Design and Analysis of Algorithms

Practice Questions

Q1) Given a set N of integers and an integer y, determine if there exit two elements in N whose absolute difference is equal to y and also print those numbers. The algorithm should take O(n lg n) time. Justify why your algorithm runs in O(n lg n) time. [10 Marks]

e.g. Let N = 3 , 7, 2, 1, 4, 10

y = 1

there are three pairs of elements in N whose absolute difference is 1

Pair 1 = |3 - 2| = |-1| = 1

Pair 2 = |3 - 4|= |-1| = 1

Pair 3 = |2 -1| = 1

Ans:

1. sort elements (NlogN)
2. for each element x use binary search (logN per element) to find |x - y| in sorted list.
3. Total time (NlogN + NlogN) = (2NlogN) = Theta(NlogN)

Another algorithm:

The following algorithm solves the problem:
1. Sort the elements in S.
2. Form the set S_ = {z : z = x − y for some y ∈ S}.
3. Sort the elements in S_.
4. If any value in S appears more than once, remove all but one instance. Do the same for S_.
5. Merge the two sorted sets S and S_.
6. There exist two elements in S whose sum is exactly x if and only if the same value appears in consecutive positions in the merged output. To justify the claim in step 4, First observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in S whose sum is exactly x if and only if the same value appears twice in the merged output.

Suppose that some value w appears twice. Then w appeared once in S and once in S_. Because w appeared in S_, there exists some y ∈ S such that w = x − y, or x = w + y. Since w ∈ S, the elements w and y are in S and sum to x. Conversely, suppose that there are values w, y ∈ S such that w + y

= x. Then, since x − y = w, the value w appears in S_. Thus, w is in both S and S_, and so it will appear twice in the merged output.

Steps 1 and 3 require O(n lg n) steps. Steps 2, 4, 5, and 6 require O(n) steps. Thus the overall running time is O(n lg n).

Q2) Given two sorted arrays X[ ] and Y[ ] of sizes M and N where M ≥ 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.

Hint: use binary search.

Solution

```
Merge (X, Y, M, N )

{

    Int j=0;

    int ind = 0;

    for (int i=0; i< N,  i++)

   {

        Int temp = binarySearch(X, Y[i])

        // temp contains index of last element less than Y[i] in X

       While (j < temp)

            C[ind++] = X [j++]

       C[ind++] = Y[i]

    }

}
```

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.

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

Solution
If we take closer look at examples, we can notice that the number of rotations is equal to index of minimum element. A simple linear solution is to find minimum element and returns its index.

Method 2 (Efficient Using Binary Search)
Here are also we find index of minimum element, but using Binary Search. The idea is based on below facts :

- The minimum element is the only element whose previous is greater than it. If there is no previous element element, then there is no rotation (first element is minimum). We check this condition for middle element by comparing it with (mid-1)'th and (mid+1)'th elements.
- If minimum element is not at middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.
    1. If middle element is smaller than last element, then the minimum element lies in left half
    2. Else minimum element lies in right half.

Q4) Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.
Examples :
Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
Output: 500

Input: arr[] = {1, 3, 50, 10, 9, 7, 6}
Output: 50

Corner case (No decreasing part)
Input: arr[] = {10, 20, 30, 40, 50}
Output: 50

Corner case (No increasing part)

Input: arr[] = {120, 100, 80, 20, 0}
Output: 120

Solution

Method 1 (Linear Search)
We can traverse the array and keep track of maximum and element. And finally return the maximum element.

Method 2 (Binary Search)
We can modify the standard Binary Search algorithm for the given type of arrays.
i) If the mid element is greater than both of its adjacent elements, then mid is the maximum.
ii) If mid element is greater than its next element and smaller than the previous element then maximum lies on left side of mid. Example array: {3, 50, 10, 9, 7, 6}
iii) If mid element is smaller than its next element and greater than the previous element then maximum lies on right side of mid. Example array: {2, 4, 6, 8, 10, 3, 1}

Q5) 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.

Solution:

Use partition function of Quicksort twice.

Apply partition and use red as pivot. Separate red from whites and blues.

Again apply partition on to separate blues and whites.