

## Lab 5: Fibonacci has Bad Internet

Tuesday February 4<sup>th</sup>

**Collaboration:** In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

**Setup:** Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab05 .
% cd lab05
```

**Grading:** For 3 points, do all the tasks through (2.e). For 4 points, finish the rest of the lab. Submit the file `lf.c0` in Autolab.

## Lagged Fibonacci

The regular Fibonacci numbers are given by the function  $F(i)$  where  $F(i) = i$  for  $i \in [0, 2)$  and where  $F(i) = F(i - 1) + F(i - 2)$  for  $i \geq 2$ . More explicitly:

$$\begin{cases} F(0) &= 0 \\ F(1) &= 1 \\ F(i) &= F(i - 1) + F(i - 2) \quad \text{for } i \geq 2 \end{cases}$$

The *lagged* Fibonacci numbers make use of two additional parameters  $j$  and  $k$ , where  $0 < j < k$ . They are defined by the function  $LF(i)$  where  $LF(i) = i$  for  $i \in [0, k)$  and where  $LF(i) = LF(i - j) + LF(i - k)$  otherwise. Here is a C0 function that implements this definition (note that `j` and `k` do not change during the computation):

```
1 int LF(int i, int j, int k)
2 //@requires 0 < j && j < k;
3 //@requires i >= 0;
4 {
5   if (i < k) return i;
6
7   int res = 0;
8   res += LF(i-j, j, k);
9   res += LF(i-k, j, k);
10  return res;
11 }
```

(1.a) The regular Fibonacci numbers can easily be computed on the basis of  $LF$ . Can you do so?

$$F(i) = LF(\underline{\hspace{10em}})$$

While `LF` computes the desired result accurately, it is quite slow for large inputs! This is because it repeats many of the sub-computations over and over again, which is slow, inefficient, and redundant. Can you see why?

## Memoization

To avoid these redundant computations, we introduce a data structure known as a *memo table* — in our case it will be an array of integers. The idea is that, the first time we compute the lagged Fibonacci number for  $i$ , we store it at index  $i$  in the memo table. Next time we need the  $i$ -th lagged Fibonacci number, we simply look it up in the table.

Saving the result of computations that we would do over and over is called *memoization*. This requires a bit of extra space in the form of the memo table, but it can save an enormous amount of time because it avoids recomputing these results over and over. This is known as a *space-time trade-off*, a really important concept in computer science.

- (2.a) Using the slow `LF` function, write a specification function `is_memo_table` that checks that, for all  $i \in [0, \text{len}]$  (note the inclusive upper bound!), `M[i]` is *either* 0 or `LF(i, j, k)`.

```
bool is_memo_table(int[] M, int len, int j, int k)
//@requires 0 <= len && len < \length(M);
```

1.5pt

- (2.b) Write a new recursive function `lf_memo`, which returns the same results as `LF` but uses a memo table to avoid re-computing results by writing them into an array of integers.

Before the function does any work, it should check whether the result is already in the memo table, and if so just return that value. If you do have to compute the number, store it in the memo table before returning, so that future calls will not have to do the same work again.

**DANGER!** Do not use the function `LF` in your `lf_memo` function outside of contracts! This will reintroduce the problem where we perform many redundant computations, which defeats the entire purpose of our memo table! At this point, `LF` has become a specification function for us.

```
int lf_memo(int[] M, int i, int j, int k)
//@requires 0 < j && j < k;
//@requires 0 <= i && i < \length(M);
//@requires is_memo_table(M, i, j, k);
//@ensures is_memo_table(M, i, j, k);
//@ensures \result == LF(i, j, k);
```

- (2.c) Using `lf_memo` as a helper function, write the function `fast_lf(i, j, k)` that initializes a new array and calls the helper to compute the lagged Fibonacci number.

```
int fast_lf(int i, int j, int k)
//@requires 0 < j && j < k;
//@requires 0 <= i;
//@ensures \result == LF(i, j, k);
```

- (2.d) Check that your `fast_lf` function works by running it in coin with `-d` for some small Fibonacci numbers. Then run it in coin without `-d` so that you actually notice a speedup. Running with `-d` is slow as `LF` is called in the postcondition.

- (2.e) What is the 54,321<sup>st</sup> Fibonacci number (mod  $2^{32}$ , of course)? What is the 100,000<sup>th</sup> lagged Fibonacci number with  $j = 1$  and  $k = 25$ ?

3pt

Now would be a good time to compare notes with your neighbors if you haven't done that already. How do their functions look the same? How do they look different?

## Timing code

In Unix, there is a way to determine the actual running time of a program. You use the `time` command followed by the program name (and its arguments) that you want to time. For example, to time an executable `a.out` in your current directory, you would enter:

```
% time ./a.out
Testing with n = 1000... Done. 0
4.602u 0.015s 0:04.63 99.5% 0+0k 0+0io 0pf+0w
```

The first number (the one with a `u` after it) is the best one to track for this activity. The “u” is for *user*, which is closer to what we want than the system time (the amount of time the program handed control over to the operating system) or the wall clock time.

- (3.a) In the lab directory, run the following commands:

```
% make LFtest
% make memotest
```

This will create two programs, `LFtest` and `memotest` that take an argument `n`, and print the result of lagged fibonacci performed on the arguments (`n`, `1`, `2`) — these are actually just the regular Fibonacci numbers! You can use these to time both the specification function and your new function. For example, to time the LF specification function using input 10, you would enter:

```
% time ./LFtest 10
```

Your job is to determine the asymptotic complexity (runtime) of each program expressed using big-O notation as a function of  $n$ , in its simplest, tightest form.

You can determine the complexity of a program by running it for varying values of  $n$  and then plotting your results (or looking for obvious patterns).

Keep in mind that the specification function should have a *significantly* worse asymptotic complexity than your `fast_LF` implementation.

4pt