UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIIC

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Data Structures and Algorithms

Lab Practice 4: Divide and Conquer

2019-2020 course

Contents

1 Introduction

This practice aims to achieve the following objectives:

- Review and apply the concepts of static class attributes and methods.
- Implement traditional sorting algorithms.
- Apply Divide and Conquer strategy, implementing the mergesort algorithm.
- Study, compare and experimentally analyze the time cost of the proposed algorithms.

2 Traditional sorting algorithms

In this section we review a traditional (and inefficient) sorting algorithm: the selection sort algorithm.

2.1 Selection sort

In the selection sort algorithm, the vector is decomposed into two parts: the ordered part and the unordered part. Initially, the ordered part does not contain any elements, while the unordered part contains all the elements of the vector. The procedure involves selecting, at each step of the algorithm, the minimum element of the unordered part, and exchanging it with the element located in the position towards the ordered part grows. Thus, at each step of the algorithm, the ordered part grows in one position, to the detriment of the unordered part, which decreases accordingly. This procedure continues until the unordered part contains a single element, which will correspond to the maximum element of the vector. The following video illustrates how this algorithm works: *Selection sort (YouTube)*. Its asymptotic cost is quadratic $O(n^2)$.

3 Divide and Conquer

The Divide and Conquer strategy consists of dividing a problem into smaller subproblems, solving subproblems recursively, and finally combining partial solutions to obtain a solution to the original problem. These three steps are called divide, conquer, and combine:

- Divide: breaks down the original problem into 2 or more similarly sized subproblems.
- Conquer: each subproblem is solved recursively, obtaining partial solutions.

- Combine: partial solutions are combined to reconstruct the solution to the original problem.

In this practice, we will implement the most representative algorithm of this strategy: mergesort.

3.1 Mergesort

Mergesort is an algorithm that sorts a vector using the Divide and Conquer strategy, as follows:

- Divide: the original vector is divided into two subvectors of equal or similar size (half of the original size).
- Conquer: if the vector has size 1, it is already ordered. If not, the same method is recursively called to continue with the subdivisions.
- Combine: the two ordered subvectors are merged (merge) to form a single ordered vector.

The following video illustrates how this algorithm works: *Merge sort in 3 minutes (YouTube)*. Its asymptotic cost is $O(n \log n)$. A pseudocode of the algorithm is provided in Appendix A. For more information, please check out the theory teaching material.

4 Empirical analysis of temporal complexity

To test the correctness of the implementations of these algorithms, as well as to empirically measure their temporal cost, we will follow the following methodology:

1. Generate random arrays of variable size.
2. Run sorting algorithm.
3. Measure execution statistics.

This way, we will execute both sorting algorithms with different array sizes, will analyze how these execution statistics evolve according to array size, and will compare the results obtained with the different algorithms. Specifically, the following statistics will be calculated:

- **Execution time** t. Measures algorithm's response time from user's perspective. This measurement is dependent on the platform on which the algorithm is executed (software and hardware), so it is not comparable with measurements made in other environments. In addition, it presents a high level of variability, since execution times depend on the current load or stress of the system on which the algorithm is executed. Its behaviour as a function of the size of the array will allow us to emprirically infer the asymptotic cost of the algorithm.

- **Number of steps executed by the algorithm** s. Total number of instructions executed. To simplify this calculation, only significant terms will be counted. In this specific case, it will suffice to calculate the total number of comparisons made between elements of the array. This measurement is platform independent, and completely deterministic. Furthermore, it will help us to deduce the asymptotic cost of the algorithm empirically, as we see in the following item. Note: in Appendix A, the pseudocode clearly states when an step should be counted for the mergesort algorithm (see variable `steps`).
- **Ratio between the number of steps and the size of the array** $\frac{s}{n}$. This value will inform us about the average number of comparisons made for each element of the array. In addition, it will help us to deduce the asymptotic temporal cost of the algorithm, as follows:
  - $\frac{s}{n} \approx 0 \rightarrow$ Constant cost $O(1)$.
  - $\frac{s}{n} \approx 1 \rightarrow$ Linear cost $O(n)$.
  - $\frac{s}{n} \approx \log n \rightarrow$ Cost $O(n \log n)$.
  - $\frac{s}{n} \approx n \rightarrow$ Quadratic cost $O(n^2)$.
  - ...

5 Implementation details

Both sorting algorithms will be implemented as public static methods in a class called *Sorting*. This class will also include a static class attribute of type *SortingStats* called `stats`, that will record execution statistics (see Section 4) of the last sorting algorithm run. Finally, we will create a *Main* class to perform sorting tests with random Integer arrays of different sizes, using both algorithms.

5.1 Class *Sorting*

Class *Sorting* will be a *suite* of sorting algorithms, offered as static methods. For now, due to the time constraints of our developers, it will only offer two algorithms: selection sort, and mergesort, with their respective methods `selectionSort()` and `mergeSort()`. Their implementations must allow genericity, so we must make sure that 1) the generic type implements the *Comparable* interface, and 2) we use the `compareTo()` method to perform comparisons; otherwise, we will not be able to sort arrays, or sorting will be done incorrectly. Additionally, *Sorting* will include a public static method `array2str()` that, given a generic array, will return its string representation; as well as a static class attribute `SortingStats stats`, that will record execution statistics of the last sort algorithm run. This attribute will be used internally by the sorting methods to record the

necessary execution data and statistics, and externally by the users of our *suite* to consult these statistics (as we will do in the *Main* class).

Class *Sorting* must follow this template:

```
package P4;
public class Sorting {
    public static SortingStats stats = ...;
    public static <E extends Comparable<E>> void selectionSort(E[] v) {...}
    public static <E extends Comparable<E>> void mergeSort(E[] v) {...}
    public static <E> String array2str(E[] v) {...}
    ... // + other private methods
}
```

5.2 Class *SortingStats*

Class *SortingStats* should record all the information necessary to calculate the execution statistics described in Section 4. To the user, it must offer, at least, the following public methods:

```
package P4;
public class SortingStats {
    ... // class attributes, public or private
    public int size(){...} // size of the array
    public long steps(){...} // execution steps
    public float sizeStepsRatio(){...} // steps/size ratio
    public long execTime(){...} // execution time, in nanoseconds
    ... // + other public methods used by algorithms in class Sorting
}
```

These statistics will refer to the last sorting method executed, so it will be necessary to restart them before these methods are executed again.

To measure execution times, we recommend using the *nanoTime()* method of class *java.lang.System*. This method returns the number of nanoseconds elapsed since the start of the Java virtual machine. Therefore, to determine the exact execution time, it will be necessary to obtain times corresponding to the instants in which the algorithm starts and ends, and calculate the difference.

5.3 Class *Main*

The Main class will contain a method `main()` that will carry out empirical evaluation tests of the temporal cost of the sorting algorithms implemented in this practice, using public methods from the *Sorting* and *SortingStats* classes. This would be a very basic example of usage:

```
Integer[] v = new Integer[100000]; // create an Integer array
... // assign random values to all components of array v
Sorting.mergeSort(v); // call static method mergeSort
System.out.println("Sorted array:"); // print sorted array
System.out.println(Sorting.array2str(v));
System.out.println("Execution stats:"); // print stats
System.out.println("- size: " + Sorting.stats.size());
System.out.println("- steps: " + Sorting.stats.steps());
System.out.println("- ratio: " + Sorting.stats.sizeStepsRatio());
System.out.println("- time(s): " + Sorting.stats.execTime()/1E09); // ns -> s
```

To automate tests as much as possible, random vectors of type *Integer* of different sizes will be generated. For the generation of random numbers it is recommended to use the static method *Math.random()*. Note that this method returns a `double` between 0 and 1. Therefore, it will be convenient to multiply the returned value by some constant $C > 1$ (e.g. `Integer.MAX_VALUE`) to obtain values between 0 and C.

Given the high variability in the measurement of execution times, the same test must be run several times and the average of the times obtained must be calculated.

Statistics will be shown appropriately in a table similar to this:

```
         |            SelectSort                |              MergeSort
Size     |   Steps            Ratio    Time(s) |       Steps   Ratio    Time(s)
-------------------------------------------------------------------------------
20000    |   199990000        10000    0,627   |       260925  13       0,006
50000    |   1249975000       25000    3,852   |       718231  14       0,015
100000   |   4999950000       50000    16,124  |       1536446 15       0,031
200000   |   19999900000      100000   66,954  |       3272598 16       0,087
```

Due to its versatility, it is recommended to use the method *System.out.printf()*, instead of the cassical *System.out.println()*. For instance:

```
String name = "Pau";
int age = 1;
double w = 11.572;
System.out.printf("%s is %d year(s) old and weighs %.1f kg", name, age, w);
// Output: "Pau is 1 year(s) old and weighs 11.6 kg"
```

6 Deliverables

As part of the assessment of the lab practice 4, the following exercises must be done:

1. Full implementation of classes *Sorting, SortingStats* and *Main*.

These exercises will be bundled into a single compressed file (`.zip`, `.tgz`, etc.) containing the source code of the project, and delivered via a poliformaT task created for this purpose. All `.java` files should be encapsulated within the same package called `P4` (`package P4;`).

A mergeSort pseudocode

```
algorithm mergesort(T[1..n])
  if n is small enough then sort(T) end_if
  else
    U := T[1..(n div 2)]              // DIVIDE
    V := T[1+(n div 2)..n]
    mergesort(U)                      // CONQUER (sub-array 1)
    mergesort(V)                      // CONQUER (sub-array 2)
    merge(T, U, V)                    // COMBINE
  end_if

algorithm merge(T[1..n], U[1..m], V[1..r])
  i := 1, j := 1, k := 1
  while i <= m and j <= r do
    if U[i] < V[j] then T[k++] := U[i++]
    else T[k++] := V[j++]
    end_if
    steps++  // add 1 step to execution stats
  end_while
  for s := i to m do T[k++] := U[s] end_for
  for s := j to r do T[k++] := V[s] end_for
```