**Fibonacci and a HashMap**

Recall the Fibonacci sequence 1, 1, 2, 3, 5, 8 . . . . and how it is calculated.

fib(n)

fib(n-2)

fib(n-1)

We saw before that the recursive version grew exponentially, or O(2n) because the recursive tree recalculated every value down to the base case.

fib(n-2)

fib(n-1)

fib(n-2)

fib(n-1)

**Fib 1**: Write the code for the recursive version here:

**Fib 2**: This iterative version, which uses arrays to store intermediate values, grows linearly, or O(n).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 1 |  |  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Write the code for this iterative version:

**Fib3** and **Fib4:** Storing intermediate values during recursion is the big idea in this lab. When we recur and store intermediate values, we get O(n) performance. Storing intermediate values is an example of "smart recursion," a.k.a. "dynamic programming." You will use it every day if you take AI or do USACO problems.

This lab uses System.nanoTime() to determine the relative Big-O performances of four different algorithms:

* Fib1 is recursive and does not store intermediate results
* Fib2 is iterative and stores intermediate results in an array
* Fib3 is recursive and stores intermediate results in a static ArrayList
* Fib4 is recursive and stores intermediate results in a static HashMap

Recall that a static field belongs to the class and does not get duplicated in each object. On the other hand, each object instantiated from that class has access to or shares the static field, which is exactly what you want when storing and retrieving intermediate values.

|  |
| --- |
| **object1 = new Fib3()** |
| **public int** fib() |
|  |
|  |
| **Object2 = new Fib3()** |
| **public int** fib() |
|  |

class Fib3

public static arrayList

public int fib()

Fib3’s and Fib4’s constructors need, if necessary, to instantiate the static field and add the first two Fibonacci terms to get started. Then the object’s fib method actually calculates the 40th Fibonacci number. Fib3 and Fib4 have base cases, namely, if a previous value is already in the arrayList or the hashMap, then return it. Else, calculate the next Fibonacci value and put it in the data structure. Finally, return the 40th Fibonacci number.

A sample run is below. Notice that every algorithm calculates the same value for the 40th Fibonacci number. Notice how storing intermediate results decreases the time needed.

**Sample Run**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Fib1, Recursive, no storing  
fib(40) = 102334155 (344219300 nanoseconds)  
  
Fib1 again with the same Fib1 object  
fib(40) = 102334155 (343141500 nanoseconds)  
  
Fib1 with a new Fib1 object  
fib(40) = 102334155 (350600600 nanoseconds)

Why are these times so large?

Why are they nearly the same?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Fib2, Iterative, stored in an array  
fib(40) = 102334155 (3200 nanoseconds)  
  
Fib2 again with the same Fib2 object  
fib(40) = 102334155 (600 nanoseconds)  
  
Fib2 with a new Fib2 object  
fib(40) = 102334155 (1600 nanoseconds)  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Fib3, Recursive, stored in a static arrayList  
fib(40) = 102334155 (59600 nanoseconds)  
  
Fib3 again with the same Fib3 object  
fib(40) = 102334155 (1200 nanoseconds)  
  
Fib3 with a new Fib3 object  
fib(40) = 102334155 (1100 nanoseconds)  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Fib4, Recursive, stored in a static hashMap  
fib(40) = 102334155 (97000 nanoseconds)  
  
Fib4 again with the same Fib4 object  
fib(40) = 102334155 (1900 nanoseconds)  
  
Fib4 with a new Fib4 object  
fib(40) = 102334155 (1900 nanoseconds)

Why is the third the same as the second?

Why is the second so much faster?

Why is the third so much slower?

Why is the second so much faster?