Editorial

Tasks, test data and solutions were prepared by: Nikola Dmitrović, Dominik Fistrić, Bartol Markovinović, Pavel Kliska, Bojan Štetić and Krešimir Nežmah. Implementation examples are given in the attached source code files.

Task Kemija

Prepared by: Bartol Markovinović
Necessary skills: for-loop, strings, arrays/lists

The task requires us to check if there is an equal number of atoms on each side of the equation. For each side of the equation we'll maintain a list (array) to keep track of the number of occurrences of each letter on that side. In the end we'll compare the two lists and if they are equal print `DA`, and otherwise `NE`. After each equation we must make sure to clear each of these lists.

To count how many atoms are on each side, first we separate the equation into two strings, each representing one side of the equation. In Python, this can be done using the split() function, and for other programming languages this can be done by iterating through the string and searching for a sequence of characters equaling `->`. After we have split the left and the right side, in a similar way we can split each of the sides into a list of strings representing the molecules. Then we count how many atoms are in each molecule and add this to the total for the corresponding side. To count the number of atoms in a molecule, we iterate over the string representing the molecule and for each letter we look at the number coming after it (if it doesn't exist we assume it's 1) and multiply the count with this number. We also multiply with the number at the beginning of the string if there is one.

Task Dijamant

Prepared by: Krešimir Nežmah
Necessary skills: BFS or DFS

Notice that if somewhere in the table there is a diamond shape whose edge is made out of `#`, then it's interior is also a diamond shape, but its edge is from `.` instead of `#`. Therefore, it is enough to iterate over the table and using BFS or DFS find the maximal connected regions made from `.` and for each such region check if has a diamond shape.

Let's now look at all cells (i, j) for which $i + j = k$, for some fixed value of k. It's easy to see that those cells will form a diagonal in the table going in the direction from bottom-left to top-right. Also, smaller values of k correspond to diagonals closer to the top-left corner of the table, while larger values of k correspond to diagonals closer to the bottom-right corner of the table. In similar way, the set of cells for which $i - j = k$ forms a diagonal, but in the other direction. This means that if we are given fixed numbers a, b, c and d such that $a \leq b$ and $c \leq d$, then the set of all cells (i, j) for which $a \leq i + j \leq b$ and $c \leq i - j \leq d$ actually forms a tilted rectangle. In case $b - a = d - c$ it is actually a tilted square, i.e. a diamond.

Therefore, to check if some region is a diamond or not, we keep track of the following: the number of visited cells (`cnt`), and the minimum and maximum diagonal in both directions ($a = \min i + j, b = \max i + j, c = \min i - j, d = \max i - j$). What remains is to check whether $b - a = d - c$ and to see if the number of visited cells matches the number of cells that should be in the diamond of this size $(b - a)$. In doing so, we need to calculate the number of `.` in a diamond of size n, which turns out to be $n^2 + (n - 1)^2$.

Task Fliper

Prepared by: Krešimir Nežmah
Necessary skills: sweep line, Eulerian tours

Clearly, a necessary condition is that the number of bounces in each cycle is divisible by 8. We'll show that this is also sufficient.

First off, we need to determine all the cycles in which the ball could get stuck. We can split each obstacle in two pieces, so that each piece corresponds to one side of the obstacle. Then for each x-coordinate we make a list of all pieces having that x-coordinate and sort it by their y-coordinate. After that we connect pairs of adjacent pieces one after another to mark them as being in the same cycle. The pieces at the ends of the list are connected with a special label representing that they are not part of any cycle, rather that they are connected to infinity. We do a similar thing for the y-coordinate. The described joining can be maintained in a disjoint set union structure.

In this way we've determined for each piece of an obstacle in which cycle it belongs to (or that it doesn't belong to any cycle, but that it's connected to infinity). Now we make a graph: each of the found cycles will be a node in the graph (and there is a dummy node representing infinity), and for each obstacle we add an edge between the nodes corresponding to the pieces the obstacle is made from. It is possible that an edge connects some node to itself.

Notice that the number of bounces in a cycle now represents the degree of a node. The problem can now be rephrased as follows: given a connected graph where the degree of each node is divisible by 8 (except maybe for the dummy node), color the edges of the graph in four colors so that each node (except the dummy node) is connected to an equal number of edges of each color.

The problem can now be solved using Eulerian tours. Note that if every node in a connected graph has an even degree, we can start an Eulerian tour from some node and color the edges alternately in two colors. In that way we make it so that each node except for the starting node (which also happens to be the ending node) has an equal number of edges of each of the two colors. This might also hold for the starting node, but only if the number of edges in the graph is even.

For the graph in the problem, we know the degree of each node except the dummy node is divisible by 8. The handshaking lemma guarantees that the degree of the dummy node is also even. Since the degree of each node is even and the graph is connected, we can start an Eulerian tour from the dummy node. In that way, we color the edges in two colors so that each node except maybe the dummy node has an equal number of edges of these two colors. The problem requires us to find a coloring using four colors, so we'll find Eulerian tours again, but separately on the edges of one color and then the edges of the other. Namely, if we look at only the edges of one color, the degree of each node (except maybe the dummy node) is divisible by 4. The graph will not necessarily be connected, but this is not a problem because the handshaking lemma guarantees that each component not connected to the dummy node has an even number of edges.

Task Radio

Prepared by: Pavel Kliska
Necessary skills: sieve of Eratosthenes, segment tree, set

For each broadcasting radio frequency x we can keep track of two other radio frequencies L_x i R_x, the next smallest and the next largest broadcasting frequency which makes noise with x (if L_x doesn't exist, we take it to be -1, and if R_x doesn't exist, we take it to be 10^9). The interval $[l, r]$ crates noise if $\max_{x\in[l,r]} L_x \geq l$ or $\min_{x\in[l,r]} R_x \leq r$. The minimum and maximum in the interval can be determined in $O(\log n)$ using a segment tree. What remains is to figure out a way to keep track of L_x and R_x.

Notice that two frequencies create a noise if and only if they are both divisible by some prime number. If in the sieve of Eratosthenes we store for each number one of its prime factors, then we can factor a number in $O(\log n)$ by going over each of its prime factors. For each prime we can maintain a set of broadcasting frequencies divisible by this prime. Then for each frequency there are at most $O(\log n)$ candidates for L_x and R_x.

These observations are sufficient to obtain a solution. When a frequency x becomes active, for each prime

factor of x we look at the set of broadcasting frequencies for that prime and we find the next smallest and next largest frequency in this set. The minimum of the larger frequencies will be R_x, and the maximum of the smaller frequencies will be L_x. Additionally, for each of the L_x and R_x candidates, their L and R values might change, so we must update them. The number of candiadtes is $O(\log n)$, and updating L_x or R_x takes $O(\log n)$. Therefore, we can keep track of the updates in $O(\log^2 n)$. When deactivating a frequency, we do the opposite. We erase the number x from all the sets it is in, and adjust L_x and R_x of its neighbours. Again, the number of neighbours is $O(\log n)$.

It should be noted that the time complexity is in practice even better because numbers less than 10^6 can have at most 7 distinct prime factors since $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 > 10^6$.

Task Usmjeravanje

Prepared by: Dominik Fistrić and Krešimir Nežmah
Necessary skills: ad-hoc strongly connected components

First, let's introduce the notion of Strongly Connected Components (SCC). In a directed graph, a subset of nodes is strongly connected if there exists a path between any two nodes of the subset. Each directed graph can be decomposed into strongly connected components, such that the number of these components is as small as possible. There are various algorithms which do this and we will mention them later. For now, we can notice that the size of the maximal set from the problem is actually just the number of SCCs (we take one node from each SCC and it is obvious that this is maximal). So the problem is actually asking us to orient the edges in such a way as to minimize the number of SCCs in the graph.

The first subtask can be solved by trying all the combinations of flight routes, decomposing the graph into SCCs and counting them. In this case the SCCs could be found naively. For example, for each city we can find all the cities reachable from it and then for each pair of cities if both are reachable from each other then they belong to the same component.

For the remaining subtasks one needs to notice that each SCC will consist of two intervals of consecutive nodes, one from each river. The proof of this claim is simple and is left as an exercise for the reader. Let's look at the flight routes and sort them from left to right on both rivers separately. Clearly, each city on the first river to the left of the first flight route is part of its own SCC because it's not possible to come back to it. Therefore, let's look at the first node on the first river which has a flight route and the first node on the second river which has a flight route. If these two nodes are connected by a flight route, and if they are not connected to any other flight routes, then this flight route is useless. Indeed, whichever orientation we choose for it, it's not possible to return to the city from which the route starts. Now we assume that there exists a route connecting one of the leftmost cities to some city to the right of the leftmost city on the other river. Without loss of generality assume this route connects the leftmost city of the first river with a city from the second river. If we orient this route so that it goes from the second river to the first, and if we orient the route connecting the leftmost node from the second river so that it's going from the first river to the second river, then we obtain an SCC consisting of the intervals determined by the leftmost nodes and the other ends of the routes (Here we should notice an edge case. We did not demand that the route starting from the leftmost node of the second river connects to some node to the right of the leftmost node on the first river. In fact, it's possible that this route connects precisely the two leftmost nodes. This is not a problem, as the interval on the first river will then simply be one node). Let's now look at some other flight route where at least one of its nodes is part of the current SCC. Without loss of generality assume it is a city from the first river. If the other end of the route is also a part of the current SCC, then this route doesn't effect anything because the two cities are already connected both ways. If the other city is not a part of the current SCC, then it is clearly to the right of the current interval on the second river (if it were to the left, we would have a contradiction with choosing the leftmost cities). It is easily seen that if we orient this flight route so that it goes from the second river to the first river, the SCC interval on the second river will expand to the right. Now we repeat this procedure, that is we expand the intervals of the current SCC until there is no more flight route for which at least one end is in the current SCC. Once no such routes are left, we cannot expand

the current SCC and it has the maximal possible size. Now we can repeat this algorithm by ignoring the nodes of this SCC and looking for the leftmost nodes on each river again.

After we determine the orientations of the routes, what remains is to count the number of SCCs and print it. This is possible with a naive implementation described in the first subtask. For the whole solution, one could implement one of the well-known algorithms for finding SCCs in linear time complexity such as Tarjan's or Kosaraju's. Note that it is not actually required to know any of these algorithms to solve the problem. Instead, we can easily see that if there were no routes, the answer would be $a + b$, and as we connect the nodes into SCCs, we keep track of how many nodes have been added and reduce the answer by that much.