Editorial

Tasks, test data and solutions were prepared by: Nikola Dmitrović, Karlo Franić, Gabrijel Jambrošić, Marin Kišić, Daniel Paleka, Ivan Paljak and Paula Vidas. Implementation examples are given in attached source code files.

Task: Emacs

Suggested by: Paula Vidas
Necessary skills: implementation

The number of rectangles in the image is equal to the number of upper-left corners of rectangles in the image. Cell (i, j) is an upper-left corner of some rectangle if that cell contains the character `'*'` while cells $(i-1, j)$ and $(i, j-1)$ (if they exist) contain the character `'.'`.

Alternatively, we can use BFS to find the number of connected components in four general directions which contain the character `'*'`.

Task: Političari

Suggested by: Ivan Paljak
Necessary skills: cycle detection

It was possible to score 50% (35) of total points on this task simply by implementing what was described in the task statement. In other words, you should have correctly implemented the rules by which politicians blame each other until we reach the K-th show. The time complexity of this solution is $\mathcal{O}(K)$, and the implementation details can be seen in `politicari_brute.cpp`.

To score all points on this task it was necessary to observe that the blaming process is cyclical. Since the next guest on a talk show solely depends on the previous guest and the person who blamed the previous guest, we can conclude that there are at most $\mathcal{O}(N^2)$ different shows (states). We consider two shows to be the same if they have the same guest that was blamed by the same politician in the previous show. Otherwise, the shows are considered different.

Since the total number of different shows is considerably less than the maximum episode in which we are interested in, we can simulate the process until we reach the show we have seen before (already visited state). At that moment, assuming that we keep track of some key items that have happened, we can use the power of math to calculate who will be the guest of the K-th show.

Let's assume we have realized that the i-th show will be the same as a (some prior) j-th show. In that case, we have just entered a cycle of length $(i-j)$ and can conclude that the guest which appeared in $(j + ((K-j)\%(i-j)))$-th show will also appear in the K-th show. Here, we use the $\%$ character to denote the modulo operator.

Time complexity of described solution is $\mathcal{O}(N^2)$.

Task: Matching

Suggested by: Paula Vidas and Daniel Paleka
Necessary skills: segment tree

If we connect the points that share one coordinate, the points will be divided into *cycles* and *paths*. With paths it is clear which points should be connected with line segments. If there is a path with an odd number of points, we immediately conclude that the answer is `"NE"`. For cycles we must determine whether we will draw all horizontal or all vertical lines (in the rest of the editorial we call this *orientation*).

There are at most $\frac{N}{4}$ cycles, so in a subtask where ($N \leq 40$) we can try all $2^{\text{broj ciklusa}}$ possible cycle orientations and check if there is one where line segments don't intersect.

The main idea is as follows: at the beginning the paths determine some line segments that must be drawn. These line segments determine the orientations of those cycles that they intersect, which determine the orientations of some other cycles, etc. If at some point during this process we draw a line segment that intersects some line segment that is already drawn or we conclude that both cycle orientations are invalid, then the answer is `"NE"`.

For a subtask where $N \leq 2000$, we can do that in $\mathcal{O}(N^2)$:

First, we find all paths and cycles and place all line segments from paths into a *FIFO queue* of the lines that need to be drawn. While the queue is not empty, we pop a line segment from it and check whether it intersects any of the previously drawn line segments (then the answer is `"NE"`). Then we traverse over not-yet-oriented cycles and orient those cycles which intersect with the line segment we are about to draw (orientation of the cycle needs to be different than the orientation of the line segment). In the end we might have some leftover cycles which we can orient in the same orientation (doesn't matter which one).

For the final subtask where $N \leq 10^5$, we need to check those line segment intersections more efficiently. We will use a segment tree to devise a data structure which supports the following operations:

- add a line segment $(x_1, y) - (x_2, y)$
- remove a previously added line segment $(x_1, y) - (x_2, y)$
- determine for some x and y_1 what is the smallest $y_2 \geq y_1$ such that there exists a line segment with y coordinate equal to y_2 which contains the point (x, y_2) (or determine that no such y_2 exists)

In each node of the segment tree we will store a set of all y coordinates whose x coordinates are in an interval for which that node is responsible. The first two operations boil down to standard addition/deletion of y to corresponding nodes. To answer the third operation, we will find such y_2 for each node responsible for x (e.g. by using `lower_bound` in `std::set`) and return minimal such value. The complexity of all three operations is $\mathcal{O}(\log^2 N)$. For implementation details, check the official solution.

Line segment which intersects the line segment $(x, y_1) - (x, y_2)$ exists if the answer to the third query for x and y_1 is less than or equal to y_2.

We will use two such data structures – one for horizontal and one for vertical line segments – in which we will store the line segments of not-yet-oriented cycles and two additional data structures to store already drawn line segments. We leave out the rest of the details as an exercise to the reader.

Task: Putovanje

Suggested by: Gabrijel Jambrošić i Marin Kišić
Necessary skills: lca, small-to-large

Note that we really care about the number of times we need to traverse each of the roads. Once we know that, it is pretty easy to decide whether we will buy multiple single-pass tickets or a single multi-pass ticket.

In the first subtask, it was enough to use an algorithm like BFS or DFS to find a path between X and $X+1$ while increasing the counter of traversals for each edge.

In the second subtask our tree is actually a chain. Let's think about that chain as an array. Let's also keep around another array called `dp`. Now, for every pair X and $X+1$ we can increase `dp[min(pos[X], pos[X+1])]` by 1 and decrease `dp[max(pos[X], pos[X+1])]` by 1 where `pos[x]` denotes the position of node x in our chain. After we have done this for all neighbouring pairs, we can go over the `dp` array and add its elements into some variable `cnt`. Note that in each moment of this traversal that variable holds the number of times we have traversed the corresponding edge in a chain.

To score all points we will slightly modify this algorithm to make it work on a tree. Let's root the tree in an arbitrary node. Let L be the lowest common ancestor of X and $(X+1)$. Let's now increase `dp[X]` by 1, increase `dp[X+1]` by 1 and decrease `dp[L]` by 2. Now we can simply find out the number of traversals of each road by calculating the sum of `dp` values in a subtree of that road.

This solution is implemented in `putovanje_lca.cpp`.

There is an alternative solution of the same time complexity which uses the so-called *small-to-large* trick. That solution is implemented in `putovanje_stl.cpp`. You can read about a very similar idea on this link.

Task: Nivelle

Suggested by: Daniel Paleka
Necessary skills: sliding window technique, two pointers

We can implement a solution of time complexity $\mathcal{O}(N^2)$ which for every substring calculates the number of different letters in it. If we use the so-called sliding window technique, we need to keep track how many times each letter appears and note each time when a new letter starts or stops appearing. For implementation details, check the attached (slower) source code.

Note that the numerator of the expression we want to minimize, i.e. the number of different characters in a substring, can either be 1, 2, ..., 26. Therefore, it is enough to fix its value in every possible way and determine the largest possible substring which has exactly that many different characters. Now we have 26 values to compare and pick the smallest one.

A simple implementation calculates for each starting character the longest substring which contains exactly K different characters. If we calculate for each character and each position the first next appearance of that character, for each starting character we can quickly check ≤ 26 strings which go to *the next new character*.