Editorial

Tasks, test data and solutions were prepared by: Gabrijel Jambrošić, Marin Kišić, Pavel Kliska, Daniel Paleka and Paula Vidas.

Implementation examples are given in attached source code files.

Task Knjige

Prepared by: Marin Kišić
Necessary skills: ad hoc

This task has many solutions, and we will describe one of them.

Notice that if we can get all books to be on the right shelf, sorted from thickest to thinnest (in the order from top to bottom), then we can just move them one by one back to the left shelf.

We can accomplish that by repeating the following algorithm as long as the left shelf is not empty:

- Use the left hand to move books from the left shelf to the right shelf until we get the thinnest book from the left shelf to be on top.
- Take the top book from the left shelf in the right hand.
- Return all books that were moved in the first step back to the left shelf using the left hand.
- Put the book from the right hand on the right shelf.

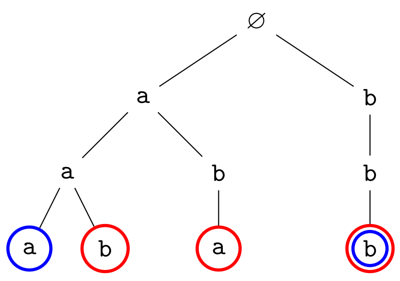
Task Vlak

Prepared by: Paula Vidas
Necessary skills: trie

We will use the *trie* data structure to solve the task. Explanation of trie can be found on many places, we recommend Codeforces, Wikipedia, Topcoder.

In short, trie is a tree with an empty string in the root, and letters in other vertices. When we build a trie using some set of strings, by concatenating all letters on some path from root to some other vertex we can get all string from the set and their prefixes.

Trie for the first example is:



We build the trie using all of Nina's and Emilija's words. We color the vertices that are ends of Nina's words blue, and verticies that are ends of Emilija's words red. (It is possible that a vertex is both blue and red.) The game starts in the root, and a move is just traversing some downward edge of the current vertex. Nina is on the move on even levels, and Emilija on odd levels.

Nina can make a move if there exists some blue vertex in that childs subtree, and a the same thing holds for Emilija and red veritces.

The winner can be determined by simple dynamic programming. We can calculate this from the bottom to the top. The player that is on the move wins if she can move to a vertex in which she also wins, otherwise the other player wins. The answer is the winner in the root.

Task Sateliti

Prepared by: Paula Vidas, Marin Kišić
Necessary skills: two-dimensional hashing, binary search

Note that a matrix obtained by circular shifts of rows and columns is uniquely determined by selecting the cell to be shifted to the upper left corner.

To simplify the solution, we will make a $2n \times 2m$ matrix $(a_{i,j})$ by taking four copies of the matrix, and replacing the characters `'*'` and `'.'` with 0 and 1. We need to determine the lexicographically smallest $n \times m$ submatrix of that matrix.

We can use two-dimensional hashing to compare submatrices: we take the hash of the element $a_{i,j}$ to be $a_{i,j}p^i q^j$, where p and q are distinct primes, e.g. 2 and 3. The hash of some submatrix is equal the sum of the hashes of its elements. If we precalculate the two-dimensional prefix sums of hashes, we can efficiently get that value for any submatrix. Of course, as the numbers can get very large, we calculate the hashes modulo some large prime number, e.g. $10^9 + 7$.

Two submatrices can be lexicographically compared by using binary search to find the first element in which they differ, and then comparing that element. If we first search the row, and then the column, of that element, we just need to know how to determine if some two submatrices are equal. To do that, we use the computed hash. Before we check if the hashes are equal, we need to divide (or multiply) them by appropriate powers od p and q, so that powers next to corresponding elements match.

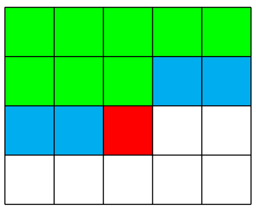
To find the lexicographically smallest $n \times m$ submatrix, we can go through all such submatrices, and compare them with the lexicographically smallest already processed submatrix.

If at the beginning we precompute all the necessary powers of p and q in the complexity $O(n + m)$, the hash prefix sums can be computed in complexity $O(nm)$. Two submatrices can be lexicographically compared in time $O(\log(nm))$, so the total complexity is $O(nm \log(nm))$.

Task Selotejp

Prepared by: Gabrijel Jambrošić and Marin Kišić
Necessary skills: dynamic programming, bitmasks

We solve the task using dynamic programming. The state is the row and column of the current square, and a bitmask that represents what happened on the last m squares, i.e. the current square, squares that are in the current row and on the left side, and squares in the previous row that are on the right side of the current square.



The current square is colored red, and other squares that are represented in the bitmask are colored blue. Let bit 0 mean that the square is covered by a horizontal piece of tape, and bit 1 mean that it is covered by a vertical one. If we want to use a horizontal piece to cover the current square, then if the square on the left is also covered by a horizontal piece, we can just continue using that piece, otherwise we need a new piece of tape. Simmilar thing holds if we want to use a vertical piece.

The transitions are:

- if bits corresponding to the current and the left square are both 0:
$$dp[i][j][mask] = \min(dp[i][j-1][mask],\ dp[i][j-1][mask \oplus 2^j])$$
- if the bit corresponding to the current square is 0, and the left square doesn't exist or its bit is equal to 1:
$$dp[i][j][mask] = \min(dp[i][j-1][mask],\ dp[i][j-1][mask \oplus 2^j]) + 1,$$
and if j equals 0, we do the transition using $dp[i-1][m-1]$
- if the bit corresponding to the current square is 1:
$$dp[i][j][mask] = \min(dp[i][j-1][mask],\ dp[i][j-1][mask \oplus 2^j] + 1).$$

We also need to take care of the open squares, we can't continue to use a piece that covers one of them. The complexity is $O(nm \cdot 2^m)$.

Task Specijacija

Prepared by: Marin Kišić, Paula Vidas, Pavel Kliska and Daniel Paleka
Necessary skills: persistent segment tree, lowest common ancestor

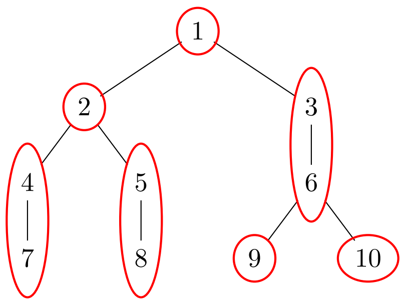
The subtask where $n \leq 1000$ could be solved by constructing the whole tree, and then for each query we can find lowest common ancestor of the given nodes.

Similarly, we can solve the subtask in which there is only one query, The only thing is that we cannot build the whole tree, but rather move towards the root from the given nodes until we reach the same node.

Below we will describe both the *offline* and *online* solution. The offline solution is worth 50 points, while for more we need to answer queries in an online fashion. The offline solution means we can load all queries first and respond to them in the order that suits us, while the online solution requires that each query is answered immediately after loading it.

For the offline solution, we will go from the bottom level to the topmost and maintain sets of queries in the subtrees. Note that when we move from one level to another, only one pair of sets merges. We can support this by merging the smaller set into the larger one. The first time some query appears twice in a set, the node represented by that set is the answer for the query, and we can remove the query from consideration.

For the online solution, we first need to build some data structure to help us with answering queries. The idea is to build a *reduced tree*. Nodes of the tree will be the components of the nodes of the original tree, obtained by removing the edges between the node a_i and its two children for each i, and an edge will exist between some two nodes if there is an edge between the corresponding components in the original tree.



The tree parametrized by $a = (1, 2, 6)$, with the components marked.

For each query, it is enough to find the components of the given nodes, and then the solution is the minimum of: the label of the first node from the query, the label of the second node from the query, and the label of the bottom of the component that is the lowest common ancestor of the components from the query on the reduced tree.

Two more implementation details are left: how to build the reduced tree, and how to find components of nodes online. Both issues can be solved with a persistent segment tree.

As in the offline solution, we will go from the bottom level to the top, and each level will have "its own" segment tree (thus we actually need persistence). In the leaves of the segment tree, we keep which component currently represents that node. When we go from one level to another, we connect the two components as follows: replace the left component with the label of the new component, and replace the right component by zero, which means there is no component there anymore. We connect the new component with the left and the right components with an edge in the reduced tree.

Note that now by supporting the query "give me the k-th leaf that has value greater than zero", we can easily find out in which components the nodes from the query are located. For implementation details, please see the official solutions.