Tutorial n°6 - Sorting Algorithms

The focus of this lab is to study various algorithms to sort a table of integers and study their complexities. We will consider that the table has to be sorted in ascending order.

1 Preliminary work

1. Define a class *CTable* that will contain a dynamic table of integers (use integer pointers) and the number of elements.
2. Add a builder that randomly initializes the table with integers for a given number of elements, and the default one, without argument.
3. Add a function to display the table.
4. Add common functions such as insertion of a new element at a given position, read an element, and swap 2 elements.

2 Simple algorithm: Bubble Sort

On each pass, bubble sort scans the array, comparing each pair of adjacent elements. If two adjacent elements are out of order, they are swapped. As long as at least one swap is performed along the scan, another pass is computed.

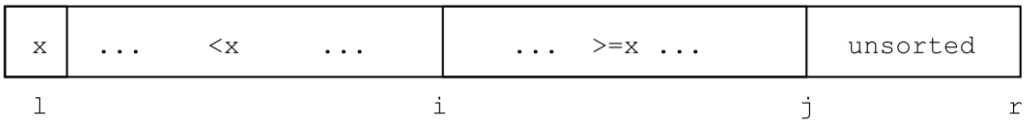
1. Add a function to the class that realizes the Bubble Sort of the table.
2. Study the complexity of this algorithm as a function of the size of the table.

3 Quicksort

Quicksort works in a "divide and conquer" manner, as follows:

- split the initial list of numbers into parts around a "pivot"; all the values in the first part are less than the pivot; all the values in the second part are greater than or equal to the pivot.
- recursively sort the two parts.

The first element $a[l]$ is used as a pivot (which we will call x). It then divides the array of r elements into two regions: elements less than x, and elements greater than or equal to x. This is a snapshot of the partition of the table during computation:

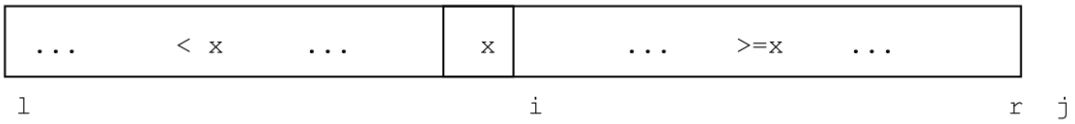


Partitioning operates by examining element $a[j]$. There are two possible cases:

- $a[j] < x$
  $a[j]$ belongs in the $< x$ partition, so swap $a[j]$ with $a[i]$, then increment both i and j
- $a[j] >= x$
  $a[j]$ belongs in the $\geqslant x$ partition, which is where it is now; nothing needs to be done to $a[j]$, so simply increment j

After processing all the elements in the array (*i.e.* $j > r$), both regions will be complete.

The final step is to move the pivot between the two regions. This is done by swapping $a[l]$ with $a[i-1]$, giving us:



1. Add a function to the class that realizes the Quicksort of the table.
2. Study the complexity of this algorithm as a function of the size of the table.
3. Can you think about a case that would increase the complexity?

4 Other simple algorithms

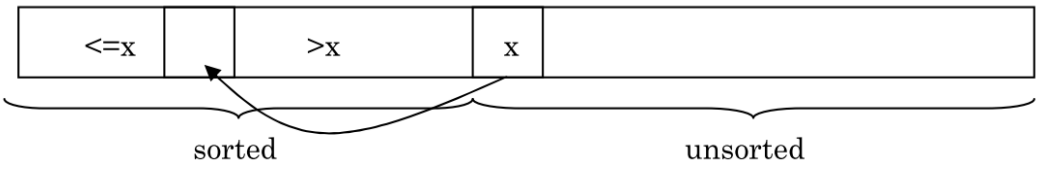
4.1 Selection Sort

In selection sort the array is divided into two parts: the first part that is sorted and the second part that is not sorted yet. Initially the sorted part is empty and the unsorted part consists of the whole array. In each step, the algorithm searches through the unsorted part, finds the smallest element and puts it at the end of the sorted part.

1. Add a function to the class that realizes the Selection Sort of the table.
2. Study the complexity of this algorithm as a function of the size of the table.

4.2 Insertion Sort

The initialization of the algorithm is similar to the selection sort, dividing the array into a sorted and an unsorted part. Each step of the algorithm picks the first item of the unsorted array (x) and inserts it into the right slot of the sorted array. The elements of the sorted array $> x$ are shifted one location forward.



1. Add a function to the class that realizes the Insertion Sort of the table.
2. Study the complexity of this algorithm as a function of the size of the table.