EDA (ETS de Ingeniería Informática). Academic Year 2020-2021
Lab 2. Efficiency's empirical analysis of two sorting D&C algorithms (one session)

Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València

1 Objectives

The purpose of this lab is twofold. On the one hand, after doing it, the student should be able to compare two "fast" Divide-and-Conquer (D&C) sorting algorithms -Merge Sort and Quick Sort- using empirical analysis. On the other hand, the student should be able to decide which of these two algorithms is the best one for each data type by contrasting the effect which the comparison of individual elements has on their runtimes.

2 Problem description

The choice of the best algorithm to sort in-place (or *in situ*) the elements of an array is just a question of taking into account the following three characteristics and, then, to arrive to a trade off among them:

- Its efficiency expressed in terms of the size of the problem; this leads to choose elementary sorting algorithms (Insertion Sort, for example) when the number of data to be sorted is small and "fast" sorting algorithms (Merge or Quick Sort), when it is huge.
- The number of comparisons and movements (exchanges or copies) that performs and the effective (run time) cost of these operations. Given that Merge Sort does a lot less comparisons than Quick Sort but a lot more movements ...
  - Merge Sort will be preferable always and when the effective cost of the data comparisons is notable and, as it happens in Java, the data movement cost is irrelevant because it only affects the references to the data and not the data itself (Pointer sorting);
  - Quick Sort will be preferable when the effective cost of data movements is as much or more important than the cost of the comparisons because, as in C++, they affect the data itself.
- Its stability, that is to say it preserves the relative order of equal elements in the array. Such a characteristic only results irrelevant when a unique order is used to sort the array, as it happens with primitive-type data. Given that Merge Sort is stable and Quick Sort is not, ... Merge Sort will be preferable when alternate orderings can be used, depending upon the situation, to sort comparable objects.

This explains why the `java.util` standard library offers a collection of overloaded methods to sort arrays of primitive-type data (for instance, `static void Arrays.sort(int[])`) and arrays of reference-type data (for instance, `static void Arrays.sort(Object[])`): in order to trade speed and memory usage for stability and guaranteed performance, Quick Sort is used to implement the primitive-type methods, whereas Merge Sort is used to implement the reference-type methods.

In this lab, the student will do a fair comparison of Merge Sort and Quick Sort by putting their empirical analysis at the service of the following tasks:

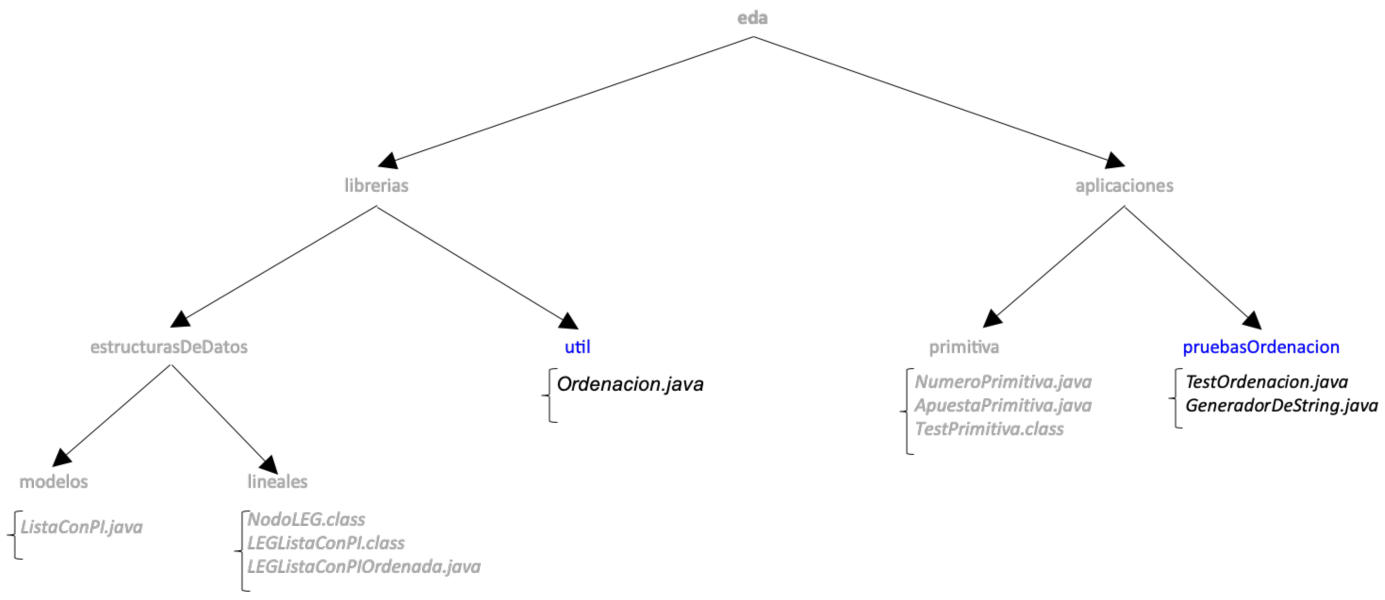
- Getting a tight code implementation of Merge Sort, faster than the straightforward version presented in the lectures of Unit 2; as you can imagine, it is not necessary to do the same with Quick Sort because the method `quickSort` introduced in Unit 2 is already such a version.
- Analyzing how the cost of individual elements's comparison affects the actual runtime of Merge Sort and Quick Sort and so deciding which of these two algorithms is the best one for each data type.

For the sake of simplicity, the arrays used in this study will be of type `Integer` and `String`. Note that, on the one hand, these Java classes already implement the `Comparable` interface, which simplifies the coding effort to be done. On the other hand, the time complexity of their `compareTo` methods serves perfectly for the above mentioned tasks: as the time required to compare two `Integer`s is constant, it will not interfere in the runtime estimation of any version of Merge Sort or Quick Sort; however, as the time required to compare two `String`s requires a time proportional to the index position at which they differ, it will allow to check the effect the comparison of individual elements has on the runtime of any version of these algorithms, by conveniently initializing the `String`s to be sorted.

3 Lab Activities

Before doing the activities described below in this section, it is necessary that the student updates the structure of packages and files of his *BlueJ eda* project by carrying out the following steps:

- Start *BlueJ*.
  - Open the *eda* project.
  - Open the *aplicaciones* package and create a new package called *pruebasOrdenacion*; it will contain the application's classes of the second lab activity.
- ***Quit BlueJ* by selecting *Quit* from the *Project* menu.**
- Download the classes available in *PoliformaT* into their corresponding folders as shown in the figure below.



  NOTE:

  - The class `Ordenacion` contains the static generic methods which implement some sorting algorithms as well as the auxiliary method `sonIguales`, which checks whether two given arrays are equal to one another using the `compareTo` method.
  - The class `TestOrdenacion` implements the methods to carry out the empirical analysis of the `Ordenacion` class sorting methods (test data generation as well as measurement and tabulation of average running times for different sizes of the problem).
  - The class `GeneradorDeString` represents a random generator of `String`s with the same `n` initial characters.

- (Re)Start *BlueJ* and open the package *librerias.util* of the *eda* project.

3.1 Increasing the performance of Merge Sort

The `mergeSort1(T[])` method of the `Ordenacion` class is quite similar to the one presented in the Unit 2 lectures. Its overall runtime can be easily improved by avoiding to copy twice the elements to be merged at each one of its "conquer" steps (method `merge1`): when examining its code, it is clear that the two first loops of `merge1` merge and copy the elements of two ordered sections of an array `v` into an auxiliary array `aux`, then the content of `aux` is copied back to the original array `v` in the last loop of `merge1`.

In this activity the student should add to class `Ordenacion` a new method, called `mergeSort2(T[])`, that must avoid such a double copy and, thus, be faster than `mergeSort1(T[])`. To do this, the following steps must be carried out:

i. Write a new version of method `merge1` that returns the merge array `aux`, instead of `void`. This is a very simple way of eliminating the second copy of the elements to be merged, i.e., the one of the already merged elements from `aux` to `v`. Consequently, the new and faster version of `merge1` will have the following specification and header:

```
/** Returns the merge array of v1 and v2.
 *  @param v1  is sorted in ascending order and its elements implement Comparable
 *  @param v2  is sorted in ascending order and its elements implement Comparable
 *  @return T[], the resulting merged array
 */
private static <T extends Comparable<T>> T[] merge2(T[] v1, T[] v2)
```

ii. Write a new version of method `mergeSort1(T[], int, int)` by using `merge2`. As the parameters of `merge2` are sorted arrays, this version cannot perform an in-place sort and, thus, it must have the following specification and header:

```
/** Returns an ordered array with the elements of subarray v[i, f].
 *  @param v  an array of Comparable elements
 *  @param i  the left-most index of the subarray
 *  @param f  the right-most index of the subarray
 *  @return T[], the resulting sorted array
 *  PRECONDITION: i<=f
 */
private static <T extends Comparable<T>> T[] mergeSort2(T[] v, int i, int f)
```

Additionally, in order to avoid as much as possible the cost of generating too many arrays of size 1, the body of this method has to include two base cases: one for arrays of size 2 and one for arrays of size 1.

iii. Write a new version of method `mergeSort1(T[])` by using `mergeSort2(T[], int, int)` while maintaining the typical header of an in-place sorting method, in which the return type is `void` instead of `T[]`. The new and faster version of `mergeSort1(T[])` will have the following specification and header:

```
/** Sorts the array v in ascending order.
 *  @param v  the array of Comparable elements to be sorted
 */
public static <T extends Comparable<T>> void mergeSort2(T[] v)
```

3.2 Testing the new version of Merge Sort

Before carrying out their empirical analysis, it is necessary to test the correctness of the method `mergeSort2(T[])`. The most simple way to do so is to compare its results with those of any other sorting algorithm that is known to be correct, for example `quickSort`.

Since the `comprobar` method of the `TestOrdenacion` class partially implements this test, the student only has to complete its code to check the correctness of `mergeSort2(T[])`; in order to do so, the methods `quickSort` and `sonIguales` of the `Ordenacion` class should be used.

3.3 Comparing the D&C sorting's methods of the `Ordenacion` class

In order to compare the D&C sorting methods of the `Ordenacion` class, first it is necessary to perform their empirical analysis. Currently, the `temporizar` method of the `TestOrdenacion` class already does it for the `quickSort` and `mergeSort1(T[])` methods. Specifically, as its code shows, `temporizar` performs the time measurements for those methods under the following conditions:

- The sorting methods' arguments are arrays of randomly generated `Integers` (`crearAleatorioInteger` method).
- The time measurements are done for arrays' sizes between 10.000 and 100.000, with increments of 10.000.
- A method's average runtime is obtained after running it 200 times for each array size and each time with a different array, in order to guarantee significant results.

- The clock used to measure each method's runtime is `nanoTime`, the method of `java.lang.System` class that returns the current value of the most precise available system timer in nanoseconds (although resolution may be lower).

Since the `temporizar` method also tabulates the average runtimes of the `quickSort` and `mergeSort1(T[])` methods, it should be clear that, to carry out this activity the student only needs ...

1. To add to the code of `temporizar` the corresponding lines for the empirical analysis of `mergesort2(T[])`.
2. To compile and run `temporizar` (by executing `TestOrdenacion`).
3. To extract the corresponding conclusions about which of the three methods is the fastest.

3.4 Plot and fit the empirical analysis results

The results of Activity 3.3 must be graphically represented and adjusted by using `gnuplot` (`plot` and `fit` commands); if it was necessary, the student could recall the most frequently used `gnuplot` commands, as well as an example of its use, in the document "A summary of `gnuplot` commands" available in *PoliformaT*.

3.5 Comparing the D&C sorting's methods of the `Ordenacion` class when the cost of the individual comparisons increases

To perform this activity the student must use the class `GeneradorDeString` that, as said before, randomly generates `Strings` with the same `n` initial characters. So, for example, in order to generate two `Strings` with the same `50` first characters with the help of this class, the instructions to use are:

```
GeneradorDeString g = new GeneradorDeString(50);
String ejem1 = g.genera();
String ejem2 = g.genera();
// ejem1 and ejem2 have the same 50 initial characters
```

Specifically, based on the above example, the student has to modify the code of the `crearAleatorioString` method in class `TestOrdenacion` so that it returns an array of `talla String` randomly generated with the same `n` initial characters.

Once done, Activities 3.3 and 3.4 have to be carried out again, but this time with arrays of `Strings` with the same `50` first characters and using the method `temporizarString` in class `TestOrdenacion` ... Which of the three D&C sorting's methods of the `Ordenacion` class is now the fastest? Why?

4 Other Activities

If the student wants to continue working a little bit more on the comparison of Merge and Quick Sort after the lab session, we suggest the two following activities:

- Repeating the last timing but for Strings with the same 10 and 80 initial characters.
- Adding to your experimental setting the standard method `java.util.Arrays.sort(Object[] a)`.