

# Online Contests Solutions

Saman Saadi



# Contents

<b>1</b>	<b>LeetCode</b>	<b>1</b>
1.1	Longest Substring Without Repeating Characters . . . . .	1
<b>2</b>	<b>HackerRank</b>	<b>3</b>
2.1	New Year Chaos . . . . .	3
2.2	Minimum Swaps 2 . . . . .	4
2.3	Count Triplets . . . . .	4
2.4	Fraudulent Activity Notifications . . . . .	5
2.5	Merge Sort: Counting Inversions . . . . .	7
<b>3</b>	<b>TopCoder</b>	<b>9</b>
3.1	SRM 428 . . . . .	9
3.1.1	ThePalindrome . . . . .	9



# Chapter 1

## LeetCode

### 1.1 Longest Substring Without Repeating Characters

The **question** is to find the length of the longest substring without repeating characters. For example in "abcabcbb" the answer is "abc".

We can use mathematical induction. Assume  $s[i..j-1]$  is a valid substring (it doesn't have any repetitive character). Now we want to add  $s[j]$  to it. If  $s[j]$  is not in  $s[i..j-1]$ , then we can easily add  $s[j]$  to the list and claim  $s[i..j]$  is also valid. Otherwise there is exactly one  $i \leq k \leq j-1$  that  $s[k] = s[j]$ .

Because of  $s[j]$ , we cannot extend  $s[i..j-1]$  anymore, but we may be able to extend  $s[k+1..j]$ . The running time of the following implementation is  $O(n)$ :

```
int lengthOfLongestSubstring(string s) {
    int res = 0;
    unordered_map<char, int> mp;
    for (int i = 0, j = 0; i < s.length() && j < s.length(); ++j)
    {
        auto iter = mp.find(s[j]);
        if (iter != mp.end())
        {
            int k = iter->second;
            for (; i <= k; ++i)
                mp.erase(s[i]);
        }
        mp[s[j]] = j;
        res = max(res, j - i + 1);
    }
    return res;
}
```



## Chapter 2

# HackerRank

### 2.1 New Year Chaos

You can find the question in this [link](#).

We define  $index_i$  as the current index for person  $i$ . For example if we have 1, 2, 3, 4 and 4 bribes 3, the queue looks like 1, 2, 4, 3. So  $index_4 = 3$ . Since no body can bribe more than 2 times,  $index_i \geq i - 2$  for  $1 \leq i \leq n$ . Consider person  $n$ . No body can bribe that person. So  $n - 2 \leq index_n \leq n$ . After we retruned that person to his actual place we can consider  $n - 1$ . So we have  $n - 3 \leq index_{n-1} \leq n - 1$  (note that at this moment  $index_n = n$ ).

```
void minimumBribes(vector<int> q) {

    const auto& n = q.size();
    int res = 0;
    for (int num = n; num > 0; --num)
    {
        for (int i = max(0, num - 3); i < num - 1; ++i)
        {
            if (q[i] == num)
            {
                ++res;
                swap(q[i], q[i + 1]);
            }
        }
        if (q[num - 1] != num)
        {
            cout << "Too chaotic" << endl;
            return;
        }
    }
    cout << res << endl;
}
```

## 2.2 Minimum Swaps 2

See the problem statement in this [link](#).

Note that this solution is based on **Selection Sort** in which the number of swaps are minimum. According to Wikipedia: "One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps,  $n - 1$  in the worst case.". Although Selection sort has minimum number of swaps among all sorts algorithms, it has  $O(n^2)$  comparisons. Since the final result is  $\{1, 2, \dots, n\}$ , it's like we have the set in sorted order so we can bypass comparisons and use Selection Sort advantage which is the minimum number of swaps.

We define  $index_i$  as the current index of number  $i$ . Suppose we have  $n$  numbers, so  $1 \leq index_i \leq n$ . The goal is to have  $index_i = i$ . Without losing generality suppose  $i < j \wedge index_i = j$ . There are two cases to consider:

1. If  $index_j = i$ , then by swapping  $arr_i$  and  $arr_j$ , we put both  $i$  and  $j$  in their corresponding positions.
2. If  $index_j = k \wedge k \neq i \wedge k \neq j$ . In this case by swapping  $arr_i$  and  $arr_j$  we only put  $i$  in its corresponding position. So we need to do an extra swap to put  $j$  in its correct position.

We can start from  $i = 1$  to  $i = n$  and make sure  $i$  is in correct position; otherwise we perform a swap. In each iteration we fix the position of one or two numbers. A good example is  $\{4, 3, 2, 1\}$ .

```
int minimumSwaps(vector<int> arr) {
    const auto& n = arr.size();
    vector<int> index(n + 1);

    for (int i = 0; i < n; ++i)
        index[arr[i]] = i;
    int cnt = 0;
    for (int num = 1; num <= n; ++num)
    {
        if (index[num] != num - 1)
        {
            ++cnt;
            index[arr[num - 1]] = index[num];
            swap(arr[index[num]], arr[num - 1]);
            index[num] = num - 1;
        }
    }
    return cnt;
}
```

## 2.3 Count Triplets

**Problem statement.**



We use dynamic programming to solve it. For mathematical induciton we define  $cnt[num][n]$  like this:

$$cnt[a_{i_1}][0] = |\{a_{i_0} \in arr \mid a_{i_1} = a_{i_0} \times r \wedge i_1 < i_2\}|$$

$$cnt[a_{i_2}][1] = |\{(a_{i_0}, a_{i_1}) \in arr \times arr \mid a_{i_k} = a_{i_{k-1}} \times r \wedge i_{k-1} < i_k \text{ for } 1 \leq k \leq 2\}|$$

So the final answer is:

$$\sum_{n \in arr} cnt[n][1]$$

Then for each number  $n$  we have

$$cnt[n \times r][0] = cnt[n \times r][0] + 1$$

$$cnt[n \times r][1] = cnt[n \times r][1] + cnt[n][0]$$

Since  $r = 1$ , the order of assignments are very important.

```

long countTriplets(vector<long> arr, long r) {
    const auto n = arr.size();
    unordered_map<long, array<long, 2>> cnt;
    //cnt[a[j]][0] = |\{a[i]\}| in which i < j and
    //                a[j] = a[i] * r
    //cnt[a[k]][1] = |\{a[i], a[j]\}| in which
    //                i < j < k and
    //a[k] = a[j] * r and a[j] = a[i] * r

    long res = 0;
    for (const auto& num : arr)
    {
        res += cnt[num][1];
        const auto next = num * r;
        cnt[next][1] += cnt[num][0];
        ++cnt[next][0];
    }
    return res;
}

```

## 2.4 Fraudulent Activity Notifications

### Problem Statement

Basically we want a  $O(n \log n)$  algorithm to find median of a sequence, when we removed the first element and add another one. So we need two binary search trees. In the first one the maximum element is the median itself and in the second one the minimum element is the second median in case of  $d = 2k$  or a value greater than median when  $d = 2k + 1$ . So if  $d = 2k$  both of these binary search trees always have  $k$  element. When  $d = 2k + 1$ , the first one always has  $k + 1$  elements and the second one has  $k$  elements. Let's call them *lessEqual* and *greaterEqual*.

If both removing element and new element belong to the same tree, nothing extra is required. So we only need to remove one element and add the new one. If the removing element is from *lessEqual*, we must remove the minimum element from *greaterEqual* and add it to *lessEqual*. If the removing element is from *greaterEqual*, we must remove the maximum element from *lessEqual* and add it to *greaterEqual*. By doing that the maximum element is *lessEqual* is median. In case of  $d = 2k$ , the minimum element in *greaterEqual* is the second median. The running time of this algorithm is  $O(n \log n)$ .

```
int activityNotifications(vector<int> expenditure, int d)
{
    multiset<int, greater<int>> lessEqual;
    multiset<int> greaterEqual;

    vector<int> init(d);
    copy(expenditure.begin(), expenditure.begin() + d,
        init.begin());
    sort(init.begin(), init.end());

    const bool isEven = (d & 1) == 0;

    int medianIndex = (d - 1) / 2;
    int i;
    for (i = 0; i <= medianIndex; ++i)
        lessEqual.insert(init[i]);
    for (; i < d; ++i)
        greaterEqual.insert(init[i]);

    int res = 0;
    for (int i = d; i < expenditure.size(); ++i)
    {
        const int median1 = *lessEqual.begin();
        if (isEven)
        {
            const int median2 = *greaterEqual.begin();
            if (expenditure[i] >= (median1 + median2))
                ++res;
        }
        else
        {
            if (expenditure[i] >= 2 * median1)
                ++res;
        }

        const auto removed = expenditure[i - d];

        if (removed <= median1 &&
            expenditure[i] <= median1)
        {
            lessEqual.erase(lessEqual.find(removed));
            lessEqual.insert(expenditure[i]);
        }
        else if (removed > median1 &&
            expenditure[i] > median1)
        {
            greaterEqual.erase(greaterEqual.find(removed));
        }
    }
}
```

```

        greaterEqual.insert(expenditure[i]);
    }
    else if ( removed <= median1)
    {
        //For handling d=1, it should first:
        greaterEqual.insert(expenditure[i]);
        lessEqual.erase(lessEqual.find(removed));
        lessEqual.insert(*greaterEqual.begin());
        greaterEqual.erase(greaterEqual.begin());
    }
    else
    {
        //For handling d=1, it should be first:
        lessEqual.insert(expenditure[i]);
        greaterEqual.erase(greaterEqual.find(removed));
        greaterEqual.insert(*lessEqual.begin());
        lessEqual.erase(lessEqual.begin());
    }
}
return res;
}

```

## 2.5 Merge Sort: Counting Inversions

### Problem Statement

We can solve it using merge sort. Suppose we have  $arr[left..right]$ . We break it into two subproblem  $arr[left..mid]$  and  $arr[mid + 1..right]$ . Both of them are sorted. According to merge sort algorithm  $left \leq i \leq mid$  and  $mid + 1 \leq j \leq right$ . In other words we already put  $arr[left..i - 1]$  and  $arr[mid + 1..j - 1]$  into their correct positions. So when  $arr[j] < arr[i]$ , it means  $arr[j] < arr[i] \leq arr[x]$  in which  $i + 1 \leq x \leq mid$ . So we need to have  $mid - i + 1$  swaps.

```

long mergeSort(vector<int>& arr, int leftIndex,
               int rightIndex)
{
    if (leftIndex == rightIndex)
        return 0;

    long res = 0;
    int midIndex = (leftIndex + rightIndex) / 2;
    res = mergeSort(arr, leftIndex, midIndex);
    res += mergeSort(arr, midIndex + 1, rightIndex);

    vector<int> sorted(rightIndex - leftIndex + 1);
    int i, j, k;
    for (i = leftIndex, j = midIndex + 1, k = 0;
         i <= midIndex && j <= rightIndex;)
    {
        if (arr[i] <= arr[j])
            sorted[k++] = arr[i++];
        else
        {

```

```
        res += midIndex - i + 1;
        sorted[k++] = arr[j++];
    }
}
if (i <= midIndex)
    copy(arr.begin() + i,
         arr.begin() + midIndex + 1,
         sorted.begin() + k);
else
    copy(arr.begin() + j ,
         arr.begin() + rightIndex + 1,
         sorted.begin() + k);
copy(sorted.begin(), sorted.end(),
     arr.begin() + leftIndex);
return res;
}
// Complete the countInversions function below.
long countInversions(vector<int> arr) {
    return mergeSort(arr, 0, arr.size() - 1);
}
```

## Chapter 3

# TopCoder

### 3.1 SRM 428

#### 3.1.1 ThePalindrome

We want to add the minimum number of characters to the end of string to make it a palindrome. The straightforward approach is to try add the first  $i$  characters in reverse for all  $0 \leq i \leq n - 1$  in which  $n$  is the length of string. So we start from  $i = 0$  and check whether the string is palindrome. If it's not we check for  $i = 1$  and so on. The running time of this algorithm is  $O(n^2)$ .

```
bool isPalindrome(const string& str)
{
    int left = 0, right = str.length() - 1;
    for (; left < right && str[left] == str[right];
        ++left, --right);
    return left >= right;
}

int find(string s)
{
    for (int i = 0; i < s.length(); ++i)
    {
        string tmp = s + string(s.rend() - i, s.rend());
        if (isPalindrome(tmp))
            return tmp.length();
    }
    throw 1;
}
```

Since  $n \leq 50$ , this algorithm is fast. We can make it  $O(n \log_2 n)$  if we use binary search tree to find the minimum  $i$ .

There is another approach. Let's assume we have string  $S = s_1 s_2 \dots s_n$ . We define  $S' = s_n \dots s_2 s_1$ . Suppose we can write string  $S$  as  $QP$ . In other words,  $S$  is the concatenation of two strings  $Q$  and  $P$ . We assume  $P$  is palindrome but  $Q$  is not. We can make  $S$  palindrome if we convert  $QP$  to  $QPQ'$  we call this

new String  $Z$ .  $Z$  is palindrome because if we reverse it we have:

$$\begin{aligned} Z &: QPQ' \\ Z' &: QP'Q' \end{aligned}$$

Since we want the length of  $Q$  be minimum, we must find the maximal  $P$ :

```
bool isPalindrome(const string& str, int start)
{
    int left = start, right = str.length() - 1;
    for (; left < right && str[left] == str[right];
        ++left, --right);
    return left >= right;
}

int find(string s)
{
    for (int i = 0; i < s.length(); ++i)
    {
        if (isPalindrome(s, i))
            return s.length() + i;
    }
    throw 1;
}
```

As the previous implementation the running time is  $O(n^2)$  but it's easy to convert it to  $O(n \log_2 n)$  using binary search.

This implementation has a unique feature. We can convert it to an  $O(n)$  algorithm using **KMP algorithm**. Suppose  $S = QP$  where  $P$  is a palindrome. We want to find palindrome postfix  $P$  which its length is maximum among all palindrome post-fixes. We need to run KMP pre-compute calculation on  $S' = P'Q'$ . Then we run KMP algorithm as if we want to find whether  $S'$  is a substring of  $S$ . Suppose we use  $i$  as an index for  $S$  and  $j$  as an index for  $S'$ . The algorithm start with  $i = 0$  and ends when  $i = \text{len}(S)$  in which  $S'[0..j-1]$  is  $P'$  or  $P$  (since it's palindrome).

```
vector<int> calculateNext(const string& B)
{
    vector<int> next(B.length());
    next[0] = -2;
    next[1] = -1;
    int i, j;
    for (i = 2; i < B.length(); ++i)
    {
        j = next[i - 1] + 1;
        for (; j >= 0 && B[j] != B[i - 1]; j = next[j] + 1);
        next[i] = j;
    }
    return next;
}

int find(string A)
{
    const string B = string(A.rbegin(), A.rend());
    const auto next = calculateNext(B);
```

```
int i, j;
for (i = 0, j = 0; i < A.length() && j < B.length();)
{
    if (A[i] == B[j])
        ++i, ++j;
    else if ((j = next[j] + 1) < 0)
    {
        //Since B.front() == A.back(), it's impossible
        //i == A.length() here:
        ++i, j = 0;
    }
}
int palindromeLen = j;
return A.length() + A.length() - palindromeLen;
}
```