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

Enrico Malizia and Riccardo Treglia

DISI, University of Bologna, Italy

Lab 1 – Exercise

Aim of this lab session is to compare the time performance of two sorting algorithms, namely *insertion sort* and *merge sort*, on different sizes of inputs.

In the following, the *sorting* (combinatorial) problem takes as input a *list of integers* (*e.g.,* `[3,1,4]`, `[0]`, `[]`), and outputs its sorted version. Concretely, a *sorting function* is a `Python` function that takes as input a list of integers and returns it *sorted in ascending order* (*e.g.,* respectively `[1,3,4]`, `[0]`, `[]`) *as a new list*, *without* modifying the original list.

1 Implementing the Sorting Algorithms

Your goal is to implement sorting algorithms "from scratch". As a result, in this section, your code must play by the following rules:

- you may not import any module;
- you may manipulate integers, lists of integers and Booleans, but no other type of data;
- on lists, you may only use the following primitives:
  - the index (*e.g.,* `myList[0]`) and slice (*e.g.,* `myList[:2]`) operators;
  - the `len` and `range` functions;
  - the `insert` and `append` functions;
  - integer list literals (*e.g.,* `[3,1,4]`, `[0]`, `[]`).

1.1 Insertion Sort

Here is some pseudocode for the Insertion Sort algorithm:

```
function insertionSort(list unsortedList):
  sortedList = []
  for insertedElement in unsortedList
    insertInSortedList(sortedList, insertedElement)
  return sortedList

function insertInSortedList(list sortedList, integer insertedElement):
  for i = 0 to sortedList.length-1:
    if sortedList[i] >= insertedElement:
      sortedList.insert(i, insertedElement)
      return
  sortedList.append(insertedElement)
```

As you can see, the algorithm proceeds by iteratively putting the i-th element of the input list in the appropriate position in the list of already-sorted elements, until the whole input list has been processed.

Write a `Python` *sorting function* `insertionSort` that implements Insertion Sort. Do not forget to test it on a few examples (*e.g.,* `[]`, `[0]`, `[1,0]`, `[0,1]`, `[0,0,0]`, and `[3, 2, 0, 4, 1, 3, 2]`, which should give `[0, 1, 2, 2, 3, 3, 4]`).

1.2 Merge Sort

Here is some pseudocode for the Merge Sort algorithm:

```
function mergeSort(list unsortedList):
  if unsortedList.length <= 1:
    return unsortedList.copy()
  center = unsortedList.length / 2
  sortedList1 = mergeSort(unsortedList[:center])
  sortedList2 = mergeSort(unsortedList[center:])
  return mergeSortedLists(sortedList1, sortedList2)
```

The algorithm, as you can see, is deeply different from Insertion Sort. In particular, it proceeds by splitting the input list into two, recursively solving the problem for the two sub-lists, and then merging the two sorted lists into one by way of a `mergeSortedLists` sub-algorithm.

Write a `Python` function `mergeSortedLists` that takes as arguments two lists *sorted in ascending order* and merges them into one list *still sorted in ascending order*. The function must go through each list *only once*. Do not forget to test it on a few examples (*e.g.,* `[]` and `[0,1]` give `[0,1]`; `[1, 3, 4, 6, 8]` and `[0, 2, 3, 5]` give `[0, 1, 2, 3, 3, 4, 5, 6, 8]`). Then, write a *sorting function* `mergeSort` that implements Merge Sort, following the pseudocode above. Do not forget to test it on the same examples as `insertionSort`.

2 Measuring Time Performance

You may now import the `random` and `cProfile` modules and use them as you wish.

Write a function `randomIntList` that takes as argument a non-negative integer n and returns a random (*uniformly distributed*) list of n integers between 0 (included) and 1 000 000 000 (included).

2.1 Counting comparisons

For a large class of sorting algorithms, including Isertion Sort and Merge Sort, the total number of comparisons between integers needed to complete the algorithm is known to be a good proxy for its time performance.

Modify the code of both `insertionSort` and `mergeSort` so that they count the number of such comparisons, then test both functions on random lists of 10, 100, 1 000, 10 000, and then 100 000 integers, and compare the results.

Based on these data, for each function, propose a simple numeric equation approximating the relation between the length n of the list and number k of comparisons needed to sort it.

2.2 Using `cProfile`

Using `cProfile`, measure the time it takes each of the sorting functions to sort a random list of 10, 100, 1 000, 10 000, and then 100 000 integers.

Based on these data, for each function, propose a simple numeric equation approximating the relation between the length n of the list and the time t (in seconds) that it takes to sort it.

Based on the results of this section and the previous one, to what extent would you say that, as far these two algorithms are concerned, the total number of comparisons is indeed a good proxy for time performance?

Optional

Compare the performance on very short lists (*e.g.,* of length 5): what do you notice? In order to get meaningful results, you should measure the time it takes to sort your list a large number of times (*e.g.,* 100 000) instead of just once.

In the light of this result, how would you modify the `MergeSort` function to make it run faster? Implement that change ant test it.