Making Recursive Fibonacci Efficient

By: Ethan J. Nephew

Introduction

The recursive Fibonacci function doesn't break under a larger parameter, but it does take longer to complete the recursive stack. This paper is meant to examine why this phenomenon takes place and how a solution can be crafted.

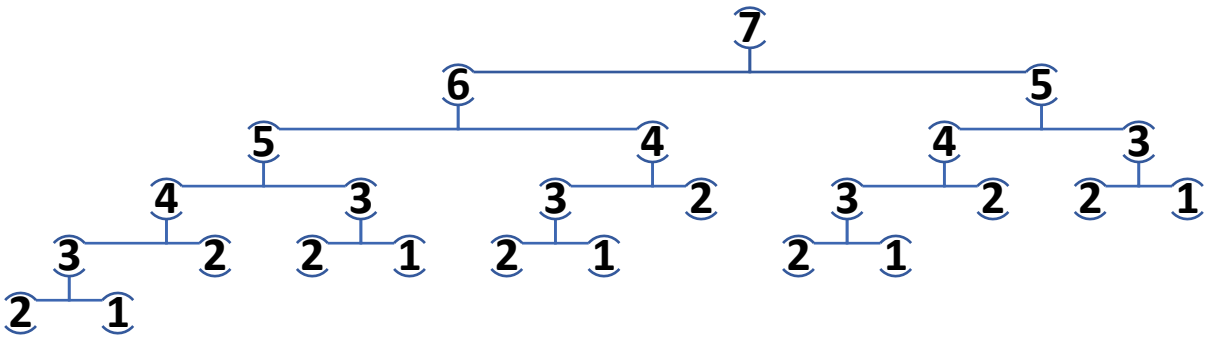
The Method

```
public static long fib(long n) {
   if (n == 0)
      return 0;
   if (n == 1)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
}
```

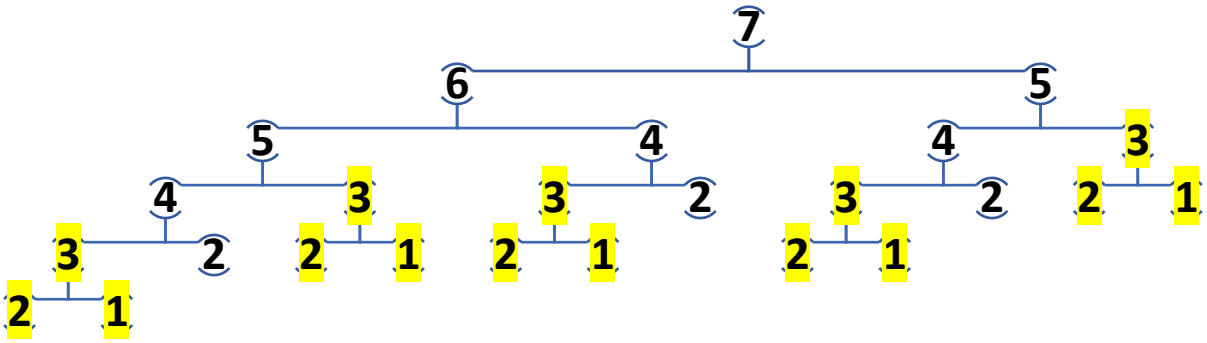
A recursive Fibonacci function can consist of two base cases and a recursive call. The first base case is if the parameter is zero. If the parameter is one, then the function will return zero. Likewise, the second base case is if the parameter is one – this will return one too. The last portion of the method block will be the recursive call that will be used to calculate the sum of the previous two Fibonacci numbers and return them.

Lifting the Veil

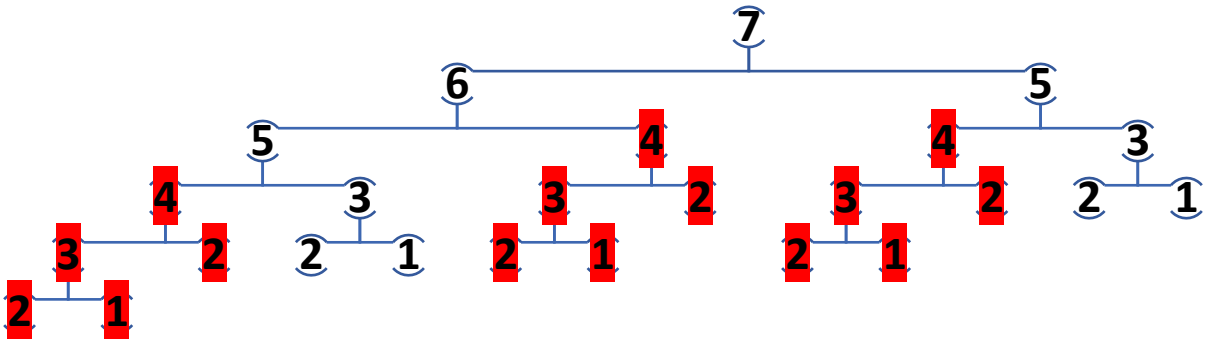
Generally, how a recursive method behaves can be described in the form of a tree. The computer is tasked with exploring each branch of the tree individually and when every branch is completely explored, only then can the method be resolved. For the method in question to be resolved it must reach the base case for every limb of the tree.



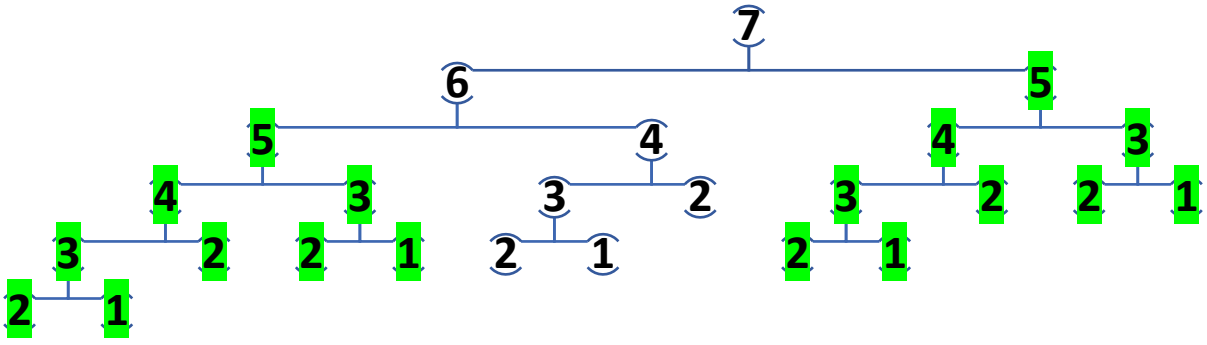
The above tree is a visualization of how the method behaves when seven is passed as a parameter. Something that can be observed is that there is a lot of repeated work taking place.



The three sub-tree is calculated five separate times.



The four sub-tree is calculated three separate times.



The five sub-tree is calculated two separate times.

These sub-trees are examples of redundant work taking place. Therefore, the method is inefficient.

Time Complexity

This behavior can be described in terms of it's time complexity. For every non-base case branch base that occurs, there are two branches that must be resolved for that branch base to be resolved. This results in a time complexity that is exponential or O(2^n). The n in the expression is the parameter that will be passed to the method. So how many steps will a computer take to resolve fib(50)?

2^50 = 1,125,899,906,842,624

This is a lot of work for a computer to handle. Many of these individual calculations signify a case a redundant labor taking place. One way of reducing the time complexity of the problem is by storing the individual base branch calculations in a quick to access data structure, such as a HashMap.

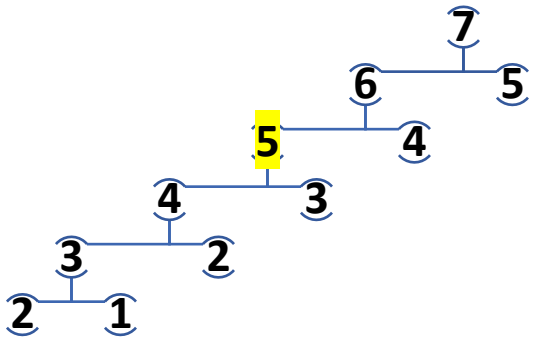
```
public static Map<Long, Long> hashMap = new HashMap<>();

public static long fibMemoization(long n) {
   long fibNumber = 0;
   if (n <= 1) {
      return n;
   } else if (hashMap.containsKey(n)){
      return hashMap.get(n);
   } else {
      fibNumber = fibMemoization(n - 1) + fibMemoization(n - 2);
      hashMap.put(n, fibNumber);
      return fibNumber;
   }
}
```

The inclusion of the HashMap can be viewed as the addition of a base case and the addition of a storage statement. The inclusion of this data structure results in the reduction of the time complexity of the method. By eliminating every specific case of redundant work that the computer must perform, calculations that would otherwise take hours, can be accomplished in milliseconds. In this case, fib(50) can now be resolved in 99 method calls.

This process is referred to as memorization. In this case, memorization has been used to reduce an exponential time complexity of O(2^n) to a generalized to O(2n). Changing the method from an exponential to a linear time complexity results in a substantial gain of efficiency when a large parameter is used.

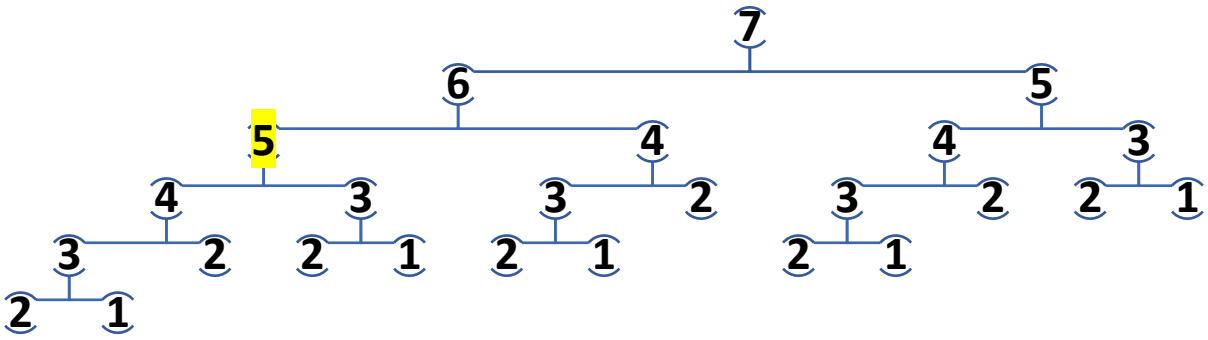
Exploring the Behavior

This is a visualization of how fib(7) is calculated by the computer.



fib(7) -> fib(6) -> fib(5) fib(4) -> fib(3) -> fib(2)

When the stack returns up, each branch base has the necessary information stored in the HashMap. For example, when the five node is reached, the three node will have already been calculated.



Previously, when the five node was reached on the return stack, fib(3) would have to be recalculated, however, it can be observed in the visualization of the memorized version of the tree, fib(3) was already calculated and saved. This reduction in redundant labor improves the performance of the algorithm drastically.