CIS 303 Analysis of Algorithms
Current Semester Here

Trevor Butler
Assignment 4b

10/30/2020

Problem

Without using Java's build-in stack, create a way to implement the Ackerman function simulating recursion using a uniquely created "stack". Gather the results using provided sets of numbers and compare them to the recursive implementation provided in the ***Ackermann.java*** file. Once the data is recorded, create a graph to visual interpret the results.

Hypothesis

Based on my knowledge regarding stacks and recursion, it is presumed that the recursion will be more expensive in run-time as the function calls itself every iteration. But while analyzing the "stack" implementation created, it appears this may be more expensive as running through the while loop in ***ackermannStack(long m, long n, PrintWriter pw)*** results in multiple iterations of if statements and calling other functions which could add up over time. Despite this, it is still proposed that the recursive function provided will be more expensive overall compared to the newly created stack implementation. This is concluded being when $n == 0$, there will be an additional call of itself which will add run-time.

Methods

1. Your implementation:

```
public static long ackermannStack(long m, long n, PrintWriter pw) {
    AckerStack ackStack = new AckerStack(MAX_STACK_SIZE);
    pw.print("(" + m + "," + n + ")\t \t");
    long startTime = System.nanoTime();
    ackStack.ackPush(m);
    while (!ackStack.isStackEmpty()) {
        m = ackStack.ackPop();
        if (m == 0) {
            n++;
        } else if (n == 0 && m > 0) {
```

```
                ackStack.ackPush(m - 1);
                n++;
            } else {
                ackStack.ackPush(m - 1);
                ackStack.ackPush(m);
                n--;
            }
        }
        long stopTime = System.nanoTime();
        long totaltime = stopTime - startTime;
        pw.println(totaltime);
        pw.close();
        return n;
    }
```

*There are also custom stack methods that were created as Java's stack ADT was prohibited. ackPush(long i), ackPop(), and isStackEmpty() which are the equivalent to Java's push(item E), pop(), and isEmpty() respectively. These are not provided in this document as their operations are assumed, their declarations are in the *AckerStack.java* file.

2. General approach:
   The general approach was implement a stack-based (not Java's built-in stack) solution to simulate recursion. While the "stack" is not empty, an implemented Ackerman function is looped through until the stack is empty. The run-time is calculated and output to a file where the pair of numbers used is additionally recorded. A file is also generated to retrieve the time as well as data entered while executing the provided ***Ackermann.java***. With these two files, an analysis can be conducted to determine which has better time complexity.

3. Data structures:
   The main data structure used with this implementation was an array that acted as a stack. This was done by storing and removing data in the same way a stack does. The array only permitted data to be pushed on or popped off, never looking past the top element. This allowed a "stack" to be created to complete the computations without using Java's stack.

4. Experiment details.

   (a) Size of N you used:
   15 for each implementation, 30 total

   (b) Case(s) tested (best, worst, random, etc.):
   (0,0), (1,0), (1,10), (1,20), (2,0), (2,10), (2,20), (3,1), (3,2), (3,3), (3,4), (3,5), (3,6), (3,7), (3,8)

   (c) Number of iterations/copies of the experiment:
   There were 15 iterations of the experiment ran for both ***Ackermann.java*** and ***ackermannStack(long m, long n, PrintWriter pw)*** resulting in a total of 30 cases ran.

   (d) Metrics.
   Run-time in nanoseconds

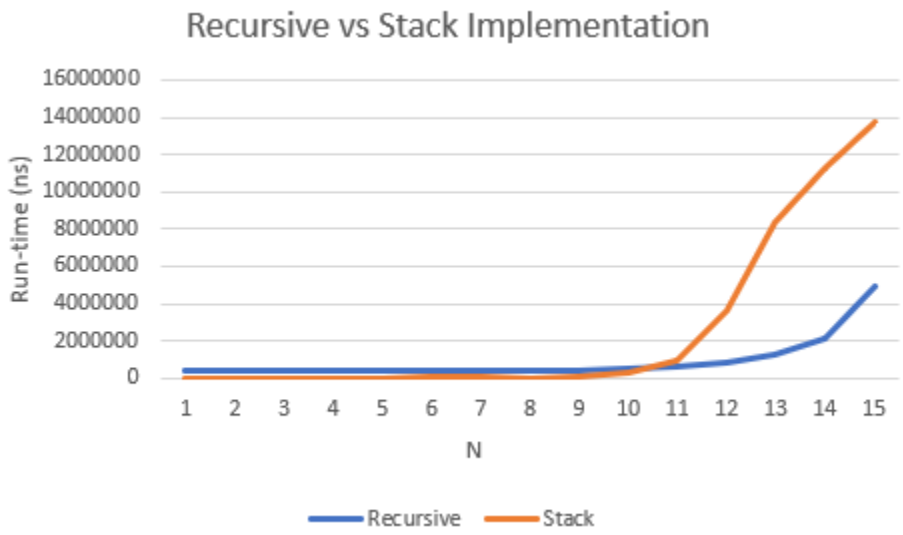

Figure 1: Graph of recorded run-time of both ***Ackermann.java*** and ***ackermannStack(long m, long n, PrintWriter pw)*** computations.

Results and Discussion

After pondering for some time to create a solution to execute the problem while following the set parameters given, an array was chosen to simulate a stack. This was done by treating the created array as if it was a stack, not looking past the top element, and only pushing on/popping off elements being a stack is a LIFO (last in first out). A linked-list was also considered but ultimately rejected as the array premise carried out seemed to be more straightforward.

The data collected was puzzling upon first glance as it did not seem to have a definite conclusion of which application was more efficient. After adding the data to an Excel file and creating a visual graph it was much easier to conclude the results. It is shown on the interval between N = 10 and N = 11 that the stack application beings to add time at an alarming rate.

Going back to the text file results you can see the switch between the two inputs (3, 3) and (3, 4). This switch is justified being when $(3,3)$ is input the given recursive method clocks at 470900 nanoseconds compared to the stacks result of 260900 nanoseconds. But when $(3,4)$ is input the recursive method results in 260900 nanoseconds and the stack clocking 909800 nanoseconds. This trend follows for the remaining inputs of $(3,5) - (3,8)$ resulting in the stack increasing run-time after $(3,3)$. This leads the conclusion that after a certain input of (m,n) (in this case (3, 3)) the stack results in a greater run-time than the recursive solution.

Conclusion

The given hypothesis was supported but also not supported to an extent. It was proposed that the recursive application would take longer than the stack implementation which was true,

but as explained, after the data set $(3, 3)$ was entered the recursion emerged as faster. The stack implementation began to build run-time as n increased with the value of m remaining the same. It would be fascinating to see how these times compared when using more/larger inputs of (m, n) and if the same outcome would hold of recursion being faster as n increases.