Data Structures – Exercise 6

Due date: 31.5.2016 until 24:00.

The Assignment

In this assignment:

1. You will write a Java implementation of the following sorting algorithms: quick-sort, merge-sort, radix-sort and insertion-sort.
2. You will write a Java implementation of the following order statistics (selection) algorithm: Randomized Selection.
3. For each algorithm: we define $f(n)$ to be the average running time of the sorting/selection algorithm in seconds on an array of size n. Our goal is to calculate an estimate for $f(n_1)/f(n_2)$ where n_1 = 400000 and n_1 = 800000 = $2n_1$. We will compare the calculated estimate of $f(n_1)/f(n_2)$ to the theoretical estimate learnt in class.

Creating Random Arrays of Doubles

In order to create random array of doubles you are going to create a class called *RandomArrayFactory* that will have a single public method:
double[] getRandomArray(int size) – This method will allocate an array of doubles of length *size* (i.e. with indices 0,…,size-1) and will fill it with random doubles in $[0,1)$. In order to generate random doubles in $[0,1)$ use *Math.random().*
Note *that the number of digits after the point should be rounded to 10 digits (i.e – 0.1234567891) so you will need to use an auxiliary method for the rounding.*

Implementing The Algorithms

Each sorting algorithm will be implemented in its own class. Each of the sorting classes will implement the interface *Sorter* (Provided to you) that contains a single method:
void sort(double[] ar) – This method receives a (possibly) un-sorted array of double *ar* with elements in $[0,1)$. *Sort* sorts the array *ar*.

The selection algorithm will be implemented in its own class and will implement the interface *Selector* (Provided to you) that contains a single method:
double select(double[] ar, int i) - This method receives a (possibly) un-sorted array of double *ar* with elements in $[0,1)$ and a rank. Select returns the element whose rank is *i* in the array ar.

You will write the 5 following classes:

1. *InsertionSort* – Implements insertion-sort.

2. *QuickSort* – Implements quicksort. When choosing a pivot, you may simply choose the last or first element of the sub-array to be the pivot, instead of choosing the median of the first, middle and last element as shown in the lecture.
3. *MergeSort* – Implements merge-sort.
4. *RadixSort* – Implements radix-sort.
5. *RandSelect* – Implements Randomized Selection. When choosing a pivot, you may simply choose the last or first element of the sub-array to be the pivot, instead of choosing the median of the first, middle and last element as shown in the *lecture*. (We will select the Median element in the exercise).
   *Note that there is **no** need to implement median of medians*.

Remarks regarding implementation in Java:

1. Note that arrays in java are 0-based, while in the course book and in some parts of the lectures arrays are 1-based. Thus, when translating the pseudo-code to Java you need to be especially careful about array indexes and loop boundaries.
2. When implementing the function *merge* (as part of merge-sort) the pseudo-code uses the symbol ∞. You may use ∞ = Double.MAX_VALUE.
3. For rounding the number of digits use the static parameter NUMBER_OF_DIGITS (in Algorithm*Comparison.java*)

Estimating The Running Times of each Algorithm

For each of the sorting/selection algorithms we define $f(n)$ to be the average running time in seconds of the sorting/selection algorithm.
We define n_1 = 400000 and n_1 = 800000 = $2n_1$.
The supplied main function (in Algorithm*Comparison.java*) calculates and estimate of $f(n_1)$ and $f(n_2)$ for quick-sort, radix-sort, merge-sort and Randomized Selection using the classes you implemented, and outputs the results to the screen. The estimate is based of 100 random arrays.
Answer the following questions (to be submitted):

1. In class we learned that the average time complexity of quick-sort is $\Theta(n\log n)$. We can strengthen this result and show that average time complexity of quick-sort is approximately $cn\log n$ for some constant $c>0$. Assume there exists a constant d that represents the time it takes in seconds for your computer to complete a single instruction. We get that for large values of n we have: $f(n)\approx dcn\log(n)$.
   a. Using the theoretical estimation $f(n)\approx dcn\log(n)$, What is $f(n_1)/f(n_2)$?
   b. What is the value of $f(n_1)/f(n_2)$ based on the values of $f(n_1)$ and $f(n_2)$ calculated by our java program?

c. What is the relative error between the theoretical estimation of $f(n_1)/f(n_2)$ and the calculated estimation of $f(n_1)/f(n_2)$?

Remark: The relative error between two values $a,b>0$ is $\frac{|a-b|}{\min\{a,b\}}$.

2. In class we learned that the average time complexity of merge-sort is $\Theta(n\log n)$. We can strengthen this result and show that average time complexity of merge-sort is approximately $cn\log n$ for some constant $c>0$. Assume there exists a constant d that represents the time it takes in seconds for your computer to complete a single instruction. We get that for large values of n we have: $f(n)\approx dcn\log(n)$.
   a. Using the theoretical estimation $f(n)\approx dcn\log(n)$, What is $f(n_1)/f(n_2)$?
   b. What is the value of $f(n_1)/f(n_2)$ based on the values of $f(n_1)$ and $f(n_2)$ calculated by our java program?
   c. What is the relative error between the theoretical estimation of $f(n_1)/f(n_2)$ and the calculated estimation of $f(n_1)/f(n_2)$?
3. In class we learned that the average (and worst) time complexity of radix-sort is $\theta(dn)$. We can strengthen this result and show that average time complexity of radix-sort is approximately cdn for some constant $c>0$. Assume there exists a constant d that represents the time it takes in seconds for your computer to complete a single instruction. We get that for large values of n we have: $f(n)\approx dcn$.
   a. Using the theoretical estimation $f(n)\approx dcn$, What is $f(n_1)/f(n_2)$?
   b. What is the value of $f(n_1)/f(n_2)$ based on the values of $f(n_1)$ and $f(n_2)$ calculated by our java program?
   c. What is the relative error between the theoretical estimation of $f(n_1)/f(n_2)$ and the calculated estimation of $f(n_1)/f(n_2)$?
4. In class we learned that the average time complexity of *Randomized Selection is* $O(n)$. We can strengthen this result and show that average time complexity of *Randomized Selection* is approximately *dcn* for some constant $c>0$. Assume there exists a constant d that represents the time it takes in seconds for your computer to complete a single instruction. We get that for large values of n we have: $f(n)\approx dcn$.

   You will need to select the median element (k = n/2).
   a. Using the theoretical estimation $f(n)\approx dcn$, What is $f(n_1)/f(n_2)$?
   b. What is the value of $f(n_1)/f(n_2)$ based on the values of $f(n_1)$ and $f(n_2)$ calculated by our java program?

c. What is the relative error between the theoretical estimation of $f(n_1)/f(n_2)$ and the calculated estimation of $f(n_1)/f(n_2)$?

Remark: The relative error between two values $a,b>0$ is $\frac{|a-b|}{\min\{a,b\}}$.

5. In order to estimate the average running time of quick-sort on arrays of size n we averaged the running time of quick-sort on many random arrays of size n. Could we get a good estimation of the average time complexity of quick-sort just by taking the running time of quick-sort on a single fixed array of size n? Explain.
6. In order to estimate the average running time of merge-sort on arrays of size n we averaged the running time of quick-sort on many random arrays of size n. Could we get a good estimation of the average time complexity of merge-sort just by taking the running time of merge-sort on a single fixed array of size n? Explain.
   Hint: Read the pseudo-code of merge-sort given in the lecture (the functions *MergeSort* and *Merge*). Consider the following question: Let n be a fixed positive integer. When merge-sort is given as input an array of size n, does the run time on the input depend on the values of the keys?

General Guidelines

- You are not allowed to change any of the given *.java files.
- All methods should be readable and well documented.
- All your methods should be efficient.
- You may add classes and methods that were not defined in the given API, as long as they conform to proper Object Oriented Design.
- You are responsible for testing your code.
- You should provide full documentation of all classes and methods.

Submission Instructions

- Programming exercises can be done in pairs.
- Do not add package declarations to java files
- **Submit in a single zip containing all of the following files via moodle:**
  - InsertionSort.java
  - RadixSort.java
  - MergeSort.java
  - QuickSort.java
  - RandSelect.java

    - The output of the main method to the console
    - Your answers to the 6 questions.
- The name of the zip file should include your ID and the number of the exercise, in the following format:
    - (Two students) ID1_ID2_Ex06
    - (A single student) ID_Ex06
    - Example: 987654321_123456789_Ex06