Introduction to Mathematical Software 4th Exercise Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
PD Dr. Ulf Lorenz
Christian Brandenburg

winter term 2009/2010
23/11/2009

Important Notice

Before continuing with this exercise sheet, you should have finished **Exercises 2.1 and 2.2**, as well as **Exercise 3.1**. If you had difficulties with the recursion in **Exercise 3.2**, have a look at **Exercise 4.1** and go back to **Exercise 3.2** afterwards.

The exercises marked with ** are advanced exercises for students who already have programming experience but, of course, everybody can give them a try.

If you still have difficulties with or questions about the material covered in these exercises, don't hesitate to ask questions or join the office hours!

Exercise 4.1 Recursion – Fibonacci Numbers

In this exercise we consider the recursive definition of the Fibonacci numbers, which is given by

$$\begin{aligned} \mathrm{Fib}(n+2) &= \mathrm{Fib}(n+1) + \mathrm{Fib}(n) \\ \mathrm{Fib}(0) &= 1 \\ \mathrm{Fib}(1) &= 1 \end{aligned}$$

for $n \in \mathbb{N}$.

From this definition, we can easily derive a recursive algorithm (given in pseudocode):

Function 1 Fib(n)

```
if n < 2 then
  return 1
else
  return Fib(n-1) + Fib(n-2)
end if
```

As you can see, for $n \geq 2$, the function `Fib` calls itself two times with smaller values of n, i.e. `Fib` calls itself recursively. Thus, the recursive Fibonacci algorithm is a direct implementation of the mathematical definition for the Fibonacci numbers. The *base case* of the recursion is when $n = 1$ or $n = 0$, in which case `Fib` returns 1 and, most importantly, *does not* call itself anymore (otherwise, the recursion would never stop).

The *recursion tree* for the computation of Fibonacci's number with $n = 4$ with the algorithm above is given by Figure 1. The root (at the top of the tree!) denotes the initial call of the `Fib`-function, the downward arcs denote the `Fib`-function calling itself with the corresponding values of n. The boxes *below* the function name denote base cases, the boxes to the right with equality sign denote the return values.

a) Using pen and paper, create the *recursion tree* for `Fib(6)`.

1. Which value does n have in each recursive call; what is the value of `Fib(n)`?
2. Which are the base cases?
3. In which order is the tree traversed?
4. How often does `Fib` call itself?

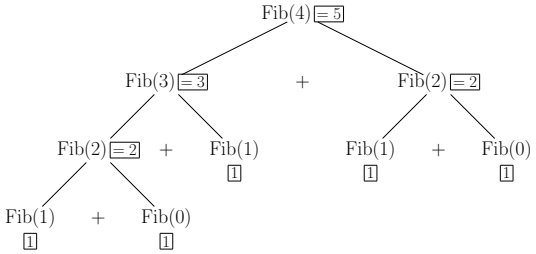

Figure 1: Recursion Tree for Fib(4)

b) Implement the `Fib` algorithm in C. Why is this algorithm a bad way to compute the Fibonacci numbers from an algorithmic point of view?

c) If you haven't done so already, solve **Exersise 3.2**.

Solution:

a) The *recursion tree* for the computation of Fibonacci's number with $n = 6$ with the algorithm above is given by Figure 2.

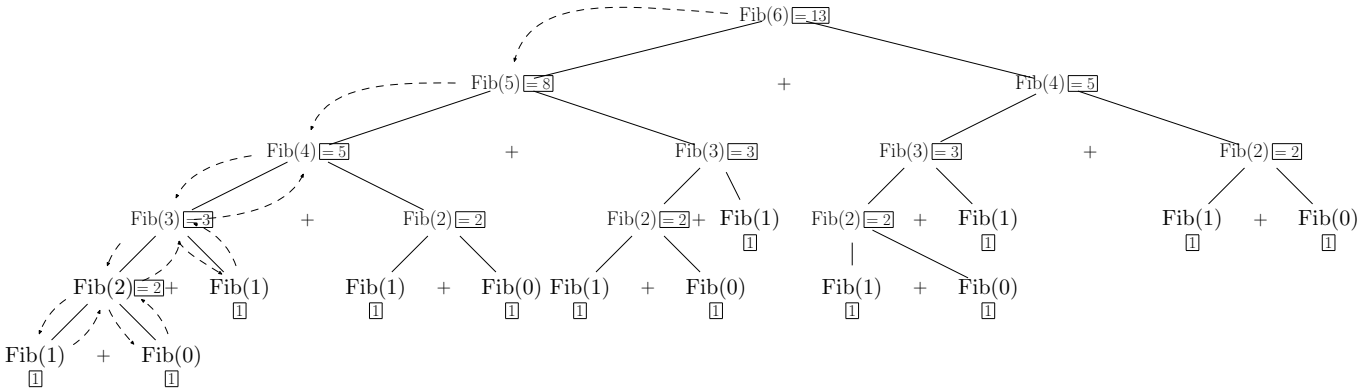

Figure 2: Recursion Tree for Fib(6)

1. you can see the values from the recursion tree.
2. the base cases are exactly the leaves of the recursion tree, where n has either the value 1 or 0.
3. the tree is traversed *depth first*, i.e. we first move down to the leftmost leave, we then go back up one level to its parent and immediately go down to the next leave.
4. `Fib(6)` calls itself 24 times.

b)

```
#include <stdio.h>

unsigned int fib(unsigned int n)
{
  if (n < 2)
    return 1;

  return fib(n-1) + fib(n-2);
}

int main(void)
{
  int n = 6;
```

```
  printf("The %d-th Fibonacci number is %d.\n", n, fib(n));
}
```

The algorithm is very inefficient as the number of recursive calls to `Fib(n)` grows exponentially in `n`. It is much more efficient to compute the Fibonacci numbers iteratively, as here the number of iterations is linear in `n`.

c) see the solution of **Exersise 3.2**.

Exercise 4.2 C++ Classes – BiggerInt

In the last lecture, you saw the implementation of a class for representing larger integers, `BiggerInt`, with an implementation of the `BiggerInt *add` class member function.

a) Obtain the file `BiggerInt.cc` from the course webpage. Compile and run it. Browse the source code and try to understand what happens. The comments should help you. If something still is unclear, don't hesitate to ask. Check that everything works as expected.

b) Implement the remaining class member functions `BiggerInt *sub`, `BiggerInt *mul`, and `BiggerInt *div`.

c) ** Replace `unsigned long low_bits;` by `std::vector<unsigned int> low_bits;` to represent arbitrarily large integers. The vector should dynamically adapt its size to hold any integer. Change the class member functions `BiggerInt`, `BiggerInt *sub`, `BiggerInt *mul`, and `BiggerInt *div` accordingly.

d) ** Adapt the recursive Fibonacci algorithm from **Exercise 4.1** for your new `BiggerInt` class and test it with some large numbers. Implement an iterative version of the Fibonacci algorithm. What do you observe?