Computer Science 228
Project 2
Quantiles

Due: 11:59 pm, Wednesday, February 25

1 Introduction

You have probably heard terms such as "median household income"[1] or "percentile score" on the ACT, SAT, or GRE. These are examples of *order statistics*. In this assignment, you will write a class consisting of utility methods for computing order statistics. You will also write a class that maintains and reports *quantiles* — a special kind of order statistics — for a collection of data.

This project is worth 150 points, and you can earn up to 15 additional (extra credit) points.

2 Order Statistics and Quantiles

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example.

- the ***minimum*** of a set of n elements is the first order statistic ($i = 1$),
- the ***maximum*** is the nth order statistic, and
- the ***median*** is the "middle" element, that is, the $\lceil n/2 \rceil$th order statistic[2].

Certain order statistics are especially useful. The q***-quantiles*** of an n-element set are the $q - 1$ order statistics that divide the sorted set into q equally-sized sets (to within 1). That is, the q-quantiles are the $\lceil 1 \cdot n/q \rceil$th, $\lceil 2 \cdot n/q \rceil$th, $\lceil 3 \cdot n/q \rceil$th, ..., $\lceil (q-1) \cdot n/q \rceil$th order statistics. Some q-quantiles have special names. There is precisely one 2-quantile: the median. The 3-quantiles are called ***tertiles*** or ***terciles***. The 4-quantiles are called ***quartiles***. The 5-quantiles are called ***quintiles***. The 10-quantiles are called ***deciles***. The 100-quantiles are called ***percentiles***. The 1000-quantiles are called ***permilles***

[1] The US Census Bureau reports these numbers for all states at `https://www.census.gov/hhes/www/income/data/statemedian/index.html`

[2] Here $\lceil x \rceil$ (read "ceiling of x") denotes the smallest integer greater than or equal to x; for instance, $\lceil 11/3 \rceil = 4$.

Quantiles are widely used as economic indicators. An example, mentioned earlier, is the median household income; another is the median value of a home[3]. One measure of the degree of income inequality in a society is the ratio of the total income of the top 10% wage earners to the total income of the bottom 10% — a higher ratio means more inequality.

Another familiar example of the use of quantiles is in reporting scores for standardized tests such as the ACT or the SAT. This data is often reported in terms of the scores for the 25th and 75th percentile of students. For instance, you may have seen a college profile such as this:

SAT Critical Reading:	500 / 610
SAT Math:	520 / 620
SAT Writing:	490 / 600

The lower number is the 25th percentile of the scores of students who enrolled in the college. The upper number is the 75th percentile of scores of students who enrolled in the college. This means that if you have an SAT math score of 640, your score would put you among the top 25% of students in the college. If you have a math score of 500, you would be in the bottom 25% for that measure.

Example. Suppose we have the following numbers

$$15, 17, 9, 35, 23, 2, 11, 18, 5, 6.$$

Then, $n = 10$, and

- the minimum is 2,
- the maximum (the 10th order statistic) is 35,
- the median is 11, which is the 5th ($5 = \lceil 10/2 \rceil$) order statistic,
- the terciles (the 3-quantiles) are 9 and 17, which are, respectively, the 4th ($4 = \lceil 1 \cdot 10/3 \rceil$) and 7th ($7 = \lceil 2 \cdot 10/3 \rceil$) order statistics, and
- the quartiles (the 4-quantiles) are 6, 11, and 18, which are, respectively, the 3rd ($3 = \lceil 1 \cdot 10/4 \rceil$), 5th ($5 = \lceil 2 \cdot 10/4 \rceil$), and 8th ($8 = \lceil 3 \cdot 10/4 \rceil$) order statistics.

2.1 Finding Order Statistics in Linear Expected Time

We can find the ith order statistic of a set of n elements by first sorting the set and then returning element i. This takes $O(n \log n)$ time if we use merge sort. There is a faster way to find order statistics, based on the partitioning idea used for quicksort. This algorithm, called SELECT, returns the ith smallest element in the array $A[\text{first}], \ldots, A[\text{last}]$ in $O(n)$ *expected* time[4], where $n =$

[3] You can find the median price of a home in Iowa at `http://www.zillow.com/ia/home-values/`

[4] The expected running time of an algorithm is its average running time over all possible inputs of size n. Expected running time can be a better measure of the algorithm's efficiency in practice than worst-case time. For instance, the expected running time of quicksort is $O(n \log n)$, even though its worst case time is $O(n^2)$; indeed, quicksort's performance in practice tends to be closer to its expected case than its worst case.

```
SELECT(A, first, last, i)
1   if (first == last)
2       return A[first]
3   p = PARTITION(A, first, last)
4   k = p − first + 1
5   if (i == k)
6       return A[p]
7   elseif (i < k)
8       return SELECT(A, first, p − 1, i)
9   else
10      return SELECT(A, p + 1, last, i − k)
```

Figure 1: Partition-based selection.

$\text{last} - \text{first} + 1$. It assumes that $1 \le i \le \text{last} - \text{first} + 1$. One of your tasks is to implement SELECT and use it as part of the `OrderStatistics` class, described in Section 3.

Algorithm SELECT, shown in Figure 1, works as follows. Line 1 handles the base case, where $A[\text{first}], \ldots, A[\text{last}]$ consists of just one element. In this case, i must equal 1, and we simply return $A[\text{first}]$ in line 2. Otherwise, the algorithm calls PARTITION in line 3, to partition the array $A[\text{first}], \ldots, A[\text{last}]$ into two (possibly empty) sub-arrays $A[\text{first}], \ldots, A[p-1]$ and $A[p+1], \ldots, A[\text{last}]$ such that

- each element in $A[\text{first}], \ldots, A[p-1]$ is less than or equal to $A[p]$ and
- each element in $A[p+1], \ldots, A[\text{last}]$ is greater than or equal to $A[p]$.

As in quicksort, we refer to $A[p]$ as the ***pivot***. Line 4 computes the number k of elements in $A[\text{first}], \ldots, A[p]$. Line 5 then checks if the pivot $A[p]$ is the ith order statistic. If so, it returns $A[p]$ in line 6. Otherwise, the algorithm determines which of the two sub-arrays contains the ith order statistic.

- If $i < k$, then the ith smallest element must be the ith order statistic among $A[\text{first}], \ldots, A[p-1]$, and line 8 recursively selects this element from that sub-array.
- If $i > k$, the element we are looking for is in sub-array $A[p+1], \ldots, A[\text{last}]$. Since we already know that there are k values that are smaller than the ith order statistic of the original array $A[\text{first}], \ldots, A[\text{last}]$ — namely, the elements $A[\text{first}], \ldots, A[p]$ — the element we are looking for must be the $(i-k)$th order statistic in $A[p+1], \ldots, A[\text{last}]$. Line 10 finds this element recursively.

It can be shown that if the pivot is chosen at random from among $A[\text{first}], \ldots, A[\text{last}]$, then the expected running time of SELECT is $O(n)$. The proof is beyond the scope of Computer Science 228, but you can find the argument in several textbooks, including Chapter 9 of Cormen et al.'s

Introduction to Algorithms (MIT Press, 2009). Note that it is also possible to find the ith order statistic in $O(n)$ *worst-case* time, but the algorithm (described in the aforementioned textbook by Cormen et al.) is perhaps too complex to be useful in practice.

2.2 Computing q-Quantiles in $O(n \log q)$ Expected Time

We can compute all $q-1$ q-quantiles of an n-element set in (expected) time $O(n \cdot q)$, by calling the SELECT algorithm of Figure 1 $q-1$ times. We obtain a faster, $O(n \cdot \log q)$, algorithm. by using divide-and-conquer. Here is a sketch of the idea.

1. Use SELECT to find the kth q-quantile, for $k = \lceil (q-1)/2 \rceil$; call this "middle" quantile u_{mid}.
2. Split the set in two parts: the set S_{low} consisting of $\lceil k \cdot n/q \rceil - 1$ elements that are less than or equal to u_{mid} and the set S_{high} consisting of the $n - \lceil k \cdot n/q \rceil$ elements greater than or equal to u_{mid}.
3. Find q-quantiles 1 through $k-1$ and q-quantiles $k+1$ through $q-1$ by using recursion on S_{low} and S_{high}, respectively.

An informal explanation of why this algorithm takes $O(n \log q)$ time is as follows. The total work is proportional to the number of levels of the recursion tree times the work per level (we have used the idea of a recursion tree to analyze binary search, quick sort, and merge sort). We always split at the middle quantile, so the height of the recursion tree of our algorithm is $O(\log q)$. The expected work per level of the recursion tree is $O(n)$, giving a total running time of $O(n \log q)$.

You can earn up to 15 extra credit points by fleshing out the details of the preceding $O(n \log q)$ algorithm, implementing it, and using it in the `Quantiles` class, described in Section 4.

3 The `OrderStatistics` Class

Your first task in this assignment is to write an `OrderStatistics` class that contains a collection of utility methods to compute order statistics for an array of `int`s. ***You may assume that all the numbers in this array are distinct.*** You must implement all of the methods specified next.

`public static int findMinimum(int[] arr)`
: Returns the minimum element (first order statistic) in array `arr`. This method must run in worst-case $O(n)$ time, where $n =$ `arr.length`, using a linear scan of the input array.

`public static int findMaximum(int[] arr)`
: Returns the maximum element (last order statistic) in array `arr`. This method must run in worst-case $O(n)$ time, where $n =$ `arr.length`, using a linear scan of the input array.

`public static int select(int[] arr, int first, int last, int i)`
: An implementation of the SELECT algorithm of Figure 1 (Section 2.1). Returns the `i`th order statistic in the subarray `arr[first]`, ..., `arr[last]`. Must run in $O(n)$ expected time, where $n =$ `last` $-$ `first` $+ 1$. This method must throw an `IllegalArgumentException` if `i` < 1 or `i` $> n$.

`public static int findOrderStatistic(int[] arr, int i)`

Returns the `i`th order statistic of array `arr` in $O(n)$ expected time, where $n =$ `arr.length`. This method must throw an `IllegalArgumentException` if `i` < 1 or `i` $> n$.

`public static int findMedian(int[] arr)`

Returns the median ($\lceil n/2 \rceil$th order statistic) in array `arr` in $O(n)$ expected time, where $n =$ `arr.length`.

4 The `Quantiles` class

Write a `Quantiles` class that uses the methods of the `OrderStatistics` class to compute and maintain the q-quantiles for an n-element `int` array. ***You may assume that this input array contains no repeated elements.***

`Quantiles` ***must be immutable.*** Furthermore, a **`Quantiles`** ***object should not store the input data array.*** Instead, a `Quantiles` object maintains only following information about the data arey.

```
private int n; // the length of the original data array
private int q; // the number of quantiles
private int[] quantiles; // stores the q-1 q-quantiles
private int topTotal;
private int bottomTotal;
```

Instance variables `topTotal` and `bottomTotal` store the total sum of the elements in the top and bottom quartile groups. They are used in the `getTopTotal()` and `getBottomTotal()` methods (see below).

For full credit, `Quantiles` must have all of the methods specified next, except for the optional fast constructor. The optional constructor, if implemented correctly, is worth an additional 15 (extra credit) points.

`public Quantiles(int[] data, int q)`

A constructor that creates a new `Quantiles` object, whose `quantiles` array has length `q` -1, and contains the `q`-quantiles of the `data` array. Throws an `IllegalArgumentException` if `q` < 1 or if `q` $> n$, where $n =$ `data.length`. The expected time complexity of this method must be $O(n \cdot$ `q`$)$, or better.

`public Quantiles(int[] data)`

A constructor that creates a new `Quantiles` object, whose `quantiles` array, has length three, and is initialized to contain the three quartiles of `data`. Throws an `IllegalArgumentException` if $n < 4$, where $n =$ `data.length`. The expected time complexity of this method must be $O(n)$.

`public Quantiles(int[] data, int q, boolean fast)`

An ***optional*** constructor that creates a new `Quantiles` object, whose `quantiles` array has length `q` $- 1$, and is initialized to contain the `q`-quantiles of the `data` array. Throws an

`IllegalArgumentException` if `q` < 1 or f `q` $> n$, where $n =$ `data.length`. If `fast` is `true`, then the expected time complexity of this method must be $O(n \log \mathtt{q})$, where $n =$ `data.length` (for this, you must implement the method such as the one outlined in Section 2.2. If `fast` is `false`, then the expected time complexity of this method is only required to be $O(n \cdot \mathtt{q})$

`public int getQuantile(int k)`

Returns the `k`-th q-quantile of `this` object. Throws an `IllegalArgumentException` if `k` < 1 or if `k` is greater than the number of quantiles of `this` object. This method must take $O(1)$ time in the worst case.

`public int quantileQuery(int x)`

Returns the index of the quantile group that contains `x`:

- If x is less than or equal to the first quantile, then return 1.
- If x is strictly greater than the last quantile (quantile $q - 1$), then return q.
- Otherwise, return the smallest index k such that x is less than or equal to the kth q-quantile.

This method must take $O(\log k)$ time in the worst case.

`public int getTopTotal()`

Returns the sum of all values that are strictly higher than the $(q - 1)$th q-quantile in the original data array. This method must take $O(1)$ time in the worst case.

`public int getBottomTotal()`

Returns the sum of all vales that are smaller than or equal to the first quantile in the original data array. This method must take $O(1)$ time in the worst case.

`public float ineqRatio()`

Returns the ratio of `getTopTotal()` to `getBottomTotal()` for `this` object.

`public int getQ()`

Returns the number of quantiles in `this` object. This method must take $O(1)$ time in the worst case.

`public int size()`

Returns the length of the original `data` array. This method must take $O(1)$ time in the worst case.

`public String toString()`

Overrides the `toString()` method, returning a `String` in the following format:

```
n, q, [quantiles_array], topTotal, bottomTotal\n
```

where [quantiles_array] is represented as [u1, u2, . . . , uq-1], where ui is the ith quantile. For example, suppose data is an array consisting of the numbers in the example of page 2, and we declare

```
Quantiles quants = new Quantiles(data);
```

Then, quants.toString() would return the string

```
10, 4, [6, 11, 18], 58, 13\n
```

5 Suggestions for Implementation

We suggest that you proceed as follows.

1. Implement the SELECT algorithm of Figure 1.
2. Use SELECT to implement the OrderStatistics class.
3. Implement the simple $O(n \cdot q)$ method to compute q-quantiles.
4. Use this q-quantile algorithm to implement all the methods of the Quantiles class, except for the $O(n \log q)$ constructor (the one with the "fast" option).
5. After implementing and testing all other methods of OrdeStatistics and Quantiles, try to implement the $O(n \log q)$ quantiles algorithm and the $O(n \log q)$ Quantiles constructor.

The extra credit $O(n \log q)$ quantiles algorithm is considerably more complex than the basic $O(n \cdot q)$ algorithm. Do not get bogged down in trying to implement the more sophisticated algorithm, at the expense of completing the rest of the project.

6 Submitting your Assignment

Write your classes in the edu.iastate.cs228.hw2 package. Turn in the zip file not your class files. Please follow the guideline posted under Documents & Links on Blackboard Learn. Include the Javadoc tag @author in each class source file. Your zip file should be named

Firstname_Lastname_HW2.zip.

Coding style and documentation will be approximately 15% of your grade. For this assignment, you are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.