15-122: Principles of Imperative Computation

Lab Week 4 — Rob Simmons

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. Feel free to talk with your neighbors about the problems if you get confused or stumped!

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

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

Grading: For partial credit, do all the tasks through (1.c). For full credit, also complete task (2.a).

Lagged Fibonacci

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

To calculate the *lagged* Fibonacci numbers, we give two additional numbers l and k, where $0 < k < l$. Then we can say that the lagged Fibonacci numbers are given by the function $LF(i)$ where $LF(i) = i$ for $i \in [0, l)$ and where $LF(i) = LF(i-l) + LF(i-k)$.

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

(1.a) Copy the code above into the `lf.c0` file in your `lab-lf` directory. Use `coin` to confirm that the regular Fibonacci numbers are just the lagged Fibonacci numbers where $k = 1$ and $l = 2$:

```
% rlwrap coin -d lf.c0
C0 interpreter (coin)
Type '#help' for help or '#quit' to exit.
--> LF(1,2,1);
1 (int)
--> LF(1,2,2);
1 (int)
--> LF(1,2,3);
2 (int)
--> LF(1,2,4);
3 (int)
--> LF(1,2,5);
5 (int)
```

Using `rlwrap` to wrap `coin` as shown above will make your life much easier: `rlwrap` gives you the ability to press the up key to see and edit the last thing you wrote, much like you can do on the command line.

(1.b) For what values of n does it start getting too slow to calculate `LF(1,2,n)` in less than a few seconds?

The reason this calculation gets so slow for such low values of n is because the recursive calls in `LF` do a lot of repeated computation.

(1.c) Trace the execution of LF(1,2,5) in full on a large sheet of paper. Look over it and make sure you understand what it is decribing. How many times in this trace do we (re) compute `LF(1,2,2)`?

Memoization

Because of the re-computation you saw in (1.c), this is a case where we can save ourselves a *lot* of time by using a little space in the form of a *memo table*.

(2.a) Using `LF` as a specification function, write a specification function that checks that, for all `i` $\in [0, \texttt{len}]$ (note the inclusive upper bound!), `A[i]` is *either* 0 or `LF(k, l, i)`.

```
1  bool is_memo_table(int[] A, int k, int l, int len)
2  //@requires 0 <= len && len < \length(A);
```

(2.b) Write a new recursive function `lf_memo`, which uses a memo table to avoid re-computing values by writing them into an array of integers.

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

```
1  int lf_memo(int[] A, int k, int l, int n)
2  //@requires 0 < k && k < l;
3  //@requires 0 <= n && n < \length(A);
4  //@requires is_memo_table(A, k, l, n);
5  //@ensures is_memo_table(A, k, l, n);
6  //@ensures \result == LF(k, l, n);
```

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

```
1  int fast_lf(int k, int l, int n)
2  //@requires 0 < k && k < l;
3  //@requires 0 <= n;
4  //@ensures \result == LF(k, l, n);
```

(2.d) Check that your `lf_memo` function works by running it 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 $54,321^{\text{st}}$ Fibonacci number (mod 2^{32}, of course). What is the $100,000^{\text{th}}$ lagged Fibonacci number with $k = 1$ and $l = 25$?

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?

(2.f) Trace out the computation of `fast_lf(1,2,5)` mimicing the provided trace of `LF(1,2,5)`. Include the state of the array.