Task KARTE	Author: Mislav Balunović

In order to determine how many cards of the same suits are missing, it is enough to read every third character from the input and add how many cards from each suit we saw from the input. In the end, we subtract the calculated number from 13 for each suit and output the result if we didn't find an error.

To check whether there is an error, it is enough to use two nested for loops to check if positions i, j exists such that the substrings of length 3 that begin at positions i, j equal.

Alternatively, it is possible to use a two-dimensional array to store if a card of a certain suit had already been found while we input the cards from the string. For implementation details, consult the official solution.

Necessary skills: arrays
Category: ad-hoc

Task AKCIJA	Author: Antonio Jurić

Let us first notice that it doesn't make sense to sort the books in groups that contain less than 3 books because the discount is only valid for a group consisting of 3 books (to refresh our memory, the discount said that we'd get the cheapest of 3 books for free). This means that we need to sort **all** the books into **groups of 3**, so that there's only **one** group that has less than 3 books in the end (because the number of books that the customer decided to purchase is not a multiple of the number 3).

Let us now find the **two most expensive** books from the set of all books (their prices can be equal to each other). Because there is no other book more expensive than them, we can't create a group where we'd get a discount for those two books (because they are the most expensive ones), but we can then take the **third** most expensive book and put it in the group with the two most expensive ones. This way we'll get the discount for the third most expensive book. Now we have decreased the initial set of books by three.

By repeating this procedure while we still have books in the set and by ensuring that we get the discount for the most expensive book (the third most expensive book out of all the remaining ones) in every grouping, we make sure that the final price is going to be the minimal possible. If the number of books is not a multiple of the number 3, that means that we'll have 1 or 2 books in the last grouping and we'll have to pay for both, but that doesn't undermine the described algorithm.

Necessary skills: analysis
Category: ad-hoc

Task BALONI	**Author: Dominik Gleich**

There are multiple strategies to pop the balloons in this task. We will mention only the one implemented in the official solution.

The optimal strategy is the following: pick the first balloon from the left that isn't still popped and shoot an arrow in its height. Let's try and prove the validity of this algorithm.

The leftmost balloon must be hit by an arrow in a way that at least one arrow begins at its spot. The only question is whether it is sometimes better to first pop a higher balloon to the right and only then this one.
Let's notice two cases:
1) In the case when the set of balloons that would be popped by shooting an arrow into that higher balloon and the first balloon don't intersect, it is clearly not important which one of the two we pop first.
2) In the case when these two sets intersect, let's notice that this intersection is a suffix of these two sets. In other words, if an intersection exists, it is made out of several rightmost elements, which means that is also irrelevant which one we pop first.

The described algorithm can be implemented using the STL structure called set. The question which the set is going to answer is "What is the first balloon that hasn't been popped yet, is in the height **H-1** and is located to the right of the current position **P**?"
We need to have **H** sets, one per height from 1 to **H**. The **H**$^{\mathbf{th}}$ set will store the positions where the balloons at height **H** appear at.
With simple queries to the set and popping the elements from the set, it is still possible to maintain the structure that answers the required question.
The time complexity of this algorithm is O(**N lg N**), the memory complexity O(**N**).
The solution of complexity O(**N**) is also possible for this task and we leave this as a practice to the reader. (Hint: try to shoot all the arrows at the same time.)

Necessary skills: greedy algorithms

category: ad-hoc

Task TOPOVI	**Author: Mislav Balunović**

Let $\mathbf{R_i}$ denote the total XOR of all rooks located in row **i**. Analogously, we define $\mathbf{C_i}$ as the total XOR of all rooks located in the column **i**. Let us notice that the total number of attacked fields is equal to the number of pairs (**i**, **j**) such that $\mathbf{R_i} \neq \mathbf{C_j}$ (this is a direct consequence of the fact that the field (**i**, **j**) is attacked if and only if the total XOR in row **i** is different that the total XOR in column **j**).

In the beginning, we can calculate for each **k** how many rows **i** there are such that $\mathbf{R_i} = \mathbf{k}$, and how many columns **j** there are such that $\mathbf{C_j} = \mathbf{k}$. It is easy to calculate the number of attacked fields with this information.

When a move occurs, we need to be able to efficiently calculate the change in the number of attacked fields. When a rook moves from field (**r**, **c**), we calculate the number of attacked fields in row **r** or column **c** and subtract it from the total number of attacked fields. When a rook moves to field (**r**, **c**), we calculate the number of attacked fields in row **r** or column **c** and add it to the total number of attacked fields.

If we use a binary tree (i.e. map in C++) for storing the data, the total time complexity of this algorithm is O(**Q** lg **N**), and the memory complexity O(**N**).

Necessary skills: binary tree, combinatorics
Category: ad-hoc

Task RELATIVNOST	**Author: Dominik Gleich**

Let us first solve the task for the case when there are no changes in requirements and we solve it under constant number of requested colored paintings $\mathbf{a_i}$ and black and white paintings $\mathbf{b_i}$. We need to determine how many ways there are to choose different configurations of selling the paintings with the requirement that at least **C** paintings are colored. The initial problem when there are no changes in requirements is solved using dynamic programming. Let $\mathbf{dp_{ij}}$ denote the number of ways to choose the initial **i** transactions so that exactly **j** requirements are satisfied with colored paintings. The relation used to calculate $\mathbf{dp_{ij}}$ is:

$$\mathbf{dp_{ij} = dp_{i-1\,j} * b_i + dp_{i-1\,j-1} * a_i}$$

In the end, the solution is the sum of all $\mathbf{dp_{n\ i}}$ for each **i** greater than or equal to **C**. The calculation can be simplified even more if we define $\mathbf{dp_{i\ K}}$ as the number of ways for not exactly **K** colored paintings, but at least **K** colored paintings. The solution that calculated this relation every time the requirements change has the complexity **O(N * C)** and was good enough for 30% of total points.

In order to get all the points, you needed to find a way to efficiently maintain the relation value after changes have been made. The inspiration comes from tournament tree. If you are not familiar with this structure, be sure to read the tutorial at http://wiki.xfer.hr/tournament/. Under the assumption that now you know how tournament tree works, we continue solving the task.

The solution will be maintained in a way that every node of the tournament tree is used to store the number of ways to choose the paintings in that subtree for each number of chosen colored paintings from 0 to **K**.

We are left with maintaining that relation after we change the requirements of one person. It is evident that it is fairly simple to calculate the change in result for each node in the tournament tree. The question is how to join two nodes.

Here we can notice that the number of ways of choosing the transaction over the entire subtree (consisting of the left - **L** and right child - **R**) with at least **i** purchases of colored paintings is $\mathbf{T_s = \sum L_i * R_{s-i}}$ for each **i <= s**.
The total time complexity is **O((N + Q) lg N * K^2)**.

Necessary skills: data structures, dynamic programming
Category: dynamic programming

Task UZASTOPNI	**Authors: Dominik Gleich, Mislav Balunović**

Let $\mathbf{S_x}$ denote the set of all the possible joke intervals that can be found at the party if we observe person **x** and all their direct or indirect subordinates (the subtree of person **x**).

Let us assume that for a person **x** we have calculated the values of sets **S** for each direct subordinate node to that person. Then we can calculate $\mathbf{S_x}$ using dynamic programming.

Let $\mathbf{Q_l}$ denote the set of all right sides of the interval of jokes that can be heard in the subtree of person **x** and whose left side of the interval is equal to **l**.
We iterate in the descending order over all the possible left sides **l** and for each **l** we iterate over all the possible right sides **r** so that there is a child of node **x** that has the interval [**l**, **r**] in its set and we calculate the following relation:

$$\mathbf{Q_l} = \mathbf{Q_l} \cup \mathbf{Q_{r+1}}$$

A special case when **l** = $\mathbf{v_x}$, then simply means:

$$\mathbf{Q_l} = \{\mathbf{v_x}\} \cup \mathbf{Q_{l+1}}$$

The time complexity of this algorithm is $\mathbf{O(NK^3)}$ where **K** is the number of different jokes, and **N** is the number of nodes.
The official solution uses a bitset data structure in order to accelerate calculating the union operation 32 times. For implementation details, consult the official solution.

Necessary skills: dfs, dynamic programming, bitset
Category: dynamic programming