CIS 303 Analysis of Algorithms
Spring 2020

Matthew Scaccia
Assignment 4b

04/01/2020

Problem

The problem assigned for 4b is to implement the Ackermann function as a stack, without using Java's stack, in order to simulate recursion. Once this is done we are to then time test our stack based implementation against the version provided and compare the results here.

Hypothesis

I hypothesize that for my implementation of Ackermann's function my program will actually run orders of magnitude slower than its recursive counterpart. The trade off being we can go deeper into the call stack than a typical recursive algorithm. I feel that due to the nature of the recursive algorithm using the native call stack it is bypassing a layer of excess complexity. I hypothesize that the addition of a loop with a variable n value sitting on top of the call stack is what will cause the largest hit to performance. Furthermore, having to make a 3 Tuple object to handle the data for Ackermann: m, n, and operation will also eat up some resources. This *should* leave my implementation of Ackermann at a significant disadvantage. As the calls go deeper for Ackermann, the iterative time should increase and outpace the recursive version.

Methods

1. **My implementation:**

```
public static long computeAckermann(long m, long n){
    LStack<Tuple> stack = new LStack<Tuple>();
    long result = 0;
    stack.push(new Tuple(m,n,0));
    while(stack.length() > 0){
        Tuple temp = stack.pop();
        if(temp.getOp() == 1)
            stack.push(new Tuple(temp.getM(), result, 0));
        if(temp.getM() == 0)
            result = temp.getN() + 1;
        else if(temp.getN() == 0)
            stack.push(new Tuple(temp.getM()-1, 1, 0));
        else{
            stack.push(new Tuple(temp.getM()-1, 0, 1));
            stack.push(new Tuple(temp.getM(), temp.getN()-1, 0));
        }
    }
    return result;
}
```

2. **General approach:** My general approach was to research the problem and mateiral as much as possible and become familiar with it. First I read the attached documentation in moodle specifically the how to simulate recursion with a stack guide. I then decided to deconstruct the python code and remake it in Java. At this point I settled on using Tuple's to handle moving the data around my stack based implementations. I then went a step further and took the textbook code for List and Stack and implemented all six source files in Java. I then went on to study the books examples of converting recursive algorithms into stacks and why this was important to us as computer scientists: we may need more stack space than what it allotted to us. I then took the book examples of factorial and Towers of Hanoi and remade them in Java, in my own code. Finally, I studied Ackermann, hand traced my states, and made sure to understand the algrothim before attempting to convert it to a stack. I used a Tuple with *m* and *n* values as well as an *operational* bit to flag whether or not the value I just pop'd off a stack was a nested call.

   On the testing side of the program I first implemented my rendition of Ackermann in a seperate file, tested it, and then integrated it into the original Ackermann.java file. I then modified this file to handle both it's original task of computing and displaying the value as well as performing the assigned experiment.

3. **Data structures:** For this project I used not just one stack but the books array based stack(for juggling the experiments set values) and the books linked list based stack for handling my Tuple objects and simulating Ackermann. I also used a simple Tuple object, inspired by the list feature in Python, to handle my Ackermann values.

4. **Experiment details:** I tested many things along my way to the main experiment. I stress tested the books stack objects on stack based Factorial and Towers of Hanoi. I further tested my own Ackermann as a stack against the table of values provided and then against the Princeton algorithm we were given.

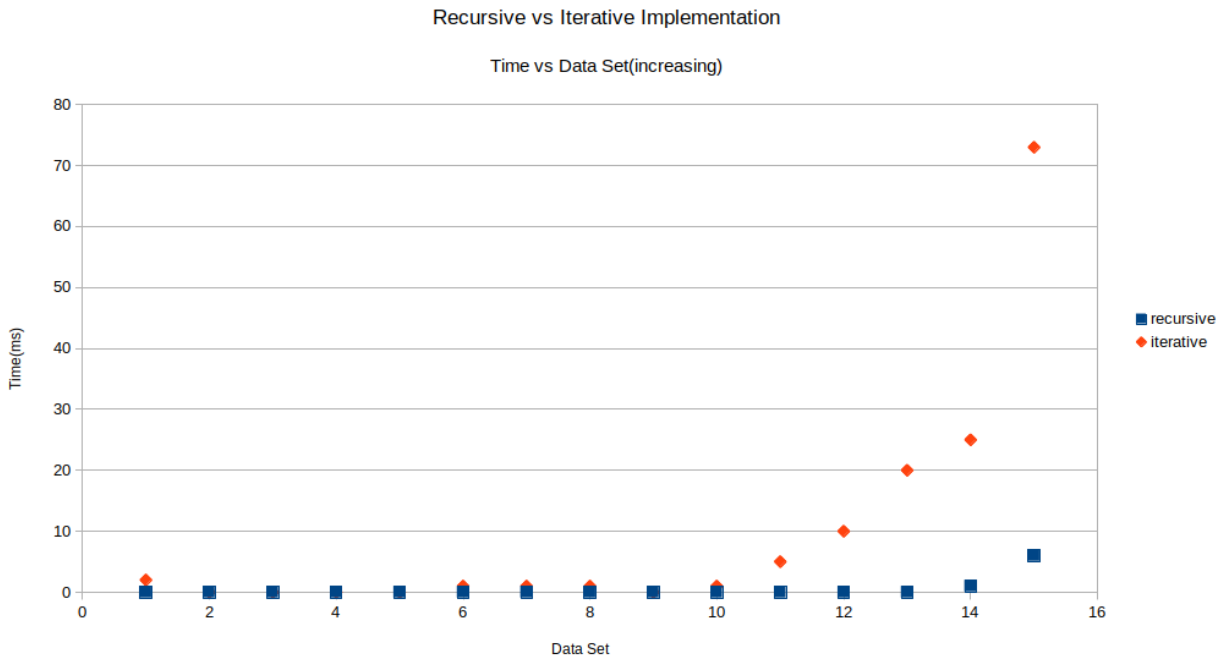
   For the actual experiment I took the base Ackermann.java program and modified it to handle both the original code and the experiment code. I made two separate methods to time both implementations of the Ackermann function. I had the program take snapshots of time before and after the Ackermann function was computed for the given data set entry. I then compared these times and stored the total elapsed time per data set entry in a separate file designated for each implementation.

   (a) I used the sample size of 15 data values provided for the main experiment. When testing my implementation of Ackermann I chose to use the smaller values as I could hand trace them and see how my stack would behave.

   (b) I ran the experiment once to generate my graph and formally test my hypothesis. I ran it multiple times outside this to compare the times generated between the two algorithms. Prior to that I tested my own Ackermann stack implementation on a variety of values from the table on Wikipedia.

   (c) The metrics I used for this assignment we're the set numbers, one through fifteen, and the elapsed time to calculate Ackermann at that data point.

Results



Discussion

The data from my experiments show that my hypothesis was correct, as my data set values grew larger the time to compute Ackermann with a stack overtook that of its recursive sibling. I tested this multiple times comparing the results to verify the integrity of my findings. I stand by my hypothesis that the gain of addition call stack depth comes with a performance hit. I feel that due to the nature of the algorithm and the fact we are adding a second layer of data manipulation on top of the call stack we are slowing down the speed of the program significantly.

Conclusion

Again, I feel that my hypothesis was right and the results back that claim up. Given more time, which was an issue for me in this assignment, I would run more tests. I would like to expand the scope beyond Ackermann as well. The assignment has peaked my interest on the topic of simulating recursion with a stack object. I would like to go back and do a similar experiment on Towers of Hanoi and Factorial and gauge their performance against the depth of the recursive calls. I would also like to see if I could trim down my algorithm or if there is a better way to implement it to improve performance.