Lab 02

**Adam Rich**

Section EN.605.202.87

Spring 2018

April 3, 2018

**Approach**

"Step 1: go to the toy store." I sat down with an physical Towers of Hanoi puzzle for quite some time to wrap my head around this lab. Dr. Chlan said in the lecture that the goal was to move the tower to "either of the other posts". The instructions on my puzzle said to move from post A to post C. Since Dr. Chlan said it didn't matter, I decided to do the latter and force each of my solutions to move the tower from post A to post C. However, I wrote the recursive and iterative functions to take the starting and ending peg as parameters.

**Recursive Solution**

The recursive solution basically says: if you want to move stack N to peg C, first move stack N-1 to peg B, move disk N, then move stack N-1 to peg C. It is a very short piece of code, a perfect example of recursion.

The base case is moving 1 disk from peg X to peg Y.

Every call to the recursive function executes a print statement to the output, representing the instruction for one move. For a stack of size N there are 2^N - 1 print statements. The computational complexity of the problem is O(2^N).

**Iterative Solution**

I initially thought it would be clever to write a routine that would take any puzzle state and figure out the best next move. This is an incredibly hard problem because one also has to consider states that are not optimal, i.e. are not found in the list of instructions for the most efficient solution. However, one way to approach this would be with three stacks and a series of rules to identify the right disk to move and where to move it.

The solution that I opted for is quite a simple one, actually. It comes from recognizing the following patterns

1. Every other instruction says to move disk 1
2. For instruction 2^k, e.g. 1, 2, 4, 8, etc., you are moving disk k+1
3. You never put an odd disk on top of an odd one, or an even on an even
4. To move the whole stack from A to C, if N is even the first move is to peg B, otherwise peg C
5. Every 2nd move is to move disk 1, every 4th is to move disk 2, every 8th to move disk 3, etc.
6. Every time a disk moves it is to an adjacent peg, with wrap around, but always in the same direction (either clockwise or counterclockwise)
7. The direction of the movement depends on whether the disk is even or odd

With these rules, which I did not "prove" but just found through doing the puzzle several times, an iterative solution can be developed that is also quite simple and does not require many lines of code.

After clearing it with Dr. Cost, I used the function Long.numberOfTrainingZeros, which returns the number of zeros to the right of the right-most one-bit in the binary representation of a number. This is equal to the power of 2 in the prime factorization for a number. This comes from rule #5 above and tells you which disk you are moving. So, the only thing that is needed is to know what step you are at and where each of the disks is to then know what the next move should be.

The computational complexity of this function is driven by the for loop which iterates over the number of moves, so it has the same complexity as the recursive function, O(2^N).

**Timings**

Here are the timings obtained for N = 1 to 32.



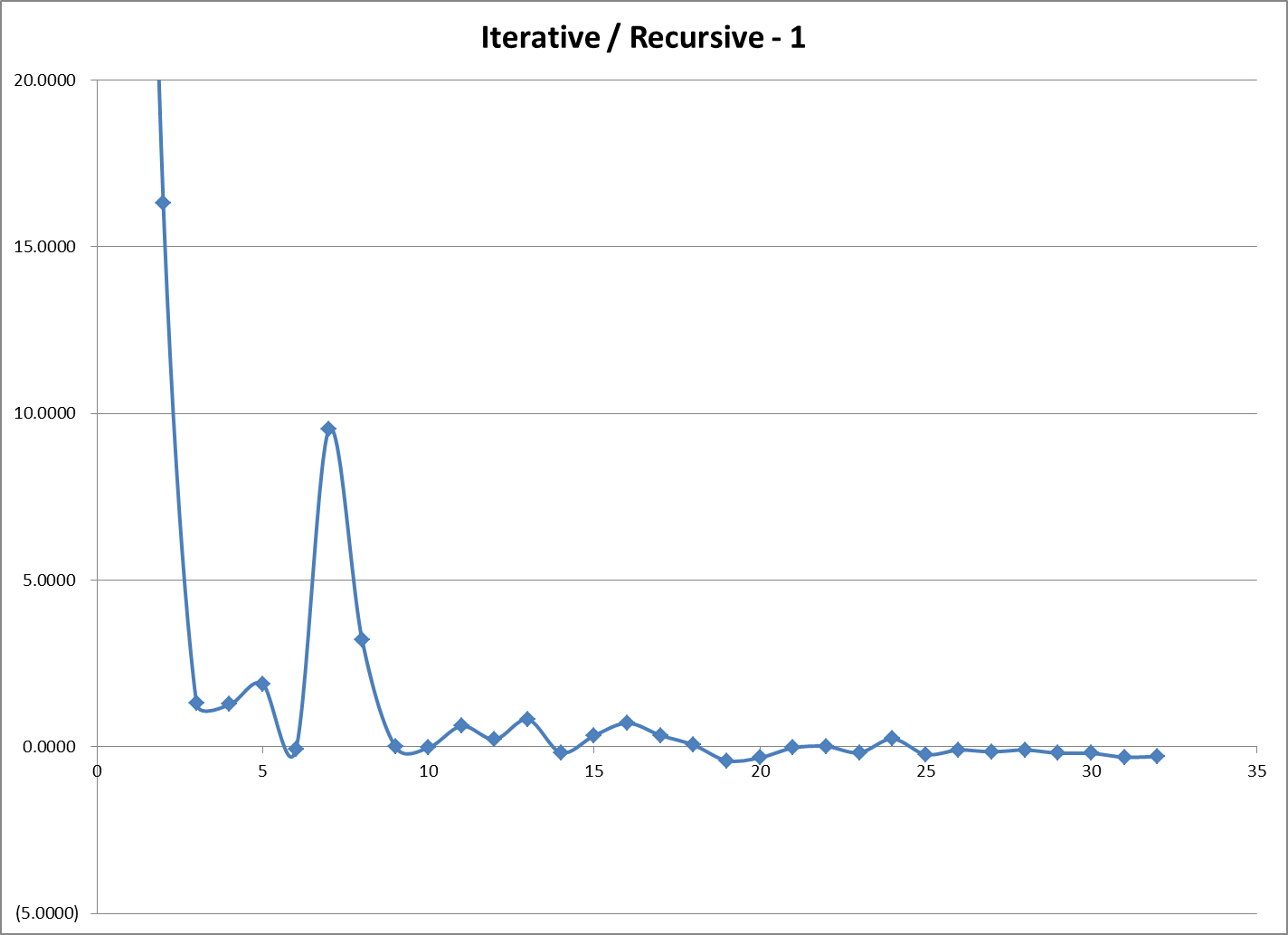
Some things to note:

1. These were run on a virtual machine which shares resources with other users, and the iterative and recursive solutions were run at different times of the day
2. The free space on the drive being written to is 183 GB. Files were created for each N, each method independently and then deleted to free up space before running the next file (timings were stored to a separate file to keep them for analysis)
3. The output provided with the lab has timings that will be different from these because they were run on a different machine. However, the relationship between recursive to iterative for N = 1 to 10 will be roughly the same

For N = 7, the iterative result is most likely an outlier. It could have been caused by several factors external to the process running the experiment, e.g. other resource-intensive operations on the same server, unexpected locking of disk space, Windows giving priority to its own processes, etc.

For N = 1 to 18 the recursive solution is almost always better. This may be because there is a fixed number of calculations per move in the iterative process and in the recursive as there are more disks the overhead, in terms of nested calls of the function, is higher. The overhead of recursion is a smaller price to pay than the fixed number of calculations in the iterative procedure up to about N = 18.

However, beyond that point it appears that the iterative solution begins to be more and more efficient, probably for the same reasons. The maintenance of the recursive process will be more taxing as N is larger.



Another way to see this relationship is to look at the ratio of iterative over recursive less 1.0. Points below the line represent when iterative is faster than recursive. As N gets larger I would expect that iterative would perform better most of the time.

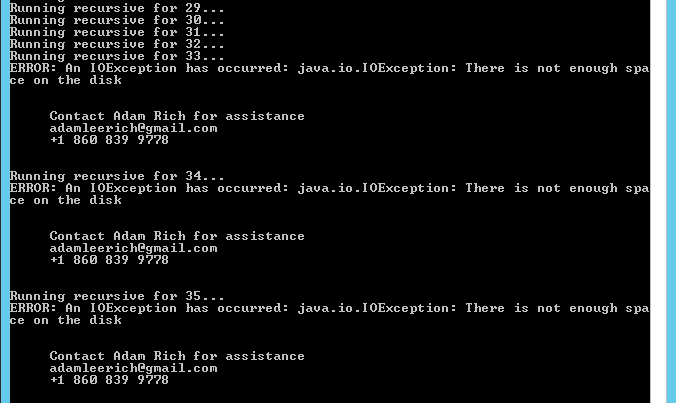
**Space Complexity**

I found that for large N the space requirements are a big hurdle.

Each instruction file requires 2^N - 1 instructions. When they are of the form: "Move disk 5 from tower A to tower B" they require 36 or 37 bytes of file space. I was able to solve ahead of time the N at which my machine would "crap out", to use a technical term.

183 GB = 1.83e11 bytes, divided by 37 = 4.95e9 instructions. Log2(4.95e9) = 32.2

I did get a screenshot of what happens when trying to run N higher than that:



To print an instruction file for N=50 would require 42,000 TBs of disk space!

**Timing Mechanics**

Because I wrote the function to take the starting and ending pegs as parameters, I felt that it was important to do some checking of inputs, even though it is a private function in the application class. There are three or so error checking statements that were included in the timing, but the front and end matter in the main function were not.

I used the system function nanoTime to measure the start and end of the procedure calls, with the difference being the number of nanoseconds it took to create the instruction files.

**Exponents and Overflow**

Based on my research, which may have been incomplete, it does not appear that Java has an exponentiation function that returns integers. Math.pow returns doubles. So, to calculate the number of moves for my iterative process, I decided to use something like the following code, m being the number of moves, n the number of disks

long m = (1 << n) - 1;

The bitwise shift operator acts like multiplying by two, so this is equivalent to 2^N - 1. However, the above piece of code unexpectedly does not work for N > 31, even when N is also declared as a long. That is because the "1" is considered an integer and it remains an integer even when shifting by a long. To allow this to work for N > 31, I had to use the following

long m = ((long)1 << n) - 1;

**Submitted Output versus Timed Results**

The file called "output/timings.txt" is the timings file created for the 20 sample instruction sheets included with my submission. N = 1 to 10, one set for iterative, one for recursive. Each file follows this naming format

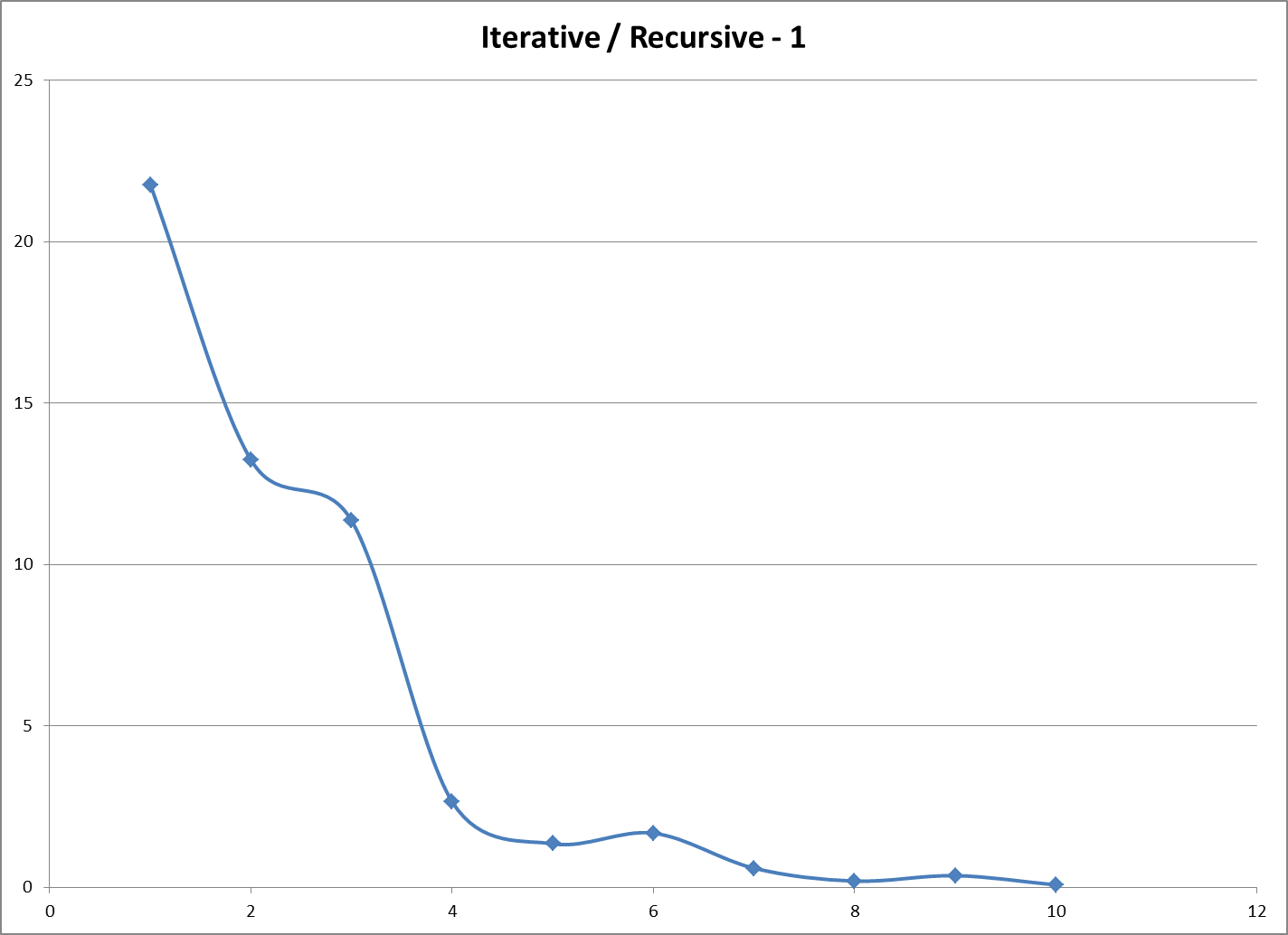
output/tower-1i.txt

The number is the number of disks, the suffix is "i" for iterative, "r" for recursive.

These files will have different timings that the chart above because they were run on a different machine (one that was not able to be dedicated to playing Towers of Hanoi for over 10 hours!). The timings for these 20 runs are below, with numbers kept as nanoseconds.



The relationship for small N seems to hold, and actually look smoother for this sample.



**DOS Batch "Helper" Scripts**

Several DOS-style BAT files were created to assist in running these tests. They are documented in the README file.

If you have Git installed, you can use "compare-with-git.bat" to quickly compare that the instruction lists for n = 1 to 10 for recursive and iterative are the same. No guarantee that they are correct as they could both be wrong together!

**Command Line Args**

The method for running the app and the expected command line args are also documented in the README file and with the source code. It seems prudent to discuss here as well simply to document my conforming to the expectations of the lab. The arguments are:

1. the number of disks
2. the name of the file where the instructions should be written
3. what method to use, 1 = recursive, 0 = iterative
4. an optional "timings" file to which is appended meta data and execution time for this run  
   if no file is given a default is used

An input file was not necessary. Argument processing, or "fixing" is handled in a private function called "populateVars" where errors are thrown or private class variables are populated.

**Things Learned**

I learned a lot about the Towers of Hanoi puzzle and can now solve it without really thinking, for an arbitrary number of disks! I also learned that it is very important to consider things like space and time complexity and integer overflow when dealing with very large problems.

**What would I do differently?**

I still would like to solve the problem of what is the next optimal step given any state of the puzzle. Also, I might have tried memoization, although I don't expect that it would have performed anywhere near as good as recursion because of all the overhead and the repeated for loops required to iterate over the stored instructions (and that fact that six versions would have to be saved for each N!).

**IDE**

My IDE was notepad++ and the command line. Instructions on how to build and run the code, from the command line at the project base folder, are in the README file.

**github**

I used github, a private repo, to backup and track important chunks of development.