15-122: Principles of Imperative Computation — Spring 2023

Lab 05: Fibonacci Has Bad Internet — Tuesday February 7th

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 other students in this lab!

Setup:
Download the lab handout and code from the course website https://web2.qatar.cmu.edu/∼mhhammou/15122-s23/schedule.html, and move it to your private directory in your unix.qatar.cmu.edu machine. Following that create a directory, move the handout to it, and unzip the handout file by executing the following commands:

```
% mkdir lab_05
% mv 05-handout.tgz lab_05
% cd lab_05
% tar -xvf 05-handout.tgz
```

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 $i >= 2$. 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):

```
int LF(int i, int j, int k)
//@requires 0 < j && j < k;
//@requires i >= 0;
{
  if (i < k) return i;

  int res = 0;
  res += LF(i-j, j, k);
  res += LF(i-k, j, k);
  return res;
}
```

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

$F(i) = LF($______________________________________$)$

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` $\in [0, \texttt{len}]$ (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 54,321st Fibonacci number (mod 2^{32}, of course)? What is the 100,000th lagged Fibonacci number with $j = 1$ and $k = 25$?

3pt How do their functions look the same? How do they look different?

(2.f) What is the worst-case asymptotic complexity of the call `LF(n, 1, 2)`? A way to approach this problem is to estimate the number of recursive calls during this computation: try drawing a diagram that visualizes this!

(2.g) What is the worst-case asymptotic complexity of the call `fast_LF(n, 1, 2)`? A way to approach this problem is to count how many times this call will write to the memo table. Can you see why?

4pt

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
real    0m1.027s
user    0m0.952s
sys     0m0.074s
```

The second number (*user* time) is the best one to track for this activity. It 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 "real" time, which is the sum of the two.

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

```
% cc0 -o LFtest lf.c0 LFtest.c0
% cc0 -o memotest lf.c0 memotest.c0
```

This will create the executables `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
```

Timing `LFtest` and `memotest` with increasing values of `n` will allow you to confirm the big-O complexity of the functions `LF` and `fast_LF` you found earlier.

This is a common approach: You can first analyze the asymptotic complexity of a program by looking at its code. Then you can confirm your hypothesis by running the program for varying values of n and plotting your results.