COMS 4231: Analysis of Algorithms I, Fall 2020

Problem Set 2, due Friday September 25, 11:59pm on Gradescope

Please follow the homework submission guidelines posted on Courseworks.

- **In all problems that ask you to give an algorithm, include a justification of the correctness of the algorithm and of its running time.**
- **All times bounds are worst-case, unless specified otherwise.**

Problem 1. [10 points] Consider the following game. There are n cards with the numbers $1,\ldots,n$ written on them, where each of the numbers $1,\ldots,n$ is written on exactly one of the cards. The cards are face down on the table, so we do not see their numbers. We turn over the cards one by one. Before turning each card, you guess the number written on the card; then you see the card, and you gain 1 point if you guessed correctly and 0 if incorrectly. A random strategy is to guess in each step uniformly at random a number that has not appeared so far in the previous cards.

1. What is the probability that the guess in the i-th step is correct?
2. Show that the expected total number of points that you gain using the random strategy is $\Theta(\log n)$.

Hint. Indicator random variables may be useful here.

Problem 2. [25 points] An array $A[1\ldots n]$ is said to have a *majority element* if the same element appears in (strictly) more than $n/2$ positions. We want to design an efficient algorithm to determine whether a given array has a majority element, and if so, to find that element. The elements of the array are not necessarily from an ordered domain like the integers, and so we cannot perform comparisons of the form "is $A[i] > A[j]$?". The only operations on the elements that we can perform are equality tests of the form "is $A[i] = A[j]$?", and this test takes constant time.

1. [10 points] Show that if the array A contains a majority element, then that element is also a majority element of either the first half of the array (i.e. $A[1...\lfloor n/2 \rfloor]$) or the second half of the array (or both).

Use this property to design a divide and conquer algorithm that solves the majority element problem in time $O(n\log n)$.

2. [15 points] A different divide and conquer algorithm is based on the following idea. Pair up the elements of A arbitrarily to get $\lfloor n/2 \rfloor$ pairs (for example, pair $A[1]$ with $A[2]$, $A[3]$ with $A[4]$ etc). For each pair, if the two elements of the pair are equal then keep only one copy, and if the two elements are not equal then discard both of them.

a. Show that if n is even and A has a majority element x, then x is also a majority element among the elements that are kept.

b. If n is odd, then give a rule regarding whether or not to keep the last element that was not paired, so that the property of part (a) holds in this case too, i.e. if A has a majority element x then x is also a majority element among the elements that are kept. Show that the property holds.
c. Use the above property of parts (a) and (b) to design a O(n)-time algorithm for the majority element problem.

Problem 3. [13 points] We are given an unsorted array A of n numbers, and a positive integer $k < n$. We want to find the k elements of A that are closest in value to the first element of A; these elements can be output in any order, and in case of a tie you can choose arbitrarily. For example, if A is the array [7, 2, 5, 15, 6, 12, 10, 18, 0, -4] and k =3 then the algorithm should return 5, 6, 10 (in some order). If k=4 then the algorithm should return 2, 5, 6, 10 or 5, 6, 12, 10. Give an O(n)-time algorithm for this problem.
(Note that k is not a constant, it is part of the input; your algorithm should run in O(n) time for any value of k, for example for k=log$n$ or k=n/5.)

Problem 4. [17 points] Given an unsorted array with n elements, and a positive integer $k < n$, we wish to find the k-1 elements of rank $\left\lceil \frac{n}{k} \right\rceil$, $\left\lceil \frac{2n}{k} \right\rceil$, …, $\left\lceil \frac{(k-1)n}{k} \right\rceil$. Give an O($nlogk$)-time algorithm for this problem.

Problem 5. [20 points] Do Problem 8.6 ("Lower bound on merging sorted lists") in CLRS, page 208.

Problem 6. [15 points] We are given r lists $L_1, L_2, ..., L_r$ of integers in the range 1 to n. The sum of the sizes of the lists is n. Give an O(n)-time algorithm to sort all the lists; i.e., the algorithm should compute lists $L_1', L_2', ..., L_r'$, where L_i' contains in sorted order the same elements as L_i for each i=1,…,r.
Note: You cannot assume that r is a constant. For example, your algorithm should run in O(n) time if we have $\sqrt{n}$ lists of $\sqrt{n}$ elements each, or lists of different lengths. Also, there are no other restrictions on the numbers in the lists; in particular, the same integer may appear in different lists, or may appear multiple times on the same list.
Hint: Radix sort or bucket sorting may be useful here. One solution starts by forming a collection of pairs (i, a), for i = 1, ..., r, and $a \in L_i$.