Online Contests Solutions

Saman Saadi

Contents

Chapter 1

HackerRank

1.1 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;
```

```
}
```

1.2 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.”. Altourh Selection sort has minimum number of swaps among all sorts agorithms, it has $O(n^2)$ comparisons. Since the final result is $\{1, 2, \ldots, n\}$, 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;
}
```

1.3 Count Triplets

Problem statement.

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

$$cnt[a_{i_1}][0] = |\{a_{i_0} \in arr \mid a_{i_1} = a_{i_0} \times r \ \wedge \ i_1 < i_2\}|$$
$$cnt[a_{i_2}][1] = |\{(a_{i_0}, a_{i_1}) \in arr \times arr \mid a_{i_k} = a_{i_{k-1}} \times r \ \wedge \ i_{k-1} < i_k \text{ for } 1 \le k \le 2\}|$$

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;
}
```

1.4 Fraudulent Activity Notifications

Problem Statement

Basically we want a $O(nlogn)$ algorithm to find median of a sequuence, 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 secon 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 $lessEqaul$ 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 $O(nlogn)$.

```
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;
}
```