Hawk Weisman

CMPSC112 S2013

Laboratory #2

Lab Report

**Deliverable One: A brief description of recursion**

Recursion is the act of a function calling itself one or more times. In Java, this occurs when a method contains a call to itself within its body. There are certain requirements for every correctly written recursive method. These include:

* A recursive method must test for a condition; otherwise, it will never terminate.
* A recursive method must have a base case. This is the 'target' case which the method is trying to reach. Once the base case is reached, the recursion stops.
* A recursive method must progress towards the base case. If it does not, then it will never terminate.

In the case of the fib() method used in this laboratory, the base case is if (k <= 2)and the else clause is the recursive case. This means that if *k* is not less than or equal to 2, then the method will call itself once again. Fortunately, this is a correctly-written recursive function, and the the recursive case contains code that will bring the value of*k* closer to 2, bringing the program toward the base case. Since each Fibonacci number is the sum of the previous two Fibonacci numbers, a call to fib(*k*) will call fib(*k*-1) and fib(*k*-2). This means that, for example, a call to fib(5) will look like this:

This "branching tree" recursion results in a great deal of fib() methods being placed on the stack, and therefore, it has a very high memory overhead. Alternatively, if the recursive method only calls one copy of itself, the memory overhead will be much lower.

fib(5) / \ fib(4) fib(3) / \ / \ fib(3) fib(2) fib(2) fib(1) / \

fib(2) fib(1) *//fib(2) and fib(1) are base cases*

**Deliverable Two: Primitive Data Types in Java**

Java contains eight primitive data types, six of which are used for storing numbers of various lengths. The bit depth, or amount of bits in memory allocated for a variable of that type, determines the maximum and minimum values the data type can store. For example, the int Fibonacci method used in this lab could not correctly calculate Fibonacci numbers over 47, because those values are too large to be stored in an int variable. The numerical data types are: 

* byte, a one-byte (8-bit) signed two's complement integer. Since *n* bits have 2^*n*permutations, a byte can store up to 256 possible unique values. Because the byte data type is signed, it stores integers from -128 to 127, inclusive.
* short, a 16-bit signed two's complement integer. 16 bits have 65536 possible unique permutations, so a short can store integers from -32,768 to 32,767, inclusive.
* int, a 32-bit signed two's complement integer. 32 bits have 4294967296 possible unique permutations, so an int can store integers between -2,147,483,648 and 2,147,483,647 (inclusive). Ints tend to be the most commonly used integer data type in Java, because they provide a middle ground between short and long.
* long, a 64-bit signed two's complement integers. Because 64 bits have approximately 1.84E19 permutations, a long can store numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (inclusive). Longs are used when calculations involving very large numbers are necessary.
* float, a 32-bit floating-point number. Floats can store numbers with a decimal component in single precision.
* double, a 64-bit floating-point number. Doubles are twice as long as floats and therefore can store decimal values in double-precision.

The two remaining primitive data types are boolean, which stores a single true or false value, and char, which stores a single Unicode character.

**Deliverable Three: Pitfalls and Problems.**

The main issue I struggled with in this laboratory assignment was the number of test cases that had to be run. In order to test both algorithmic techniques and data types five times with the numbers 1, 5, 10, 15, 20, 25, 30, 35, 40, and 45, 200 individual test cases needed to be run. I solved this issue by automating the test cases using a perl script. Unfortunately, I was not very familiar with the perl scripting language before this laboratory assignment, but I was able to get some instruction from my colleague Max Cline. With help from Max, I was able to write a simple script to automatically run the necessary test cases, which I distributed to my lab partners.  
  
Another problem that needed to be solved in order to complete this lab assignment was the statistical analysis of experimental results. While there are existent software tools for conducting statistical analysis, including calculating the arithmetic mean as required for this assignment, I chose to write my own Java program. I made this choice primarily because I thought writing a simple statistical analysis tool would be an interesting and educational experience. A number of problems had to be solved in order to make this tool work correctly, but with advice from my colleagues and information from the Internet, I was able to create a working statistical analysis tool.  
  
My group also had some difficulty organizing the division of labor among its members. In order to solve this problem, we created a document on Google Docs, which we used to coordinate what tasks were assigned to which group members and to track their completion status. We also used this document for sharing the software tools we used to complete the laboratory assignment, including test scripts and my statistical analysis program. I found this to be very useful, and will probably use a similar system to facilitate collaboration in the future.   
  
I had seriously considered using a purpose-built project management system, such as GitHub's issue tracking tool, but I decided that the time it would take to set up such a system would outweigh the potential gains in productivity. Additionally, I knew my colleagues were not familiar with such tools, and the time taken for them to learn how to use these systems would probably not be worth it. In the end, I decided it was more important to focus on the project.  
  
The final major issue I had to deal with involved presenting our experimental data. I had created a graph showing the time taken to run wthe various Fibonacci implementations, but the recursive implementation ran for so long on high Fibonacci numbers that the bars for the iterative implementations were not even visible on the graph. To solve this problem, I chose to present the data using a logarithmic rather than linear scale.

**Analysis of Experimental Results**

Based on my analysis of the results of the experiment, it appears that the iterative Fibonacci implementation has similar performance regardless of the Fibonacci number being calculated, while the recursive implementation takes longer and longer as the Fibonacci number being calculated increases. I hypothesize that this is because as the Fibonacci number gets higher, the number of fib() methods called increases exponentially, while the iterative implementation always contains the same number of method calls.

The performance of the long and int implementations seemed to be roughly the same. While there was some variance in speed, it did not seem to follow a specific trend and appeared to be caused by background system processes.

However, there was a major difference in output between the long and int implementations. Because the int data type has a lower maximum value, it is unable to calculate Fibonacci numbers above the 47th, while the long implementation can calculate those Fibonacci numbers. The int implementation frequently returns an incorrect negative number for Fibonacci numbers above the 47th, presumably due to the sign bit being set by mistake.