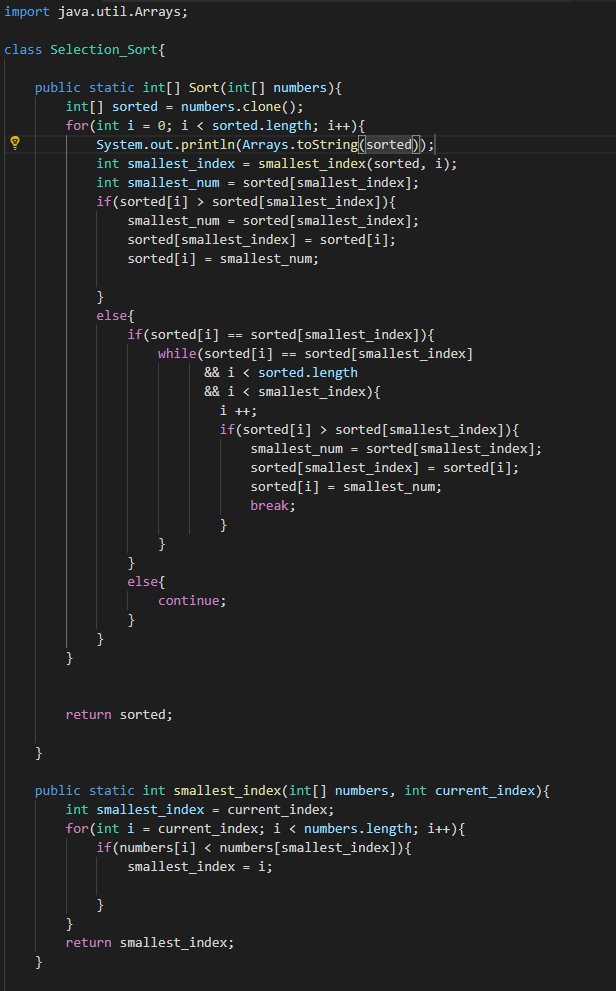
## IN2010 H18

## Mandatory assignment3: Sorting

**Haiyuec**

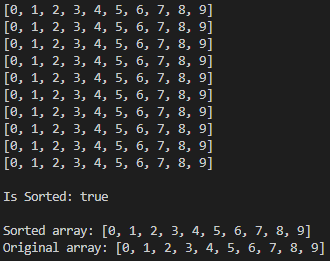
**Selection sort:**

**Code**:



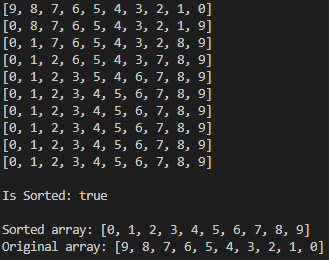
**Patterns:**

**Ascending values:**

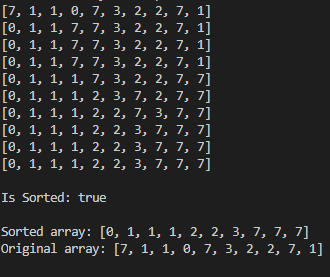


We can see that selection sort does not do any real changes when the input array is already sorted in an ascending order. For every iteration the algorithm finds the smallest element in the sub-array, which in this case is always the first element, and swaps the smallest element with the first element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

**Descending values:**



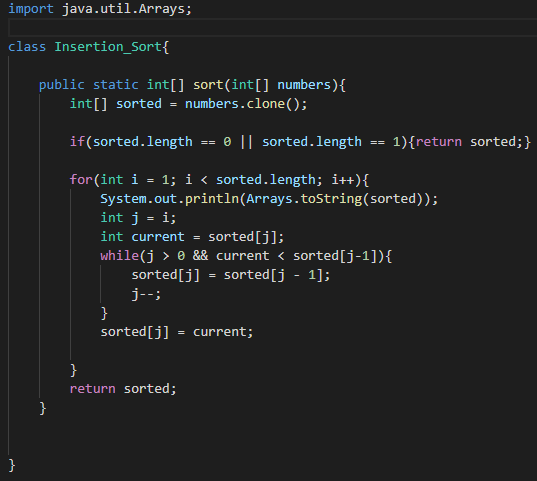
We can see that selection sort finds the smallest element in the sub-array and swaps it with the first element in the sub-array. When the input values are sorted in a descending order, it is almost like there are two indices that iterates the array from the beginning and from the end, and swaps the elements 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.

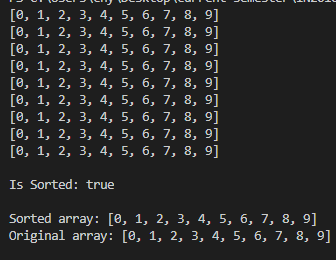
**Insertion sort:**

**Code**:



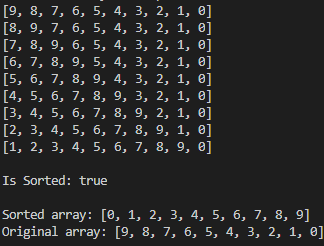
**Patterns:**

**Ascending values:**



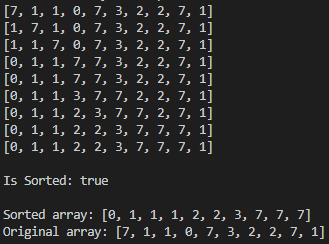
When the input array is already sorted in an ascending order, the algorithm does not do any changes. This is because insertion sort checks if the current element is smaller than the previous element, if it is, the element is moved forward in the array until the current element is not smaller than the 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”.

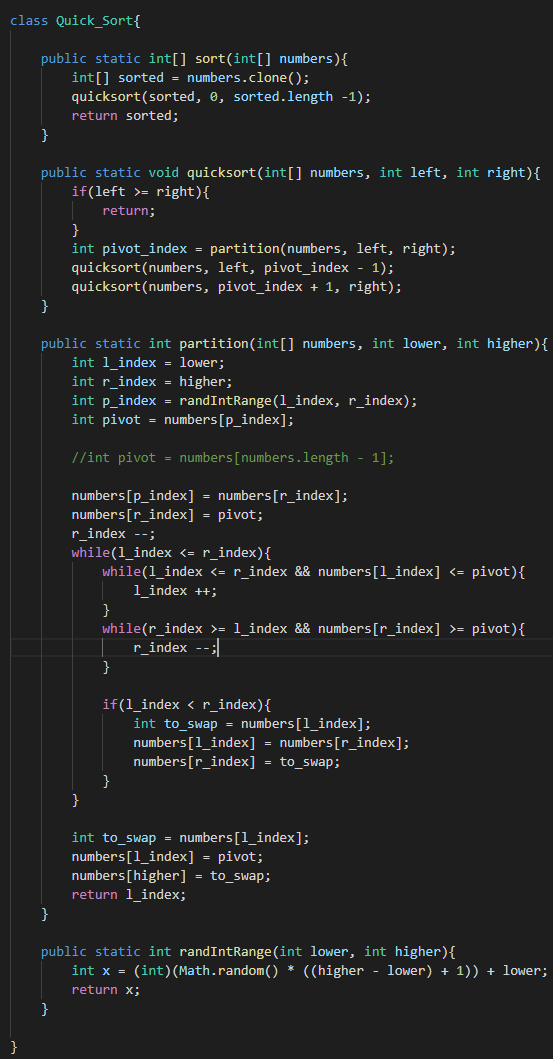
**Random values:**



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

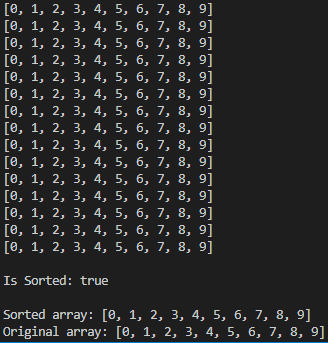
**Quick sort:**

**Code**:

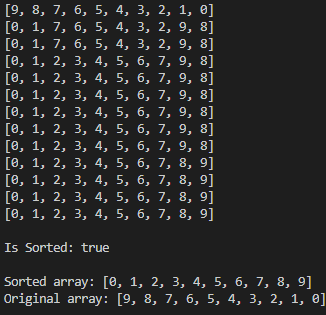


**Patterns:**

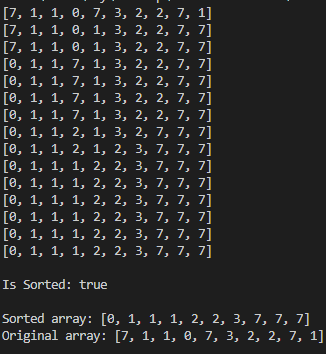
**Ascending values:**



**Descending values:**



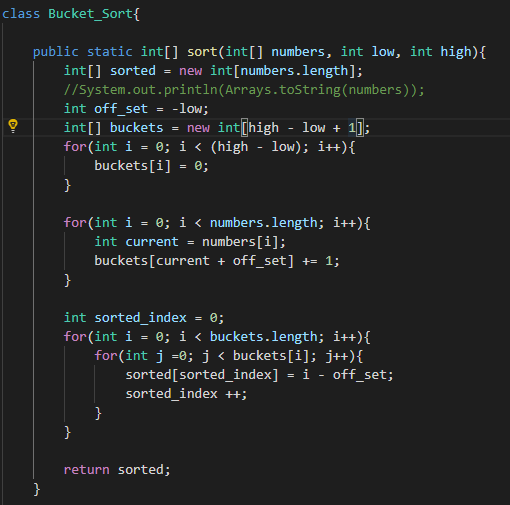
**Random values:**



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 an array with the values as indices, and number of occurrences as values. I have also added an off\_set variable to create a shift in the “keys”, so that this implementation could handle negative values

**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:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ascending | Nr. Elements | Selection sort | Insertion sort | Quick sort | Bucket sort | Array.sort |
|  | 1000 | 4 | 0 | 1 | 2 | 77 |
|  | 5000 | 8 | 0 | 0 | 3 | 0 |
|  | 25000 | 75 | 0 | 4 | 14 | 0 |
|  | 125000 | 1582 | 1 | 5 | 18 | 1 |
|  | 625000 | 37738 | 2 | 28 | 96 | 1 |
|  |  |  |  |  |  |  |
| Descending | 1000 | 0 | 0 | 0 | 0 | 0 |
|  | 5000 | 2 | 0 | 0 | 0 | 0 |
|  | 25000 | 73 | 0 | 1 | 1 | 0 |
|  | 125000 | 1483 | 0 | 7 | 6 | 0 |
|  | 625000 | 37924 | 1 | 32 | 57 | 2 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Random | 1000 | 1 | 2 | 0 | 9 | 0 |
| range (-10000 to 10000) | 5000 | 8 | 12 | 3 | 4 | 2 |
|  | 25000 | 161 | 73 | 22 | 4 | 2 |
|  | 125000 | 3779 | 1281 | 13 | 9 | 18 |
|  | 625000 | 94337 | 35225 | 74 | 23 | 81 |
|  |  |  |  |  |  |  |
| Random | 1000 | 5 | 2 | 1 | 476 | 0 |
| range | 5000 | 15 | 12 | 1 | 486 | 1 |
| (-10000000 to 10000000) | 25000 | 205 | 74 | 3 | 469 | 4 |
|  | 125000 | 3797 | 1277 | 13 | 497 | 25 |
|  | 625000 | 93010 | 35217 | 81 | 711 | 88 |

The places where JAVAs Arrays.sort() method are better are marked in red. I did not understand what the assignment meant by testing with no limited on the variation of the inputs, as the bucket sort algorithm need the upper and lower range to work. I instead tested the algorithms with two drastically different variations of range.

None of the results are actually surprising.

The selection sort algorithm has O(n2) and performed in fact the worst of all the algorithms.

The insertion sort has the best performance for ascending and descending values. This is not that surprising as those are the best case scenarios for the algorithm. Insertion sort performs O(n) operations. However, the performance falls significantly when the inputs are random. This is much closer to the worst case scenario for insertion sort, where is has O(n). Because these input are not exactly the worst of the worst, insertion sort still performs better than selection sort.

The quick sort algorithm has the most consistent random input performance. It is beaten by the bucket sort algorithm when sorting random numbers within small intervals, but is far more efficient than bucket sort when the interval is greater. This is not that surprising, because the quick sort algorithm does not care about the variation in the inputs, and therefore has consistent performance. This is true to the complexity of the algorithm, which is O(nlog(n)).

The bucket sort algorithm performed very well in sorting random numbers with small variations. This is caused by the specific implementation of this algorithm. My implementation creates a bucket for every possible value in the input array, which causes significant overhead. This can certainly be improved by implementing the algorithm differently. One of my ideas is to not create any buckets in the beginning of the algorithm, and only create new ones when the number is new and does not belong to any existing bucket. This way we can avoid some of the problems regarding to the unnecessary amount of buckets.

There is another problem with this approach to the bucket sort algorithm. By creating a bucket for every possible value in the input, it is very easy to exceed the RAM limit on the machine. Therefore it is better to create buckets that are for a small interval, for instance 1000, and use other algorithms, such as insertion sort or quick sort, to sort the individual buckets. This way we can reduce the space usage significantly and avoid some of the problems. The real world result also matches the theoretical complexity.