Design and Analysis of Algorithms

Homework # 2

Total Marks = 55

Q1) Suppose you have an unsorted array A of colors *red*, *white* and *blue.* You want to sort this array so that all *reds* are before all *whites*, followed by all *blues*. Only operations available to you for this purpose are: equality comparison A[i] = = c where c is one of the three colors, and swap(i, j) which swaps the colors at indices i and j in A. How to sort this array in O(n) worst case time and O(1) additional space. Assume that some satellite data is also there with these colors so counting the number of reds, whites and blues will not solve the problem.[10 Marks]

Q2) Given two sorted arrays X[ ] and Y[ ] of sizes M and N where $M \geq N$, devise an algorithm to merge them into a new sorted array C[ ] using O(N lg M) comparison operations. Suppose arrays M and N are indexed from 1 to M and from 1 to N respectively. [10 Marks]

Hint: use binary search.

Q3) Consider an array of distinct numbers sorted in increasing order. The array has been rotated (clockwise) k number of times. Given such an array, find the value of k. The solution should be efficient and use divide and conquer approach. [10 Marks]

Examples:
Input : arr[] = {15, 18, 2, 3, 6, 12}
Output: 2
Explanation : Initial array must be {2, 3, 6, 12, 15, 18}. We get the given array after rotating the initial array twice.

Input : arr[] = {7, 9, 11, 12, 5}
Output = 4

Q4) [5+5 = 10 Marks] An array A[1 : : : n] is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is A[i] > A[j]?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is A[i] = A[j]?" in constant time.

(a) Show how to solve this problem in O(n log n) time. (*Hint:* Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.) [5 Marks]

(b) Can you give a linear-time algorithm? (Use the following divide-and-conquer approach: [5 Marks]

- Pair up the elements of A arbitrarily, to get n=2 pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

  Show that after this procedure there are at most n=2 elements left, and that they have a majority element if and only if A does.)

Q5) [10+5 = 15 Marks]

(a) In Data Structures, you studied binary heaps. Binary heaps support the insert and extractMin
functions in $O(lgn)$, and getMin in $O(1)$. Moreover, you can build a heap of n elements in just $O(n)$. Refresh your knowledge of heaps from chapter no. 6 of your algorithms text book.
Now implement Merge Sort, Heap Sort, and Quick Sort in C++ and perform the following experiment:
1. Generate an Array A of $\mathbf{10^9}$ random numbers. Make its copies B and C. Sort A using Merge Sort, B using Heap Sort, and C using Quick Sort.
2. During the sorting process, count the total number of comparisons between array elements made by each algorithm. You may do this by using a global less-than-or-equal-to
function to compare numbers, which increments a count variable each time it is called.
3. Repeat this process 5 times to compute the average number of comparisons made by each algorithm.
4. Present these average counts in a table. These counts give you an indication of how the different algorithms compare asymptotically (in big-O terms) for a large value of n.

(b) Now compare the same algorithms in terms of practical time, i.e. the actual running time. Simply, repeat the previous example but use the chrono library to compute the actual times taken by each algorithm, and report the average value of the time for each algorithm.