Lab lecture exercises – 28 October 2016

Sorting algorithms are algorithms that leave the elements in an array unchanged, but bring them into an order so that the array is sorted. We consider two important sorting algorithms. The first, `selectionSort`. It is relatively easy, but inefficient and cannot be used on big arrays. The second, `quickSort`, is more complicated, but also one of the most efficient sorting algorithms. First however, we write a method that checks whether an array is sorted in increasing order.

1. **Method for Checking the Sortedness of an array**

   Write a method `public static boolean isSorted(double[] arr)` which tests whether a given array is sorted in increasing order. E.g., the array
   `a1 = {1.0, 1.1, 2.0, 2.0, 3.0}` is sorted, but the array
   `a2 = {1.0, 1.1, 2.1, 2.0, 3.0}` is not.

2. `selectionSort`

   `selectionSort` is an algorithm for sorting arrays (e.g., of type `double[]`). The idea is to consider the array consisting of two parts: the initial part (from the left) which is sorted (and initially empty) and the rest which is unsorted (and initially the whole array). In each iteration the smallest element from the unsorted part is selected and put at the end of the sorted part by swapping. That is, in the first round the smallest element in the whole array is selected and swapped with the element at position 0. In the next round the smallest element in the unsorted part of the array (from position 1 on) is determined and swapped with the element in position 1 and so on until the whole array is sorted. In pseudo code this is:

   ```
   int min;
   for (int i=0; i < a.length; i++){
         determine min as minimum between i and a.length -1;
         swap a[i] and min;
       }
   ```

   Implement a method `public static double[] selectionSort(double[] a)` in a class `Sorting.java` implementing this algorithm. Experiment with the method with some randomly generated arrays using the class `SortingMain.java` in the lab material for week 5 (from Canvas).

3. `quickSort`

   In order to get a more efficient algorithm it is necessary to reduce the problem size dramatically (ideally by halving it) in each single step. `quickSort` does this by determining a so-called `pivot` so that all elements of the array that are smaller than the pivot end up in the left sub-array and all elements that are bigger than the pivot end up in the right sub-array. Then the method is applied again to the left and the right sub-arrays until the sub-arrays consist only of one element, which are trivially sorted. Assume we are working on an array `a` with entries `a[0],...,a[size-1]`, then we can split the set of indices. That is, we consider the two sub-arrays arrays `a[0],...,a[m-1]` as well as `a[m + 1],...,a[size-1]` and pivot `a[m]` and swap elements between the two sub-arrays so that all the elements in the first are smaller than or equal to `a[m]` and `a[m]` is smaller than or equal to all the elements in the second.

   The same process is repeated on the sub-array until every sub-array consists only of one element, that is, the sub-array is trivially sorted.

   In our implementation, the pivot should be chosen to be the element in the middle between the start and the end index of the sub-array. (In general, more sophisticated ways are used.)

   Implement a corresponding method
   `public static double[] quickSort(double[] a)` in the class `Sorting.java`. Experiment with the method with some randomly generated arrays using the class `SortingMain.java` in the lab material for week 5 (from Canvas). You will find a stub `Sorting.java` in the lab material for week 5 (from Canvas).