CyberChallenge.IT 2022 - Programming
Commented solutions

Contents

1 Problem "Range Cover"

1.1 Problem Statement

In this task you are given in input N different ranges and a number K. Each range is defined as a pair of integers [start, end], where both start and end are included in the range. You goal is to count how many integers are contained in exactly K of these ranges.
For example, given $K = 3$ and $N = 6$ ranges: [3, 10], [0, 5], [6, 13], [1, 15], [13, 19] and [15, 18], only 10 integers are covered exactly by $K = 3$ ranges, in particular:

- 3 is covered by the ranges: [3, 10], [0, 5] and [1, 15]
- 4 is covered by the ranges: [3, 10], [0, 5] and [1, 15]
- 5 is covered by the ranges: [3, 10], [0, 5] and [1, 15]
- 6 is covered by the ranges: [3, 10], [6, 13] and [1, 15]
- 7 is covered by the ranges: [3, 10], [6, 13] and [1, 15]
- 8 is covered by the ranges: [3, 10], [6, 13] and [1, 15]
- 9 is covered by the ranges: [3, 10], [6, 13] and [1, 15]
- 10 is covered by the ranges: [3, 10], [6, 13] and [1, 15]
- 13 is covered by the ranges: [6, 13], [1, 15] and [13, 19]
- 15 is covered by the ranges: [1, 15], [13, 19] and [15, 18]

All the over indexes are covered by less then $K = 3$ ranges.

1.2 Problem Details

Input

The first line of the input contains two space-separated integers **N** and **K** representing the number of ranges available and the number of index of overlapping ranges to found.
The next **N** lines contain two space-separated integers each, representing the starting and the ending point of the coordinates (included).

Output

The output must contains only one integer, representing how many indexes are covered by exactly **K** ranges.

Scoring

- Subtask 1 (40 points): $N = 10$ and ranges are between 0 and 10.
- Subtask 2 (40 points): $N = 100$ and ranges are between 0 and 10 000.
- Subtask 3 (20 points): $N = 10\,000$ and ranges are between 0 and 10^{15}.

1.3 Solution

Subtask 1 (40 points)

For this subtask, a simple bruteforce approach works. We start with an array of length 11 (the numbers between 0 and 10) V initialized to all zero elements and, for each range $[a, b]$, we loop from a to b (included) adding 1 to each element V_i. At the end we count how many elements of V are exactly equal to K. The time complexity of this approach is, in worst case, $O(NL)$, where L is the maximum possible length of a range.

Subtask 2 (40 points)

For this subtask we propose a slightly more clever approach. We start as in the previous case, with an array of length 10001 V initialized to 0. This time we don't loop through each range but instead we take every range $[a, b]$, increment V_a and decrement V_{b+1}. For each element V_i with $i > 0$ we set V_i to $V_i + V_{i-1}$ obtaining the same representation as the previous subtask and then proceed as before, counting the number of entries of V that are exactly equal to K. This approach has time complexity $O(N)$.

Subtask 3 (20 points)

Notice that, altough the algorithm for Subtask 2 would be fine also here in terms of time complexity, it has a memory complexity of $O(L)$ that is too much with the constraints of this last subtask. The idea here is to keep the same approach as the previous subtask but without storing the complete array. Instead, we store only the relevant cells: the start and the end of each range. In order to do this we can for example store an array of pairs of the form $(a, 1)$ and $(b + 1, -1)$ for each range $[a, b]$ and then proceed in a similar way as before, looping on these pairs instead of the full array. This solution is $O(N)$ in both time and space complexity.

1.4 Source Code

C++

```cpp
#include <bits/stdc++.h>

using namespace std;

int main(){
    int N, K, level = 0;
    long long int answer = 0;
    cin >> N >> K;
    vector<array<long long int, 2>> ranges(2*N);

    for(int i=0; i<N; i++) {
        cin >> ranges[2*i][0];
        cin >> ranges[2*i+1][0];
        ranges[2*i][1] = 1;
        ranges[2*i+1][1] = -1;
    }

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

    for(int i = 0; i<2*N-1; i++){
        level += ranges[i][1];
        if(level == K)
            answer += ranges[i+1][0] - ranges[i][0];
    }

    cout << answer << endl;

    return 0;
}
```

Python

```python
#!/bin/env python3

N, K = map(int, input().strip().split(" "))

events = []
```

```
count, level = 0, 0

for _ in range(N):
    start, end = map(int, input().strip().split(" "))
    events.extend([start, +1], [end+1, -1])

events.sort()

for i in range(len(events)-1):
    level += events[i][1]
    if level == K:
        count += (events[i+1][0] - events[i][0])

print(count)
```

2 Problem "Password Protection Officer"

2.1 Problem Statement

For us at CC (Coste Challenge), password protection is very important, for this reason we want you as our new PPO (Password Protection Officer).
The password policy for our website is very hard to verify, in particular, when a user is changing it, the new password should follow all these requirements:

1. The new password must be at least 8 characters long.
2. The new password must be at most 16 characters long.
3. The new password must contains both lowercase and uppercase letters.
4. The new password must contains at least one digit and one special character.
5. The new password must not contain two consecutive identical characters.
6. The new password must not be derivable by deleting, substituting or adding exactly one character from the old password.

Write a program that, given a list of pairs of new and old passwords, returns if the new passwords follows all the requirements.

2.2 Problem Details

Input

The first line of the input contains the integer **N**, the number of password checks to perform.
The following **N** lines contains two space-separated strings each, the new password and the old password respectively.

Output

The output must contain N lines, one for each password checking. Each line should contains 1 if the new password is valid, 0 otherwise.

Scoring

For each of the subtasks, the following constraints are met:

- Subtask 1 (30 points): If the password is not valid, it violates one of the requirements from 1 to 3.
- Subtask 2 (30 points): If the password is not valid, it violates one of the requirements from 1 to 5.
- Subtask 3 (40 points): If the password is not valid, it violates one of the requirements from 1 to 6.

2.3 Solution

This problem is an implementation task. No clever ideas or algorithmic knowledge is necessary to solve it, but it requires a good familiarity with a programming language and some implementation skills.

Subtask 1 (30 points)

In order to solve this subtask we need to satisfy the first 3 constraints. The first 2 are pretty simple: we take the length of the new password and check that it lies between 8 and 16 characters. For the third one, let's consider for example lowercase letters. We start by initializing a boolean variable to false and we loop through the new password. If the current character is a lowercase letter, we set the boolean to true. At the end of the string, if the boolean is still false, the check is not passed, otherwise it is. A similar check works also for uppercase letters.

Subtask 2 (30 points)

For this subtask we need to extend the previous solution to constraints 4 and 5. Constraint 4 can be solved in the exact same way as constraint 3, looping through the new password with 2 more boolean variables. For the constraint 5 we loop again through the string (using an index from 0 to the length minus one) and we check if there exist an index i such that the character at index i and the one at index $i+1$ are equal. If not, the check is passed.

Subtask 3 (40 points)

This last subtask is a little bit more tedious. In particolar, it can become very long to code if approached in the wrong way. The proposed solution, calling $p = p_0p_1...p_n$ the new password and $o = o_0o_1...o_m$ the old one, is to compute all the substrings of the two passwords of the form $p_0p_1...p_ip_{i+2}...p_n$ and $o_0o_1...o_jo_{j+2}...o_m$ with $0 \leq i \leq n$ and $0 \leq j \leq m$. We call these two sets P and O respectively. Then it boils down to check:

- if the intersection of P and O is empty,
- if p is not in O,
- if o is not in P.

If all these constraints are satisfied, then check 6 is passed, otherwise it is not. This completes the last point of the task.

2.4 Source Code

C++

```cpp
#include <bits/stdc++.h>

using namespace std;

bool valid(string p, string o){
    bool lc = false,  uc = false, dig = false, spec = false;
    string punctuation = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~";

    if(p.length() < 8 || p.length() > 16)
        return false;

    for(int i = 0; i<p.length(); i++){
        if(int('A') <= int(p[i]) && int('Z') >= int(p[i]))
            uc = true;
        else if(int('a') <= int(p[i]) && int('z') >= int(p[i]))
            lc = true;
        else if(int('0') <= int(p[i]) && int('9') >= int(p[i]))
            dig = true;
        else if(punctuation.find(p[i]) != string::npos)
            spec = true;

        if(i<p.length()-1){
            if(p[i] == p[i+1])
                return false;

            string old_sliced = o.substr(0, i) + o.substr(i+1);
            string new_sliced = p.substr(0, i) + p.substr(i+1);

            if(p == old_sliced || new_sliced == old_sliced || o == new_sliced)
                return false;
        }

    }

```

```
    return lc && uc && dig && spec;
}

int main(){
    int N;
    string new_psw, old_psw;

    cin >> N;

    for(int i=0; i<N; i++) {
        cin >> new_psw >> old_psw;
        if(valid(new_psw, old_psw))
            cout << 1 << endl;
        else
            cout << 0 << endl;
    }

    return 0;
}
```

Python

```
#!/bin/env python3

from string import *


def valid1(p):
    return 8 <= len(p) <= 16 and any(c in ascii_lowercase for c in p) and any(c in ascii_uppercase for c in p)


def valid2(p):
    return valid1(p) and any(c in digits for c in p) and any(c in punctuation for c in p)
           and all(p[i] != p[i+1] for i in range(len(p)-1))


def valid3(p, old):
    # Cancellazione da old
    check1 = not any(p == (old[:i] + old[i+1:]) for i in range(len(old)-1))
    # Sostituzione da old
    check2 = len(p) == len(old) and not any(
        (p[:i] + p[i+1:]) == (old[:i] + old[i+1:]) for i in range(len(old)-1))
    # Aggiunta da old
    check3 = not any(old == (p[:i] + p[i+1:])
                     for i in range(len(old)-1))
    return valid2(p) and check1 and check2 and check3


N = int(input())
for _ in range(N):
    p1, p2 = input().split(" ")
    print(1 if valid3(p1, p2) else 0)
```

3 Problem "Swap the numbers"

3.1 Problem Statement

In this task you are given a vector V of length N. The vector V is generated such that is ordered, except for a single element, e.g. in $V = [1, 2, 9, 5, 6]$ only the number 9 is not placed correctly.
Your task is to count how many swaps of two elements of the vector (even not adjacent) are required to sort increasingly the vector.
Given the previous vector V, the solution is thus 2 as:

1. Swap (9, 6) to get $V = [1, 2, 6, 5, 9]$
2. Swap (6, 5) to get $V = [1, 2, 5, 6, 9]$

3.2 Problem Details

Input

The first line of the input contains one integer **N**, the length of the vector V.
The following line contains **N** space-separated integers, the vector V.

Output

The output must contains a single line with an integer, representing the number of swaps needed to sort the vector V.

Scoring

For each of the subtasks, the following constraints are met:

- Subtask 1 (50 points): In this subtask $N = 100$ and V contains all the numbers from 0 to $N - 1$.
- Subtask 2 (50 points): In this subtask $N \leq 100\,000$ and V can contain any number.

3.3 Solution

Subtask 1 (50 points)

In this first subtask we have that all the elements are distinct. A simple solution then is to create a sorted copy of the given array (let's call the initial array V and the copy V'), find the misplaced element in V and compare the position of this element in the two arrays. Their difference, in absolute value, will be the answer.

Subtask 2 (50 points)

For this second subtask the only difference is that elements can be repeated. Modifying the previous algorithm to work also in this case is not too hard, but can be a little tedious to implement for some edge cases. Instead, we propose a cleaner solution that works in both cases. Let's consider again our two arrays V and V': by construction there exists a subarray V'' (not necessarily continuous) of V' such that all the elements of V'' are different than the corresponding elements of V and the length of V'' is maximum (for example: if $V = [1, 3, 1, 2, 3, 3, 4]$ and $V' = [1, 1, 2, 3, 3, 3, 4]$, we have $V'' = [3, 1, 2]$). Moreover, this subarray has the property that either its maximum is at position 0, or its minimum is in the last position, and all the other elements are sorted. What we want to do then is simply sorting this subarray, that, if its length is L, can be done with $L - 1$ swaps.

3.4 Source Code

C++

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
```

```
    int N, ans = -1;
    cin >> N;

    vector<int> V(N), V2;

    for(int i = 0;i<N;i++)
        cin >> V[i];

    V2 = V;
    sort(V2.begin(), V2.end());

    for(int i = 0; i<N; i++){
        if(V2[i] != V[i]) ans++;
    }

    cout << max(0, ans) << endl;

    return 0;
}
```

Python

```
#!/bin/env python3

import sys

def swap(N, V):
    V2 = list(sorted(V))
    ans = -1

    for i in range(N):
        if V2[i] != V[i]:
            ans += 1

    return max(0, ans)

N = int(fin.readline().strip())
V = list(map(int, fin.readline().strip().split(" ")))
print(swap(N, V))
```