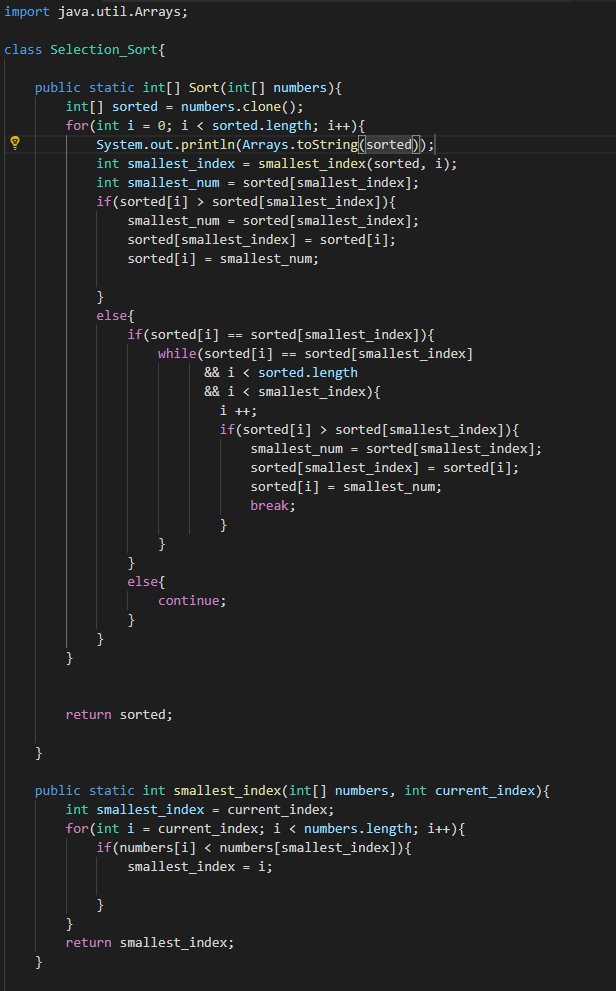
## IN2010 H18

## Mandatory assignment3: Sorting

**Haiyuec**

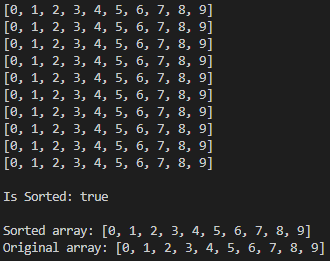
**Selection sort:**

**Code**:



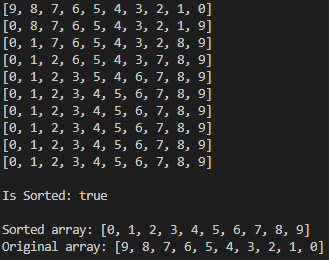
**Patterns:**

**Ascending values:**

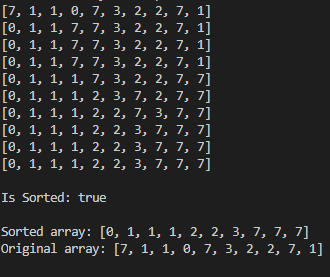


We can see that selection sort does not do any real changes when the input array is already sorted in an ascending order. For every iteration the algorithm finds the smallest element in the sub-array, which in this case is always the first element, and swaps the smallest element with the first element in the sub-array. Thus, the algorithm does not actually change the array.

**Descending values:**



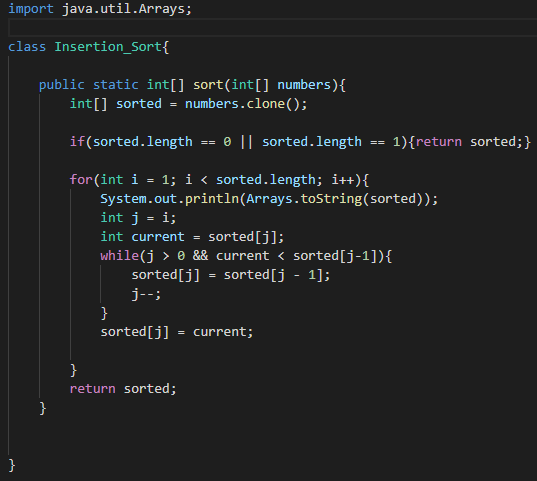
We can see that selection sort finds the smallest element in the sub-array and swaps it with the first element in the sub-array. When the input values are sorted in a descending order, it is almost like there are two indices that iterates the array from the beginning and from the end, and swaps the elements as the iteration continues, until the two indices meet in the middle. In this example, this process takes 5 iterations. After 5 iterations, the array is already correctly sorted and the algorithm does not do any meaningful changes to the array.

**Random values:**

In this example, we can see the “normal” pattern of the selection sort algorithm. The algorithm chooses the smallest element from the sub-array, swapping it with the first element in the sub-array, and continues until it has iterated all the indices of the input array.

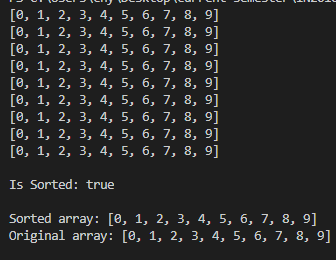
**Insertion sort:**

**Code**:



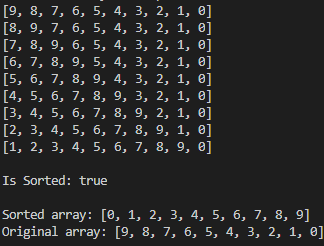
**Patterns:**

**Ascending values:**



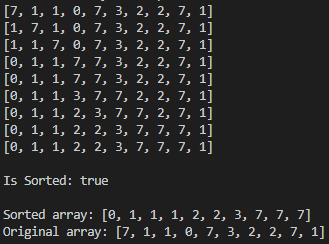
When the input array is already sorted in an ascending order, the algorithm does not do any changes. This is because insertion sort checks if the current element is smaller than the previous element, if it is, the element is moved forward in the array until the current element is not smaller than the previous element, or is currently the first element in the array. When the input array is already sorted in the ascending order, this operation never happens.

**Descending values:**



When the input array is sorted in descending order, the algorithm for every iteration “picks up” the current element and inserts it at the beginning of the array. This “pushes” the other elements and causing a “shifting effect”.

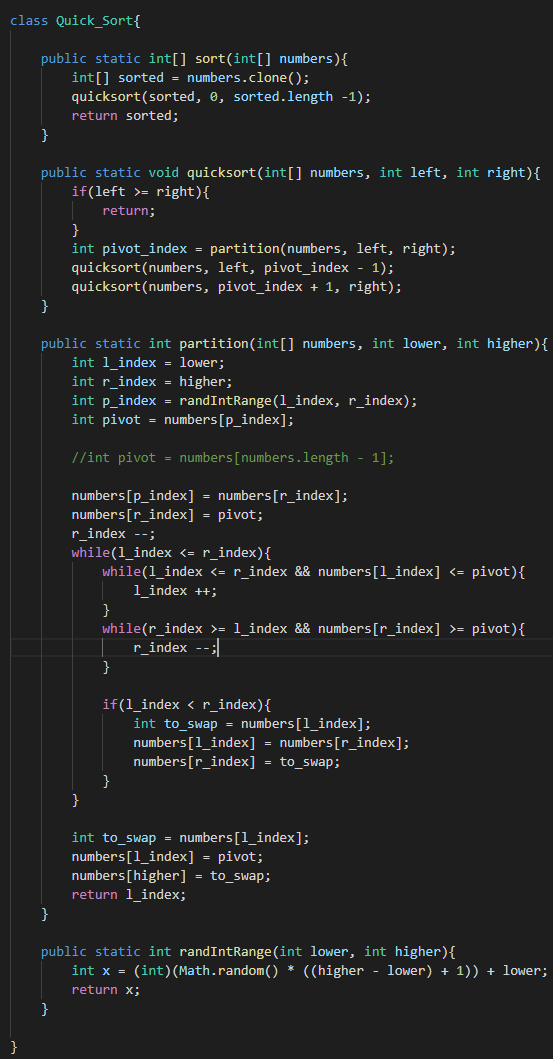
**Random values:**



This example is for random inputs, and it works just as the algorithm description.

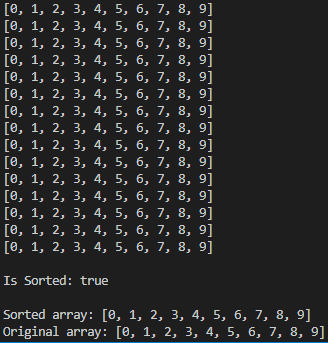
**Quick sort:**

**Code**:

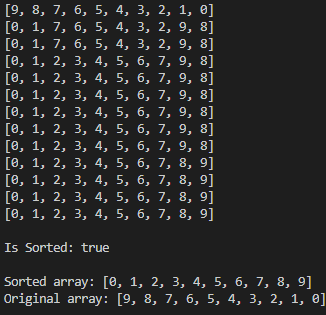


**Patterns:**

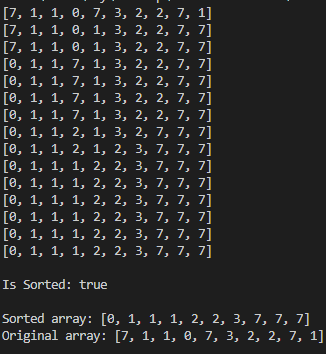
**Ascending values:**



**Descending values:**



**Random values:**



I did not see any special patterns for sorting numbers for the quick sort algorithm. However I did notice that if the input array are too large, it would cause OutOfMemoryError. This is because this implementation uses recursion, if the input is too large, the JavaVM would run out of stack space. This problem could be solved by implementing this algorithm in an iterative manner, which is also described in the book.

**Bucket sort:**

**Code**:



I chose to implement the algorithm with a hashmap with linkedlists rather than the array as described in the book. I did this change to handle input arrays that has negative values. I know also that there are other ways to tackle this particular problem, such as having a variable to offset the index, which are faster than this hashmap implementation. This was the easiest way so I did it like this.

**Patterns:**

The bucket sort algorithm does not sort the array in place as the other algorithms, so no iteration printing are collected.

There were no particular patterns with this algorithm regarding the sorting. However, do to the fact that the algorithm needs to check every bucket after the numbers are assigned, this algorithm would perform better when there is a large number of repeating numbers in the input array.

**Speed Tests:**