Data Structures and Algorithms

Problem Set #2

Topics : Divide and Conquer, Sorting

Problems

1. **Fixed Point** : Givena *sorted* array of *distinct* integers $A[1...N]$, you want to find out whether there is an index i for which $A[i] = i$. A simple linear time approach is easy to come up with. Can you do it on $O(\log n)$

2. **Remove Duplicates** : You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$

3. **Two Sum** : You are given an an array of n elements, and a number k. You need to determine whether there exists a pair with sum k. An $O(n^2)$ approach is pretty obvious, just consider every pair and check whether their sum is equal to k. Show how to do this in $O(n \log n$. **Hint** : Sort the array and use the 2 pointer approach.

4. **Minimum Difference** : You are given an an array of n elements. Find the minimum difference betweeen 2 elements in the array in $O(n \log n)$.

5. **Minimum Number of Platforms** : Given the arrival and departure time of n trains, find the minimum number of platforms needed to accommodate all of them. You can assume that the arrival departure time are positive integers.

6. **Unimodal Array Maximum** : You are given an array $A[1, ...N]$ of n *distinct* integers that is unimodal, that is, for some index $p \; \epsilon \; (1, 2, ...N)$, the values in the array entries increase upto the index p and then decrease from the index $p + 1$ till the index n. The problem is to find the peak entry p by reading as few array entries as possible. Show how to find the peak index p in time $O(\log n)$

7. **Counting Inversions** : Given an array $A[1, ...N]$ (with possible repetitions), We say that for $1 \leq i < j \leq n$, a pair (i, j) is an inversion if $a_i > a_j$. Our goal is to count the number of inversions in an array. An $O(n^2)$ algorithm is apparent. Show how to do this in $O(n \log n)$.

8. **Maximum Sum Subarray** : Given an array consisting of positive and negative integers, write an algorithm to find the maximum sum sub array. A subarray is defined as a contiguous sequence of elements. A subarray can of be length 1 as well as of length N, but it cannot be empty. Note that you only need to return the sum and not the subarray. Start out with an $O(n^3)$ solution. Then improve this to $O(n^2)$. Further, improve it $O(n \log n)$. Later in the course, we'll see that this can be even further optimized to $O(n)$

9. **Buying and Selling Stocks** : You are given an array $A[1, ...N]$, where the entries of the array a_i represents the stock price at the $i - th$ day. A transaction is the process of buying and selling a particular stock. Suppose you are allowed to do *atmost* one transaction. What is the maximum profit that you can earn? Give an $O(n^2)$ algorithm along with another efficient algorithm with complexity $O(n \log n)$

   Hint : Consider the difference array **Diff**, where **Diff**$[i] = A[i+1] - A[i]$. Can you now use the *Maximum Sum Subarray* algorithm taught in the class to solve this quesetion? What does a subarray of the *Diff* array represent in terms of the original array?

   - $A = [7, 1, 5, 3, 6, 4]$ The maximum profit that you can earn is 5 (By buying on day 2 at price 1 and selling on day day 5 at price 6).
   - $A = [5, 4, 3, 2, 1]$ The maximum profit that you can earn is 0. (By not doing any transaction)

   Note : There are actually 2 different solutions with complexity $O(n \log n)$. The first one uses the algorithm developed in *Maximum Sum Subarray* as it is (as described by the above hints). The second one is a pure divide and conquer based approach. Start out by dividing the array into 2 roughly equal parts. Now, the buy and sell day may lie entirely on the left half, or entirely on the right half, or they can cross. When they cross, it means, you buy at left half and sell at the right half. What would optimal strategy to do this? Can you now create a recursive relation?

10. **Minimum Swaps to Sort** : You are given an array $A[1, ...N]$. It may also contain duplicate elements. You need to sort the array using minimum number of swaps performed. However, you cannot swap any 2 elements directly. You can only swap **adjacent** elements. Describe an *efficient* algorithm to determine the minimum number of swaps required to sort the array

    - $A = [3, 2, 1]$ The minimum swaps is 3. We can achieve this by swapping 2 and 3, resulting in $[2, 3, 1]$. We then swap 1 and 3 resulting in $[2, 1, 3]$. Finally we swap 1 and 2, giving us the sorted array.
    - $A = [5, 2, 3, 4, 1]$ The minimum swaps is 7 and not 1.
    - $A = [1, 20, 6, 4, 5]$ The minimum swaps is 5