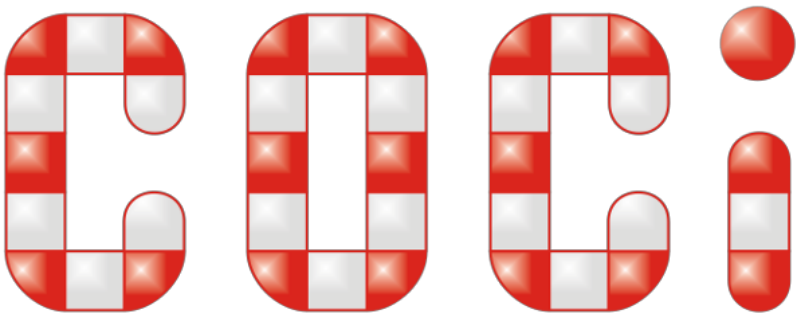

CROATIAN OPEN COMPETITION IN INFORMATICS

5th round, January 18th, 2015

solutions

Task FUNGHI	**Author:** Adrian Satja Kurdija

Initially, let's assume that we aren't given a "circular", but a regular array of **N** elements.
In a such array, how would we find the largest sum of four consecutive elements? We would choose the position of the quadruple using a for loop and check whether its sum is larger than the maximum sum so far. The position of the quadruple must be such that it stays within the array.

In a "circular" array, we must also take into account the quadruples that exceed the array's boundary and continue from the beginning of the array. In order to elegantly include such quadruples in the above algorithm, it is sufficient to double the given array. In other words, concatenate it to itself, creating an array of 2**N** elements (in our case 16). This array contains all the consecutive quadruples we need so the application of the described algorithm gives us the required maximum. For implementation details, consult the attached source code written in, of course, Python.

Necessary skills: finding the maximum element in an array
Category: ad-hoc

Task ZMIJA	**Author:** Ivan Paljak

Let's denote the lower left corner of the screen (the snake's initial position) with the coordinate **(0,0)** and the upper right corner with the coordinate **(R - 1, S - 1)**. Let **h** be the largest row number that contains at least one apple. It is evident that the number of presses of button B is equal to **h**.

We are left with the problem of the number of presses of button A. Let's assume that the snake is located at row **r** and column **s**. Depending on the parity of number **r**, snake is facing to the left (odd **r**) or to the right (even **r**). Let's analyze the **necessary** steps the snake must do in the given situation.
If the snake is facing to the left and the row **r** contains an apple that is located at column number **smaller** than **s**, it is necessary to keep pressing the A button until we get to at least that apple. Additionally, if the row above (numbered **r + 1**) contains an apple that is located at column number **smaller** than **s**, it is necessary to keep pressing the A button until we get to at least that column (otherwise, we won't be able to eat that apple). Only then can we press button B and move upwards.

Analogously, if the snake is facing to the right, it is necessary to keep pressing the A button to position the snake to at least **the maximum** column number that contains an apple in rows **r** or **r+1**.

Is it possible to reduce the number of **necessary** steps in one of the following rows by exceeding the number of necessary steps in the current row? It is visible that we increase

(or don't change) the number of necessary steps in the row above when we exceed the number of necessary steps in the current row. Inductively, it is clear that by using necessary movements in each row we will obtain the path of optimal length.

Necessary skills: greedy algorithms, mathematical problem analysis
Category: ad-hoc

Task TRAKTOR	**Author:** Matej Gradiček

The goal of this task is to find the first moment when there are exactly **K** mushrooms in a row, column or line parallel to the diagonal.

The solution that was enough for 50 points represents the meadow in a matrix and, when adding a new mushroom, checks whether there are exactly **K** mushrooms in a row, column or on the lines where the new mushrooms is located.

The solution that was enough for all points remembers for each x and y coordinate how many mushrooms there are with that x or y coordinate. Let's observe the lines parallel to the diagonals of the meadow. There are two types of such lines. For the first type, the value of x-y is constant, whereas for the second type the value of x+y s constant and there are $2*10^5$-1 such lines for each diagonal. For each such line, we can remember how many mushrooms are located on it.

Now we are only left with finding the first moment when some of the counters gets to **K** when a new mushroom is growing.

Total complexity is O(**N**).

Necessary skills: mathematical problem analysis
Category: ad-hoc

Task ZGODAN	**Author:** Adrian Satja Kurdija

Given number **N**, let's find the first digit from the left that violates the alternating parity, in other words, that is of the same parity as the previous digit. We have two options:

A. Change the current digit and leave all the previous digits unchanged.
B. Change a digit previous to this one.

Option A consists of two suboptions: increasing and decreasing the digit. If the digit is 0, it can only be increased; if it's 9, it can only be decreased; otherwise it can be both increased or decreased. It is evident that it's not worth increasing or decreasing the digit by more than one.

After increasing or decreasing the digit, what to do with the following digits? If we have increased the digit, the resulting number is surely larger than **N** so, in order to get closer to it, the following digits should be as small as possible: 0101… or 1010..., depending on the parity of the increased digit. Analogously, if we have decreased the digit, the resulting number is surely smaller than **N** so, in order to get closer to it, the following digits should be as large as possible: 9898… or 8989…, depending on the parity of the decreased digit.

Of course, if we had the option of both increasing and decreasing the digit, it is necessary to find a number closer to **N** between the two resulting numbers. We can do this by subtracting and comparing the differences (Python has built-in support for arbitrarily large numbers).

The constraints in the task allowed to also consider option B. We can attempt to increase and decrease each digit (so that, of course, it remains of different parity than the previous digit), construct the rest of the number using the above mentioned principle and compare the obtained difference with the minimal so far, storing the best solution. However, this wasn't necessary because option B is actually never better than A. The proof is left as an exercise to the reader.

Necessary skills: mathematical problem analysis, handling bignums
Category: ad-hoc

Task JABUKE	**Author: Marin Tomić**

The simplest approach to solving this task, which was enough for 30% of points, is to check for all possible trees if they are candidates for the closest tree per each query. This algorithm has the complexity of **O(GRS)**.

In order to reduce complexity per query, we need to reduce the number of candidates for the closest tree. Let's solve a simpler problem. For a given location of the fall of an apple (**r**, **c**), determine the closest tree located at column **s**.

If we denote rows with $\mathbf{r}_1$ and $\mathbf{r}_2$ so that $\mathbf{r}_1$ is the first line above row **r** such that column **s** contains an apple, and $\mathbf{r}_2$ is the first line below row **r** such that column **s** contains an apple, it is clear that ($\mathbf{r}_1$ **s**) and ($\mathbf{r}_2$, s) are the only possible candidates for the closest tree in that column.

Therefore, in order to answer the initial question, we need to check 2 trees per column. If we had a quick way of determining exactly which two trees these are per column, the answer to each query could be found in complexity **O(S)**.

To quickly find all possible candidates per column, we can maintain two matrices `g[r][s]` and `d[r][s]`. Matrix `g[r][s]` stores the first tree above position `(r, s)`, and matrix

`d[r][s]` stores the first tree below position `(r, s)`, given the fact that both fields contain r if there is a tree located at field `(r, s)`.

To maintain the matrices described above, when inserting a new tree, we need to update values in the column where the tree grew, which results in **O(R)** operations per query.

The final complexity of this algorithm is **O(G(R+S))**, which was sufficient for all points.

There is an alternative solution that uses BFS.

Necessary skills: complexity calculation, preprocessing

Task: DIVLJAK	**Author: Dominik Gleich**

The solution sufficient for partial points was to check whether some of the barbarians' strings is a substring of the word Tarzan gives out and if there is, increment the counter of that word by 1, repeating this procedure each time Tarzan gives out another word. Regarding the solution for 100% of points, there are multiple solutions, both offline and online. Here we will describe an online solution.

To begin with, let's construct an Aho-Corasick tree, which is a prefix tree. Let's define p(X) as the string obtained by traversing from the root of the tree to node X in the prefix tree.

For every node X, the Aho-Corasick tree remembers the deepest node Y such that p(Y) is a suffix of p(X). Let's call such node fail(X).

Additionally, we need to remember for each node X, node Z, where node Z is the deepest node on which some of the barbarian's strings ends, and at the same time p(Z) is a suffix of p(X). Let's call such node back(X).

Now, let's construct an explicit tree from back(X) links. Therefore, for each node on the Aho-Corasick tree, make a new node in the new tree so that the parent of the new node, node X, is node back(X) in the new tree.

Let's call the new node of some node X in the old tree exp(X). Having the tree constructed this way, by traveling to the root of the new tree from a node exp(X) we walk through all suffixes of string p(X) that are at the same time words written on the tablets of the barbarians.

Now, let's see what operations need to be implemented.
During the update operation (in other words, when Tarzan gives a new word), it is necessary to update all strings that are in it as a substring. How do we find such strings?
Let P be the newly given word. Let pref(i) be the prefix of word P up to position i.

We will use a classic traversal of Aho-Corasick tree that, for each prefix of string P prex(i), finds the longest suffix of that prefix that exists in the Aho-Corasick tree.
Formally, for each pref(i), where i is the position of the prefix in string P, it finds the deepest node X such that p(X) is the suffix of pref(i). Let's call such X for a pref(i), up(i). If we walk towards the node in our newly generated tree from every node exp(up(i)), we will walk through all words that are substring of the word P and are written on the tablets of the barbarians.

How to find up(i) for each i? Let's assume that we have up(i) for an i, and try to calculate up(i) for i+1. Let's call the letter appearing at position i+1 in the string L. In case there is an edge for letter L on node up(i), then up(i+1) is simply the node we get to when moving for edge L from node up(i). In the case when such edge doesn't exist, we try to find such edge for fail(up(i)). Therefore, we try to find such edge for the next longest suffix that is the suffix of the current suffix pref(i). The second operation is repeated until we get back to the root of the tree or until we find an edge for letter L.

The construction of this structure is possible in linear time per number of characters O(L). Also, the construction of the new tree is possible in linear time.

Up to now, we currently need to be able to increase all nodes in the union of paths from the root of the new tree to the set of nodes up(i), for each i, by exactly 1.

Implementing this operation is possible by using heavy-light decomposition and logarithmic structure. Let's decompose the tree to a set of chains such that the number of chains changed on the payh from the root of the tree to node exp(X) is of the size O(lgL), where L is the number of characters in the tree. The remaining task is to increase a part of that chain by exactly 1. To not increase each string only once, but as many times as it appears in string P, we would simply increase the corresponding part of each chain during our traversal from exp(up(i)) to the root of the tree by 1. This increase can be very easily done by using logarithmic structure for each chain.

How will we increase if we want to increase each node only once?
We can notice that, on each chain, we will eventually increase only one of its prefixes by exactly 1 at the end of each update. Accordingly, we can, on each chain, remember up to what prefix had that chain been increased, and in the case that the current increase increases a few other nodes, we increase only that difference set between nodes. On a concrete example, let's say that so far in this update, for some chain the prefix of size 5 had been increased by 1. Now comes a new change on the chain that requires for us to increase the prefix of size 8. In this case we will not increase the prefix of size 8, but will only increase position [6, 8]. This is the way we handle the increase of each string by exactly 1. There are a few other ways in which we could handle this increase, but this one is implemented in the official solution.

The total complexity of this approach is O(L * lg^2L), where L is the number of total characters. The space complexity is O(L * alphabet size).

Figuring out the offline solution of the same complexity is left as an exercise to the reader. (Hint: suffix array, union find, bst.)

Necessary skills: Aho-Corasick, Heavy-Light decomposition and Fenwick tree or Suffix Array, Union find, Segment tree and Binary search tree.

Category: data structures