85301 – Algorithms and Data Structures in Biology (2022/23)

Enrico Malizia and Riccardo Treglia

DISI, University of Bologna, Italy

Lab 2 – Exercise

In this lab session, we will continue the exercise we saw last week, and try to solve it in a principled way, in light of the theoretical results we saw in the meantime, but also using tools like `numpy` and `scipy`, which have been introduced in today's lab session.

1 Implementing the two Sorting Algorithms

We assume that you have implemented both *Insertion Sort* and *Merge Sort* in `Python`. If you have not done so yet, it is time for you to have your two programs up and working as soon as possible.

This exercise sheet contains extensive text and guidance, but very few actual instructions: to help you, instructions are displayed **in bold**.

2 Measuring Time Performance

Last week, we naïvely measured the time performances of the two sorting algorithms at hand, without really basing the experimental analysis on their analytical properties. We know that the two algorithms at hand have different time complexities: *Insertion Sort* has complexity $O(n^2)$, whereas *Merge Sort* has complexity $O(n \lg n)$. In other words, *Merge Sort* is, analytically speaking, more efficient than *Insertion Sort*. The purpose of today's exercise session is to check whether the aforementioned analysis matches the experimental data.

2.1 A More Systematic Measurement of the Time Performances

Since the execution time of an algorithm may vary on *different* input of the *same* size, we need to perform some statistical analysis. Intuitively, we run many times the algorithm on *different random* inputs of the *same size*, we keep track of the algorithm computation time in all these attempts, and then we average over these values.

For example, let's assume that we want to accomplish this task: measuring the average execution time of Selection Sort on a random array of length m, and average this computation time over n attempts.

In order to do this in a more systematic way, we can measure the algorithm computation time with the help of the `Python` module `codetiming`.[1] We can use the timing measurement features offered by the module `codetiming` in the following way.

```python
from codetiming import Timer      # import the Timer facilities

# [...] some more code here

for _ in range(n):                # we perform the time measurement n times
    r_input = randomIntList(m)    # we generate a random array of m integers
    with Timer(name = "insertion_sort", logger = None):
        insertionSort(r_input)
```

We here use the `Timer` facilities because we do not want to include in the time measurement the time needed to generate the random array. The parameter `name = "insertionSort"` is simply a label that

[1]Before using this module, you need to install it. To install the module, either in CPython or Thonny, you can refer to the guide on installing Python on your machine, and follow the instructions to install the additional modules, but in this case the additional module to install is `codetiming`.

we use to store together all the time measurements that we perform under that name, so that we can have access to them later. The time measured is the time elapsed in the execution of the code block beneath the statement `with Timer`. It is important to use the parameter `logger = None` otherwise the timer will print all the measurements.

Once we have performed the measurements, we can have access to them in this way:

```
# we use the label "insertion_sort" because we want to access
# the time measurements stored under the name "insertion_sort"

Timer.timers.max("insertion_sort") # to obtain the maximum for the specified label
Timer.timers.min("insertion_sort") # to obtain the minimum for the specified label
Timer.timers.mean("insertion_sort") # to obtain the average for the specified label
Timer.timers.stdev("insertion_sort") # to obtain the standard deviation for the specified label
Timer.timers.count("insertion_sort") # to obtain the number of measurements for the specified label
Timer.timers.total("insertion_sort") # to obtain the overall time execution for the specified label

Timer.timers.pop("insertion_sort") # to clear the statistics for the specified label
Timer.timers.clear() # to clear all statistics (for all labels)
```

Given that we can use the labels to discriminate measurements of different parts of the code, we can actually take time measurements for different blocks of code (and algorithms) at the same time. This can be helpful in the following context.

In order to fairly compare the performances of two sorting algorithms, we should test them on the very *same* inputs, and in particular on the very *same random* inputs. How can we be sure that the two algorithms are executed on the same random inputs? Well, the probability of generating twice the same sequence of random inputs is pretty low—let's say that it is almost zero.[2] What we can do is generating one random array and have the same array sorted multiple times by different algorithms. Then we measure the sorting time of the different algorithms on the same inputs and we keep distinct statistics for the distinct algorithms. We can easily do this by using different labels.

```
from codetiming import Timer      # import the Timer facilities

# [...] some more code here

for _ in range(n):                 # we perform the time measurement n times

    r_input = randomIntList(m)     # we generate a random array of m integers

    # remember that we designed our sorting algorithms so that the sorted array
    # is returned by the function, and we do not sort the passed array; for
    # this reason, in this specific case, we do not need to clone the array

    with Timer(name = "insertion_sort", logger = None):
        insertionSort(r_input)

    with Timer(name = "merge_sort", logger = None):
        mergeSort(r_input)
```

It is important to notice that, if we want to use in our code a timer with the same label multiple times but in different contexts (i.e., in contexts for which it does not make sense to sum up and group together the timings), then we need to store the gathered statistics that we need somewhere, and then clear the statistics associated with that label before proceeding.

- **Write a Python function `testSortAlgsOneLength(length, repetitions)`** that takes as parameters two integers, `length` and `repetitions`, and returns (more details below) the average time needed by Insertion Sort and Merge Sort, respectively, to sort arrays of length `length`. The average is evaluated over a number of repetitions `repetitions` passed as parameter to the function.

  Regarding the output of the function, the function returns a list of two elements, in which the first and second element are the average computation time for Insertion Sort and Merge Sort, respectively.

  Write your function so that, if the user does not explicitly specify a number of repetitions, then the number of repetitions carried out by default by the function is `10000`.

[2]We could overcome this by feeding the random generator with the same initial seed. However, if the algorithms themselves use the random generator, then this can be a bit problematic.

For example, if we invoke the function as `testSortAlgsOneLength(500, 10000)`, we mean that we want to know the average computation time for Insertion Sort and Marge Sort, respectively, to sort arrays of length 500, and the average is computed over 10000 repetitions. An answer like

```
[0.005558163550000131, 0.0030588481500000695]
```

means that the specific implementations of Insertion Sort and Marge Sort that we are testing on our specific hardware take an average 0.00556 seconds and 0.00306 seconds, respectively, to sort arrays of 500 elements (when averaged over 10000 repetitions).

- **Write a Python function `testSortAlgs(lengths, repetitions)`** where in this case `lengths` is a list of lengths over which performing the test. For each of these lengths, the average computation time for Insertion Sort and Merge Sort is evaluated over a number of repetitions equal to `repetitions`.

  In this case, the output of the function is a list of two lists, in which the first and second list are the average computation times for Insertion Sort and Merge Sort, respectively, for the lengths that we passed as input.

  Also in this case, write your function so that, if the user does not explicitly specify a number of repetitions, then the number of repetitions carried out by default by the function is `10000`.

  For example, if we invoke `testSortAlgs([100, 500, 1000], 10000)`, we mean that we want to know the average computation time for Insertion Sort and Marge Sort, respectively, to sort arrays of length 100, 500, and 1000, respectively, and the average is computed over 10000 repetitions. An answer like

  ```
  [[0.00029203723000000535, 0.005562382629999666 , 0.022565693400000288],
   [0.0004975963499999807 , 0.0030624656100001375, 0.00674039318000062 ]]
  ```

  means that the specific implementations of Insertion Sort and Marge Sort that we are testing on our specific hardware take an average 0.000292 seconds, 0.00556 seconds, and 0.02257 seconds, for Insertion Sort to sort arrays of length 100, 200, and 1000, respectively; and 0.000498 seconds, 0.00306 seconds, and 0.00674 seconds, for Merge Sort to sort arrays of length 100, 200, and 1000, respectively.[3]

2.2 Collecting the Data

- **Create a list `dataLengths`** of lengths for which you will measure the running time of both sorting algorithms. For example, you can take `dataLengths = [0, 200, 400, 600, ..., 4000]`. You can choose a different list if this one takes too long to be tested, but try to cover a wide range (at least up to 1000) and to have enough data points (at least 20).

- **Compute the lists of running times** for the functions `insertionSort` and `mergeSort` and store the results in `dataInsertionSortTimes` and `dataMergeSortTimes`, respectively.

3 Building the Models

We will use the module `scipy.optimize` to fit the theoretical models to the data. As an example, we will show how to proceed for the (terrible) implementation of `fibonacci` seen last week:

```
>>> import numpy
>>> import scipy.optimize
>>> def testFibonacciOneNumber(argument, repetitions):
    ...
>>> def testFibonacci(arguments, repetitions):
    ...
>>> dataFibArgs = list(range(39))
>>> dataFibTimes = testFibonacci(dataFibArgs, 100)
```

The curve looked exponential, i.e., of the form $t = a \cdot b^n$, so we will look for a model of that shape.

[3]Do you notice anything strange in the statistics above?

```
>>> def exponential(x, a, b):
      return a * numpy.power(b, x)
>>> scipy.optimize.curve_fit(exponential, dataFibArgs, dataFibTimes)

(array([2.44416322e-07, 1.62303070e+00]),
 array([[ 3.91860448e-16, -6.95485164e-11],
        [-6.95485164e-11,  1.23523799e-05]]))
```

The function `scipy.optimize.curve_fit` returns two `numpy` arrays: the first contains the estimates of the parameters, the second the estimated covariance matrix of these estimators. For now, we will focus on the first: we have $a = 2.444 \cdot 10^{-7}$ and $b = 1.623$, so the estimated running time is $t = 2.444 \cdot 10^{-7} \cdot 1.623^n$. Observe that this estimated running time is for this specific implementation of the algorithm running on the specific machine of the test. On different computers, the function of the estimated running time will be different, because the execution time is different. (For this reason, to experimentally compare the performances of different algorithms, besides testing them on the same inputs, the comparison has to be carried out over the running times required by the different algorithms on the *same* computer, otherwise the comparison would be meaningless.)

- **Now, do the same for the *insertion sort* data** (the model is $t = a \cdot x^2 + b \cdot x$) **and the *merge sort* data** (the model is $t = a \cdot x \cdot \log(x) + b \cdot x$).

4 Plotting the Data and the Models

We will plot the data and the model within a LaTeX document using `pfgplots`. First, we use `Python` to format the data in a way that `pfgplots` will understand:

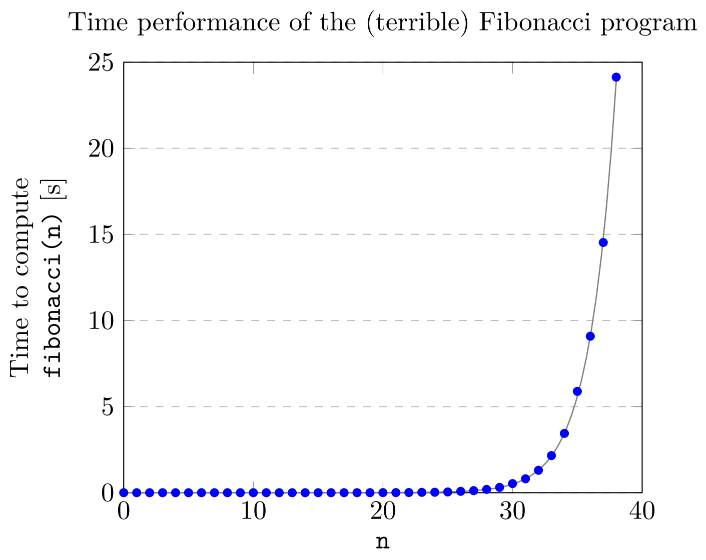
```
>>> def printFibonacciData(xdata, ydata):
      for (n,t) in zip(xdata, ydata):
        print ((n, round(t, 3)), end=' ')
      print()

>>> printFibonacciData(dataFibArgs, dataFibTimes)
(0, 0.0) (1, 0.0) (2, 0.0) (3, 0.0) (4, 0.0) (5, 0.0) (6, 0.0) (7, 0.0) (8, 0.0)
(9, 0.0) (10, 0.0) (11, 0.0) (12, 0.0) (13, 0.0) (14, 0.0) (15, 0.0) (16, 0.001)
(17, 0.001) (18, 0.002) (19, 0.003) (20, 0.005) (21, 0.007) (22, 0.01) (23, 0.02)
(24, 0.026) (25, 0.044) (26, 0.079) (27, 0.12) (28, 0.191) (29, 0.31) (30, 0.529)
(31, 0.809) (32, 1.301) (33, 2.156) (34, 3.443) (35, 5.885) (36, 9.086) (37, 14.531) (38, 24.133)
```

With these, we can write a LaTeX source file:

```
\documentclass[a4paper]{article}

\usepackage{amsmath}
\usepackage{pgfplots}

\begin{document}

\begin{center}
\begin{tikzpicture}
\begin{axis}[
  title={Time performance of the (terrible) Fibonacci program},
  xlabel={\texttt{n}},
  ylabel={$\begin{array}{l}\text{Time to compute}\\
        \texttt{fibonacci(n)}\text{ [s]}\end{array}$},
  xmin=0, xmax=40,
  ymin=0, ymax=25.000,
  xtick={0,10,20,30,40},
  ytick={0,5,10,15,20,25},
  legend pos=north west,
  ymajorgrids=true,
  grid style=dashed]

  % Plot the model : t = 2.444e-7 * 1.623^n
  \addplot[gray,line width=0.5,domain=0:38,samples=100] {2.444e-7*1.623^x};

  % Plot the data
  \addplot[color=blue, only marks, mark size=1.5]
    coordinates{
```

```
    (0, 0.0) (1, 0.0) (2, 0.0) (3, 0.0) (4, 0.0) (5, 0.0) (6, 0.0) (7, 0.0) (8, 0.0) (9, 0.0) (10, 0.0)
     (11, 0.0) (12, 0.0) (13, 0.0) (14, 0.0) (15, 0.0) (16, 0.001) (17, 0.001) (18, 0.002) (19, 0.003) (20,
      0.005) (21, 0.007) (22, 0.01) (23, 0.02) (24, 0.026) (25, 0.044) (26, 0.079) (27, 0.12) (28, 0.191)
     (29, 0.31) (30, 0.529) (31, 0.809) (32, 1.301) (33, 2.156) (34, 3.443) (35, 5.885) (36, 9.086) (37,
     14.531) (38, 24.133)};
\end{axis}
\end{tikzpicture}
\end{center}
\end{document}
```

After we compile it, the resulting PDF file contains the graph:



As you can see, this looks like a pretty close match.

- **Now, do the same for the *insertion sort* data and (separately) for the *merge sort* data.** After that, in addition, you can plot them on the same graph in order to compare them.