605.202: Data Structures

Lab 2 Analysis

Renee Ti Chou

Due Date: March 21, 2017

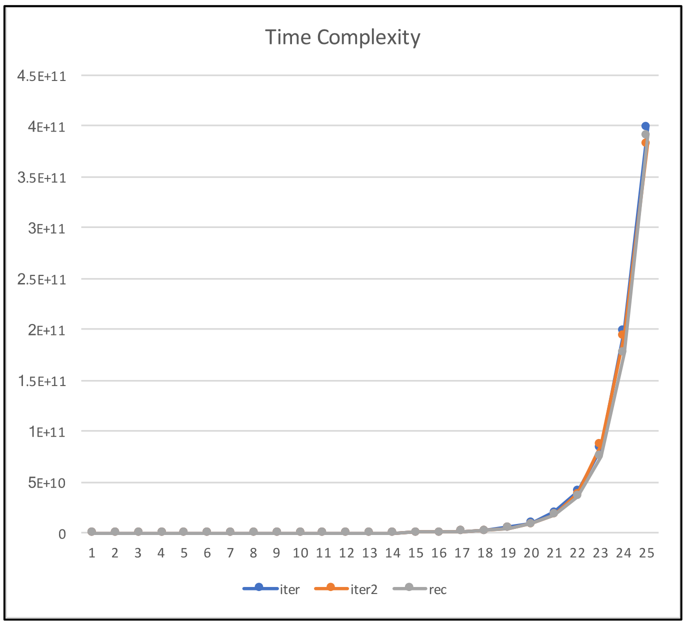
Lab 2 Analysis

In this assignment, there are three different solutions to the problem, one using the recursive method, and others using the iterative method, which are respectively modulated to the *RecHanoi*, *IterHanoi*, and *IterHanoiFromRec* classes. The purposes are to compare the efficiency and efficacy of different approaches and discuss what factors contribute to a better solution to the Towers of Hanoi problem.

I. Issues of Efficiency

1) Time complexity

n	rec	iter	iter2	n	rec	iter	iter2
1	171845.2	97469.6	65692.8	14	182403487.3	161617133.3	164798448.8
2	190214.5	247337.5	201314.2	15	294087130.6	354699179.9	390645381.1
3	440499.6	508741.6	363572.3	16	661355072	719227607.9	605776079.1
4	758684.8	1053718.4	834281.8	17	1216180618	1335914984	1157503920
5	1271116.3	3411233.1	1862238.4	18	2354220182	2709552266	2579410320
6	2599303.1	3013332	1920031	19	4675555933	5322076027	4494584159
7	3626425.8	6471221.6	3527355.2	20	9040961356	9997255778	9151706634
8	6048635	4876694.2	4169523.7	21	17953116484	20324704044	17944542867
9	7176118.3	9574017.2	15384876.8	22	35914435565	40473708084	38514149276
10	16706932.8	25134530.9	19468255.6	23	75129295046	83462374340	86762788878
11	35060250.8	24471254.7	62036889.6	24	1.77071E+11	1.99166E+11	1.94303E+11
12	62625843.4	63846149	50557782	25	3.90441E+11	3.98441E+11	3.83033E+11
13	76276368	97391213.5	105750096.7		(nanoseconds)	(nanoseconds)	(nanoseconds)



The values in the table are the averages of 10 samples. The *rec* column, *iter* column, and *iter2* column are from the *RecHanoi* solution, the *IterHanoi* solution, and the *IterHanoiFromRec* solution, respectively. The time increases doubly with the value of n, and it takes about 3.5 hours to run the 29th Tower of Hanoi for all of the solutions. In order to generate 10 samples within the limited time, I ran to the 25th Tower of Hanoi for each solution, and it took about 6 hours to accomplish all of the works.

i. *RecHanoi* solution

The complexity of this solution depends on the number of recursive calls because it will change with the value of n. For a particular value n, the times of calls is $1 + 2 + 2^2 + \ldots +2^{n-1} = 2^n - 1$. Thus, the solution has time cost of $O(2^n)$. From both the table and the plot above, we can clearly see the times double with the increase of n. According to this statistical analysis, to run the 50th Tower of Hanoi, my computer will take more than $10^5 * 2^{50} / 10^9 / 60 / 60 / 24 / 365 = 3{,}570$ years to accomplish it.

ii. *IterHanoi* solution

This iterative solution contains a *for* loop, so the time complexity would be linear to the iterative times. Because of the *Math.pow(2, n) - 1* function, the iterative times is about 2^n for each n. Thus, the time complexity for this solution is $O(2^n)$. From the result shown above, the line of *iter* overlaps the line of *rec*, which indicates that the time complexity for both solutions are the same.

iii. *IterHanoiFromRec* solution

The theoretical calculation of the time complexity for this solution becomes more complex because it involves two while loops. Before the statistical analysis, I expected, intuitively, that it would have higher time cost than the others. However, from the table and the plot above we can know that this solution also has time complexity of $O(2^n)$. All of the three lines increase dramatically after $n = 20$, demonstrating that the algorithms for this problem are all computationally expensive.

2) Space complexity

i. *RecHanoi* solution

The space complexity is related to the size of the system stack. The maximum number of items pushed into the stack increase with the value of n linearly, so the space cost is $O(n)$.

ii. *IterHanoi* solution

The space cost for this solution mainly depends on the integer arrays. With each n value, the space needed is $3n + 9$, which displays a linear relationship. Hence, the space complexity is $O(3n)$.

iii. *IterHanoiFromRec* solution

Since there are four arrays of size n created when the value of n is passed in, the space complexity is O(4n). Compared to the *IterHanoi* solution, which also involves the array structure, when n is extremely large, there may be a perceptible difference between the space costs of these two solutions because they have a different coefficient.

II. Justification for Design Decisions

1) Lab2

Lab2 contains the driver and is designed to print out the three solutions into a single file. It contains for loops to drive the process from 1 to the assigned number disks. I used *System.setOut* method to redirect the output stream since the *print* methods are separated in different classes. Considering encapsulation, I left these classes less modified so that the things would not be ruined carelessly.

For concise and clarity purposes, the output format is designed to constitute units of three solutions for each size of tower. In this way, we can easily check if there are any differences or mistakes among the three solutions. An optional choice is written for the user to generate the table of times, in which the steps of solutions would be printed out to the console. The file format is comma separated values, which would be convenient for subsequent analysis when imported into other software such as Excel.

2) RecHanoi

This class solves the problem recursively and contains two methods. The *recTower* method is the basic recursive method which has four parameters because the poles are going to swap with each other during each recursive call. The *recTime* method returns the time in nanoseconds for each run of a particular tower size, and it would also print out movement steps. Both of the methods are public, so the user can choose between them depending on the user's actual needs.

3) IterHanoi

This class has two public methods, and the parameters passed in are designed to correspond to those of the recursive solution, making them the control variables for time analysis. It solves the problem iteratively by simulating the actual moves of the Hanoi towers using the array structure. There are three integer arrays created to represent the poles, with name of the poles stuffed in, the bottom of poles indicated by -1's, and the top of each pole referenced by values in the first element in the arrays.

Because the second pole is going to be the destination of the tower, the numbers of disks passed in the function are divided into the odd number and the even number groups. There is a rule for the Hanoi tower that in every three steps the movements between specific two poles will repeat. The algorithm follows this rule, and the conditions for the direction of moves are separated to three private methods.

4) IterHanoiFromRec

This solution is converted directly from the recursive version. Since the *recTower* method in the *recHanoi* class has an inherent binary tree structure with the two recursive calls, the *iterTower2* method in this class simulate a stack using the array to mimic the iterative inorder traversal of a binary tree.

In order to have corresponded parameters as *recTower*, the root of the tree is taken out of the while loop and will be independently assigned the passed-in values. Another modification I made for the recursion-to-iteration conversion is due to change of the poles in each call. When a left child is created, pole B and pole C will exchange; whereas when a right child is created, pole A and pole C will swap.

The *disk* variable represents the order of the disk. In the binary tree of this problem, the nodes at the same level will have the same value, which is also the order of the disk. Thus, the decrease of the value of *disk* by one is equivalent to moving down to the left child or the right child of a tree node.

5) IllegalParameterException

This exception class provides specific error message for the iterative methods, which helps the user who does not know the exactly implementation in the methods to realize that an illegal parameter is passed in. This error handling is not used for the recursive method since it could affect the analysis of time because of the addition of the *if* statement, and also, I think the user can understand what the error message of *stack overflow* means when there is an inappropriate number of disks passed in.

III. Data Structures Implementation

I chose the array structure for the iterative solutions of this lab because I wanted to use it to simulate stacks, and I thought it would be a good practice to write the code without using the source code of a stack. The array structure is already built in Java, and it could be a good choice because it has the advantage of random access. In addition, the size of the array is the number of disks, so I don't have to guess the amount of space for the array in advance.

IV. What I learned

I learned the beauty of the stack, and that the stack is a simple structure which can be used in many places to facilitate our works. I used the stack structure in two different ways to assist me in writing the two iterative solutions. The first one is using the stack to simulate the poles of Hanoi towers, and the second one is using it to mimic the system stack provided for recursion. The stacks were intermediate steps in the developing process, and I finally converted them into array structures, because I wanted to challenge myself to accomplish the lab without using the source code of a stack.

Through the practice of the array structure, I did realize that Java methods actually pass in parameters by values rather than references. The situation I encountered was that when I was trying to move the disk between two arrays, I made the *move* method have the index of the disk position passed in. The result was that there is no effect on the movement. Finally, I figured out that passing in the array instead of the array index would successfully move the disk, and I speculated the reason is that the “value” of the reference to the array can passed in by the *move* method.

Another thing I learned is that recursive calls in the same method can inherently form a regular tree structure, which is very useful and helpful to me to convert the algorithm from recursion to iteration. In the Towers of Hanoi problem, it has a binary tree structure because there are two recursive calls in the recursive method. Figuring out the underlying structure, it then becomes easier to convert the recursive method into an iterative version.

V. What I might do differently next time

What I might do differently next time would be changing the arrays in the iteration solutions into two dimensional arrays. I am considering this because it may make the code look more succinct and clearer by reducing the number of declarations. Also, I might want to analyze the time complexity for this new version to see if the alteration can lead to any difference.

Other things I might want to do differently are making another driver for this lab and having the *printTimeTable* method to be put in the additional driver. I think it could be more convenient because in this way the user will not be forced to get the output file for movements first and then get the file for table of times. By creating an additional driver, it may help the user to collect the samples for statistical analysis more conveniently.