CyberChallenge.IT 2021 - Programming
Commented solutions

Contents

1 Problem "Unbalancer"

1.1 Problem Statement

As first task, you are required to write the worst possibile software, in particular we need you to write a load (un)balancer.
In our local server we have N available workers, for each of them we know the number of tasks it has to execute. The load (un)balancer can move any amount of available tasks from one worker to another worker. Your job is to write the load (un)balancer in a way that maximimize the unbalance of the server with K possibile moves available.
The unbalance of server is defined as the highest possibile difference of tasks between two workers.
Write a program that, given the number of tasks and moves available, returns the maximum possible unbalance between two workers.

1.2 Problem Details

Input

Your program must read the input data from the standard input.
The first line of the input contains two space-separated integers **N** and **K** representing the number of worker and the number of movements.
The next line contains **N** space-separated integers representing the number of tasks in each workers.

Output

Your program must write the output data into the standard output.
The output must contains only one integer, representing the maximum difference possibile between the workers after K moves.

Scoring

For each of the test cases the program will be tested, the following constraints are met:

- $K = 1$ and $N \leq 10$ for at least 25% of all the test cases.
- $K \leq 10$ and $N \leq 100$ for at least 50% of all the test cases.
- $K \leq 100$ and $N \leq 1\,000$ for at least 75% of all the test cases.
- $K \leq 1\,000$ and $N \leq 10\,000$ for all the test cases.

1.3 Solution

This is the easiest problem in the problem set. It requires a basic idea to be solved, but then the implementation is straightforward, with no big programming knowledge required.

Subtask 1 (25 points)

Since the amount of tasks for a certain worker can not be lower than 0, the idea here is pretty simple: we want to leave one worker with 0 tasks, and put the maximum possible amount of them in another. What is this maximum? Since we have only one move, the best possible choice is to merge the two workers with the highest number of tasks, so moving the maximum to the 2nd maximum or viceversa. This can be done by simply looping 2 times on the array, and returning the sum of the two highest values.

Subtasks 2 and 3 (25+25 points)

For subtasks 2 and 3, we extend the reasoning of the previous point to $K > 1$. It is again intuitive that we simply need to stack the $K + 1$ maximums in a single worker, and for these subtasks it is sufficient to do it by looping $K + 1$ times on the list of workers finding everytime the maximum and removing it from the list. This has complexity $O((K + 1)N)$ and, if implemented carefully, can work for both subtasks.

Subtask 4 (25 points)

As a last improvement, we can avoid looping on the list of workers $K+1$ times by simply sorting the list and summing up the first $K+1$ elements. It is sufficient to do it in $O(N \log N)$ with a whatever standard sorting algorithm, but notice that this can be optimized to $O(N)$ with, for example, a counting sort.

1.4 Source Code

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int N, K;
    cin >> N >> K;

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

    int solution = 0;

    sort(V.begin(), V.end());
    reverse(V.begin(), V.end());
    for(int i=0; i<K+1; i++) solution += V[i];

    cout << solution << endl;

    return 0;
}
```

Python

```python
#!/bin/env python3

import sys

fin = sys.stdin
fout = sys.stdout

def main():

    (N, K) = map(int, fin.readline().strip().split())
    V = list(map(int, fin.readline().strip().split()))

    solution = sum(list(sorted(V))[-(K+1):])

    print(solution)

if __name__ == "__main__":
    main()
```

2 Problem "Evolution"

2.1 Problem Statement

From our last virus scan we have found that unfortunately our systems are infected with a last generation malware that expands itself in memory.
For the problem, our memory can be considered as a bidimensional grid of N rows and M columns, where each cell can be of three types:

- Empty cell (symbol `.`) that contains no data.
- Data cell (symbol `+`) that contains valid information.
- Malware cell (symbol `*`) that contains part of the malware.

Each cell on the grid, after someone run a software on the system (for simplicity called execution **round**), can update its value according to the number of non-empty cells in its neighborhood (in the 8 possible directions, or less if a border cell) and in particular:

- An empty cell that has more than 4 non-empty cells becomes a data cell.
- A data cell that has more than 4 non-empty cells becomes a malware cell.
- A data cell that has less than 4 non-empty cells becomes an empty cell.
- A malware cell that has more than 4 non-empty cells becomes a data cell.
- A malware cell that has less than 4 non-empty cells becomes an empty cell.

Write a program that given the initial configuration of a memory finds its final configuration after k rounds.

2.2 Problem Details

Input

Your program must read the input data from the standard input.
The first line of the input contains three space-separated integers **N**, **M** and **K**, respectively: the number of rows of the memory, the number of columns of the memory and the number of rounds to simulate.
The following **N** lines will contain a string of **M** characters representing the initial configuration of the memory.

Output

Your program must write the output data into the standard output.
The output must contain **N** lines, each of which will contain a string of M characters representing the final configuration of the memory after **K** round.

Scoring

For each of the test cases the program will be tested, the following constraints are met:

- $K = 1$ and $N, M \leq 10$ for at least 25% of all the test cases.
- $K \leq 10$ and $N, M \leq 50$ for at least 50% of all the test cases.
- $K \leq 20$ and $N, M \leq 100$ for all the test cases.

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 (25 points)

The problem is a simulation task, in the style of Conway's "Game of Life" and other cellular automata problems. In order to solve it, it is sufficient to follow carefully the rules in the problem statement. The complexity of the simulation is $O(MNK)$. For this first subtask we have $K = 1$. We keep a $N \times M$ matrix G with the current state of the game and a new matrix G' that initially has all the cells marked as empty. For each cell in G, we keep track of the number of empty and non-empty cells among its 8 neighbors (looping on them) and update G' accordingly.

Subtask 2 (25 points)

In this subtask, we have to tackle the case $K > 1$. To do it, we wrap the code for the previous subtask in another loop of length K. At the end of each iteration, we change G to the final state of G' and re-initialize G' with all empty entries. Notice that, especially for Python users, you must be careful in doing the copy step between G and G' by value and not by reference.

Subtask 3 (50 points)

This logic for this subtask is the exact same as the previous one, just with larger inputs. If your implementation is correct and optimized, you should have no problem in getting also these points.

2.4 Source Code

C++

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <array>

using namespace std;

int main() {
    int N, M, K;
    cin >> N >> M >> K;

    vector<array<int,2>> dir = {
        {-1, -1},
        {-1,  0},
        {-1, +1},

        { 0, -1},
        { 0, +1},

        {+1, -1},
        {+1,  0},
        {+1, +1},
    };

    vector<string> G(N, string(M, '.'));

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

    for(int k=0; k<K; k++) {
        vector<string> G2(N, string(M, '.'));

        for(int i=0; i<N; i++) {
            for(int j=0; j<M; j++) {
                int count = 0;

```

```
                for(array<int, 2> d : dir) {
                    int di = i+d[0];
                    int dj = j+d[1];
                    if(di < 0 || di >= N) continue;
                    if(dj < 0 || dj >= M) continue;
                    count += 1 - (G[di][dj] == '.');
                }

                G2[i][j] = G[i][j];

                if(G[i][j] == '.') {
                    if(count > 4) G2[i][j] = '+';
                }
                else if(G[i][j] == '+') {
                    if(count < 4) G2[i][j] = '.';
                    if(count > 4) G2[i][j] = '*';
                }
                else {
                    if(count < 4) G2[i][j] = '.';
                    if(count > 4) G2[i][j] = '+';
                }
            }
        }

        G = G2;
    }

    for(int i=0; i<N; i++)
        cout << G[i] << endl;

    return 0;
}

```

Python

```
#!/bin/env python3

import sys

def solve(fin, fout):
    (N, M, K) = map(int, fin.readline().strip().split())

    V = []
    for i in range(N):
        V.append(list(fin.readline().strip()))

    for _ in range(K):
        V2 = [["." for _ in range(M)] for _ in range(N)]

        for i in range(N):
            for j in range(M):
                f = {".": 0, "+": 0, "*": 0}

                for (di, dj) in [(-1, -1), (-1, 0), (-1, 1), (1, -1), (1, 0), (1, 1), (0, 1), (0, -1)]:
                    if 0 <= (i+di) < N and 0 <= (j+dj) < M:
                        f[V[i+di][j+dj]] += 1

                empty = f["."]
```

```
                full = f["+"] + f["*"]
                co = V[i][j]
                cn = co

                if co == '.':
                    if full > 4:
                        cn = '+'
                elif co == '+':
                    if full < 4:
                        cn = '.'
                    elif full > 4:
                        cn = '*'
                else:
                    if full < 4:
                        cn = '.'
                    elif full > 4:
                        cn = '+'

                V2[i][j] = cn

        V = [[V2[i][j] for j in range(M)] for i in range(N)]

    for v in V:
        print("".join(v))

if __name__ == "__main__":
    solve(sys.stdin, sys.stdout)

```

3 Problem "Password"

3.1 Problem Statement

For our brand new website we have developed a new custom way to store our hashed password.
To calculate the hash of a password P we perform the following operations:

- Generate two random strings of random length (may be zero) that will be used as salts, respectively called $salt1$ and $salt2$.
- Generate a random permutation $P2$ of the original password P.

The hash value H of the password P is thus $H = salt1 + P2 + salt2$ (where $+$ is the string concatenation operator).
But now we miss the function that given an hash verify the password!
Write a program that, given a set of T pairs of password P and hash H, returns if each password is verified or not.

3.2 Problem Details

Input

Your program must read the input data from the standard input.
The first line of the input contains an integer **T** representing the number of passwords to verify.
For each of the verification requested, the input contains two lines with the respective password P and hash H to verify.

Output

Your program must write the output data into the standard output.
The output must contain **T** lines representing the result of each password verification. Each line must contain the number **1** if the password is verified, **0** otherwise.

Scoring

For each of the test cases the program will be tested, the following constraints are met:

- $T \leq 5$ and $|P| \leq |H| \leq 10$ for at least 25% of all the test cases.
- $T \leq 10$ and $|P| \leq |H| \leq 100$ for at least 50% of all the test cases.
- $T \leq 15$ and $|P| \leq |H| \leq 1\,000$ for at least 75% of all the test cases.
- $T \leq 20$ and $|P| \leq |H| \leq 10\,000$ for all the test cases.

3.3 Solution

This is the hardest problem in the problem set. It requires a mix of clever ideas and implementation skills to be fully solved.

Subtask 1 (25 points)

In order to solve this subtask, we start by sorting P in increasing order. Let's call this sorted value P_o. We then loop on H with two indices, i and j (with j starting at $i + 1$), and for each iteration we sort the slice starting from index i and ending in index j of H and check if it is equal to P_o. If this is the case, we return true, otherwise, if we never found a match for all values of i and j, we return false. This approach has time complexity $O(n^3 \log n)$, where n is the length of H.

Subtask 2 (25 points)

For this subtask, a slight improvement of the previous method is sufficient. Instead of looping on H with 2 indices, we can use only one and take $j = i + \text{len}(P)$, since all the other cases can not lead to a solution because the length of P_o and the one of the slice will not match. With this approach, we reduce the complexity by a factor n, leading to $O(n^2 \log n)$.

Subtask 3 (25 points)

For this subtask, we change completely our approach. Although a good implementation of the previous approach could possibly also pass this subtask, we describe here a $O(n^2)$ solution that will be useful for the last subtask. First of all, we initialize a vector (or a map) V_P, where we will store, for each letter, the number of appearances of that letter in P. This can be done in $O(n)$ by simply initializing V_P to 0 and looping through P, incrementing the value of the corresponding letter each time. Then we loop again over H with two indices i, j (again with j starting from $i+1$) and do the same for H: for each loop over i we initialize V_H to 0; then we loop on j starting from position i to the end of H incrementing the value of the corresponding letters in V_H. If *at any point* V_P is equal to V_H we return true, otherwise we return false at the end of the process.

Subtask 4 (25 points)

For this last subtask, we want to shrink the previous solution to $O(n)$. This can be done by optimizing the creation of V_H: instead of looping on it with 2 indices we can have a sliding window of fixed length (the length of P) and use a single loop. This removes a factor n from the complexity and is enough to solve all the subtasks.

3.4 Source Code

C++

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

using namespace std;

void solve()
{
    vector<int> cntp(26), cnt(26);
    vector<bool> eq(26);
    int sum = 0;

    string p, h;
    cin >> p >> h;

    auto chg = [&](int c, int val) {
        sum -= eq[c];
        cnt[c] += val;
        eq[c] = (cntp[c] == cnt[c]);
        sum += eq[c];
    };

    for (int i = 0; i < p.size(); i++)
        ++cntp[p[i] - 'a'];

    for (int i = 0; i < 26; i++)
    {
        eq[i] = (cnt[i] == cntp[i]);
        sum += eq[i];
    }

    int m = p.size();
    int n = h.size();

    for (int i = 0; i < n; i++)
    {
        chg(h[i] - 'a', 1);
        if (i >= m)
            chg(h[i - m] - 'a', -1);
        if (sum == 26)
        {
            cout << 1 << endl;
```

```
            return;
        }
    }
    cout << 0 << endl;
}

int main()
{
    int tc;
    cin >> tc;
    for (int i = 0; i < tc; i++)
        solve();
    return 0;
}
```

Python

```
#!/bin/env python3

import sys

def check(P, H):
    cntp = [0 for _ in range(26)]
    cnt = [0 for _ in range(26)]
    eq = [False for _ in range(26)]

    s = 0

    def chg(c, val):
        s = -1 if eq[c] else 0
        cnt[c] += val
        eq[c] = (cntp[c] == cnt[c])
        s += (1 if eq[c] else 0)
        return s

    for p in P:
        cntp[ord(p) - ord('a')] += 1

    for i in range(26):
        eq[i] = (cnt[i] == cntp[i])
        s += eq[i]

    m = len(P)
    n = len(H)

    for i in range(n):
        s += chg(ord(H[i]) - ord('a'), 1)
        if i >= m:
            s += chg(ord(H[i-m]) - ord('a'), -1)
        if s == 26:
            return True

    return False

def solve(fin, fout):

    T = int(fin.readline().strip())

    for _ in range(T):
```

```
43          P = fin.readline().strip()
44          H = fin.readline().strip()
45          print("1" if check(P, H) else "0")
46
47  if __name__ == "__main__":
48      solve(sys.stdin, sys.stdout)
49
```