SEARCHING & SORTING - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/minimum-absolute-difference/
- **Difficulty level**: Easy

Explanation:

Sort the given array. Traverse the array once and calculate the minimum difference between consecutive elements. Then, traverse the array again and pick out those pairs of consecutive elements whose difference is the same as the calculated minimum difference.

Solution:

```
class Solution {
public:
    vector<vector<int>> minimumAbsDifference(vector<int>& arr) {

        vector<vector<int>> output;

        sort(arr.begin(), arr.end());

        int minDiff = INT_MAX;
        int n = arr.size();

        for(int i=0; i<n-1; i++)
        {
            int diff = abs(arr[i]-arr[i+1]);
```

```
            minDiff = min(minDiff, diff);
        }

        for(int i=0; i<n-1; i++)
        {
            int diff = abs(arr[i]-arr[i+1]);

            if(minDiff == diff)
            {
                vector<int> pair {arr[i], arr[i+1]};
                output.push_back(pair);
            }
        }

        return output;
    }
};
```

Complexity:

- Time: O(n log n)
- Space: O(n)
  Space complexity is O(1) if the output array does not count as extra space for space complexity analysis.

Question 2:

- **Problem link**: https://leetcode.com/problems/largest-perimeter-triangle/
- **Difficulty level**: Easy

Explanation:

For `a >= b >= c`, `a,b,c` can form a triangle if `a < b + c`.

1. We sort the `A`
2. Try to get a triangle with 3 biggest numbers.
3. If `A[n-1] < A[n-2] + A[n-3]`, we get a triangle.
   If `A[n-1] >= A[n-2] + A[n-3] >= A[i] + A[j]`, we cannot get any triangle with `A[n-1]`
4. repeat step2 and step3 with the left numbers.

Solution:

```
int largestPerimeter(vector<int>& A) {
        sort(A.begin(), A.end());
        for (int i = A.size() - 1 ; i > 1; --i)
            if (A[i] < A[i - 1] + A[i - 2])
                return A[i] + A[i - 1] + A[i - 2];
        return 0;
    }
```

Complexity:

- Time: O(n log n)
- Space: O(1)

Question 3:

- **Problem link**: https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/
- **Difficulty level**: Easy

Explanation:

There are two good approaches to solve this problem.

Hint: As the given array is sorted, we might use two pointer technique / binary search.

Two pointer technique:

We can exploit the sorted property of an array.

Let's say for indices i & j such that i<j,

if sum of elements A[i]+A[j] == target, then we are done

else check whether A[i]+A[j] > target, then the last element has to be removed to reduce the value of sum (j--).

Similarly if A[i]+A[j] < target, we have to add little to increase the sum (i++).

Binary Search technique:

Consider each element and find if (target-nums[i]) exist in array range [i+1,N-1]? If yes, we are done, else increment one step, then again repeat the process.

Solution:

Two pointer technique:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int i=0,j=nums.size()-1;
        while(i<j){
            int sum = nums[i] + nums[j];
            if(sum==target) return {i+1,j+1};
            else if(sum>target) j--;
            else i++;
        }
        return {}; // not found
    }
};
```

Binary search technique:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
```

```
        int N = nums.size();
        for(int i=0;i<N-1;i++){
            int lo = i+1;
            int hi = N;
            while(lo<hi){
                int mid = lo + (hi-lo)/2;
                if(nums[mid]==target-nums[i]) return {i+1,mid+1};
                else if(nums[mid]>=target-nums[i]) hi=mid;
                else lo = mid+1;
            }
            if(lo!=nums.size() && nums[lo]==target-nums[i]) return
{i+1,lo+1};
        }
        return {}; // not found
    }
};
```

Complexity:

- Time:
  O(n) for two pointer
  O(n log n) for binary search
- Space: O(1)

Question 4:

- **Problem link**: https://leetcode.com/problems/find-the-duplicate-number/
- **Difficulty level**: Medium/Hard

Explanation:

Use two pointers: the hare and the tortoise. The hare one goes forward two steps each time, while the tortoise one goes only one step each time. They must meet the same item when hare==tortoise. In fact, they meet in a circle, the duplicate number must be the entry point of the circle when visiting the array from nums[0]. Next we just need to

find the entry point. We use a point(we can use the hare before) to visit from begining with one step each time, and do the same job to tortoise. When hare==tortoise, they meet at the entry point of the circle. For a more detailed explanation and the mathematical proof and correctness of the above method, refer to the solution on leetcode.

Solution:

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int hare=nums[nums[0]];
        int tortoise=nums[0];
        while(hare!=tortoise)
        {
            tortoise=nums[tortoise];
            hare=nums[nums[hare]];
        }
        tortoise=0;
        while(tortoise!=hare)
        {
            hare=nums[hare];
            tortoise=nums[tortoise];
        }
        return tortoise;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 5:

- **Problem link**: https://leetcode.com/problems/maximum-element-after-decreasing-and-rearranging/
- **Difficulty level**: Medium

Explanation:

Sort the given array. Make the first element of the sorted array as 1. Maintain a variable to store the largest element. Traverse through the array starting from the second element and check if the absolute difference between the current element and its previous element is less than or equal to 1 or not. If not, then, make the current element as 1 plus the previous element. Update the max element.

Solution:

```
class Solution {
public:
    int maximumElementAfterDecrementingAndRearranging(vector<int>&
arr) {
        sort(arr.begin(), arr.end());

        arr[0] = 1;
        int maxNum = 1;

        for(int i=1; i<arr.size(); i++)
        {
            if(abs(arr[i]-arr[i-1]) > 1)
            {
                arr[i] = arr[i-1]+1;
            }

            maxNum = max(maxNum, arr[i]);
        }
```

```
        return maxNum;
    }
};
```

Complexity:

- Time: O(n log n)
- Space: O(1)

Question 6:

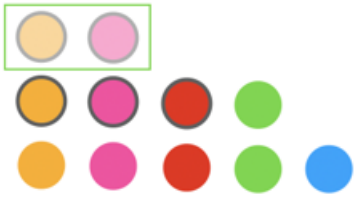
- **Problem link**: https://leetcode.com/problems/sell-diminishing-valued-colored-balls/
- **Difficulty level**: Medium

Explanation:

We compare the current element `A[w-1]` with `A[w]` (if they are the same we keep scanning until we find a different value), let the height difference be h.

We take h balls from `A[0]` to `A[w-1]`, `h * w` balls in total. In this way, all the elements with index in range `[0, w-1]` have the same height as `A[w]`.

When `w = 1, h = 2`, we take 2 balls with value (5 + 4) * 1 = 9.



When `w = 2, h = 1`, we take 2 balls with value 3 * 2 = 6

When w = 3, A[i] == A[i+1], skip.

When w = 4, h = 1, if we take all 4 balls, we will have 8 balls in total which are more than what we need. Instead, we should take T / w = 3 / 4 = 0 rows, and T % w = 3 % 4 = 3 remainder balls. So we take 0 * 4 + 3 = 3 balls with value 3 * 2 = 6.

Now T = 0, we get 7 balls with total value 21.

Solution:

```
class Solution {
public:
    int maxProfit(vector<int>& A, int T) {
        sort(rbegin(A), rend(A)); // sort in descending order
        long mod = 1e9+7, N = A.size(), cur = A[0], ans = 0, i = 0;
        while (T) {
            while (i < N && A[i] == cur) ++i; // handle those same
numbers together
            long next = i == N ? 0 : A[i];
     long h = cur - next, r = 0, cnt = min((long)T, i * h);
            if (T < i * h) { // the i * h balls are more than what we
need.
                h = T / i; // we just reduce the height by `T / i`
instead
                r = T % i;
            }
            long val = cur - h; // each of the remainder `r` balls
has value `cur - h`.
            ans = (ans + (cur + val + 1) * h / 2 * i + val * r) %
mod;
            T -= cnt;
            cur = next;
        }
        return ans;
    }
};
```

Complexity:

- Time: O(n log n)
- Space: O(1)

Question 7:

- **Problem link**: https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/
- **Difficulty level**: Medium

Explanation:

A modified binary search can be used to find the first and last occurrence of the target element separately. So essentially, two binary searches will be used. For the first occurrence, either the target element should be the first element in the array, or different from the previous element. For the last occurrence, the target element should be either the last element of the array or different from the next element.

Solution:

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {

        int l=0,r=nums.size()-1;
        int start=-1,end=-1;
        while(l<=r)
        {
            int mid=l+(r-l)/2;
            if(nums[mid]==target and (mid==0 or (mid-1>=0 and
nums[mid]>nums[mid-1])))
            {
```

```
                start=mid;
                break;
            }
            else if(nums[mid]>=target)
            {
                r=mid-1;
            }
            else l=mid+1;
        }
        l=0,r=nums.size()-1;
        while(l<=r)
        {
            int mid=l+(r-l)/2;
            if(nums[mid]==target and (mid==nums.size()-1 or
(mid+1<nums.size() and nums[mid]<nums[mid+1])))
            {
                end=mid;
                break;
            }
            else if(nums[mid]<=target)
            {
                l=mid+1;
            }
            else r=mid-1;
        }
        return {start,end};
    }
};
```

Complexity:

- Time: O(logn)
- Space: O(1)

Question 8:

- **Problem link**: https://leetcode.com/problems/first-missing-positive/
- **Difficulty level**: Hard

Explanation:

The max value the answer can take is n+1 where n is the size of the array. Since we know that the ans is in the range [1,n+1], we first clean the array such that all the values <=0 and >n are changed to 1 (After checking that 1 is present. If not, 1 is the ans.). Then while traversing the array, we set the index of the found element as negative(multiplying by -1) if found(0 index for n). Then we traverse the array again and report the positive index as the missing element or ans.

Solution:

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int n=nums.size();
        int contains1=0;

        for(int i=0;i<n;i++)
        {
            if(nums[i]==1)
            {
                contains1=1;
            }
            if(nums[i]<=0 or nums[i]>n)
            {
                nums[i]=1;
            }
        }
```

```
        if(!contains1) return 1;
        if(nums.size()==1) return 2;

        for(int i=0;i<n;i++)
        {
            if(nums[abs(nums[i])%n]>0)
            {
                nums[abs(nums[i])%n]*=-1;
            }
        }
        for(int i=1;i<n;i++)
        {
            if(nums[i]>0)
            {
                return i;
            }
        }
        if(nums[0]>0) return n;
        return n+1;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)