TD 5: Comparison Sorting

This assignment will give you practice about sorting methods.

Pendant ce TD, vous suivrez dans le cours qui vous est donné la spécification des algorithmes et vous les implémentez au fur et à mesure. Bien sûr, ces algorithmes sont accessibles partout sur le web (et dans les livres :-)). Essayez de les implémenter seuls, néanmoins ;-)

During these directed study sessions, you will follow the course material that has been given to you regarding the specification of the algorithms and you will implement them as you go. Of course, these algorithms are accessible everywhere on the web (and in books :-)). Try to implement them on your own, nevertheless ;-)"

⇨ **Test classes are given to you, do not hesitate to complete them and even share them with the entire class! But let's stay positive!**

When all the sorting methods are implemented, you can run *ForRuntimePerformanceTesting*, to automatically compare runtimes.

Part 1: simple sorting methods (15 min)

In this part you must implement **at least one of the two most common quadratic sorting algorithms**:

- *selection sort*
- *insertion sort*

Complete the class template SimpleSorting by writing all the methods. For both methods, give the best and worst cases running time complexity and give an array configuration when they occur.

Supporting file

- SimpleSorting.java and corresponding tests

Part 2: heap sort method (15 min)

In this part, you must implement the *heap sort* algorithm using the BinaryHeap class you designed in the previous lab.
Complete the class template HeapSort by writing all the methods.

Supporting file

- HeapSort.java

if you failed to design the *BinaryHeap* class in the previous lab, you can pick up the class in the resources.

Please note that the sorting function includes building the heap... in this version.

Part 3: merge sort method (30 min)

In this part, you must implement the recursive merge sort algorithm. This algorithm is not in place and needs to use a second array of the same size as the array to sort. Complete the class template MergeSort by writing all the methods.

Supporting file

- MergeSort.java

Part 4: quick sort method (45 min)

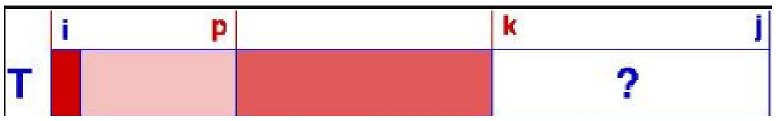
If you are running late, move on to the next exercise to better understand the impact of complexity on time. For those who are more advanced, feel free to analyze other measures such as required memory size.

In this part you must implement the recursive *quick sort* algorithm. This algorithm is based on a partition method which must be carefully designed.
Your implementation should follow these requirements:

- the partition method must be the one describes below
- your algorithm should switch to insertion sort algorithm from part 1 below a cutoff you have to find yourself by experimenting
- you must provide a new insertion sort method to match the new context (i.e., this new method should have more than one parameter)

The partition method should work as follow:

- choose the pivot as the median of 3 like explained on slides
- swap the pivot with the first item in the array slice
- partition using a loop which matches the following invariant:



For all a in [i+1, p], b in [p+1, k[, T[a] < T[i] <= T[b]

Supporting file

- QuickSort.java

Part 5 : Compare runtimes (30 min)

For those who are more advanced, analyze other measures such as required memory size.

Using and adapting *ForRuntimePerformanceTesting* compare the runtimes of these different algorithms **against java sort runtimes.**