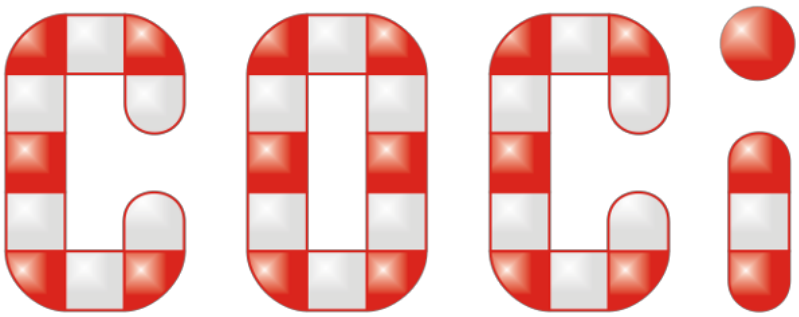

CROATIAN OPEN COMPETITION IN INFORMATICS

3rd round, November 29th, 2014

solutions

Task STROJOPIS	Author: Adrian Satja Kurdija

One of the possible solutions is to define eight arrays of characters (strings) where each of them consists of the characters corresponding to one finger. Using a for loop over the input string, in the inner loop we check whether which of the defined eight strings contains the current character and increment (by one) the counter of number of presses the corresponding finger made. Those counters are elements of an array of eight elements that we will output in the end.
For an alternative solution, consult the attached code *strojopis.py*.

Necessary skills: working with array of characters (string), nested for loop
Category: ad-hoc

Task DOM	Author: Marin Tomić, Antonio Jurić

Let us observe that the problem can be represented with a graph: **the nodes** are going to be TV channels and the pairs of favourite and hated channel of each pensioners are going to be **directed edges** from the hated to the favourite channel.

In case a channel is hated by more than one pensioner (and as such should have more than one edge coming out of it towards their favourite channels) it is sufficient to remember **only one** edge: the edge of the youngest pensioner because he is the one who gets up and changes the channel in this case.

Since the choice of changing the channels is now unambiguously defined, it is easy to come up with a solution: we traverse the constructed graph starting from the given channel **P** and count how many channels we changed so far (denoted by **K**) and which channels we've already viewed (already visited in our graph).

If we reach a channel we've visited **before**, it is obvious that the same channels are going to repeat (meaning we've entered a **cycle**) so the output is -1.

If we reach a channel that is **hated by nobody** (not a single edge is coming out of it towards another channel), the channels are going to stop changing so the total number of channel changes is **K** (movements in the graph), which is our solution.

Necessary skills: graph theory, cycles
Category: ad-hoc

Task SILUETA	**Author: Ivan Paljak**

Firstly, we will describe a naive approach. We could, for example, first draw the painted image. In other words, fill the skyscrapers' area with '#'. After filling that out, the image from the first example from the task would look like this:

```
........####
####....####
####..######
####..######
####..######
************
```

Now we are left with whiting out the interior of the skyscrapers. Notice that the character '#' becomes '.' only if none of the surrounding characters (in all 8 directions) is '.'. By implementing this algorithm, we get the wanted image.

The perimeter can be calculated from the given image by carefully counting the characters '#'. First, let us add to the perimeter all the characters '#'. Notice that some of the characters '#' participate in building two sides, whereas some of them don't participate at all. The perimeter should be increased by the number of characters participating in two sides and decreased by the number of characters participating in none. The characters being added to the perimeter are marked with red and the ones being subtracted are marked with blue:

```
........####
####....#..#
#..#..##..#
#..#..#....#
#..#..#....#
************
```

It is clear that the characters that are being added are in fact the upper corners of the skyscrapers and are of the shape:

```
##        ##
#.  ili   .#
```

whereas the characters being subtracted are the points of contact between two skyscrapers and are of the shape:

```
.#        #.
##  ili   ##
```

Solutions such as this, their the complexity being `O(sum_of_areas)`, were enough to get 50% of total points.

Let `maxh[i]` denote the maximum height of a skyscraper that stretches over the i^{th} column. The elements of this array can be calculated in the complexity `O(N*(max{Ri}-min{Li}))`.

Calculating the perimeter comes down to linearly iterating over the array `maxh` where we add to the final solution the absolute difference between the adjacent elements (two vertical sides) and one for each column where `maxh[i]` > 0 (horizontal sides).
The similar procedure is used to reconstruct the image.
This implementation was sufficient for all points.

Necessary skills: complexity analysis, implementation
Category: ad-hoc

Task HONI	**Author: Adrian Satja Kurdija**

Let the ordered pair (x, y) denote the contestant that won x points in the first round and y points in the second round.

Let's concentrate on a contestant (a, b). In order to find his best possible place, we need to count the contestants (x, y) that are surely better than (a, b) on the total ranking list: their number incremented by 1 is the required best place of contestant (a, b).

If x > a and y > b, contestant (x, y) is surely better than (a, b). Are there any other contestants that are better? No, there aren't! The rest of the contestants can be in front of (a, b) after the first two rounds for a maximum of 650 points (because they weren't better than him at least in one round), so if all of them win 0 points in the third round, (a, b) catches up if he wins 650 points. Therefore, it is sufficient to only count the contestants (x, y) such that x > a and y > b.

Let us now find the worst possible place for contestant (a, b). Let us assume that he wins 0 points in the third round. Which contestants (x, y) can be better than him on the total ranking list? Clearly, those aren't the ones with x < a and y < b because they also win 0 points. Let us assume that all other contestants (x, y) win 650 points in the third round. In at least one round, they had more or equal points as (a, b). If they had equal points as (a, b), and lost from (a, b) by 650 points in the remaining round, after three rounds they will have the equal sum of points as (a, b) so they are not better than him. In all other cases, they are better.

In a nutshell: in the worst possible scenario for (a, b), contestants (x, y) that are NOT better than him are the ones with x < a and y < b and those with x = a, y = 0, b = 650 or y = b, x = 0, a = 650. The number of such contestants subtracted from N - 1 gives us the number of contestants that are better than (a, b) so that number increased by 1 gives the worst possible position of contestant (a, b) after the first three rounds.

Therefore, the task comes down to counting pairs (x, y) such that x > a and y > b, as well as those such that x < a and y < b. If the number of pairs (x, y) is stored in a matrix at position [x][y], we need to be able to quickly find the sum of a rectangle in that matrix. We

do this by first dynamically calculating the sums of all rectangles starting at corner (0, 0) and quickly calculate the sum of any rectangle by their mutual subtractions.

Common mistakes of contestants on this task:

- some of them who used logarithmic structure (Fenwick tree) forgot that it starts from 1, not from 0
- a lot of them didn't notice that contestant (0, b) cannot overtake contestant (650, b) even though the latter one doesn't "majorize" him. In all other "non-majorizing" cases, overtaking is possible.

Necessary skills: mathematical problem analysis, preprocessing
Category: ad-hoc

Task STOGOVI	**Author: Ivan Katanić**

Let us imagine a tree of stacks such that each stack from the task corresponds to a path to a certain node in the tree from the root of the tree. Each node will have an integer **x** assigned to it, excluding the root, which will be assigned zero. Every stack label **v** will then have a tree node assigned to it, call it $\mathbf{f_v}$. Precisely the number of the node $\mathbf{f_v}$ is the number from the top of stack **v**, and climbing up to the root we would see the whole stack. All the while ignoring the zero at the root, of course.

Let us make sure that this is really possible. At the beginning of the game, we have one empty stack and make a tree with the node zero. While copying stack **v** and adding a new number we simply add a new node to the tree, assign a number to it and set $\mathbf{f_v}$ as its parent. While copying stack **v** and removing element from its top, we will output the number in node $\mathbf{f_v}$ and assign the parent of $\mathbf{f_v}$ to the new stack. It is evident that the tree-like structure is stable after each operation and that all operations are of constant complexity.

We are still left with efficiently answering the query of type c. For starters, let us notice that the numbers placed on stacks are always ascending. In other words, no two tree nodes will have the same number **x** written in them. So the question "how many of the same numbers are in stacks **v** and **w**" is equal to the question "how many of the same nodes, excluding the root, is on the path from the root to $\mathbf{f_v}$ and on the path from the root to $\mathbf{f_w}$". All such nodes will be in the first part of the path from the root to $\mathbf{f_v}$/$\mathbf{f_w}$ because once the paths part ways, they will never meet again. Therefore, it is sufficient to find the last such node and output how many node there are on the path from the root to itself.

Such node is also called the lowest common ancestor in a tree and there are various methods to find it. The most popular solution is "jumps of powers of 2 length" (for more information, consult this link: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor), of the complexity O(N log N) for preprocessing and O(log N) per query.

Additionally, we need to notice that in this task we don’t have a complete tree for which we can initially build a jumping table, but it isn’t a problem. When adding a new node (the only moment when the tree changes), we will calculate all its jumps in the complexity O(log N) because the jumps of its ancestors have already been calculated.

The total time and memory complexity of this solution is O(N log N).

Task KAMIONI	**Author: Mislav Bradač**

Let us imagine a simplified version of the task where we only have two trucks and one query. An event is defined as a pair (**id**, **t**) which denotes that the truck with index **id** is turning around at time **t** (in other words, **t** minutes after departure).

After calculating the events and sorting by time, we process one event at a time and keep track of the turnaround time and movement direction for each truck. This information is sufficient to calculate for each moment **t** which truck is located in the city with a smaller index. The encounter is detected when for times $\mathbf{t_i}$ and $\mathbf{t_{i+1}}$ (consecutive events times) the following holds:

- $\mathbf{a_i} < \mathbf{b_i}$ and $\mathbf{a_{i+1}} > \mathbf{b_{i+1}}$ or $\mathbf{a_i} > \mathbf{b_i}$ and $\mathbf{a_{i+1}} < \mathbf{b_{i+1}}$, $\mathbf{a_i}$ - the city in which the first truck is located at moment $\mathbf{t_i}$, $\mathbf{b_i}$ - the city in which the second truck is located at moment $\mathbf{t_i}$.

This claim follows from the fact that two lines can have at most one intersection or completely overlap (but they won't because of the remark from the task).

It was possible to get 50% of points if every query was individually solved in the aforementioned way. This is done by looking at events for the two trucks we wanted to know the number of encounters for in the current query.

When we solve the task for 50% of points, it becomes clear that, in order to fully solve the task, we will need to find a way to solve multiple queries at once. Let us observe the following algorithm:

- assign the query to just one truck from the pair
- calculate the events of all trucks and sort the events by time
- process events in order and while doing so keep track for every truck: the moment when it last turned around, the position it last turned around and its movement direction
- while processing event (**id**, **t**) of truck with index **id**, for each query assigned to that truck we ask ourselves whether the other truck from the pair is located to the left or to the right of it in time **t**; let us notice that the condition about encounters from the explanation of the 50% solution is still valid even though we are observing the relative position of trucks from the query while processing the event for only one truck from the pair (caution is needed only when processing the event after the aliens abduct a truck)

Let **S** be the sum of all $\mathbf{K_i}$. If we always assign the query to the truck with the smaller number of cities in its route, we reach a complexity of (**S** sqrt **S**). It is easy to prove the complexity if we split the queries to small and large (a query is large if both trucks from the pair have more than sqrt **S** cities in their route, otherwise it is small). We can have **M** small queries and for every small query we need to determine the relative position of trucks at most sqrt **S** times. We can have at most sqrt **S** large queries (the ones with more than sqrt **S** cities in their route) and for each of them the second trucks from the query can determine the relative position at most **S** times. Therefore, the complexity of processing

small queries is O(**M** sqrt **S**) and large O(**S** sqrt **S**) which gives us the total complexity of O(**S** sqrt **S**).

The task has an alternative solution of the same complexity that splits the trucks to those who have more than sqrt **S** cities on their route and calculates their number of encounters with each of the other trucks and to trucks that have less than sqrt **S** cities on their route and calculates only the queries that haven't been calculated while processing the large queries. More details on the alternative solution can be found in the attached file *alternativno_kamioni.cpp*.

Necessary skills: complexity analysis
Category: ad-hoc, sweep