CS 240 Project 3

Alex Vondrak

Fall 2011

Overview

Hash tables are insanely useful in everyday coding. There's a lot you can do with them. In this project, we'll explore their use & effectivness in a process called *memoization*, while also getting some practice with recursion.

Background

Fibonacci Numbers

We'll be making use of a famous recursively-defined sequence of numbers which you're probably familiar with: the *Fibonacci* sequence. In mathematical notation, the n^{th} Fibonacci number (denoted $F(n)$) can be computed by the following recurrence relation.

$$\begin{aligned} F(1) &= 1 \\ F(2) &= 1 \\ F(n) &= F(n-2) + F(n-1) \end{aligned}$$

In English, the n^{th} Fibonacci number is equal to the sum of the previous two Fibonacci numbers, and the sequence starts out with the two numbers 1 and 1.* Thus, the first 10 Fibonacci numbers are

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55$$

There's plenty more information about the characteristics and importance of Fibonacci numbers in the Wikipedia article available at `http://en.wikipedia.org/wiki/Fibonacci_number`.

Memoization

Memoization is an optimization technique we can use to speed up the execution of a program at the cost of increased memory usage. Memoization avoids repeating the calculation of function calls we've seen previously. To do this, we keep a hash table that maps previous input values to the output value that was generated the first time we did the computation—we *cache* the results of previous computations.

*Some sources like to start the sequence at 0 and 1. It doesn't matter much for our purposes. For this project, let's just stick with starting at 1 and 1.

In this project, we'll compute Fibonacci numbers both with and without memoization. At a high level, we can think of how we'd recursively caluclate, say, $F(6)$:

$$\begin{aligned} F(6) &= F(4) + F(5) \\ &= F(2) + \underline{F(3)} + \underline{F(3)} + F(4) \\ &= 1 + F(1) + F(2) + F(1) + F(2) + F(2) + \underline{F(3)} \\ &= 1 + 1 + 1 + 1 + 1 + 1 + F(1) + F(2) \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 8 \end{aligned}$$

Notice that we wind up computing the value of $F(3)$ three separate times. We also calculate $F(4)$ twice.

If we calculate $F(3)$ recursively, we spend time allocating call stack frames, invoking the recursive procedures, and deriving the sum—even if we've already calculated it before. However, if we memoize the Fibonacci algorithm, we instead keep a **`static`** hash table that persists between calls to the Fibonacci method. Any time we're asked to calculate some $F(n)$, we look in the hash table to see if we have the key n.

- If it isn't in our cache, we calculate the value as we normally would (in the Fibonacci algorithm, by recursive calls to F). We store the result of having done this in the hash table for the benefit of future function calls.
- If it is in our cache, that means we've already computed and stored the result of the computation we'd otherwise do, but we don't need to duplicate the work. Instead, we return the value associated with the key in the hash table. For instance, suppose we've already calculated $F(3)$ in a previous call by recursing down to $F(1) + F(2) = 1 + 1 = 2$. Any subsequent time we compute $F(3)$, we'll have the key-value pair $3 \mapsto 2$ in the hash table, so we can just return 2 immediately for future calls to $F(3)$.

What To Do

Code

Write three classes:

- The class `Unmemoized` will have a single **`static`** method called `fib` that takes in an **`int n`** and computes the $\texttt{n}^{\text{th}}$ Fibonacci number recursively. That way, a call to the unmemoized version of the Fibonacci algorithm looks like `Unmemoized.fib(n)`.
- The class `Memoized` will have a single **`static`** method called `fib` that takes in an **`int n`** and computes the $\texttt{n}^{\text{th}}$ Fibonacci number recursively, but also memoizes the results. That way, a call to the memoized version of the Fibonacci algorithm looks like `Memoized.fib(n)`.
- The class `TimeFibonacci` implements a `main` method (and any other helper methods you need for the sake of clear code) that performs the experiment described in the next section.

Experiment

With the memoized and unmemoized versions of the recursive Fibonacci algorithm, we'll run an experiment to see how much memoization actually helps us. You'll do this by timing how long it takes both methods to compute the same result.

For $\texttt{n} = 1, 2, 3, \ldots, 45$, compute the values of the $\texttt{n}^{\text{th}}$ Fibonacci number using both your regular and memoized versions of the algorithm.

- Verify that the results are equal no matter which version is used.
- Report the running time of each method individually in milliseconds. (*Hint:* use `System.currentTimeMillis`.)
- After computing each running time, report the average running times of both methods across all the values of `n` that were tested.

For example, the first few lines of output might look like

```
$ java TimeFibonacci
Unmemoized.fib(1) = 1 (took 0 ms); Memoized.fib(1) = 1 (took 1 ms)   ... equal
Unmemoized.fib(2) = 1 (took 0 ms); Memoized.fib(2) = 1 (took 0 ms)   ... equal
Unmemoized.fib(3) = 2 (took 0 ms); Memoized.fib(3) = 2 (took 1 ms)   ... equal
Unmemoized.fib(4) = 3 (took 0 ms); Memoized.fib(4) = 3 (took 0 ms)   ... equal
Unmemoized.fib(5) = 5 (took 0 ms); Memoized.fib(5) = 5 (took 0 ms)   ... equal
```

However, you might get different running times than the above. At the end, report the average running time for the unmemoized version (based on these few lines, $(0 + 0 + 0 + 0 + 0)/5 = 0$ms) and for the memoized version (based on these few lines, $(1 + 0 + 1 + 0 + 0)/5 = 0.4$ms).

Include the above output (when run from your computer) as a comment in your source code.

Finally, in your source code, include the answers to the following questions:

1. Which version ran faster on average? Why?
2. Why did we limit `n` up to only 45? What happens when we try larger values of `n`? (Play around with your code until you notice something break.)