Comparison of a few implementations of the Fibonacci sequence

Li-Thiao-Té Sébastien

July 10, 2019

1 Introduction

In this document, we use Lepton to compare a few implementations of the computation of the Fibonacci sequence (OCaml, Python and C). This is intended to demonstrate Lepton's features, such as

- embedding source code inside a document (literate programming)
- code restructuring for better documentation
- embedding executable instructions

From the point of view of (scientific) applications, these features make it possible to

- include and distribute the actual source code
- distribute the instructions necessary for compiling and running the code
- certify that the embedded source code is correct by executing it
- running analysis scripts and generating figures
- embedding different programming languages in the same document / same platform
- simplifying code re-use by distributing only a single file.

Note that code chunks are colorized according to the rules of the `Pygments` Python beautifier.

2 Problem statement

The Fibonacci sequence is defined as the sequence of integers F_n such that

$$F_0 = 1 \tag{1}$$

$$F_1 = 1 \tag{2}$$

$$F_n = F_{n-1} + F_{n-2} \tag{3}$$

The goal is to define function that returns F_n given the integer n.

3 Implementation

The proposed implementations in this document are taken from the Rosetta Code project `https://rosettacode.org/wiki/Fibonacci_sequence`. Note that only the OCaml implementation is actually correct.

3.1 Recursive implementation in OCaml

We define a `fibonacci` function in OCaml in the following code chunk. The contents of this chunk are sent by Lepton to an instance of the OCaml interpreter, and the output (the type of the `fibonacci` Ocaml object) is captured below automatically.

Code chunk 1: «ocaml»

```
let rec fibonacci = function
  | 0 -> 1
  | 1 -> 1
  | n -> fibonacci (n-1) + fibonacci (n-2)
;;
```

Interpret with `ocaml`

```
val fibonacci : int -> int = <fun>
```

To check that the function is correct, let us ask OCaml for the first few numbers in the sequence.

Code chunk 2: «ocaml (part 2)»

```
fibonacci 0;;
fibonacci 1;;
fibonacci 2;;
fibonacci 3;;
```

Interpret with `ocaml`

```
- : int = 1
- : int = 1
- : int = 2
- : int = 3
```

3.2 Iterative implementation in Python

We define a `fibIter` function in Python in the following code chunk. The contents of this chunk are sent by Lepton to an instance of the Python interpreter. There is no output on success.

Code chunk 3: «python»

```
def fibIter(n):
    if n < 2:
        return n
    fibPrev = 1
    fib = 1
    for num in xrange(2, n): fibPrev, fib = fib, fib + fibPrev
    return fib
```

Interpret with python

```
```

To check that the function is correct, let us ask Python for the first few numbers in the sequence.

Code chunk 4: «python (part 2)»

```
for i in range(0,4): print fibIter(i),
```

Interpret with python

```
0 1 1 2
```

3.3 Iterative implementation in C

In a compiled language such as C, we need to define the function `fibC` first, then include it in a program to use it. Let us start with the function definition.

Code chunk 5: «fibC»

```
long long int fibC(int n) {
	int fnow = 0, fnext = 1, tempf;
	while(--n>0){
		tempf = fnow + fnext;
		fnow = fnext;
		fnext = tempf;
		}
		return fnext;
}
```

We include this in a program with a `main` function. This code chunk contains a reference to the definition of the `fibC` function, and Lepton will replace the reference with the corresponding source code.

Code chunk 6: «main.c»

fibC

```
#include <stdlib.h>
#include <stdio.h>

<<fibC>>

int main(int argc, char **argv)
{
	int i, n;
	if (argc < 2) return 1;

	for (i = 1; i < argc; i++) {
		n = atoi(argv[i]);
		if (n < 0) {
			printf("bad input: %s\n", argv[i]);
			continue;
		}

		printf("%i\n", fibC(n));
	}
	return 0;
}
```

We configured the `main.c` code chunk such that its (expanded) contents are written to disk. We can now compile it with the following shell commands.

Code chunk 7: «shell»

```
gcc main.c -o a.out
./a.out 0 1 2 3
```

Interpret with shell

```
1
1
1
2
```

4 Comparison of running times

In this section, we compare the running times of the three proposed implementations. Let us first indicate the system configuration that performed this comparison using shell commands. Note that Python writes to `stderr`, and we have to redirect its output so that it appears in the PDF document.

Code chunk 8: «shell (part 2)»

```
uname -a
ocaml --version
python --version 2>&1
gcc --version
```

Interpret with `shell`

```
Linux lepton5530 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5 (2019-06-19) x86_64 GNU/Linux
The OCaml toplevel, version 4.05.0
Python 2.7.16
gcc (Debian 8.3.0-7) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The time necessary for computing the number F_n depends on the algorithm, language, as well as n. We will compute the running times for several values of n, then assemble the results into a plot.

4.1 Ocaml

Let us define a function `time` to measure the time necessary in OCaml. This function uses the `Sys` module in the standard library.

Code chunk 9: «ocaml (part 3)»

```
let time niter n =
  let start = Sys.time() in
  for i = 1 to niter do ignore (fibonacci n) done;
  (Sys.time() -. start) /. float_of_int niter
;;
time 100 10;;
time 100 15;;
time 100 20;;
```

Interpret with `ocaml`

```
val time : int -> int -> float = <fun>
- : float = 1.77000000000000084e-06
- : float = 2.07299999999999862e-05
- : float = 0.00021098
```

Writing the results to disk.

4.2 Results and discussion

We use Gnuplot for making the figures.

Code chunk 10: «runningtimes.dat»

```
# this is a test file and does not contain actual results
0.0017
0.019
0.2
```

Code chunk 11: «plot.in»

```
set terminal pdf
set output "runningtimes.pdf"
plot "runningtimes.dat" using 1 with linespoints
```

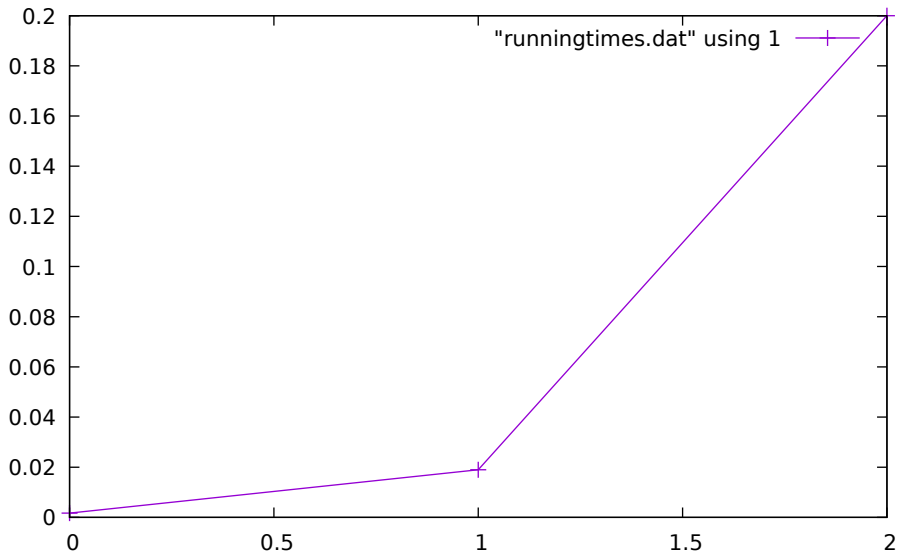
Code chunk 12: «shell (part 3)»

```
gnuplot plot.in
```

Interpret with `shell`

```

```



The proposed recursive implementation uses an algorithm with exponential complexity.

5 Conclusion

This document shows how to compute the Fibonacci sequence in three different programming languages, with one recursive implementation and two iterative implementations. Using Lepton makes it possible to

- provide everything in a single executable file that makes it easy to reproduce the results
- embed source code and executable instructions in a readable manner
- restructure the source code for easier human comprehension
- run compilation commands and analysis scripts to ensure that the figures were generated with the provided source code.