GREEDY - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/can-place-flowers/
- **Difficulty level**: Easy

Explanation:

The solution is very simple. We can find out the extra maximum number of flowers, count, that can be planted for the given flowerbed arrangement. To do so, we can traverse over all the elements of the flowerbed and find out those elements which are 0(implying an empty position). For every such element, we check if its both adjacent positions are also empty. If so, we can plant a flower at the current position without violating the no-adjacent-flowers-rule. For the first and last elements, we need not check the previous and the next adjacent positions respectively. If count obtained is greater than or equal to n then return true otherwise false.

Solution:

```
class Solution {
public:
    bool canPlaceFlowers(vector<int>& flowerbed, int n) {
        int m=flowerbed.size(); //size of flowerbed
        int i=0,count=0;
        while (i < m) {
            if(flowerbed[i]==0 && (i==0 || flowerbed[i-1]==0) && (i==m-1 ||
flowerbed[i+1]==0))
            {
                flowerbed[i] = 1;
                count++;
            }
            i++;
        }
        return count >= n;

    }
};
```

Complexity:

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

Question 2:

- **Problem link**: https://leetcode.com/problems/water-bottles/
- **Difficulty level**: Easy

Explanation:

As long as there are no less than numExchange bottles left, we can drink them and use the empty bottles to exchange full bottle(s).

Solution:

```
class Solution {
public:
    int numWaterBottles(int numBottles, int numExchange) {
        int ans = numBottles;
        while (numBottles >= numExchange) {
            int remainder = numBottles % numExchange;
            numBottles /= numExchange;
            ans += numBottles;
            numBottles += remainder;
        }
        return ans;
    }
};
```

Complexity:

- Time: O($\log_{numExchange}$(numBottles))
- Extra space: O(1)

Question 3:

- **Problem link**: https://leetcode.com/problems/majority-element/
- **Difficulty level**: Easy

Explanation: This can be solved by **Using Moore's Voting Algorithm):**

Approach: We can have some way of counting instances of the majority element as +1 and instances of any other element as -1, summing them would make it obvious that the majority element is indeed the majority element.

Algorithm:

1. Loop through each element and maintains a count of majority element, and a majority element value, maj_value
2. If the next element is same as maj_value then increment the count if the next element is not same then decrement the count.
3. if the count reaches 0 then changes the maj_value to the current element value and set the count again to 1.
4. If the majority element always exists in the array then simply return maj_value

5. Else traverse through the array again and find the count of the majority element found.
6. If the count is greater than half the size of the array, print the element
7. Else print that there is no majority element

Solution:

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int maj_value = nums[0];
        int count=1;

        for(int i=1;i<nums.size();i++){
            if(nums[i] == maj_value){
                count++;
            }
            else{
                if(count==0){
                    maj_value=nums[i];
                    count=1;
                }
                else{
                    count--;
                }
            }
        }
        // if majority element always exists in the array
        return maj_value;
        // if majority element does not always exists in the array
        // count=0;
        // for(int i=0;i<nums.size();i++){
        //     if(nums[i] == maj_value)
        //         count++;
        // }
        // if(count > nums.size()/2)
        //     return maj_value;
        // else
        //     return -1;
    }
};
```

Complexity:

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

Question 4:

- **Problem link**: https://practice.geeksforgeeks.org/problems/n-meetings-in-one-room
- **Difficulty level**: Easy

Explanation:

The idea is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

Algorithm:

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array and print it.
3. Do the following for the remaining activities in the sorted array.

a. If the start time of this activity is greater than to the finish time of the previously selected activity then select this activity and print it.

Solution:

```
class Solution
{
    public:
    //Function to find the maximum number of meetings that can
    //be performed in a meeting room.
    static bool comp(pair<int,int>&p1,pair<int,int>&p2){
        return p1.second < p2.second;
    }

    int maxMeetings(int start[], int end[], int n)
    {
        // Your code here
        vector<pair<int,int>>v;

        for(int i=0;i<n;i++){
            v.push_back(make_pair(start[i],end[i]));
        }

        sort(v.begin(),v.end(),comp);

        int prev_finish_time = v[0].second;
        int count=1;
        for(int i=1;i<v.size();i++){
            if(v[i].first > prev_finish_time){
                count++;
                prev_finish_time = v[i].second;
            }
        }
        return count;

    }
};
```

Complexity:

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

Question 5:

- **Problem link**: https://practice.geeksforgeeks.org/problems/stock-buy-and-sell
- **Difficulty level**: Medium

Explanation: The idea here is to use local minima as buying point and local maxima as selling point.

Algorithm:

1. Find the local minima and store it as starting index. If not exists, return.
2. Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
3. Update the solution (Increment count of buy sell pairs)
4. Repeat the above steps if end is not reached.

Solution:

```
class Solution{
```

```
public:
    //Function to find the days of buying and selling stock for max profit.
    vector<vector<int> > stockBuySell(vector<int> A, int n){
        // code here
        int i=0;
        vector<vector<int>>ans;
        while (i < n - 1) {

                // Find Local Minima
                // Note that the limit is (n-2) as we are
                // comparing present element to the next element
                while ((i < n - 1) && (A[i + 1] <= A[i]))
                    i++;

                // If we reached the end, break
                // as no further solution possible
                if (i == n - 1)
                    break;

                // Store the index of minima
                int buy = i++;

                // Find Local Maxima
                // Note that the limit is (n-1) as we are
                // comparing to previous element
                while ((i < n) && (A[i] >= A[i - 1]))
                    i++;

                // Store the index of maxima
                vector<int>v;
```

```
            int sell = i - 1;
            v.push_back(buy);
            v.push_back(sell);
            ans.push_back(v);
          }
        return ans;

    }
};
```

Complexity:

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

Question 6:

- **Problem link**: https://leetcode.com/problems/jump-game/
- **Difficulty level**: Medium

Solution:

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int step = 0;
        for (size_t i = 0; i < nums.size() - 1; ++i) {
            step = max(step, nums[i]);
            if (step <= 0) {
                return false;
            }
            step--;
        }
        return true;
    }
};
```

Complexity:

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

Question 7:

- **Problem link**: https://leetcode.com/problems/jump-game-ii/
- **Difficulty level**: Medium

Explanation:

Since each element of our input array (N) represents the maximum jump length and not the definite jump length, that means we can visit any index between the current index (i) and i + N[i]. Stretching that to its logical conclusion, we can safely iterate through N while keeping track of the furthest index reachable (next) at any given moment (next = max(next, i + N[i])). We'll know we've found our solution once next reaches or passes the last index (next >= N.length - 1).

The difficulty then lies in keeping track of how many jumps it takes to reach that point. We can't simply count the number of times we update next, as we may see that happen more than once while still in the current jump's range. In fact, we can't be sure of the best next jump until we reach the end of the current jump's range.

So in addition to next, we'll also need to keep track of the current jump's endpoint (curr) as well as the number of jumps taken so far (ans).

Since we'll want to return ans at the earliest possibility, we should base it on next, as noted earlier. With careful initial definitions for curr and next, we can start our iteration at i = 0 and ans = 0 without the need for edge case return expressions.

Solution:

```
class Solution {
public:
    int jump(vector<int>& N) {
        int len = N.size() - 1, curr = -1, next = 0, ans = 0;
        for (int i = 0; next < len; i++) {
            if (i > curr) ans++, curr = next;
            next = max(next, N[i] + i);
        };
        return ans;
    }
};
```

Complexity:

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

Question 8:

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

Explanation: An important observation here is to notice that the performance of the team will depend on two variables, that is, efficiency and speed. The idea here is to fix one variable, here efficiency, and devise a solution to find maximum performance by choosing remaining (k-1) engineers with higher speeds.
The detailed solution and explanation can be found here.

Solution:

```
class Solution {
public:
    int maxPerformance(int n, vector<int>& speed, vector<int>& efficiency, int k) {

        vector<pair<int, int>> ess(n);
        for (int i = 0; i < n; ++i)
            ess[i] = {efficiency[i], speed[i]};
        sort(rbegin(ess), rend(ess));
        long sumS = 0, res = 0;
        priority_queue <int, vector<int>, greater<int>> pq; //min heap
        for(auto& [e, s]: ess){
            pq.emplace(s);
            sumS += s;
            if (pq.size() > k) {
                sumS -= pq.top();
                pq.pop();
            }
            res = max(res, sumS * e);
        }
        return res % (int)(1e9+7);
    }
};
```

Complexity:

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