SEARCHING & SORTING - SET 2 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/find-smallest-letter-greater-than-target/
- **Difficulty level**: Easy

Explanation:

Start a procedure similar to binary search with the left pointer at index of 1st element and right pointer at index of last element. Initialize the result with the first character, so that if there exists no other greater character than the target character, the first character will be returned. Now, if the 'mid' character is greater than the target character, store it as the result and shift your search to the range letters[left ... mid-1] for a tighter greater element. Else search in the range letters[mid+1, right] for a greater character.

Solution:

```
class Solution {
public:
    char nextGreatestLetter(vector<char>& letters, char target) {

        int n= letters.size();

        int left = 0;
        int right = n-1;
        int res = letters[0];
```

```
        while(left <= right)
        {
            int mid = left + (right-left)/2;

            if(letters[mid] > target)
            {
                res = letters[mid];
                right = mid-1;
            }
            else
            {
                left = mid+1;
            }
        }

        return res;
    }
};
```

Complexity:

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

Question 2:

- **Problem link**: https://leetcode.com/problems/first-unique-character-in-a-string/
- **Difficulty level**: Easy

Explanation:

The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway. The idea is to go through the string and save in a hash map(or an array) the number of times each character appears in the string. That would take O(*N*) time, where `N` is a number of characters in the string. And then we go through the string the second time, this time we use the hash map as a reference to check if a character is unique or not. If the character is unique, one could just return its index. The complexity of the second iteration is O(*N*) as well.

Solution:

```
class Solution {
public:
    int firstUniqChar(string s) {
        vector<int>v(26);
        int l=s.length();
        for(int i=0;i<l;i++)
        {
            if(v[s[i]-'a']==0)
            {
                v[s[i]-'a']=1;
            }
            else
            {
                v[s[i]-'a']=-1;
            }
        }
        for(int i=0;i<l;i++)
        {
            if(v[s[i]-'a']==1)
            {
                return i;
            }
        }
```

```
        return -1;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1) as O(26)~O(1) for any length of given string

Question 3:

- **Problem link**: https://leetcode.com/problems/sort-array-by-increasing-frequency/
- **Difficulty level**: Easy

Explanation:

Sort by two keys. In the below code count[x] sorts by frequency, and if two values are equal, it will sort the keys in decreasing order.

Solution:

```
   vector<int> frequencySort(vector<int>& A) {
        unordered_map<int, int> count;
        for (int a: A)
            count[a]++;
        sort(begin(A), end(A), [&](int a, int b) {
            return count[a] == count[b] ? a > b : count[a] <
count[b];
        });
        return A;
    }
```

Complexity:

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

Question 4:

- **Problem link**: https://leetcode.com/problems/increasing-decreasing-string/
- **Difficulty level**: Easy

Explanation:

Idea is simple . count the frequency of each character .Then traverse from top(a) to bottom(z) if freq of char is non zero append it into ans. After that, traverse back from bottom(z) to top(a) and do the same thing. Count the number of appending characters each time. Do this until the count will become equal to the length of the string or freq of each character become zero. If count is equal to length of string then return the final string.

Solution:

```
class Solution {
public:
    string sortString(string s) {
        int n =s.length();

        int freq[26]={0};

        for(char ch:s)
            freq[ch-'a']++;

        string str;
        int count=0;
        while(count!=n){
            for(int i=0; i<26; i++)
```

```
                if(freq[i]>0){
                    str+='a'+i;
                    freq[i]--;
                    count++;
                }
            for(int i=25; i>=0; i--)
                if(freq[i]>0){
                    str+='a'+i;
                    freq[i]--;
                    count++;
                }
        }

        return str;
    }
};
```

Complexity:

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

Question 5:

- **Problem link**: https://leetcode.com/problems/magnetic-force-between-two-balls/
- **Difficulty level**: Medium

Explanation:

Sort the array. Now the minimum force between two balls should be in the range [0, position[n-1]) where 'n' is the size of the 'position' array. Start a binary search- like process where 0 is the lowest value and position[n-1] is the highest value. Every time, calculate the 'mid' value and check if it is possible to have this value as the minimum

force between two balls. If yes, then, this value is a possible answer. So, store this value and then, search for a higher value in the range [mid+1, right]. If no, search for a value in the range [left, mid-1].

Solution:

```
class Solution {
public:
    int maxDistance(vector<int>& position, int m) {

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

        int n = position.size();
        int left = 0;
        int right = position[n-1];
        int mid;
        int res = 0;

        while(left <= right)
        {
            mid = left + (right-left)/2;

            if(isPossible(position, m, mid))
            {
                res = mid;
                left = mid+1;
            }
            else
            {
                right = mid-1;
            }
        }

        return res;
    }
```

```
    bool isPossible(vector<int> &position, int m, int diff)
    {
        int previous = position[0];
        int count = 1;

        for(int i=1; i<position.size(); i++)
        {
            if(position[i] - previous >= diff)
            {
                previous = position[i];
                count++;

                if(count == m)
                    return true;
            }
        }

        return false;
    }
};
```

Complexity:

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

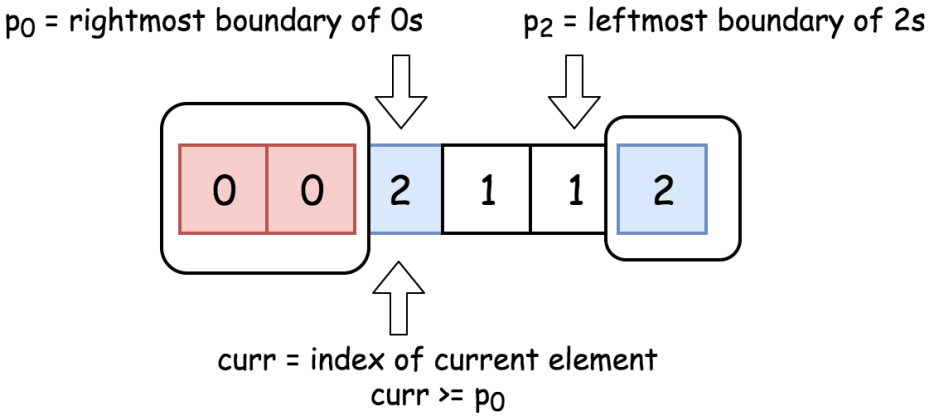
Question 6:

- **Problem link**: https://leetcode.com/problems/sort-colors/
- **Difficulty level**: Medium

Explanation:

The problem is known as Dutch National Flag Problem and first was proposed by Edsger W. Dijkstra. The idea is to attribute a color to each number and then to arrange them following the order of colors on the Dutch flag.

Let's use here three pointers to track the rightmost boundary of zeros, the leftmost boundary of twos and the current element under the consideration.



The idea of solution is to move `curr` pointer along the array, if `nums[curr] = 0` - swap it with `nums[p0]`, if `nums[curr] = 2` - swap it with `nums[p2]`.

Solution:

```
class Solution {
  public:
  /*
  Dutch National Flag problem solution.
  */
  void sortColors(vector<int>& nums) {
    // for all idx < p0 : nums[idx < p0] = 0
```

```
    // curr is an index of element under consideration
    int p0 = 0, curr = 0;
    // for all idx > p2 : nums[idx > p2] = 2
    int p2 = nums.size() - 1;

    while (curr <= p2) {
      if (nums[curr] == 0) {
        swap(nums[curr++], nums[p0++]);
      }
      else if (nums[curr] == 2) {
        swap(nums[curr], nums[p2--]);
      }
      else curr++;
    }
  }
};
```

Complexity:

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

Question 7:

- **Problem link**: https://leetcode.com/problems/maximum-number-of-events-that-can-be-attended/
- **Difficulty level**: Medium

Explanation:

Sort events increased by start time. Priority queue `pq` keeps the current open events.

Iterate from the day 1 to day 100000, each day, we add new events starting on day `d` to the queue `pq`. Also we remove the events that are already closed. Then we greedily attend the event that ends soonest. If we can attend a meeting, we increment the result `res`.

Solution:

```
int maxEvents(vector<vector<int>>& A) {
      priority_queue <int, vector<int>, greater<int>> pq;
      sort(A.begin(), A.end());
      int i = 0, res = 0, d = 0, n = A.size();
      while (pq.size() > 0 || i < n) {
          if (pq.size() == 0)
              d = A[i][0];
          while (i < n && A[i][0] <= d)
              pq.push(A[i++][1]);
          pq.pop();
          ++res, ++d;
          while (pq.size() > 0 && pq.top() < d)
              pq.pop(;

      }
      return res;
  }
```

Complexity:

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

Question 8:

- **Problem link**: https://leetcode.com/problems/maximum-performance-of-a-team/
- **Difficulty level**: Hard

Explanation:

We use a greedy approach to solve it optimizely.

For given efficiency we will pick at most k-1 other engineers whose efficiency is more than least one. If the number is more than k-1 then pick the fastest k-1 among them.

Sort workers based on their efficiency. Starting from the most efficient worker, compute and track the maximum performance. use `sumOfMinHeap` to track all engineers' speed.If available workers exceed `k`, remove the 'slowest' worker from the sum. To do that, we can use a min heap.

For more information read the solution tab of the question.

Solution:

```
class Solution {
public:
    int maxPerformance(int n, vector<int>& speed, vector<int>&
efficiency, int k) {
    // make pair of eff vs speed for each engineer
    vector<pair<int, int>> eff_speed;

    for (int i = 0; i < n; ++i)
        eff_speed.push_back({efficiency[i], speed[i]});

    // sorting based on the efficiency
    sort(eff_speed.begin(), eff_speed.end());

    long sumOfMinHeap = 0;
    long result = 0;

    // implementing min heap
    priority_queue <int, vector<int>, greater<int> > pq_speed;

    // engineer i have least efficiency among i to n-1
```

```
    // if min heap size become greater than k remove engineer with
least speed
    for (int i = n - 1; i >= 0; --i) {
        sumOfMinHeap += eff_speed[i].second;
        pq_speed.push(eff_speed[i].second);

        if (pq_speed.size() > k) {
            sumOfMinHeap -= pq_speed.top();
            pq_speed.pop();
        }

        result = max(result, sumOfMinHeap * eff_speed[i].first);
    }
    return result % 1000000007;
    }
};
```

Complexity:

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