COSC 1437 JAVA DATA STRUCTURE

Lab 7: Sorting in Java

In this lab you will implement three different sorting algorithms: **insertion sort, quicksort and merge sort**. You will also compare their performance on various sorts of input data.

The main file

Start by downloading Sorting_lab.java. This file is where you will put your three sorting algorithms. Opening it, you will see three sorting methods:

```
public static void insertionSort(int[] array)
public static void quickSort(int[] array)
public static int[] mergeSort(int[] array)
```

At the moment, these three are just skeleton implementations that throw an `UnsupportedOperationException` when called. **Your main job is to finish the implementation.**

Just below these three methods, you will find skeleton implementations of some helper methods that you might find useful:

- Versions of `quickSort` and `mergeSort` that are given a parameter that specifies which part of the array to sort.
- A method `int partition(int[] array, int begin, int end)` that does the partitioning step from Quicksort. After it's finished, it should return the final location of the pivot element.
- A method `int[] merge(int[] left, int[] right)` that merges two sorted arrays into one, used for mergesort.

There is also a method `void swap(int[] array, int i, int j)` that swaps two elements of an array, that you may find useful. It is already fully implemented.

The benchmark tool

You should also download Bench.java. This is a benchmarking program for your three implementations. To run it, compile both Sorting_lab.java and Bench.java and run the

`Bench` class. You can already run it: try it and it will print out a benchmark report consisting of lots of tables. Right now they all look like this:

```
Arrays of length 1000
================================================================
Algorithm      |         Random |     95% sorted |         Sorted
Insertion sort |              - |              - |              -
Quicksort      |              - |              - |              -
Merge sort     |              - |              - |              -
```

This particular table shows the performance of the three algorithms for arrays of 1000 elements (see the heading at the top). Because you haven't implemented any of the algorithms yet, all of the table cells are filled with dashes, but eventually it will look more like this:

```
Arrays of length 1000
================================================================
Algorithm      |         Random |     95% sorted |         Sorted
Insertion sort |       0.242677 |       0.022095 |       0.002044
Quicksort      |       0.045372 |       0.042710 |       0.455633
Merge sort     |       0.075152 |       0.050483 |       0.047600
```

The benchmark program tries your algorithms on three sorts of data:

- Completely random arrays (the leftmost column)
- Arrays that are already in the right order (the rightmost column)
- Arrays where 95% of elements are in the right order, but the rest are chosen at random (the middle column)

All times are reported in milliseconds. So here we can see that insertion sort took 0.242ms for a random list of 1000 elements while quicksort on a 95% sorted list of 1000 elements took 0.0427ms.

Once you have implemented one of the algorithms, you can re-run Lab1.java and the numbers for that algorithm will be filled in. If there is a bug in your algorithm, the table will not contain numbers, but rather a description of what went wrong:

- A dash "`-`" if your method throws an `UnsupportedOperationException` (which is what you get at the moment).
- The string `INCORRECT` if your method gave the wrong answer.
- The string `EXCEPTION` if your method threw an exception.
- The string `STACK OVERFLOW` if your method recurses too deep and overflows the stack.

The testing tool

Probably your implementation will have bugs at first. At least, mine did! In that case, the benchmark report will just say `INCORRECT` or `EXCEPTION`. Not very helpful!
To help you track down bugs, there is a testing program, Test.java. This program tests a sorting algorithm on arrays of various sizes and prints a message when it goes wrong. You need to tell it what algorithm to use: near the top of the file there is a line that reads `Bench.insertionSort`. By changing it to `Bench.quickSort` or `Bench.mergeSort` you can change which algorithm is tested. After doing that, just compile all three `.java` files and run the `Test` class.

A sample output of the testing program is:

```
Testing on arrays of size 0...
Testing on arrays of size 1...
Testing on arrays of size 2...
Failed!
Input array: {2, 4}
Expected answer: {2, 4}
Actual answer: {2, 0}
```

This shows that we tried to sort the array `{2, 4}`, which should give the result `{2, 4}`. However, the sorting algorithm (mergesort in this case) mistakenly gave the answer `{2, 0}`. Once you have this test case, you could for example add a `main` method to `Sorting_lab.java` that just tries this particular test case, then add `println` statements to trace execution and see what goes wrong.

Note that if your algorithm is correct, the testing program will just try out bigger and bigger arrays until the end of time. So you should only use it if the benchmarking program reports a problem.

Your job

Your first job is to implement the three sorting algorithms as follows:

- `insertionSort` should implement insertion sort. It should use the optimisation discussed in the lecture where you move elements upwards instead of swapping.
- `quickSort` should implement quicksort, picking the first element of the array as the pivot.
  Note: your version of quicksort might crash with `STACK OVERFLOW` for large sorted inputs. This is allowed for sorted arrays of size 30000 and 100000 but not in any other cases.
- `mergeSort` should implement mergesort. As `mergeSort` is not in-place, it returns a new array instead of modifying the existing one.

You will probably find the existing skeleton code and helper methods useful, but they are only a suggested way to structure your implementation: if you have a better idea, go for it! But you must not change the types of the sorting methods themselves.

Once all the algorithms are working, your second job is to compare their performance by looking at the benchmark results. It's up to you what you look at, but in the end you should write a few sentences to say when you would recommend each algorithm. Questions you could answer include:

- Is there a size cutoff at which a different algorithm becomes best?
- How does the type of test data affect which algorithm you should choose?
- Which should you choose if you don't know anything about the test data?
- Are there circumstances when you should definitely avoid a certain algorithm?

Submission

You should submit the following things to Blackboard:

- **Your final `Sorting_lab.java`.**
- **The output of the benchmark program.**
- **Your performance recommendations.**

Your submission will be rejected if:

- Your code is unreadable or extremely hard to understand. We expect reasonable code quality! Try to indent your code sensibly, avoid copying and pasting if you can, and comment parts that seem tricky.
- The benchmark reports any unimplemented ("`-`"), `INCORRECT` or `EXCEPTION` answers.
- The benchmark reports any `STACK OVERFLOWS`. **Exception:** Quicksort on arrays of size 30000 and 100000 may cause a stack overflow.