recursive (nanoseconds)**	**Iterative (nanoseconds)**
1	2020886	110762	14	1583684	6214162
2	4065	9472	15	2809237	10396223
3	5288	16186	16	7631884	5451812
4	16778	31028	17	35109258	15627306
5	18788	76643	18	22472675	66310976
6	44306	133678	19	59461678	307369045
7	640302	245272	20	313312908	150328701
8	157039	460962	21	735743182	792225360
9	190845	679004	22	1484421641	1558769422
10	330284	1011761	23	2986982136	3446965784
11	503396	964584	24	27079151700	34436208320
12	229879	998127	25	Out of space	Time out
13	862002	2812211	26	Out of space	Time out

<u>Data Structures Implementation and Design Decision Details:</u>

Description:
In the implementation of this lab, the main data structures I used are Stacks and LinkedLists, which was implemented with a generic Node class that holds and object and the next Node pointer. For storing general data, such as a collection of input number and the resulting output, I used a LinkedList. The input was stored using Nodes with string objects, and the steps of the Tower of Hanoi, for both recursive and iterative implementations, were stored with Nodes of custom OutputItem, which is a string representation of the disk moved, where it came from, and where it moved to. As for processing the Tower of Hanoi iteratively, I used a Stack that is implemented by an array of integers, instantiated with a size of the tower, which is the input size provided.

Justification:
I chose to use an array-based stack for the disks in the iterative implementation because it makes the most sense in terms of intuitively representing the problem, has all the functionalities needed at an efficient time and space complexity, and is easy to implement for this problem. First, the Towers of Hanoi problem has three towers, and only the top disks from each tower can be removed, and the most intuitive representation of this functionality would be through a stack, which has LIFO ordering and access to the top most item. In an array-based stack, size is limited, but in this problem we know a set max size, which is the number of disks it can have (the source tower with start off with max and the destination tower will end up with max), so an array-based stack would be easy to implement. Additionally, by keeping a pointer of where the top item's index is, accessing the top (both in terms of removal and adding) is O(1).
As for using Node and LinkedList for the general data, it is fairly straightforward, since data to be printed out and the data of the sizes of the towers are FIFO, a linkedlist with a front and tail pointer is the best option, since it is now O(1) for adding and removing (add from the tail pointer, remove from the head pointer).

Appropriateness/Efficiency:
In this case, using a LinkedList is appropriate in this case, as explained in the prior section, a LinkedList with head and tail pointers make both adding and removing O(1) time when the data is FIFO, such as the Hanoi steps to be printed in the output file, or the list of input sizes.
Also, using an array implementation of Stack is also appropriate because of the LIFO nature of adding and removing disks from a Tower of Hanoi in the iterative process, and we know that the max size of the stack is the size of the tower provided as input, so a fixed-sized stack using an array and a pointer is a great choice for adding and removing from the most recent added due to its ability to have random access and O(1) time for adding and removing items.

<u>Learning:</u> What you learned, What you might do differently next time

From this lab, I extended what I learned in the last Lab, where I implemented my own LinkedList and Stack. However, this was different in that the data that the LinkedList and Node needed to represent were different. In the previous lab, I only had to represent String data in the Nodes, so I hardcoded in the datatype of Node as string, but in this case, I needed to represent the custom object OutputItem and string (from input), so I figured out to set the data type in the Node as an Object, and convert it to readable output through methods such as toString() or getData(), which I wrote for the OutputItem. In the last lab, the representation of the steps was not very elegant, and I tried to improve it this time by minimizing the number of different classes needed, and generalizing classes to encompass more data types. I also was able to learn how to time a java program's runtime, which would be extremely useful in the future, for deciding what data structure or ADT to use for a particular function, or finding which steps are the most costly in terms of time.

In the future, I would like to be able to add error handling for time outs. I made several attempts at adding error handling for things like OutOfSpaceError, but it would still completely become stuck when the input went above 24. I am not sure if this is the expected behavior, or if there is a way to set it up so that it would catch the error before it runs out of space.

<u>**Enhancements**</u>

For this lab, I added extensive error and edge case handling. For example, I included custom exceptions for when an input is a negative number, and excluded the ones with size zero since they should not run. I also added error handling for each step of the way, including error handling for adding to a full stack or popping from an empty stack in the iterative version, and handling errors that may occur when reading the input files. The error handling capabilities is demonstrated with the input and output files badin.txt and badout.txt, where the bad input data has led to error messages being printed.

Additionally, I set a limit for when to stop writing the steps to the output file in order to reduce clutter and the sheer size of the file. For anything with a input size of over 6, the output file would only display the time it took for the method to run, without the actual step by step process.