Comparison of a few implementations of the Fibonacci sequence

Li-Thiao-Té Sébastien

July 22, 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
- certify that the figures correspond to the provided source code
- embedding different programming languages in the same document / same platform
- simplifying code re-use by distributing only a single file.

1.1 Executing this document

This document is a script and is intended to be executed to produce a PDF file, as well as by-products such as extracted source code or data files. It requires :

- Lepton to produce a `.tex` LaTeXdocument,
- LaTeX to process the `.tex` file into PDF,
- the Pygments library for syntax highlighting and the `minted` LaTeXpackage for code beautification.

To process the document with Lepton, the user should run the command :

Code chunk 1: ≪`compilation`≫

```
lepton fibonacci.nw -o fibonacci.tex
```

To produce the PDF document, the user should run the following commands. LaTeX needs to run twice to process references.

Code chunk 2: ≪`compilation (part 2)`≫

```
# The LaTeX style file lepton.sty should be in a directory accessible to LaTeX
pdflatex fibonacci.tex
pdflatex fibonacci.tex
```

2 Problem statement

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

$$F_0 = 0 \tag{1}$$

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

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

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

3 Implementations

The proposed implementations in this document are taken from the Rosetta Code project `https://rosettacode.org/wiki/Fibonacci_sequence`.

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 3: ≪ocaml≫

```
let rec fibonacci = function
  | 0 -> 0
  | 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 4: ≪ocaml (part 2)≫

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

Interpret with `ocaml`

```
- : int = 0
- : int = 1
- : int = 1
- : int = 2
```

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 5: ≪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 6: ≪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 7: ≪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 8: ≪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. Note that this implemenation returns an incorrect value for F_0.

Code chunk 9: ≪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 is used to perform 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 10: ≪shell (part 2)≫

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

Interpret with `shell`

```
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 11: ≪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
;;
```

Interpret with `ocaml`

```
val time : int -> int -> float = <fun>
```

Writing the results to disk.

Code chunk 12: ≪ocaml (part 4)≫

```
let oc = open_out "runtimes.ocaml" in
begin
  output_string oc "# Running times of Fibonacci sequence using Ocaml, time in seconds\n";
  for i = 1 to 9 do
    output_string oc (string_of_int i ^ "\t" ^ string_of_float (time 1000 i) ^ "\n")
  done;
  close_out oc;
end;;
```

Interpret with `ocaml`

```
- : unit = ()
```

We display below the contents of the results file.

Code chunk 13: ≪shell (part 3)≫

```
cat runtimes.ocaml
```

Interpret with `shell`

```
# Running times of Fibonacci sequence using Ocaml, time in seconds
1       2.1e-08
2       2.7e-08
3       4e-08
4       8.8e-08
5       1.44e-07
6       2.4e-07
7       3.92e-07
8       6.68e-07
9       1.133e-06
```

N.B. In the above example, all code chunks with the name `ocaml` share the same interpreter process. Consequently, the `fibonacci` function defined in the first chunk is available in the subsequent chunks. This is the same for the `shell` chunks and for the `python` chunks.

4.2 Python

Similarly, we define a function `timefib` to measure the time necessary in Python. There is no output on success.

Code chunk 14: ≪python (part 3)≫

```
from time import clock

def timefib(i):
  start = clock()
  for n in range(0,1000): fibIter(i)
  end = clock()
  return (end-start)/1000

file1 = open("runtimes.python","w")
file1.write("# Running times of Fibonacci sequence using Python, time in seconds\n")
for i in range(1,10):
  file1.write(str(i))
  file1.write("\t")
  file1.write(str(timefib(i)))
  file1.write("\n")

file1.close()
```

Interpret with `python`

We display below the contents of the results file.

Code chunk 15: ≪shell (part 4)≫

```
cat runtimes.python
```

Interpret with shell

```
# Running times of Fibonacci sequence using Python, time in seconds
1        1.08e-07
2        2.08e-07
3        2.53e-07
4        3.24e-07
5        3.82e-07
6        5.18e-07
7        4.08e-07
8        3.41e-07
9        4.51e-07
```

4.3 C

In C, we write a new program to run the benchmark. Note that this benchmark uses a reference to the `fibC` function defined earlier. This ensures that the same code is used in the `main.c` and `benchmark.c` programs. Lepton automatically inserts a PDF link to the definition of the `fibC` function.

Code chunk 16: ≪benchmark.c≫

fibC

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

<<fibC>>

int main(int argc, char **argv)
{
    int i,n;
    clock_t t;

    for (n = 1; n<10; n++) {
        t = clock();
        for (i=1; i<10000; i++) fibC(n);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC / 10000; // in seconds
        printf("%i\t%e\n",n,time_taken);
    }
 }
```

We compile and run the benchmark below.

Code chunk 17: ≪shell (part 5)≫

```
gcc benchmark.c -o bench.out
echo "# Running times of Fibonacci sequence using C, time in seconds" > runtimes.c
./bench.out >> runtimes.c
head runtimes.c
```

Interpret with shell

```
# Running times of Fibonacci sequence using C, time in seconds
1        2.200000e-09
2        3.300000e-09
3        4.900000e-09
4        6.600000e-09
5        8.500000e-09
6        1.040000e-08
7        1.230000e-08
8        1.420000e-08
9        1.650000e-08
```

4.4 Results and discussion

We use Gnuplot for making the figures. We first define the gnuplot script.

Code chunk 18: ≪plot.in≫

```
set terminal pdf
set output "runningtimes.pdf"
plot "runtimes.ocaml" using 1:2 with linespoints, \
     "runtimes.python" using 1:2 with linespoints, \
     "runtimes.c" using 1:2 with linespoints
```

We execute the script in gnuplot. This writes a PDF to disk, which is then included in a LaTeX figure.

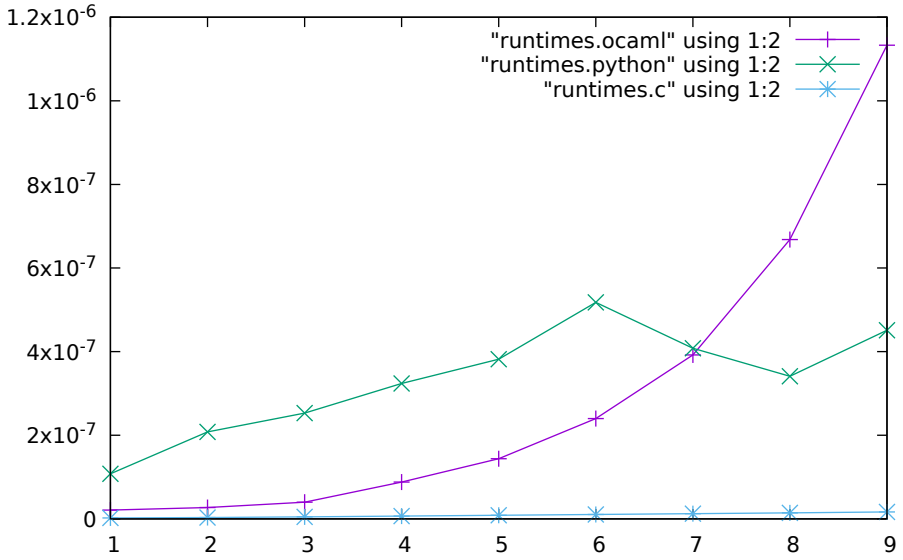

Figure 1: Running time in seconds for computing F_n as a function of n for the three proposed implementations.

Code chunk 19: ≪shell (part 6)≫

```
gnuplot plot.in # no output on success
```

Interpret with shell

```
```

In Figure 1, we plot the running times necessary to compute the Fibonacci sequence. As expected, the OCaml recursive implementation has exponential complexity in time, whereas the Python and C iterative implementations have linear time complexity. The C implementation is much faster than the Python and OCaml code, which are run inside an interactive loop whereas the C code is compiled. It would be interesting to compare with the programs produced with native code compilers of Python and OCaml or just-in-time compilation. Additionally, the recursive OCaml function runs faster than the iterative Python function for small values of n, which suggests that there is less overhead for calling functions in OCaml than in Python.

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.