Georgia Gwinnett College
School of Science and Technology
ITEC 3150: Advanced Programming
Homework Assignment 8

Problem 1 [50 Points]

In this problem you are to revisit the vote count problem from Homework Assignment 7 (the extra credit problem), with a more strict time complexity requirement.

For the NBA All-Star Game, the fans vote for their favorite players. All the votes are stored in an array list, in which the name of a player appears once every time he receives a vote. A software expert, you are hired by the Commissioner to write an app that helps decide the starting lineups of the game. Write a method **voteCount()** that on the input array list of votes, returns an array list of players in ***descending order*** of the number of votes received. The skeleton for the method **voteCount()** is provided in the file **VoteCount.java**, which also contains the **Player** class whose instance variables **name** and **count** represent a player's name and vote count respectively. The method should return an array list of Player objects in *descending order of count*. If two players have the same count, then they are ordered *alphabetically by name*.

The following is a sample run:

Input: [LeBron, Kawhi, James, Giannis, Anthony, Luka, Kawhi, LeBron, LeBron, Anthony, Luka]
Output: [(LeBron, 3), (Anthony, 2), (Kawhi, 2), (Luka, 2), (Giannis, 1), (James, 1)]
Explanation: LeBron has 3 votes which is the most of all players; he therefore appears the first in the list. LeBron is followed in the alphabetical order by Anthony, Kawhi and Luka who each have 2 votes. They are followed in the alphabetical order by Giannis and James who each have 1 vote.

Your method must have time complexity $\boldsymbol{O(n + k\log k)}$, where n is the size of the input array list (i.e. the total number of votes) and k is the number of players receiving votes. Note that the number of votes is typically a lot larger than the number of players receiving votes (for example when 100,000,000 fans vote for 100 players). Therefore, the $O(n + k\log k)$ time complexity required here is better than $O(n\log n)$ that was required in HW1. While there may be several ways to accomplish this, ***you must do so following the steps below to receive credit***:

1. Iterate through the input list and use a ***hash map*** to count the votes for each player.

2. Define a ***comparator*** for the Player class that orders players as described above. Namely, a player with a higher count precedes a player with a lower count, and two players with the same count are ordered alphabetically by name.

3. Construct a ***priority queue*** of players using the comparator defined in Step 2.

4. Iterate through the entry set of the hash map. For each map entry (i.e. key-value pair) construct a Player object and add it to the priority queue.

5. Remove players from the priority queue until it is empty.

Problem 2 [50 Points]

One way to get the k smallest elements of an unsorted list, where k is an int value, is first to use the list to build a *min priority queue* and then remove the first k elements one by one. See PriorityQueueDemo.java and ClosestPointToSource.java for examples on String and WorkOrder and Point. A generic method **kSmallest()** that uses this approach is included in **KSmallest.java**.

While this approach is simple and intuitive, its time complexity is $O(n \log n)$ where n is the size of the entire list, even when k is small. This is because for each element of the list added to the priority queue, the time complexity is *O*(log n), the height of the heap. For n elements, the total time is *O*(n log n).

In this problem you are to implement a faster solution. method, **kSmallestFaster()**, which has time complexity $\boldsymbol{O(n \log k)}$**.** Therefore, this version is more efficient when n is large and k is small (e.g. when n is a billion and $k = 10$). The skeleton for the method **kSmallestFaster()** is provided in the file **KSmallest.java**.

While there may be several ways to accomplish the desired time complexity, ***you must do so following the steps below to receive credit***:

1. Define a new comparator, $revComp$, that reverses the ordering imposed by the comparator $compare$ which is given as a parameter. (Done for you in the starter file.)

2. Construct a *max* priority queue using the comparator $revComp$.

3. Iterate through the input list A. For each element a of A, if the current size of the priority queue is less than k, then add a to the priority queue. Otherwise, compare a with the current maximum element in the priority queue. Do nothing if a is larger or equal. If a is smaller, remove the maximum element and add a to the priority queue.

4. At this point the priority queue contains exactly the k smallest elements of A. Until the priority queue is empty, remove each element and add it to a list *in the correct order*. You can decide what type of Java list to use.

Note that in Steps 3 and 4 the size of the priority queue never exceeds k. Therefore, each add or remove incurs a cost of only $O(\log k)$.

Deliverables

The two files to submit for the assignment are **VoteCount.java** and **KSmallest.java**. After completing your methods, please place both files in a single folder named **HW8**, compress the folder and upload **HW8.zip** to D2L. ==> Now please submit in Mimir.