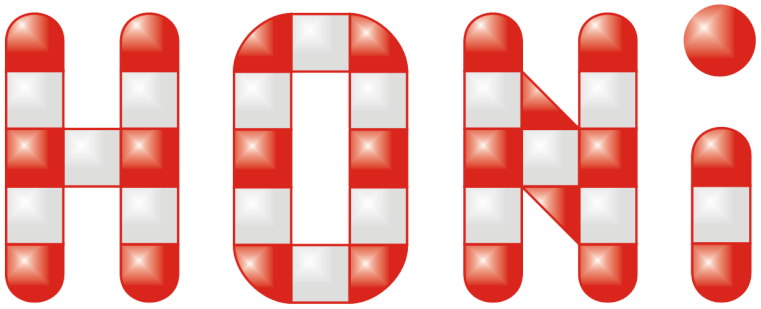

COCI 2017/2018

Round #1, October 14th, 2017

Solutions

Task Cezar	Author: Marin Kišić

We can maintain the array *deck*, where the i^{th} member denotes the number of cards left in the deck of value *i*. Initially, we set *deck[2], deck[3], deck[4], deck[5], deck[6], deck[7], deck[8], deck[9]* and *deck[11]* to 4, and *deck[10]* to 16. Now, every time we read a new card value, we decrease *deck[value_of_read_card]* by 1. Additionally, as we read the card values, we calculate their sum. In the end, let *X = 21 - sum*. The number of cards with the value greater than *X* is than equal to the sum *deck[x+1] + … + deck[11]*, whereas the number of cards with the value less than *X* is equal to *52 - n - number_of_cards_with_the_value_greater_than_X*.

Necessary skills: arrays

Category: ad hoc

Task Tetris	Author: Tonko Sabolčec, Marin Tomić

The most complex part of this task is to translate the images of Tetris figures into the program code with all of their possible rotations, so we can recognize them in a matrix. One possibility is to store one position of each figure and then programmatically generate the rotations and compared them to the content from the matrix. Another possibility is to manually write if-statements for each rotation. The first approach requires more programming skills, and the second requires more careful code-writing.

When we decide on the strategy to recognize the figures, we are left with counting the number of appearances of each one. You can find both approaches in the official solution.

Necessary skills: if, for, arrays

Category: implementation

Task Lozinke	Author: Tonko Sabolčec

We can solve this task in the following way: for each password X_i, we answer the query "How many other passwords exist that contain X_i as a substring?

If, for each password, we solve the queries by iterating over all the passwords and checking each possible substring, we can solve the task for 40% of total points. The complexity of this algorithm is $O(N^2 L^2)$, where *N* is the number of users, and *L* is the password length.

The solution can be sped up by calculating in advance, for each password, the different substrings it contains, and answer the queries by counting the number of times a substring appeared. The counting can be solved using a hash table, which is already implemented in most programming languages. The total complexity is O($N L^2$).

Necessary skills: data structure knowledge

Category: data structures, counting

Task Hokej	**Author: Vedran Kurdija**

Imagine the game as a 6 * *M* matrix, and the players as 1 * *I* blocks. We will fill up the matrix with players (blocks), without overlap. If a player covers the field (*X, Y*) in the matrix, it means they are playing position *X* in the Y^{th} minute of the game.

We fill out the matrix (game) by sorting the players by quality and iterating sequentially from the most quality player to the least quality player, and fill out the matrix row by row. If it so happens that a player's block doesn't fit entirely in the current row of the matrix, we break that block into two parts and continue with that player (block) in the following row.

This way, we made sure that we have the maximal possible sum of quality by the minute, since we used each player, starting from the most quality one, until their endurance limit (or less, in the case of the last player that might not fit entirely into the matrix).

We don't have to worry that a player will play multiple positions in the same minute, because it's endurance is at most *M*, and if they break from one row to another, they will never cover the same column (same minute of the game) in both rows (both positions), because the matrix width is *M*.

As for the output, we will store the substitute when we're moving on to a new player (new block) in the corresponding minute and position, keeping track of the previous player (block), in order to know which player is leaving the game. If it's a player from the start of the row, we don't output the substitute, but place that player with the initial six players. We will output the substitutes chronologically in the end.

We need to be careful when outputting the substitutes. If a player is of endurance *M* and breaks into two parts, from row *X* to row *X* + 1, we will output them leaving the game in row *X* + 1, and their entrance in row *X* for the same minute. Substitutes where the player enters and leaves the game in the same minute are not allowed. We will solve this problem by checking if we are dealing with a player of endurance *M*, and if they don't start in the first column, then we will place them in the matrix broken into two parts, but still keep track of the previous player, and declare them as the one leaving the game when the next player is entering.

The implementation details are not complex, and are left as an exercise to the reader.

Necessary skills: sorting, matrices

Category: greedy algorithms

Task Deda	**Author: Luka Kalinovčić**

We maintain an array that, for each child, keeps track of the station they got off on. The query comes down to finding the first element smaller than or equal to *Y* in a suffix of that array. Linear search (for loop) is obviously too slow for large test cases. Therefore, we will use a tournament tree where each vertex holds the minimum of the corresponding intervals of the observed array. With each Marica's statement, we update the tree in the standard way, from the leaves to the roots.

In the query, that starts from the root of the tree, we behave in the following way:

- Call the query for the left child if its interval contains elements in common with the required suffix and if its minimum is smaller than or equal to *Y*.
- If the query for the left child found the required element, return it.
- Otherwise, call the query for the right child and return the given element or -1 if none was found.

What is the complexity of the query? The largest number of nodes reached will be:

- the O(log *N*) nodes of the tournament tree that precisely cover the required suffix and all the nodes on the path to these, which is again O(log *N*),
- the path from the first of the nodes whose minimum is smaller than or equal to *Y* to one of the leaves, which is again O(log *N*).

Necessary skills: tournament tree (segment tree) →
https://www.hackerearth.com/practice/notes/segment-tree-and-lazy-propagation/

Category: data structures

Task Plahte	**Author: Marin Kišić**

Let's construct the following graph: let each node be a rectangle from the task, and the edge between nodes *A* and *B* will exist if the rectangle represented by node *A* is the smallest rectangle that contains the rectangle represented by node *B*. We can notice that the constructed graph is a forest of trees where the roots of the trees are rectangles that are not contained in any other rectangle. We can construct this graph using the sweep line algorithm. We scan on the x-axis from 0 to +oo and at the same time maintain the

tournament tree on the y-axis. When the imaginary scanning line reaches the left side of a rectangle, we insert its index into the tournament tree in the interval [*b*, *d*] (*b* is the lower y-coordinate, *d* is the upper). However, before inserting, we check to see in the tournament tree which rectangle currently covers that interval in order to construct an edge between them. When we reach the right side of a rectangle, we remove its index from the interval [*b*, *d*]. The complexity of this part is O(*N* log *N*). Consult the official solution for implementation details.

Now let's place each paintball ball in the node that corresponds to the smallest rectangle that contains that paintball ball. We can do this using an almost identical sweep line algorithm in the same complexity. Now we've reduced the task to calculating, for each node, the number of different paintball balls in its subtree. We can do this by traversing the nodes from the deepest to the root by using a DFS tree traversal, and for each node maintaining a set of all paintball balls that are in the current node's subtree. More precisely, when we're processing a node, we merge its set of paintball balls with the sets of its children. We need to make sure that, when merging the sets, we always merge the smaller set with the bigger one, so we don't end up with a bigger complexity of the operation. You can read more about this here. Finally, for each node, we output the size of its set.

The complexity of the second part is O(*N* $\log^2$ *N*), which is also the total complexity of the algorithm. There is a solution of the complexity O(*N* log *N*), and it is left as an exercise to the reader.

Necessary skills: sweep line, tournament tree (segment tree), trees

Category: sweep line, data structures