Editorial

Tasks, test data and solutions were prepared by: Marin Kišić, Josip Klepec, Vedran Kurdija, Daniel Paleka and Paula Vidas.

Implementation examples are given in attached source code files.

Task Šifra

Prepared by: Marin Kišić
Necessary skills: naredba ponavljanja

With the first loop we pass through the word, and when we come across a digit, we convert the next few digits into the corresponding number and skip those digits in the main loop.

As the numbers have at most three digits, we can have an additional boolean array in which the i-th element tells us whether we have found the number i in the word.

For code golf fans, there is a solution to this problem in only one line of Python, attahed in the official source codes.

Task Po

Prepared by: Vedran Kurdija
Necessary skills: stack

The problem is equivalent to starting from the given sequence and reaching all zeros in the minimum number of decreasing steps.

The main insight is that if we take a number x to zero using only one decrease, we should also take all other elements equal to x in the same step, unless there are elements less than x in between them. For example, take the sequence [4 6 4 5 4]. If we take any 4 to zero in a single step, we should decrease the segment spanning all of them.

It is also clear that the minimal nonzero element of the sequence should be taken to zero in a single decrease. This gives us a solution for $n \leq 1000$; just keep track of the minimal nonzero value, and always decrease some minimal elements.

We can make this approach fast using a segment tree, but the intended solution is simpler.

Traverse the sequence from left to right. It can be proven that each element contributes 1 step to the optimal solution, except those elements for which there is an element of equal value to the left, and there are no smaller elements between them.

We can efficiently find whether each element satisfies the above condition using a monotone stack. We push values to the top of the stack after popping the stack until the new element is not smaller than the old top of the stack. If the top and the new element have equal values, we don't add 1 to the solution. The complexity is $O(n)$.

Task Magenta

Prepared by: Vedran Kurdija
Necessary skills: ad hoc, depth-first search

To solve the first subtask with dynamic programming with $O(n^2)$ states and $O(n)$ transitions, see this link.

The main insight in the second subtask is that Paula certainly wins if the path length between the starting positions of Paula and Marin is even, and Marin if the length is odd. The reason for this is that the player

with the right parity can take on the role of a *hunter*, approaching the opponent with every move. The hunter will then drive the opponent into a leaf and win. Therefore, for the second subtask it was enough to find the path length between a and b with depth-first search and print the winner depending on the parity. The complexity is $O(n)$.

In the last subtask, the above insight is partially preserved. It is no longer true that the player with the right parity will surely win, but they can always either win or draw. The hunter will again hunt the opponent, but this time, due to the possible edges that are impassable to the hunter, the opponent may be able to escape and thus secure a draw. Before embarking on the analysis of how to escape, let us mention special cases when Paula cannot make the first move and when Marin has only blue edges. In the first case Marin wins, and in the second case Paula wins.

Barring the above special cases, Paula and Marin can make their moves. Hence, the game will then end either with the hunter catching the opponent, or the opponent managing to escape and secure a draw.

Note that if there is a edge such that both its vertices are accessible to the fleeing player, where we consider a node that can be reached from the starting point, and no vertex of that edge is accessible to the hunter, then it's a draw. If such a edge does not exist, the hunter wins. It remains to check whether there is such a edge.

We will start depth-first search from the initial position of the hunter, and for each vertex available to the hunter, write down the initial distance.

After that, we start depth-first search from the fleeing player, with the additional condition that we expand into a vertex only if the distance we from the fleeing playr to it is less than the initial distance of the hunter to that vertex, that is, if we can reach that node without colliding with hunter.

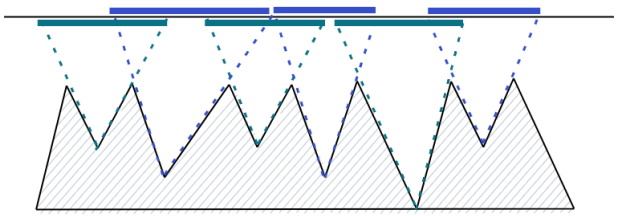
If during this search we ecounter an edge that leads us to a vertex to which we can expand, and if the current vertex, as well as the vertex to which we expand, are not available to the hunter, it is a draw because we found an edge on which the fleeing player can oscillate indefinitely. The total complexity is $O(n)$.

Task Planine

Prepared by: Daniel Paleka and Paula Vidas
Necessary skills: convex hull, line intersection formula

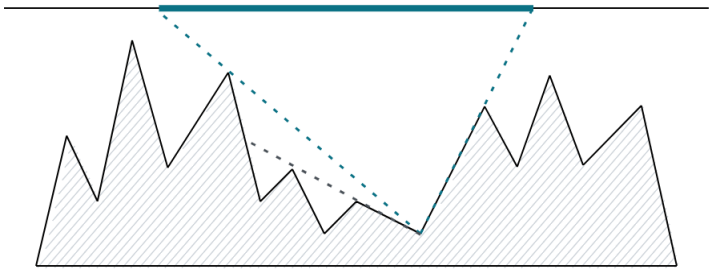
Notice that for each valley, there is a segment of x-coordinates for which a fairy illuminates this valley.

In the first subtask, the segments are determined by the points adjacent to the valley. We can just intersect the lines with the line $y = h$. The complexity of this step is $O(n)$.



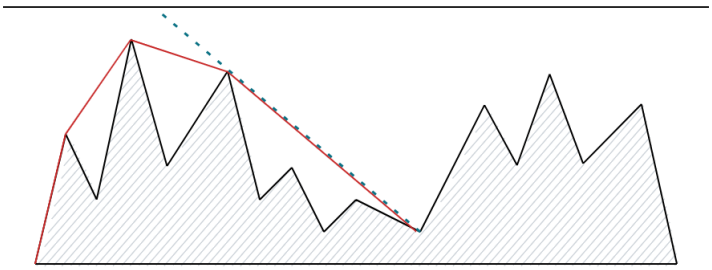
After we find the segments, we have the well-known problem of covering a set of segments with the minimum number of points. The greedy algorithm where we sort the segments by their right end is correct, and the complexity is $O(n \log n)$.

When the mountain tops have different heights, there is an issue: it can happen that a nonadjacent mountain top "blocks the view":



For the second subtask, we can look at each valley-mountain top line, and determine the segment from the intersections of the lines with the line $y = h$. The complexity is $O(n^2)$.

For the full solution, note that the mountain top which "blocks the view" is actually the penultimate point on the upper convex hull of all the points up to the current valley:



Hence, maintaining the convex hull is enough to find the left ends of all the segments. We can do the same thing (but backwards) for the rights ends. The complexity of this part is $O(n)$.

Task Sjeckanje

Prepared by: Daniel Paleka and Paula Vidas
Necessary skills: segment tree

The first subtask can be solved by dynamic programming. We find the largest possible value of each array prefix. When we are on some element, we try to cut the string in all possible places to the left, and maintainthe current minimum and maximum. The complexity is $O(n^2)$ per update.

For other subtasks, we need the following observation: It is optimal to chop the array into monotonous (ascending or descending) segments. If the segment is not monotonous, we can always cut it into two parts that will have higher total value.

The value of a monotonous segment $[a_l, a_{l+1}, \ldots, a_r]$ equals $|a_l - a_{l+1}| + |a_{l+1} - a_{l+2}| + \cdots + |a_{r-1} - a_r|$.

Let $d_i := |a_i - a_{i+1}|$. The value of the whole array is the sum $\sum |d_i|$, minus those $|d_i|$ in the places where we cut. That is, if we cut the string in place between i and $i + 1$, we won't include $|d_i|$ in the answer.

We have reduced the problem to the following: we are given an array of of integers (d_i), and we need to choose some of them so that the sum of their absolute values is maximal, without taking two adjacent numbers, one of which is strictly positive and the other strictly negative. (The least condition is because the segments of the initial sequence must be monotonic).

For the second subtask, we can recompute the maximum possible value after each update by simple dynamic programming in complexity $O(n)$: for each prefix we calculate the maximum possible value in two cases depending on whether or not we took the last element. Hence the total complexity is $O(qn)$.

We can solve the full problem using a segment tree. Each node remembers the maximum value of its segment, for each of the four cases of (not) taking the first and the last element. While merging, we have to pay attention to the signs of the elements on the border to be joined. Note that when an update occurs, at most two d_i-s (those at positions $l-1$ and r) will change, so the segment tree easily handles this easily. The total complexity is $O(n + q \log n)$.