Introduction to Computer Science II
Grundlagen der Informatik II

April 9, 2014

Lab 1 - QuickSort

Submission Deadline: May 11, 2014 @ 23:59

Submission System:
`https://gdi2.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi`

1 Introduction

A library is using an radio-frequency identification (RFID) tag system to keep track of their books. Each book has a unique ID (stored on the RFID tag) and there is a number of readers placed inside the library. Each reader has its own serial number. The read values are stored in a central file. The entries in the file are written by each reader, and have the following format:

```
Book_serial_number;ReaderID;STATUS
```

2 Task

Your task is to implement a Java program which sorts the entries in this file in *ascending order* on the `Book_serial_number` and the `ReaderID`.
The `Book_serial_number` has a higher priority than the `ReaderID`. The `STATUS` is not considered during sorting.

For sorting, you shall implement two versions of the QuickSort algorithm:

1. `QuickSortA` (choose the pivot as the first element in the list)
2. `QuickSortB` (choose the pivot as the "median" of the first, last and mid element[1] in the list)

For instance, let's consider a list containing the following elements [3, 45, 78, 1, 23, 49]. At the first iteration of the algorithm, `QuickSortA` should choose 3 as the pivot element. For `QuickSortB` the three values are 3, 49 and 78, so the median is 49.

As a basis for your implementation we provide a Java project containing a set of auxiliary classes and several input files (see section 3). Please note that you MUST use this code framework.

2.1 Input

A sample input file looks as below.

```
Z8IG4;LDXS;OK
0X6F9;ERSY;OK
YSI7Q;ERSY;OK
6C8IV;ERSY;Error
YSI7Q;4O09;OK
EMBXP;GQ9Y;OK
5MXGT;7L8Q;Error
F0C9U;7L8Q;OK
XOH3X;GQ9Y;Error
```

[1] if the number of elements in the list is even, i.e. if there are two mid elements, use the one with the smaller index

```
XOH3X;ERSY;Error
XDYF6;P8OS;OK
GFN81;7L8Q;Error
FOC9U;7L8Q;OK
WN178;GQ9Y;OK
```

The `Book_serial_number` is a unique 5-character string, containing only capitalized letters and/or numbers. The `ReaderID` is represented by a 4-character string, also containing only capitalized letters and/or numbers. The `STATUS` is either `OK` or `Error`.

Note: The Semicolon is used as a separator in the file. An auxiliary class for reading the input file is provided, as well as several input files (see section 3).

2.2 Output

A sample output from your program (to the console) looks like this:

```
Starting QuicksortA tests!
QuicksortA [TestFile1]: Correct order! Read Ops: xx; Write Ops: yy
QuicksortA [TestFile2]: Correct order! Read Ops: xx; Write Ops: yy
Correct QuicksortA sortings: 2 out of 2 tests

Starting QuicksortA complexity tests!
Complexity QuicksortA [TestFile1]: Complexity within allowed range!
Complexity QuicksortA [TestFile2]: Complexity within allowed range!
Passed complexity tests for QuicksortA: 2 out of 2 tests

Starting QuicksortB tests!
QuicksortB [TestFile1]: Correct order! Read Ops: xx; Write Ops: yy
QuicksortB [TestFile2]: Correct order! Read Ops: xx; Write Ops: yy
Correct QuicksortB sortings: 2 out of 2 tests

Starting QuicksortB complexity tests!
Complexity QuicksortB [TestFile1]: Complexity within allowed range!
Complexity QuicksortB [TestFile2]: Complexity within allowed range!
Passed complexity tests for QuicksortB: 2 out of 2 tests
```

The output is generated by the JUnit test class automatically. Make sure that you **do not print anything** to the console from your methods in your final solution, in order not to mess up the output! In case of errors, a short explanatory message is presented.

2.3 Implementation Complexity

`Read Ops` and `Write Ops` represent the number of "read" operations (respectively "write" operations) executed by each sorting algorithm during sorting of the given input file. We use these counters as complexity measures to evaluate your implementation.

An effective implementation should not access (read and write) the elements in the list more than necessary. For instance, for the same test file, the two implementations of QuickSort may issue different outputs:

```
QuicksortA [TestFile1]: Correct order! Read Ops: 233734822; Write Ops: 50252410
QuicksortB [TestFile1]: Correct order! Read Ops: 779116; Write Ops: 167508
```

In this example, the `QuicksortB` implementation is about 300 times more effective than the `QuicksortA` implementation. Think about the number of times you access the list; try to keep this number low by reducing any unnecessary read or write operations. You are not allowed to use any temporary arrays in your code to reduce the number of accesses.

3 The Code

We provide the following Java classes and test files within the zip archive **Lab1.zip** which contains the Java project with the building blocks. The classes are split in two packages– the 'frame' and the 'lab' package. contains auxiliary classes.

3.1 The 'frame' package

DO NOT change any class in this package! The submission system will use its own copy of these classes when testing your source code.

3.1.1 `SortingLab.java`

This is a helper class that provides the sorting algorithms to the JUnit test class. It also contains the method responsible for reading input test files into a `SortArray` object.

3.1.2 `LibraryFileReader.java`

This class contains the method `readFile` for reading the input files into an ArrayList.

3.1.3 `SortArray.java`

This class stores the list to be sorted. It provides methods to read (`getElementAt`) or write (`setElementAt`) items from or to the list at a certain position. The methods provided in this class should be sufficient for you to sort the records of the input files. It also contains the two variables (`readingOperations` and `writingOperations`) that act as counters for the number of accesses to the arrays to be sorted. These are used by the JUnit tests to construct the output and test the complexity (see Section 2.2).

3.1.4 `AllTests.java`

This is the JUnit test case class. The test cases defined by this class are used to test if the input file was correctly sorted, and if the measured complexity of the sorting algorithms fits into the allowed range. Remember that Quicksort has the complexity O(n^2) in the worst case and O($nlog(n)$) in the best and average case. This class is also responsible for outputting to the console.

3.2 The 'lab' package

This package contains the files you are allowed to modify. You are free to add any additional methods or variables to the contained classes, as long you **do not change the signature (*name, parameter type and number*) of any given method**, as they will be used by the JUnit tests. Any source code changes that you make outside the 'lab' package will be ignored when you upload your source code to the submission system.

3.2.1 `QuickSort.java`

This class is the abstract superclass for the `QuickSort` classes in order to be implemented in various ways. DO NOT modify the abstract method
`public abstract void Quicksort(SortArray records, int left, int right);`
You may add new methods that all subclasses shall have in common.

3.2.2 `QuickSortA.java, QuickSortB.java`

These classes extend the abstract class `QuickSort`. In these classes you have to implement the algorithm versions Quicksort A and Quicksort B (see Section 2) in the `QuickSort` method of the corresponding class. Make sure that you use the read and write methods of the `SortArray` class to sort the lists. DO NOT bypass them in order to display lower values for the read and write operations.

3.2.3 `SortingItem.java`

This class represents entries of the list that has to be sorted. The read and write methods in the `SortArray` class return and obtain list entries as objects of the type `SortingItem`. You may add auxiliary methods if needed.

3.3 Test Files

We provide three input files for testing as follows:

- `TestFile1_r_200` - 200 entries, *randomized*;
- `TestFile2_a_200` - 200 entries, sorted in *ascending* order;
- `TestFile3_d_200` - 200 entries, sorted in *descending* order.

You are encouraged to test your solution using additional input files as well. You simply need to provide additional input files in the same directory as the given input files. No source code changes are required. Think about the assumptions made on the input. To make sure that your solution works, do test it with all the test cases provided.

4 Testing

Apart from the given input files, we will test your solution with several additional input files (larger, special cases, etc.), to confirm the correctness of your program. Your program is expected to work with **any** input file that follows the format described in section 2.1. You are also expected to be able to explain, in detail, how each version of the QuickSort algorithm works, step by step, as well as your own implementation.