

# CS 12 • Winter 2003

## In-lab Programming Exercise 2

### Recursion

wagner@cs.ucr.edu  
Department of Computer Science and Engineering  
University of California, Riverside

January 20, 2003

This second assignment is entirely devoted to recursion and is divided in four parts, all of which are mandatory. Remember that you must write an explanation section for all your work. In particular, you should use that section to answer the questions awaiting you below.

All four parts deal with the Fibonacci sequence. Fibonacci was a mathematician born in the Italian city of Pisa (where the famous Leaning Tower is) around the year 1175. His real name was Leonardo of Pisa but because his father's name was Guglielmo Bonaccio, Leonardo liked to call himself Fibonacci, short for *filius Bonacci*, which is latin for 'the son of Bonacci.' You can find more about him on the web, for example, here.

As it turns out, the history behind his famous sequence is quite interesting. Apparently, Fibonacci wanted to know how fast rabbits can breed, so he made some simplifying assumptions and created a model to describe their mating behavior. You can find more about this fascinating aspect of his research here. Our interest in the Fibonacci sequence stems from the fact that it can be described recursively:

$$F(n) = F(n - 1) + F(n - 2) \quad (n \geq 3).$$

Of course, we need starting values and the standard choices are  $F(1) = F(2) = 1$ . These give rise to the sequence  $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots\}$ . Although not obvious at first, it turns out that the Fibonacci numbers grow exponentially with  $n$ , that is, they grow *very* quickly. Your task in this exercise will be to explore some of the issues involved in computing the Fibonacci numbers.

## Part 1

(a) Write a program to **recursively** compute the Fibonacci numbers. You should first write a separate function `int FiboRec(const unsigned int n)` which returns the value of the  $n$ -th Fibonacci number. Once you have that, write your `main()` function to compute and print the first 50 Fibonacci numbers. Compute and print them one at a time; do **not** compute them all first. In other words, the output of your program should be similar to this:

| $n$ | $F(n)$         |
|-----|----------------|
| 1   | 1              |
| 2   | 1              |
| 3   | 2              |
| 4   | 3              |
| ... | ...            |
| 50  | whatever it is |

where each line is printed as its corresponding  $F(n)$  is computed. Be warned that it may take a while for the last several numbers to be computed. In particular, you may have to manually abort your program (I had to do so when trying to compute  $F(50)$ ). You should take a moment to appreciate how inefficient a recursive computation of the Fibonacci numbers is.

(b) Once you have a table of the Fibonacci numbers computed recursively, take a closer look at it. Notice that some numbers are negative. How can that be considering that every Fibonacci number is strictly positive? You should try to figure that out by yourself without asking the TAs or your friends. There's an important lesson to be learned here and it's important that you find it out by yourself. If this is a bug of some kind, what would you suggest to fix it?

(c) Now comment away the statements which correspond to the base case of the recursion and run your program. What happens? Why?

## Part 2

(a) Now do exactly as you did in Part 1 above, but do so **iteratively**. If you did things the right way, you only need to change in your `main()` which function you're calling, `FiboRec()` or `Fibolter()`. Can you tell (subjectively) how much faster the iterative version is compared to the recursive one?

(b) Once again, after you have a table of the Fibonacci numbers (now computed iteratively), take a closer look at them. Are there negative numbers still? Why?

### Part 3

Using the functions you wrote for parts 1 and 2 above, write another program that measures how long each computation takes. In other words, the goal is to print a table like the following, where time is measured in clock units (an arbitrary amount which is computer dependent):

| $n$ | $F(n)$ iter | time (iter) | $F(n)$ rec | time (rec) |
|-----|-------------|-------------|------------|------------|
| 1   | 1           | 0           | 1          | 0          |
| 2   | 1           | 0           | 1          | 0          |
| 3   | 2           | 0           | 2          | 0          |
| 4   | 3           | 0           | 3          | 0          |
| ... | ...         | ...         | ...        | ...        |
| 35  | 9227465     | 0           | 9227465    | 110        |
| ... | ...         | ...         | ...        | ...        |
| 40  | 102334155   | 0           | 102334155  | 1196       |
| ... | ...         | ...         | ...        | ...        |
| 50  | ...         | ...         | ...        | ...        |

For your convenience, I'm giving you the *entire* `main()` except for the two functions you need to implement. Am I a nice guy or what? There's a catch, however. You *still* need to explain what is happening in `main()` in your explanation section, and asking the TAs for help on that is a no-no. You're going to have to do some digging in the book to understand this code — that is intentional. By the way, as usual, free code can be downloaded from the course web page so you don't have to copy all this.

```
#include <iostream>
#include <time.h>

const int MAXN = 50;

const int N_WIDTH = 5;
const int LARGEWIDTH = 12;

int fibo_rec = 1, fibo_iter = 1;
double elapsed_rec = 0, elapsed_iter = 0;

int FiboRec(const unsigned int n);
int FiboIter(const unsigned int n);
```

```

void PrintHeader ();
void PrintOutput(const unsigned int n);

int main ()
{
    std::cout.setf(std::ios::fixed);
    std::cout.precision(0);

    PrintHeader();

    clock_t start;
    clock_t finish;

    start = clock();
    finish = clock();
    double delay = (double)(finish - start);

    for (int n = 1; n <= MAXN; n++)
    {
        start = clock();
        fibo_iter = FiboIter(n);
        finish = clock();

        elapsed_iter = (double)(finish - start);
        elapsed_iter -= delay;

        start = clock();
        fibo_rec = FiboRec(n);
        finish = clock();

        elapsed_rec = (double)(finish - start);
        elapsed_rec -= delay;

        PrintOutput(n);
    }

    return 0;
}

void PrintHeader ()

```

```

{
    std::cout.width(N_WIDTH);
    std::cout << "n" << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << "F(n)_iter" << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << "time_(iter)" << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << "F(n)_rec" << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << "time_(rec)" << std::endl;
}

void PrintOutput(const unsigned int n)
{
    std::cout.width(N_WIDTH);
    std::cout << n << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << fibo_iter << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << elapsed_iter << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << fibo_rec << "\n";

    std::cout.width(LARGE_WIDTH);
    std::cout << elapsed_rec << std::endl;
}

int FiboRec(const unsigned int n)
{ /* Ok... this is up to you to write */ }

int FiboIter(const unsigned int n)
{ /* Ok... this is up to you to write */ }

```

## Part 4

In this part, I want you to write yet another program using your recursive function to compute the Fibonacci numbers but this time I want you to print the number of recursive calls made in the computation of a given Fibonacci number. For example, in the recursive computation of  $F(4)$ , we need  $F(3)$  and  $F(2)$ . But  $F(3)$  needs  $F(2)$  and  $F(1)$ , while  $F(2)$  is a base case, just as  $F(1)$  is. Thus, in order to compute  $F(4)$  recursively we need to make 4 calls: one  $F(3)$ , two  $F(2)$ 's and one  $F(1)$ .

Your program should output a table like this:

| $n$ | $F(n)$ | num of rec calls |
|-----|--------|------------------|
| 1   | 1      | 1                |
| 2   | 1      | 1                |
| 3   | 2      | 2                |
| 4   | 3      | 4                |
| ... | ...    | ...              |
| 50  | ...    | ...              |

What can you say about how fast the number of recursive calls is growing as we increase the value of  $n$ ?

