Towers of Hanoi Recursive and Iterative

Jeffrey Wan

John Hopkins University

Author Note

It was actually pretty great to learn how to solve this problem using recursion and iteration. The recursive solution is so much more elegant than the iterative one to a degree that I didn't realize was possible previously. Thanks for the lab!

Abstract
This program solves the Towers of Hanoi problem with a variable number of disks using both iteration and recursion. The algorithms separately solve the problem and provide the same output in a file specified by the using of the application.

Design and Analysis

This application is fairly simple in design and only has three classes. The first class is the TowersOfHanoiRunner and it's responsible for creating the output files, taking in the command line inputs which include the file names for the algorithm steps and timetables, and the number of disks to start in the Hanoi game. The runner also handles edge cases like an invalid number of disks.

The second class is the TowersOfHanoiIterative which runs the algorithm for the iterative solution for the game. The algorithm is better details in the code but a few interesting points. First, it becomes clear why we switch the destination and spare poles when the numberOfDisks is divisible by 2. When you examine the cases of when the numberOfDisks is 2 and 3, it's clear why we have to switch disks: for every additional disk, we need to switch which pole we need to start building on otherwise the destination pole won't eventually be free for the nth disk. Second, the algorithm is complicated and lengthy with several logical tricks… it's cumbersome to write and to read. Third, we need to use a stack to represent the pole and this application uses three stacks to represent the three poles. A stack is chosen because that's how the pole works; the last disk places on the pole is always going to be the first removed.

The last class is the TowersOfHanoiRecursive. The main point that stands out about the algorithm is that it's elegant and simple; it's about 6 lines of code. On a high level, the algorithm makes sense. Say have have 10 disks on the source. We then need to move 9 disks from the source to the spare so that the 10th disk can be moved from the source to the destination pole. Then the 9 disks on the spare pole can be moved to the destination. That logic lends very well to recursion.

The Big(O) of the recursive solution is O(2^n) which is exponential time which makes sense since the minimum number of moves needed to solve the problem with n disks is (2^n)-1 which is an exponential equation. After some reading and thought, it seems like the space complexity of this is just O(n) at max (the max occurs when the algorithm goes down the depth of the tree at the beginning) because the base calls resolve, are removed from the stack, and even though subsequent calls may increase the stack again, it never grows larger than the initial depth which is n. It is better explained and visualized **here.**

The Big(O) of the iterative solution is the same. The game still needs (2^n)-1 moves as demonstrated by this line:

```
for (int move = 1; move <= total_num_of_moves; move++) {
```

And so the time complexity is the same. The space complexity is also the same because as number of disks grows, the number of nodes that need to be stored on the poles grows linearly and so the space complexity is O(n).

Time Results

Here is a timetable in nanoseconds:

NumberOfDisks	Recursive	Iterative
1	30767020	2493486
2	58475	54919
3	117652	82209
4	121075	138314
5	220775	392285
6	378403	355821
7	474330	444989
8	493125	461065
9	980652	941706
10	1234504	1369358
11	1354036	1665276
12	3094142	3406768
13	6531468	5160192
14	7119428	4902927
15	5660438	9808106
16	21624135	10664483
17	14060656	26764518

The time are in nanoseconds. Both algorithms seem to grow together at the same rate which makes sense given that they both have a big O of O(n^2). There seems to be exponential growth in the nanoseconds which makes sense. The recursive solution has a wider and wider tree of method calls, similar to how the recursive solution for fibonacci grows wider and higher. The number of method calls grows at a n^2 rate.

Odd observation:

This is my code:

```
for (int i = 1; i <= numberOfDisks; i++) {
    output("Number of Disks: " + String.valueOf(i));
    output("Running Recursive Towers of Hanoi...");
    Instant start = Instant.now();
    output("time: " + start);
    recursiveSteps = new TowersOfHanoiRecursive().run(i, 'A', 'B', 'C');  // the last 3 args
are the names of the rods used in the game
    Instant finish = Instant.now();
    output("time: " + finish);
    long delta = Duration.between(start, finish).toMillis();
```

```
    output(recursiveSteps);
    output("Recursive time elapsed: " + delta);

    output(DELIMITER);

    output("Number of Disks: " + String.valueOf(i));
    output("Running iterative Towers of Hanoi...");
    start = Instant.now();
    iterativeSteps = new TowersOfHanoiIterative().run(i, 'A', 'B', 'C');  // the last 3 args
are the names of the rods used in the game
    finish = Instant.now();
    delta = Duration.between(start, finish).toMillis();
    output(iterativeSteps);
    output("Iterative time elapsed: " + delta);

    output(DELIMITER);
    if (i != 1) {
        output("Decrementing number of disks for next run...");
    }
    output(DELIMITER);
}
```

In the timetables, the first run seems to have a large delta and the rest of the runs are normal (the milliseconds eventually increase and using nanoseconds gives more meaningful results). Why is there such a huge upfront cost on the first run? Why is there a large delta on the first run?

Lessons Learned

The biggest thing I learned is that a function can have a big O time complexity of O(n^2) but a space complexity of O(n). In the recursive algorithm, the way in which the space collapses as subfunction calls return is interesting and the collapsing of the function calls is why the space complexity is less than the time complexity.

The second lesson learned is that sometimes, the recursive solution is wildly simpler than an iterative one and does not have performance downsides. I think in this application, the recursive solution is the better choice.

Some additional features:

- included Javadocs in the doc folder.