GREEDY - SET 2 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/split-a-string-in-balanced-strings/
- **Difficulty level**: Easy

Explanation:
Simply count the number of L and R characters in one pass and whenever they become equal, increment the counter and reset the character counts to zero.

Solution:

```
class Solution {
public:
    int balancedStringSplit(string s) {
        int l=0,r=0,n=s.length(),cnt=0;
        for(int i=0;i<n;i++)
        {
            if(s[i]=='L') l++;
            else r++;
            if(l==r)
            {
                l=0;
                r=0;
                cnt++;
            }
        }
        return cnt;
    }
};
```

Complexity:

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

Question 2:

- **Problem link**: https://leetcode.com/problems/minimum-subsequence-in-non-increasing-order/
- **Difficulty level**: Easy

Explanation:

Steps:

1) Find the sum of all the elements of the vector.

2)Sort the vector in descending order.

3)Keep on subtracting the elements from the highest to lowest until the sum of highest elements is greater than the remaining. Also,add the elements to the answer vector.

4)The required elements till the condition meets is the final answer.

5)Return them in non decreasing order

Solution:

```
class Solution {
```

```
public:
    vector<int> minSubsequence(vector<int>& nums) {
        int sum=0,newsum=0;
        vector<int>ans;
        for(int i=0;i<nums.size();i++)
        {
            sum+=nums[i];
        }
        sort(nums.begin(),nums.end(),std::greater<>());
        for(int i=0;i<nums.size();i++)
        {
            newsum+=nums[i];
            sum=sum-nums[i];
            ans.push_back(nums[i]);
            if(newsum>sum)
                break;
        }
        sort(ans.begin(),ans.end(),std::greater<>());
        return ans;
    }
};
```

Complexity:

- Time: O(nlogn)
- Extra space: O(1) excluding output array

Question 3:

- **Problem link**:
- **Difficulty level**: Easy

Explanation:
The main focus in this question is on the digit 9 which creates all the changes otherwise for other digits we have to just increment their value by 1 but if we change the node's value with the value 9 it makes a carry which then has to be passed through the linked list.

Find the last node in the linked list which is not equal to 9. Now there are three cases:

1. If there is no such node i.e. the value of every node is 9 then the new linked list will contain all 0s and a single 1 inserted at the head of the linked list.
2. If the rightmost node which is not equal to 9 is the last node in the linked list then add 1 to this node and return the head of the linked list.
3. If the node is other than the last node i.e. every node after it is equal to 9 then add 1 to the current node and change all the nodes after it to 0.

Solution:

```
Node* addOne(Node* head){
    Node* last = NULL;
    Node* cur = head;
    while (cur->next != NULL) {
        if (cur->data != 9) {
            last = cur;
        }
        cur = cur->next;
    }
    if (cur->data != 9) {
        cur->data++;
        return head;
    }
    if (last == NULL) {
        last = new Node();
        last->data = 0;
        last->next = head;
        head = last;
    }
    last->data++;
    last = last->next;
    while (last != NULL) {
        last->data = 0;
        last = last->next;
    }
    return head;
}
```

Complexity:

❖ Time: O(n)
❖ Space: O(1)

Question 4:

❖ **Problem link**:
❖ **Difficulty level**: Medium

Explanation:

We take two pointers ‘before’ and ‘after’ to keep track of the two linked lists, the first pointing to a list with elements less than ‘x’ and the second pointing to the list with elements greater than or equal to ‘x’. Iterating through the original list, if an element is smaller than ‘x’ then move the node to ‘before’ otherwise move it to list with the pointer ‘after’. Finally, combine the two lists.

Solution:

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {

        ListNode *before_head = new ListNode(0);
        ListNode *before = before_head;
        ListNode *after_head = new ListNode(0);
        ListNode *after = after_head;

        while (head != NULL) {


            if (head->val < x) {
                before->next = head;
                before = before->next;
            } else {
                after->next = head;
                after = after->next;
            }

            head = head->next;
        }

        after->next = NULL;
        before->next = after_head->next;
        return before_head->next;
    }
};
```

Complexity:

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

Question 5:

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

Explanation:

For each position, we need to find the best potential replacement to minimize abs(n1[i] - n2[i]). We can track the biggest gain between original and replaced value among all numbers, and use it in the end.

To find the best potential replacement, we can sort numbers in n1 and use binary search to find numbers close to n2[i].

Solution:

```
class Solution {
public:
    int minAbsoluteSumDiff(vector<int>& n1, vector<int>& n2) {
      long res = 0, gain = 0;
      set<int> s(begin(n1), end(n1));
      for (int i = 0; i < n1.size(); ++i) {
         long original = abs(n1[i] - n2[i]);
         res += original;
         if (gain < original) {
            auto it = s.lower_bound(n2[i]);
            if (it != end(s))
               gain = max(gain, original - abs(*it - n2[i]));
            if (it != begin(s))
               gain = max(gain, original - abs(*prev(it) - n2[i]));
         }
      }
      return (res - gain) % 1000000007;
   }
};
```

Complexity:

- Time: O(nlogn)
- Space: O(1) excluding output array

Question 6:

- **Problem link**: https://leetcode.com/problems/gas-station/
- **Difficulty level**: Medium

Explanation:

Another efficient solution can be to find out the first gas station where the amount of gas is greater than or equal to the cost needed to be covered to reach the next gas station. Now we mark that gas station as start and now we check whether we can finish the journey towards the end point. If in the middle, at any gas station, the amount of gas is less than the cost that is needed to reach the next gas station, then we can say we cannot complete the circular tour from start. We again try to find out the next point from where we can start our journey i.e. the next gas station where the amount of gas is greater than or equal to the cost needed and we mark it as start. We need not look at any gas station in between the initial gas station marked as start and and the new start as we know that we cannot complete the journey if we start from any middle gas station because eventually we will arrive at a point where the amount of gas is less than the cost. Now we repeat the process until we reach the last gas station and update our start as and when required. After we reach our last gas station, we try to reach our first gas station from the last and let's say we have a remaining amount of gas as gas_left. Now we again start travelling from the first gas station and take the advantage of our gas_left and try to reach the start. If we can reach the start, then we may conclude that start can be our starting point.

Solution:

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost)
    {
        int i, j, n = gas.size();

        /*
         If start from i, stop before station x -> no station k from i + 1 to x - 1 can reach x.
         Bcoz if so, i can reach k and k can reach x, then i reaches x. Contradiction.
         Thus i can jump directly to x instead of i + 1, bringing complexity from O(n^2) to
O(n).
         */
        // start from station i
        for (i = 0; i < n; i += j) {
            int gas_left = 0;
            // forward j stations
            for (j = 1; j <= n; j++) {
                int k = (i + j - 1) % n;
                gas_left += gas[k] - cost[k];
                if (gas_left < 0)
                    break;
            }
            if (j > n)
                return i;
        }

        return -1;
    }
};
```

Complexity:

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

Question 7:

- **Problem link**: https://practice.geeksforgeeks.org/problems/minimum-platforms-1587115620/1
- **Difficulty level**: Medium

Explanation: The idea is to consider all events in sorted order. Once the events are in sorted order, trace the number of trains at any time keeping track of trains that have arrived, but not departed.

Algorithm:

1. Sort the arrival and departure time of trains.
2. Create two pointers i=1, and j=0 and a variable to store the number of platforms needed, plat=1(initially plat=1 because for first arriving train atleast a platform is needed and hence i starts from 1)
3. Run a loop while i<n and j<n and compare the ith element of arrival array and jth element of departure array.
4. if the arrival time is less than or equal to departure then one more platform is needed so increase the count, i.e. plat++ and increment i
5. Else if the arrival time greater than departure then increase i and j, i.e. no new platform is needed

Solution:

```
class Solution{
    public:
    //Function to find the minimum number of platforms required at the
    //railway station such that no train waits.
    int findPlatform(int arr[], int dep[], int n)
    {
        // Your code here


        int plat=1;
        sort(arr,arr+n);
        sort(dep,dep+n);
        int i=1,j=0;
        while(i<n && j<n){
            if(dep[j] >= arr[i]){
                i++;
                plat++;
            }
            else{
                i++;
                j++;
            }
        }
        return plat;
    }
};
```

Complexity:

- Time: O(n*logn)
- Space: O(1) (since no space extra required in this approach)

Question 8:

Problem link:

- **Difficulty level**: Hard

Explanation:
The idea is based on iterative linked list reverse process. We iterate through the given linked list and one by one reverse every prefix of the linked list from the left. After reversing a prefix, we find the longest common list beginning from reversed prefix and the list after the reversed prefix.

Solution:

```
int countCommon(Node *a, Node *b)
{
    int count = 0;

    // loop to count common in the list starting
    // from node a and b
    for (; a && b; a = a->next, b = b->next)

        // increment the count for same values
        if (a->data == b->data)
            ++count;
        else
            break;

    return count;
}

/*The function below returns an int denoting
the length of the longest palindrome list. */

int maxPalindrome(Node *head)
{
    //Your code here
    int result = 0;
    Node *prev = NULL, *curr = head;

    // loop till the end of the linked list
    while (curr)
    {
        // The sublist from head to current
        // reversed.
        Node *next = curr->next;
        curr->next = prev;

        // check for odd length palindrome
        // by finding longest common list elements
        // beginning from prev and from next (We
        // exclude curr)
        result = max(result,
                     2*countCommon(prev, next)+1);

        // check for even length palindrome
        // by finding longest common list elements
        // beginning from curr and from next
        result = max(result,
                     2*countCommon(curr, next));

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
    return result;
}
```

Complexity:

- Time: $O(n^2)$
- Space: $O(1)$