Editorial

Tasks, test data and solutions were prepared by: Nikola Dmitrović, Karlo Franić, Gabrijel Jambrošić, Marin Kišić, Josip Klepec, Vedran Kurdija, Daniel Paleka, Ivan Paljak, Stjepan Požgaj i Paula Vidas. Implementation examples are given in attached source code files.

Task: ACM

Suggested by: Vedran Kurdija and Marin Kišić
Necessary skills: string parsing, sorting, greedy algorithms

For 20 points, it was enough to conclude that (since there are no `'?'` in the input) the unfrozen standings will look exactly like the frozen standings. Therefore, we only needed to find which row of the frozen standings starts with `NijeZivotJedanACM` and output its index.

In the remaining 30 points, it was necessary to deduce that, in the worst case for our team, all other teams should solve all problems that they have submitted during freeze time. The intuition is clear, the more problems other teams solve, the worse it is for our heroes. Therefore, we can change all `'?'` into `'+'` for all other teams except our heroes. For them, we already know the whole row (it was given in the input).

The only thing that remains is to parse the standings and correctly calculate the penalty time for each team. Most `WA` verdicts on this task stem from wrong penalty calculations, more often by treating the penalty for each wrong submission as 20 seconds instead of 20 minutes.

Task: Slagalica

Suggested by: Gabrijel Jambrošić
Necessary skills: case analysis, greedy algorithms

The first subtask was solvable via simple case analysis with a couple of well-placed `if` statements.

The second subtask could have been solved by trying out all permutations of puzzle pieces in time complexity $\mathcal{O}(n!)$.

The third subtask could have been solved by observing that we must alternate between pieces of type 1 and 4 so it's always optimal to take the piece that fits and has the smallest number written on it. If we have some leftovers at the end, the puzzle cannot be solved.

Regarding the fourth subtask, let's assume we have already placed some pieces and the last one that was placed had a hole on its right end. The next piece that will be placed must have a bump on its left end, and that condition is satisfied only by pieces of type 1 and 2. If we place a piece of type 2, the right side remains a hole and we will again need to place a piece of type 1 or 2 next. On the other hand, if we place a piece of type 1, then in the next step we'll have to place a piece of type 3 or 4 (following a similar argument). Therefore, we could think of a piece 1 as a piece that, in a sense, changes the form of the next piece. By analogy, the same holds for piece 4. Therefore, in this subtask we will first place all pieces of type 2 (or 3, depending on the left bound), then we'll place a piece of type 1 (or 4), and finally we'll place all pieces of type 3 (or 2). We always place the piece of the proper type with the smallest possible value. We must also take extra care of a special case where there is no piece of type 1 or 4.

Finally, to solve the entire task, let's consider a naive greedy algorithm in which we always pick a piece that fits and has the smallest possible number written on it. The problem is that we may end up with some leftover pieces of types 2 or 3 (if any other type remains, the puzzle cannot be solved). We can fix this by inserting those pieces in the last place (farthest from the left) where they could fit. That place will always be adjacent to the last piece of type 1 or 4.

Time complexity: $(O)(n)$.

Task: Checker

Suggested by: Paula Vidas and Daniel Paleka
Necessary skills: mathematical induction, linked list

Claim 1: In each triangulation with $N \geq 4$ there are at least two "ears", i.e., triangles that share two sides with the input polygon.

Sketch of the proof: Induction. Claim holds for a square. Let a diagonal divide the polygon into two smaller polygons. If one of them is is a triangle, that is an ear. Otherwise, we make an inductive argument.

First we will describe the process of triangulation checking. A naive algorithm will remove ear by ear. This can be more or less efficiently implemented and you could have scored the first two subtask using an $\mathcal{O}(n^2)$ implementation.

Here we present one possible $\mathcal{O}(n \log n)$ approach.

For each diagonal ab, we construct directed edges (a, b) and (b, a). Now, let's sort all $2(N-2)$ edges by the distance of their endpoints in the starting polygon. Here, we consider the distance between two nodes as the number of polygon sides between them in clockwise direction. It is clear that the first element of this sorted array must be an ear. It can be proved that if we trim the ears in this order, in each moment the first remaining edge will be a part of the ear that needs to be removed in that step. If it is not, then the triangulation is not valid.

For implementation details see the attached solution, where we use a linked list to keep the current order of outer polygon nodes. Both color and triangulation checks can be done at the same time.

Task: Popcount

Suggested by: Ivan Paljak
Necessary skills: constructive algorithms, segment tree

Esoteric programming language, only one variable, constraints on source code length, looks like a perfect combination for solving "subtask by subtask".

In the first subtask we're allowed to use the number of commands that is (almost) the same to the number of bits of the input value. This suggests that we could solve the subtask "bit by bit". Let's assume that we have successfully solved the suffix of X bits, i.e., that suffix of A now contains the number of ones in the corresponding suffix of input value while the other bits of A remain unchanged with regards to the input value. Note that the input value itself already has a solved suffix of size 1. Suffix of size $X+1$ bits will be solved by adding the value of bit $(X+1)$ to the variable A while also setting that bit to 0. We can do that using the following command `A=((A&(((1<<N)-1)-(1<<X)))+((A&(1<<X))>>X))`. We could have solved the first subtask by u using $(N-1)$ commands of this form.

To solve the second subtask, we can use the same idea from the first subtask along with the fact that variable A can be found at most 5 times in the right side of command. Therefore, instead of solving the task "bit by bit", we will solve it using the technique "four bits by four bits". This will reduce the number of used commands 4 times.

To solve the remaining subtask, we could use an idea from the famous data structure, the `segment tree`. Let's imagine that we have built a segment tree on top of our N bit number. Each node of the tree will store the sum of bits (number of ones) in its subtree. Imagine also that those values are written in binary with the number of bits that corresponds to the length of the interval (segment) which is covered by that node.

If we were to glue the values of all leaves (from left to right) in our segment tree, we would get the input value into our program. If we glue the values of their parents in the same way, we will get the wanted

value of number A after our first command. If we were to repeat this process until the root, we will in the end store the correct value in variable A by using an order of $\mathcal{O}(\log n)$ commands, which should be enough for the last two subtasks. The only thing left to do is to find the command which makes the transition from one set of nodes to their parents in the segment tree.

For the first command, we could have taken all bits at even indices and add to them all bits at odd indices shifted to the right by one. We can do that with the expression `((...01010101&A)+((...10101010&A)>>1))`. In the next step (intervals of length 4) the expression could be `((...00110011&A)+((...11001100&A)>>2))`. By repeating this pattern we will solve the task using $\mathcal{O}(\log n)$ commands, each of which has two occurrences of A in them.

In this task, for the sake of implementation, we suggest using languages `Python` or `Java`.

Task: Zvijezda

Suggested by: Marin Kišić
Necessary skills: ccw function, binary search, basic computational geometry

Let A_i denote the i-th (modulo n) point of our input polygon, and let X denote the point from Mirko's query.

If $ccw(A_i, A_{i+1}, X) > 0$, we will write 1 on the i-th side of our polygon, otherwise we'll write 0. Now, we need to answer the question: "Is there a pair of opposite sides such that both of them have 1 written on them". Obviously, we cannot check what is written on each of the sides for every query because we would have an $\mathcal{O}(nq)$ algorithm. But, we can get a more efficient algorithm by making some observations.

Observation 1: If two sides have 1 written on them, then all sides between them (in one direction) also have 1 written on them. We can intuitively see this by imagining that X is a light source and polygon sides are walls through which the light cannot pass. We can see that illuminated sides have 0 written on them. Since all zeroes form an interval on this cyclic array, we can deduce that all ones form an interval as well.

Observation 2: the answer is `DA` if and only if an interval of ones contains at least $\frac{n}{2} + 1$ elements. This is obvious because in $\frac{n}{2} + 1$ successive sides there must be two of them which are opposite.

Now, let's consider an arbitrary side and its opposite side and observe what values are written on them. If both of them have 1 written on them, we can immediately output `DA`. If both of them have 0 written on them, we can immediately output `NE`. If one of them has 1 written on it and another one has 0 written on it, we can cut the array into two parts, each of which starts with one of the sides we have checked. One part has the form `111...000`, and the other one has the form `000...111`. In one of them we want to find the last 1 and in the other one we want to find the first 1. This can easily be done using binary search. Now we have found the endpoints of an interval of ones and can easily check of the condition from observation 2 holds.

Time complexity: $\mathcal{O}(q \log n)$