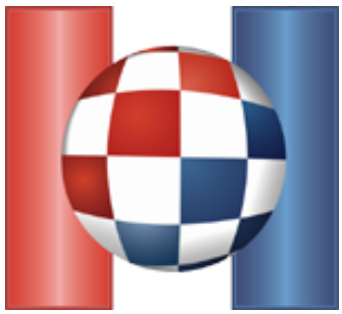
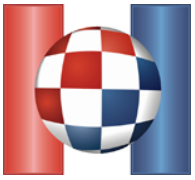


CROATIAN OLYMPIAD IN INFORMATICS 2014

algorithm descriptions



Given HTML document is comprised of a tree-like structure where the nodes are *div* elements. *Div* element A is a parent of *div* element B if B is directly contained in A. Each node in the tree is associated with *classes* which belong to a corresponding *div* element.

We can build such a structure in our programme simply by using stack and parsing carefully.

A selector can now be viewed as a series of instructions for traversing the tree. For instance, the classifier `.banana_.apple.orange > .orange` corresponds to the following series of instructions:

1. **beginning**: pick any node in the tree as a starting point for the search
2. `.banana`: check if the current node's class is `banana`, if not, end search
3. `_`: continue searching on any descendant of the current node
4. `.apple`: check if the current node's class is .apple
5. `.orange`: check if the current node's class is .orange
6. `>`: continue searching on a direct child of the current node
7. `.orange`: check if the current node's class is .orange
8. **end**: end search, this node corresponds to the selector

Such a series of instructions is easy to implement with careful parsing.

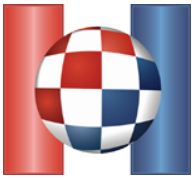
Now the tree traversal can be implemented as a recursive function which has two parameters, *node* and *pos*. The parameter *node* describes the current position in the tree, and the parameter *pos* describes the current position in the series of instructions for traversing the tree.

The recursion's transitions are now performed as described above. Therefore, for `>` we move onto a direct descendant, and for `.class` we check whether this class exists in the current node.

The transition for `_` is a bit more complicated. If we implemented it naively, by iterating over the whole list of descendants, it would be too slow. This is why the recursion will have an additional parameter *jump* that tells us whether we're in the process of moving onto a child node.

It is necessary to memoize the states of the recursion to ensure that we visit one state only once. The complexity is $O(M \cdot$ *number of states)*. The number of states is equal to $2 \cdot$ *number of nodes* $\cdot$ *number of instructions* which is approximately $N \cdot L$, where L is the maximum line length.

For implementation details, consult the source code.



We solve this task with dynamic programming. Let $dp[X]$ mark the minimal mistake of constructing a histogram with given points such that the rightmost point of the histogram is on the x-coordinate X. Now we could conceptually solve this problem by trying to connect each point on the coordinate X with the points left to it on the same height (y-coordinate), which would lead us to the correct solution of complexity $O(N^3)$, which is too slow.

This complexity implicitely assumes that the problem of determining the function $abserror$ or $diffcount$ between two points is solved in complexity of $O(N)$.

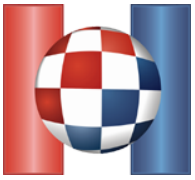
In order to speed up the given algorithm, let's notice that when calculating $dp[X]$ it is enough to observe only the previous x coordinate of a certain point (because the histogram can arbitrarily change the height without any additional errors).

This observation, along with a careful implementation, leads us to complexity $O(N^2)$ which was enough for 50% of points.

To further speed up the algorithm, it is necessary to calculate $diffcount$ and $abserror$ between two points in complexity $O(1)$ instead of $O(N)$. Let's show that this is possible for $abserror$ (the more difficult case), whereas we'll leave $diffcount$ for you to practise.

Let's notice that $abserror$ between two points on the same y coordinates a and b is equal to $abserror(a, b) = abserror(0, b) - abserror(0, a)$, where 0 marks a point (sometimes fictional) on x-coordinate 0 with equal height as a and b. In other words, it is necessary for each point p to determine only $abserror(0, p)$.

This problem can be solved for all points in complexity $O(N \log N)$ by using Fenwick tree (logarithmic structure) or segment (tournament) tree and scanline algorithm in the direction of the increasing x-coordinate. For implementation details, consult the source code.



Initially, let's notice that the memory can hold only blocked roads and built bridges because their amount will be sufficiently small. A query about the existence of a road between two cities G_1 and G_2 can be split into three cases:

1. The towns are located on opposite river banks:

   Let's find the first bridge which can be reached with roads from G_1 and from which there are roads to reach G_2. If such a bridge does not exist, then a way from G_1 to G_2 doesn't exist. The previous statement is true because, no matter what other bridge we pick, it will require the existence of all the roads our first bridge requires, and some additional roads.

2. The towns are located on the same river bank and the direction of roads on that bank is from G_1 to G_2:

   Bridges cannot help us in this case because they will lead us in the opposite direction of town G_2 and therefore we need to check whether there is a blocked road between G_1 and G_2. We can do this by finding the first blocked road which appears after town G_1. If this road is located before town G_2, a way does not exist, otherwise it exists.

3. The towns are located on the same river bank and the direction of roads on that bank is from G_2 to G_1:

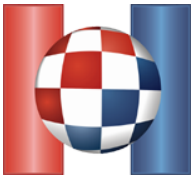
   Let's notice that we cannot reach G_2 by only using roads. We need to find the first bridge which can be reached from G_1 and cross to the other river bank. Now this case comes down to the first case. The choice of bridge is fine because of the similar reason as in the first case.

For answering queries about the first blocked road after a certain town, we can, for example, keep the blocked roads in a balanced search tree (STL set).

For answering queries about the first bridge that can be reached by roads from town G we simply find, in a similar way to queries about roads, the first bridge after town G and check whether there is a blocked road between them.

In the first case, we want to know about the first bridge after G_1 and before G_2, but it is sufficient to notice that the bridge is either first after G_1 or first before G_2 or they are equal.

The time complexity of this solution is $O(M \log N)$. For implementation details, consult the source code.



This task can be solved incrementally, meaning that we will take each member of the smaller sequence and insert it into the larger sequence, respectively. At the same time, we are left with the same subproblem with the same characteristics as the original problem and that means we can use the below described procedure multiple times until we're left with a sorted sequence.

Let's observe how it is possible to insert a whole smaller sequence into the larger one so that the first (smallest) element of the smaller sequence is in the right place. For simplicity and clarity, let's observe the following general sequence example:

$$a\ b\ c\ d\ A\ B\ C\ D\ E\ F\ G\ H$$

where lowercase letters mark the elements of the smaller sequence, and the uppercase letters mark the elements of the larger sequence. Using command cmp, we find the element after which we need to insert element a (in other words, we're looking for the biggest element of the larger sequence X for which holds $X < a$). With the help of binary search, we find the wanted X using only $O(\log n)$ operations. Let's assume in this case that the following holds: $C < a < D$. If we use command reverse on all elements between a and C, we get:

$$C\ B\ A\ d\ c\ b\ a\ D\ E\ F\ G\ H$$

Furthermore, flipping the elements $[C, A]$, $[d, a]$, we get:

$$A\ B\ C\ a\ b\ c\ d\ D\ E\ F\ G\ H$$

where we can see that the first four elements of the sequence are on their final positions, so we use the same algorithm on the given sequence, only this time we ignore the first four elements. A simple calculations shows that the total cost of all reverse orders is at most $O(N_a^2 + 2 \cdot N_b)$, which is enough for all 100% of points on this task.