Algorithms and Data Structures (ADS2)

Laboratory Sheet 1

This Lab Sheet contains material based on Lectures 1-2. This exercise is not for submission but should be completed to gain sufficient experience in the implementation of elementary sorting algorithms, and the testing thereof.

Setup
Datasets for tests can be downloaded from Moodle under Labs/Files.

Exercise
Implement two sorting algorithms for arrays of integers in Java, create a test program to check your implementations, and test the performance of each of the algorithms on a suite of test files provided. You are not allowed to use library functions and data structures e.g. `ArrayList`.

Part 1

a) Following the pseudocode for INSERTION-SORT introduced in Lecture 1 (slide 13), implement the `InsertionSort` algorithm in Java.
b) Write a test program, `TestSortingAlgorithms`, to check that your implementation is correct, namely that the output array is in *ascending order*.
c) What is the complexity of `TestSortingAlgorithms`?
d) Implement `InsertionSortDescending` to sort arrays in *descending order*.

Part 2 SELECTION-SORT is a sorting algorithm informally described as follows:

Input: an array A of integers (with indices between 0 and n-1)
Output: a permutation of the input such that A[0] ≤ A[1] ≤ … ≤ A[n-1]

Algorithm: Array A is imaginary divided into two parts - sorted one and unsorted one. At the beginning, the sorted part is empty, while unsorted one contains the whole array. The algorithm sorts A by repeatedly picking the minimum element from the unsorted subarray and moving it to the end of the sorted subarray.

a) Write pseudocode for SELECTION-SORT corresponding to the natural language description above.
b) What is the running time of SELECTION-SORT in the worst case? And in the best case? How does it compare to INSERTION-SORT?
c) Is it a stable sorting algorithm?
d) Does it sort in-place?
e) Implement the `SelectionSort` algorithm in Java following your pseudocode for SELECTION-SORT. Use `TestSortingAlgorithms` to check that your implementation is correct.

Part 3 You have been provided with a suite of test text files `int10.txt`, `int50.txt`, `int100.txt` and `int1000.txt` where each `int`*n*`.txt` contains *n* integers in randomly sorted order. Write a program `TimeSortingAlgorithms.java` to generate timing runs for `InsertionSort` and `SelectionSort` outputting the time taken to sort (an array read from) each of the text files above. Example output might be something like:

```
-------------------------------------------------------
Time taken to sort int10.txt:


InsertionSort: 310 milliseconds

SelectionSort: 530 milliseconds


-------------------------------------------------------
Time taken to sort int50.txt:
...
```

Hints

(1) You are allowed to use an intermediate `ArrayList` to read the input files and then copy it to an `int` array:

```
public int[] copyArray(ArrayList<Integer> data) {

     int n = data.size();
     int[] res = new int[n];

     for (int i = 0; i<n; i++){
         res [i] = data.get(i);
     }

     return res;
    }

public int[] readArray(String path) {

     Scanner sc = new Scanner(new File(path));
     ArrayList<Integer> data = new ArrayList<Integer>();

     while (sc.hasNextInt()) {
          data.add(sc.nextInt());
     }

     return copyArray(data);
    }
```

(2) Program `TestSortingAlgorithms` takes an array of integers as input and returns `true` if the input is sorted in ascending order, `false` otherwise.

(3) Example run of SELECTION-SORT with input A = [4,9,2,5]:
- Iteration 1: minimum of [4,9,2,5] is 2, swap with 4 to obtain [2,9,4,5]
- Iteration 2: minimum of [9,4,5] is 4, swap with 9 to obtain [2,4,9,5]
- Iteration 3: minimum of [9,5] is 5, swap with 9 to obtain [2,4,5,9]

(4) You can use the `currentTimeMillis()`(or `nanoTime()`) method of class `System` to determine the current time in milliseconds. E.g. to store the current time in variable `time1` of type `long`:

```
long time1=System.currentTimeMillis();
```

(5) If you run your `TimeSortingAlgorithms.java` program several times, you will get different (but similar) results. Ideally one would run each experiment a number of times (e.g. 10) and take the average time taken, but this would of course increase the amount of time that would be required to collect a complete set of results. If you have time, extend your program to allow for multiple runs in this way.