VI MaratonUSP Freshman Contest
Editorial

MaratonUSP

A - Kobus hates sweepstakes

Before presenting the solution, I want to comment on a few things.

The first is that it is possible, in C ++ and other languages, to compare two strings natively, which makes implementation very easy.

Another detail is that the new permutation given as an answer can be used to map words to a new alphabet. For example, the *okbusacdefghijl..z* alphabet maps *kobus* to *bacde* and *kkbus* to *bbcde*. Graphically, we have to:

o k	*b u*	*s a*	*c d*	*e f*	*g h*	*i j*	*l m*	*n p*	*q r*	*t v*	*w x*	*y z*
↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓
a b	*c d*	*e f*	*g h*	*i j*	*k l*	*m n*	*o p*	*q r*	*s t*	*u v*	*w x*	*y z*

and then we can compare the mapped words as if they were in the *normal alphabet*.

Finally, we have a function called *next_permutation* in C ++, which, when used in conjunction with a *do...while* loop can generate all permutations of a sequence of numbers, here is the reference of this method.

The first idea that may seem to solve this problem is to make the new alphabet to be *kobusacdefghijlmnpqrtvwxyz*. With this, you will transform *kobus* into *abcde*, and it just won't be first in the list if there are names like *kob*, *kobu* (*kobus* prefixes) or *kkoo*. However, this idea does not work, just look at the case where one of the names on the list is *kkbus*. In this example, our program would say that there is no solution, even with one of the possible solutions being *okbusacdefghijlmnpqrtvwxyz*.

Now, finally, let's go to the solution. The idea is similar to the initial one, however, now we are going to use all possible permutations of the word *kobus* as the beginning of the new alphabet and simply see if any makes Kobus come first.

The code below implements this idea.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {
    int n; cin >> n;

    vector<string> vs(n);
    for (int i = 0; i < n; i++) cin >> vs[i];

    string kobus = "kobus", alpha = "acdefghijlmnpqrtvwxyz";
    vector<int> permutation({0,1,2,3,4});

    // this do...while will generate every permutation of {0,1,2,3,4}
    do {

        int ok = 1;
        string perm_kob = ""; // will be a permutation of "kobus"
        for (auto x: permutation) perm_kob += kobus[x];

        string new_alphabet = perm_kob + alpha;
        map<char, char> old_to_new; // maps old alphabet to new one
        for (int i = 0; i < 26; i++)
```

```
            old_to_new[new_alphabet[i]] = 'a'+i; // 'a'+0=='a', 'a'+1=='b'

        string new_kob = kobus;
        for (char &c: new_kob) c = old_to_new[c];

        for (auto x: vs) {
            string s = x;
            for (char &c: s) // remap each name to new alphabet
                c = old_to_new[c];
            if (new_kob >= s) ok = 0; // kobus not in first place
        }

        if (ok) {
            cout << new_alphabet << endl;
            goto fim;
        };

    } while (next_permutation(permutation.begin(), permutation.end()));

    cout << "sem chance" << endl;
    fim:;
}
```

B - Guidi wants to be stronger

This problem is a direct application of the **Longest Common Subsequence** problem, which uses the dynamic programming technique. An excellent article on this problem can be read here.

The code below implements this idea.

```
#include "bits/stdc++.h"
using namespace std;

int lcs[1009][1009];

int main() {
    string a, b; cin >> a >> b;

    for (int i = 0; i <= a.size(); i++) {
        for (int j = 0; j <= b.size(); j++) {
            if (i == 0 || j == 0)
                lcs[i][j] = 0;
            else if (a[i-1] == b[j-1])
                lcs[i][j] = 1 + lcs[i-1][j-1];
            else
                lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1]);
        }
    }

    cout << lcs[a.size()][b.size()] << endl;
}
```

C - Harada and the lucky numbers

First of all, we have to note that the amount that Q can take is very small, at most 7. In addition, we can create an auxiliary vector that tells us the frequency of each value of the cards that Harada won.

With that, the idea is to check all possible subsets of lucky numbers, to check if it is possible to assemble the numbers of that subset using the cards Harada won and keep stored which is the largest subset that satisfies the condition seen so far.

The most widely used technique for generating all subsets uses BitMask. If your set has N elements, the idea is to iterate over all the values in the range $[0, 2^N - 1]$ and use the binary representation of the number to decide which set (I get the i-th only element if the i-th bit of the number is turned on), read more about this technique here.

Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {
    int q, n; cin >> q >> n;

    vector<int> v_q(q);
    for (int i = 0; i < q; i++) cin >> v_q[i];

    // frequency['0']==frequency if card with value 0
    map<char, int> frequency;
    for (int i = 0; i < n; i++) {
        int n_i; cin >> n_i;
        frequency[n_i+'0']++;
    }

    int ans = 0;
    // the mask makes it possible to generate all possible subsets
    for (int mask = 0; mask < (1<<q); mask++) {

        bool ok = 1;
        map<char, int> f = frequency;
        for (int i = 0; i < q; i++) {
            if ((mask&(1<<i)) == 0) continue; // i-th number not selected
            for (auto c: to_string(v_q[i])) {
                f[c]--;
                if (f[c] < 0) ok = 0;
            }
        }

        // __builtin_popcount tell us the number of 1 bits in the binary
        // representation of the number
        if (ok && __builtin_popcount(mask) > __builtin_popcount(ans))
            ans = mask;
    }

    cout << __builtin_popcount(ans) << endl;
    for (int i = 0; i < q; i++)
        if (ans&(1<<i)) cout << v_q[i] << ' ';
```

```
40
41      cout << endl;
42  }
```

D - Corona Mashup

The information that everyone who can participate on the x day can also participate on the days after x gives us an intuition that we should use ordering.

The easiest way to implement this problem is to use a Map, which we will call *day_count*. The idea is that $day\_count[x]$ tells us how many people can participate from x (non-cumulative value). Given that Map keeps the keys sorted in its internal structure, we can iterate over this structure and keep a variable that counts how many people can participate in the contest until the current day, always adding the respective value of the iteration key. Every time this number is divisible by 3, we update the answer.

Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {

    long long int n; cin >> n;

    // day_count[x] == 'participantes on day x'
    map<long long int, long long int> day_count;
    for (int i = 0; i < n; i++) {
        long long int x; cin >> x;
        day_count[x]++;
    }

    long long int cnt = 0, ans = -1;
    // here we iterate the elements in the map, this will give days
    // in increasing order. Also, each element in the map is a pair
    // containin a key (in the case, a day) and a value (in this
    // case) the number of participants that can participate in
    // on the given day
    for (auto x: day_count) {
        cnt += x.second;
        if (cnt%3 == 0) ans = x.first;
    }

    cout << ans << endl;
}
```

E - Learning new languages

The idea of this problem is to keep some structure that tells us which words we know, we can use both a Set and Map for that. However, when we learn a new word, how do we check that it is not possible to learn another word, given that our textitvocabulary has increased?

Since m only goes up to 100 you can do the following: for each word in the dictionary, keep a list of what words you need to know to learn it. Go through the list of words in the dictionary and see if you were able to learn any. To do this, go through the list of the respective word and check, using your Set/Map, if you know all the words on that list. If you have managed to learn a new word, go through the dictionary word list again, repeating this process until you are unable to learn a word. After that, just print out the total number of words you know.

Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {

    int n; cin >> n;

    map<string, bool> known_words;
    for (int i = 0; i < n; i++) {
        string s; cin >> s;
        known_words[s] = 1;
    }

    int m; cin >> m;
    // learn_list[s] == "words you need to know in order to learn s"
    map<string, vector<string>> learn_list;

    while (m--) {
        string s; cin >> s;
        int k; cin >> k;
        while (k--) {
            string aux; cin >> aux;
            learn_list[s].push_back(aux);
        }
    }

    bool ok = 1;
    while (ok) {
        ok = 0;
        for (auto x: learn_list) {
            if (known_words[x.first]) // already know this word
                continue;

            int cnt = 0;
            for (auto y: x.second) // add 1 if I know the necessary word
                cnt += known_words[y];

            if (cnt == x.second.size()) { // know all the necessary words
```

```
                known_words[x.first] = 1;
                ok = 1;
            }
        }
    }

    int ans = 0;
    for (auto x: known_words)
        ans += x.second; // add 1 for each known word

    cout << ans << endl;
}
```

F - Confusing Morete

To solve this problem we will use an accumulated sum vector, accumulating the values on the cards. If at any time the accumulated value was negative, Morete made a mistake. Otherwise, we will create a list of suspicious card indexes and iterate from the end to the beginning of the cards, that is, from right to left.

If at any time our accumulated sum vector becomes less than 0, we clear the elements of our list of suspicious cards, because no card to the right of that position will be suspicious, since removing it will still make us have a point with negative accumulated value.

Otherwise, we check if the value of the card in the current position is less than or equal to the minimum value seen so far (minimum value between cards in the current position until the end). If so, withdrawing this card means that the accumulated sum will never be negative, so this would be a suspicious card.

At the end, just check if the list of suspicious card indexes is empty. If so, Morete had a negative balance. Otherwise, we can print the indexes of the suspicious letters

Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

long long int r[112345], accumulated_sum[112345];

int main() {

    long long int n; cin >> n;
    for (int i = 0; i < n; i++)
        cin >> r[i];

    bool never_negative = 1;
    for (int i = 0; i < n; i++) {
        if (i > 0) accumulated_sum[i] = accumulated_sum[i-1];
        accumulated_sum[i] += r[i];

        if (accumulated_sum[i] < 0) never_negative = 0;
    }

    if (never_negative) {
        cout << "morete chapou: errou conta!" << endl;
        return 0;
    }

    vector<int> suspects; // kinda sus
    long long int mn = accumulated_sum[n-1];
    for (int i = n-1; i >= 0; i--) {

        // there cannot be any suspect card after this point
        if (accumulated_sum[i] < 0)
            suspects.clear();

        mn = min(mn, accumulated_sum[i]);
        if (mn >= r[i]) // remove this card makes always non-negative
```

```
35              suspects.push_back(i);
36      }
37
38      if (suspects.empty()) {
39          cout<<"morete chapou: ficou com saldo negativo!"<<endl;
40          return 0;
41      }
42
43      cout << suspects.size() << endl;
44      for (auto x: suspects) cout << x+1 << ' ';
45      cout << endl;
46  }
```

G - The blue dot game

This problem is based on the Green Eyes Puzzle. The answer is the minimum value between n and the number of students with blue points + 1.

Follow the implementation.

```
#include "bits/stdc++.h"
using namespace std;

int main() {

    int amt = 0, n; cin >> n;

    for (int i = 0; i < n; i++) {
        int x; cin >> x;
        amt += x; // add 1 if blue
    }

    int ans = min(n, amt+1);
    cout << ans << endl;
}
```

H - The comedian Nathan

The main idea of this problem is to use the Line Sweep technique. Basically, by putting all the N pairs into a vector and sorting it out, we can extract the answer very easily.

We can imagine that each person will form a segment on the timeline, which starts when he enters and ends when he leaves. In addition, due to the gossip character of the students, we can think that when two segments intersect, they become one. This is a generalization that helps people find the answer.

With that, we can calculate the union of all segments. For example, in the test case of the sheet, we will have the ordered vector $[[2,7],[6,9],[8,13],[21,25]]$ and its union is defined by the pairs $[2,13]$ and $[21.25]$.

If a pair of the union is defined by $[X,Y]$, we can calculate the number of jokes needed by iterating over the interval $[X,Y)$ and adding 1 to a counter every time the value of the iteration is divisible for 5. By doing this for all the pairs resulting in the union and getting the most jokes needed we can get the answer.

Homework: why can we simply iterate through the segments without running out of time? Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {

    int n; cin >> n;

    vector<pair<int,int>> v;
    for (int i = 0; i < n; i++) {
        int e, s; cin >> e >> s;
        v.push_back({e, 1}); // entering the bus will come after
        v.push_back({s, -1}); // because of the sorting
    }

    sort(v.begin(), v.end());
    int earlier_time = -1, passager_count = 0, ans = 0;

    for (auto x: v) {

        int cur_time = x.first, cur_operation = x.second;

        if (cur_operation == -1) {
            passager_count--;
            if (passager_count == 0) { // last passager to departure
                int jokes = 0;
                for (int t = earlier_time; t < cur_time; t++)
                    jokes += (t%5 == 0); // add 1 if time is divisible by 5
                ans = max(ans, jokes);
            }
        }

        if (cur_operation == 1) {
            if (passager_count == 0)
                earlier_time = cur_time;
```

```
35                 passager_count++;
36             }
37         }
38 
39         cout << ans << endl;
40 }
```

I - Competitive Mario Kart

To solve this problem you can create an auxiliary vector in which the first position is the number of points earned by placing first, the second position is the number of points earned by being in second place, and so on. The optimal answer is always given when we try to reach the best places the greatest number of times.

Follow the implementation.

```cpp
#include "bits/stdc++.h"
using namespace std;

int main() {

    int n; cin >> n;

    vector<int> ans, points({15, 12, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1});

    while (n > 0) {
        for (int i = 0; i < points.size(); i++) {
            if (n >= points[i]) {
                ans.push_back(i+1);
                n -= points[i];
                break;
            }
        }
    }

    cout << ans.size() << endl;
    for (auto x: ans) cout << x << " ";
    cout << endl;
}
```

J - Raphael singer

With the first pair *ano* and *titulo* we can estimate your year of birth. Now just check if the year of birth is going to be the same for all the next albums.

Follow the implementation.

```
#include "bits/stdc++.h"
using namespace std;

int main() {

    int birth = -1, ok = 1, n;
    cin >> n;

    for (int i = 0; i < n; i++) {

        int a, t;
        cin >> a >> t;

        if (birth == -1) {
            birth = a-t;
            continue;
        }

        if (a-t != birth) ok = 0;
    }

    if (!ok) cout << "mentiu a idade" << endl;
    else     cout << "idades corretas" << endl;
}
```