Problem A1. Cracow Sights

Author: **Pavlo Tsitsei**
Task preparation: **Pavlo Tsitsei**
Editorial: **Pavlo Tsitsei**

Block 1: $x_i^2 + y_i^2 \leq 18$ for all i:

We can change from (x, y) to (a, b) if $x^2 + y^2 = a^2 + b^2$, which means that we can change from $(-x, y)$ to (x, y) or from (x, y) to (y, x), so we can make all coordinates in the input of type $0 \leq x \leq y$ and group them together. In this way, we have 10 types of such groups (if we don't count $(0, 0)$). So we can solve it as the third block.

Block 2: $n = 1$:

Note that the distance from $(0, 0)$ to (x, y) is $\sqrt{x^2 + y^2}$, so if we change (x, y) to (a, b) where $x^2 + y^2 = a^2 + b^2$, then the distance doesn't change. So, the minimum distance is from $(0, 0)$ to the point in the input and from this point to $(0, 0)$.

Block 3: $n \leq 10$:

We can try each permutation of points that defines the path we take, and the first and last elements will be $(0, 0)$. The total complexity will be $O(n!)$.

Block 4: without additional constraints:

We can change the coordinates of each point from (x, y) to $(0, \sqrt{x^2 + y^2})$. Then, the answer is the maximum of these points times 2 (since we need to return).

Problem A2. Sakurako and a Weird Permutation

Author: **Bohdan Feysa**
Task preparation: **Bohdan Feysa**
Editorial: **Bohdan Feysa**

Block 1: $n \leq 8$:

In this block's solution, we will try to generate all possible permutation lists and check whether they are good for our permutation t. Time complexity $O(\binom{n \cdot (n-1)}{8} \cdot n)$. Which is less than $2.5 \cdot 10^8$.

Block 2: $n \leq 100$:

For our permutation, we will create its cycle decomposition as follows:

1. create its graph representation;
2. compute for each vertex the set of those that are accessible from it in $O(n^2)$;
3. find the order of those numbers in a cycle.

Now that we have a cycle decomposition of our permutation, we take its transpositions and it is our answer. Time complexity $O(n^2)$.

Block 3: $n \leq 10^5$:

In this block's solution, we can use $O(n \log n)$ time cycle decomposition of a permutation. When we have all cycles, we write out the edges of cycles and that is our answer. Because by using those permutations, we are creating an inverse of t. And $t[t^{-1}[i]] = i$, and that is what we need to find.

Block 4: full solution:

We change our code to decompose a permutation into cycles in linear time. Then we write out the edges of a cycle, and that is our answer. Time complexity $O(n)$.

Problem B1. UJGOI in Cracow

Author: **Oleksandr Tymkovych**
Task preparation: **Oleksandr Tymkovych, Pavlo Tsitsei**
Editorial: **Oleksandr Tymkovych**

Block 1: $a_i \geq 1$:

The main observation is that the array has no segment of length $\ell \in \{2, 3, \ldots, n\}$ with a major element if and only if the array has no segment of length $\ell \in \{2, 3\}$ with a major element.

Block 2: $a_i = 0$:

For $k \geq 3$ one of the possible answers is $a = [1, 2, 3, 1, 2, 3, 1, 2, 3, \ldots]$. For $k < 3$ the answer is almost always -1, except for sufficiently small n where the previous construction works as well.

Block 3: there is no $i > 1$ such that $a_i = a_{i-1} = 0$:

For each unknown element, we know it's both neighbors (if they exist). The solution is similar to $k > 5$ with some corner cases.

Block 4: $k > 5$:

For $k > 5$ we can do greedy replacing each unknown element with $\texttt{mex}\{a_{i-2}, a_{i-1}, a_i, a_{i+1}, a_{i+2}\}$.

Block 5: $k \leq 5$:

For $k \leq 5$ the author's approach included dynamic programming with some of the states meaning the last three values solving the sub-problem on the array $[a_1, \ldots, a_i]$.

Block 6: without additional constraints:

Combine the solutions of blocks 4 and 5.

Problem B2. UJGOI Scoreboard

Author: **Pavlo Tsitsei**
Task preparation: **Pavlo Tsitsei**
Editorial: **Pavlo Tsitsei**

Block 1: a_i is odd for all i, $n \leq 1000$:

Let's make a prefix sum of the array, so we can find the sum of any subarray in $O(1)$, and then go through all valid intervals. g_{odd} of that interval is not zero if the number of odd elements is odd, i.e., the length of the interval is odd. So, we can just check for each interval if the length is odd and, if so, add its sum to the answer.

Block 2: $n \leq 1000$:

Here, we need to make two additional arrays. Let it be an array *odd* and *even* where $odd_i = 1$ if a_i is odd, and otherwise it is 0. For array *even*, $even_i = 1$ if a_i is even, and otherwise it is 0. We can make a prefix sum on this array, and then we need to add $g_{odd}(l, r)$ for some valid interval l, r if the number of i such that odd_i is 1 (so the sum of elements on the subarray of array *odd*) is odd, which can be computed using prefix sum. The same goes for even, so we can just traverse all intervals.

Block 3: a_i is odd for all i:

Let's fix some element i, and we will count how many times it is included in the sum of the interval. As it was stated, a_i is included in g_{odd} if the number of odd elements on the interval is odd, so if the length of the interval is odd, which means that l and r have different parity. So, we can find it just by counting the number of r in the range from i to n such that it is odd and even, and the same for l, but in the range from 1 to i.

Block 4: without additional constraints:

Let's generalize the solution for the third block. So, if the element is odd, we need to count the number of intervals where this element is included in g_{odd}, otherwise in g_{even}. We will assume that we include it into g_{odd} always, since for g_{even} it is symmetrical. So, for element i it is included in $g_{odd}(l, r)$ if a_i is odd and the interval from l to r has an odd number of elements. Let's make an array *odd* like in the second block, but with the idea from the third. So we fix some i, and we need that the parity of odd_r is different from the parity of odd_l. To do this, we can make two new arrays *odd*[0] and *odd*[1] where we will count the number of even elements of array *odd* on the prefix of length i and the number of odd such elements, respectively. Now, we can count the number of odd_r which are even or odd using the prefix sum on the array *odd*[0] or *odd*[1], and the same for l.

Problem C1. Sakurako and Her Trip

Author: **Bohdan Feysa**
Task preparation: **Bohdan Feysa**
Editorial: **Bohdan Feysa**

Block 1: $n, m \leq 100$:

For the first block of our task we will be iterating through every possible answer k ($1 \leq k \leq n$), create a matrix $n \times n$ and check whether there is a path from $(1, 1)$ to (n, n) which does not contain any cells closer than k to any tourist traps. This results in $O(n \cdot (n^2 + m))$ time complexity.

Block 2: $n, m \leq 10^3$:

For the second block of this task we will preserve the technique of constructing the matrix, but instead of iterating through every possible answer k we will use binary search to find the biggest possible k s.t. there is a path from $(1, 1)$ to (n, n) which does not contain any cells closer than k to any tourist traps. This results in $O(\log n \cdot (n^2 + m))$ time complexity.

Block 3: $n \leq 10^3$:

This block has the same solution, but here we need to use BFS from all m tourist traps to mark all cells that are closer than k to any tourist trap. This results in $O(\log n \cdot (n^2))$ time complexity.

Block 4: either $x_1 = x_2 = \cdots = x_m$ or $y_1 = y_2 = \cdots = y_m$:

This is a block in which we could use a formula to try and pass from $(1, 1)$ to (n, n) between each pair of closest tourist traps. Time complexity is either $O(m \log m + m \log n)$ or $O(m \log m + m)$.

Block 5: full solution:

We will create a graph out of $m + 4$ vertices. Each of the first m vertices denotes a tourist trap. Other 4 vertices denote 4 borders of the matrix.($m + 1$ - top,$m + 2$ - left,$m + 3$ - bottom,$m + 4$ - right)

We will create an edge between a pair of first m vertices if there is no path from $(1, 1)$ to (n, n) which passes between these two tourist traps while not passing through any cell which has distance from it to a tourist trap less than k. Also, we will try to connect the first m vertices to the other 4: if that tourist trap is closer to the border than k, then we add an edge. Now, if there is a path from $m + 1$ to $m + 2$ or from $m + 1$ to $m + 3$ or from $m + 2$ to $m + 4$ or from $m + 3$ to $m + 4$, then there is no way to go from $(1, 1)$ to (n, n) with given k.

Total time complexity $O(m^2 \log n)$ which passes nicely.

Problem C2. UJGOI Party

Author: **Oleksandr Tymkovych**
Task preparation: **Oleksandr Tymkovych, Pavlo Tsitsei**
Editorial: **Oleksandr Tymkovych**

Block 1: $a_i \leq 1$:

Without any increments, the XOR is at most one. When can we get 0? Consider some cases when 0 is present in the array and not.

Blocks $2, 3, 4, 5$:

Refer to the solution of block 6. The solutions to these blocks still exist, with some crucial observations missed.

Block 6: without additional constraints

One can notice that $a_i := a_i + 1$ is the same operation as $a_i := a_i \oplus \Big(a_i \oplus (a_i + 1)\Big) = a_i \oplus b_i$ where $b_i = a_i \oplus (a_i + 1)$.

Namely, we replace the "+1" with "$\oplus b_i$". In the problem, we need to minimize $a_1 \oplus \cdots \oplus a_n$, hence the problem reduces to finding a set $\mathcal{S} \subseteq [n]$ that for $x = \bigoplus_{i \in \mathcal{S}} b_i$ the value $a_1 \oplus \cdots \oplus a_n \oplus x$ is minimal possible.

The next observation is that every b_i has the form of $2^k - 1$ for some k.

To find the set $\mathcal{S}$, we can do greedy, trying to cancel out the most significant bits of $A = a_1 \oplus \cdots \oplus a_n$ using elements from b. Notice that b has at most $\lceil \log_2 10^9 \rceil \approx 30$ different values, so for each bit from A we can iterate through all different elements in b to find the proper value.

Problem D1. UJGOI Contest

Author: **Pavlo Tsitsei**
Task preparation: **Pavlo Tsitsei, Oleksandr Tymkovych**
Editorial: **Pavlo Tsitsei**

Claim:

If we have two identical queries, we can work with it as with one query. This is because of operation bitwise xor. We have that $a \oplus a = 0$ and $a \oplus b \oplus a = b$. So, if we choose two identical operations, it is equal to choosing none of them. In the next blocks, we will assume that all queries are unique.

Block 1: $n \leq 5$:

Since n is up to 5, we can have up to 10 queries, so we can just try all subsets of queries if they are correct.

Block 2: $q \leq 20$:

We can try all subsets of queries. In order not to traverse the array, we can make an array b where $a_i = b_1 \oplus b_2 \oplus \cdots \oplus b_i$, and in this array, a query from l to r is equivalent to changing just b_l and b_{r+1}. So, we can count the number of ones in the array b and when we change b_i, if it is 1, we decrease this value by one and change b_i, otherwise we increase the value and change element b_i. In order for the subset to be an answer, we need that after all queries we have zero ones.

Block 3: $q \leq 40$:

Let's assume that l is the index of the first element 1 in the array. Let $r_1, r_2, ..., r_m$ be all queries of type l, r_i in ascending order. Then, we use queries l, r_1 and delete all others and add $r_1 + 1, r_i$ for all i where $i > 1$. In this way, if we choose query $r_1 + 1, r_i$, it is the same as choosing l, r_i. So, we can just do this many times until we get an array full of zeros.

Block 4: $n \leq 64$:

Let's change our initial array to array b as in the first block. Note that it can be represented as the binary representation of some number x, and queries will be just xor of two powers of 2 ($2^{r+1} \oplus 2^l$). Then, we need to make the number equal to 0. We can try to do this greedily. Let's sort the array of queries in descending order. Then, we will make a new array sequentially. We push the biggest element of the array of queries to the end of this array. For each next element (let it be x), we traverse the array changing x to $min(x, x \oplus c_i)$ where i is the i-th element of the new array. If x is not zero, we push it to the back of the array. To get the answer, we try to change n in a way we changed x, and we need to make it 0.

Block 5: All ℓ_i are different:

We can use a greedy algorithm to solve this. Let's start from the first occurrence of element 1 in the array. Obviously, we need to make it zero, and there can be only one query starting in this position (we need to take the one that starts here. Intuitively, it is because if we change the array to array b we change only 2 elements, first and last). So we do this algorithm while we have queries or while we have some 1 in the array.

Block 6: $a_i = 1$ for all i:

If we make an array b as in the second block, it looks like $100000\ldots$, and each query will change two elements, which we can express as changing the index where element 1 is. So, we can express it as a graph where we start at vertex 1 and each edge changes your position (where edge is a query). And you need to get to vertex $n+1$, which can be done using BFS.

Block 7: $n,\ q\ \leq\ 1000$:

We will make an array b as in the second block. Now, let's make a graph where the vertex set is the vertices of the array b and the edge set is queries. Note that some number of queries can form a path and if we use all queries in the path, then only the elements that will change are the first and last. So, to solve the task, we can fix one element, which is 1, and find a path to any other element which is 1. Then, take all queries, i.e. path. If there is no other element equal to 1, there is no way to make an array equal to zero.

Block 8: without additional constraints:

We can optimize the solution for block 7 by making MST on the graph and making LCA on it. In this way, we can find a path between two vertices in $O(1)$.

Also, we can try to do it differently. We will construct a graph as in block 7 and change b_{l_i} to $b_{l_i} \oplus 1$. Now, if we direct an edge from l_i to $r_i + 1$, we need to change b_{r_i+1} to the opposite and take an edge to the answer. If we direct an edge in a different direction, we change b_{l_i} to the opposite and don't take the query. Now, we need to direct all edges in a way that some vertices have odd and others even in-degree (depending on the value of b_i). To do this, we need to find the DFS tree of it and edges not from the DFS tree directly in any direction. Then, we can greedily direct all edges in the tree, starting from the leaves. Then, just check if all vertices have the desired in-degree. The solution works in $O(n)$.

Problem D2. Sakurako and a Stolen Legend

Author: **Bohdan Feysa**
Task preparation: **Bohdan Feysa**
Editorial: **Bohdan Feysa**

Block 1: $n, m \leq 80$:

We compute all SCCs in $\mathcal{O}(n^3)$ for each of the m moments of time using a simple DFS. After that, we create an array $ans[i][j]$, $1 \leq i, j \leq n$, which stores the minimal moment of time when i and j are in the same SCC. After that, we answer all q queries in $\mathcal{O}(1)$. Total time complexity: $\mathcal{O}(n^3 \cdot m + q)$.

Block 2: $n, m \leq 10^3$:

We compute all SCCs in $\mathcal{O}(n + m)$ for each of the m moments of time using Kosaraju's algorithm. After that, we create an array $ans[i][j]$, $1 \leq i, j \leq n$, which stores the minimal moment of time when i and j are in the same SCC. After that, we answer all q queries in $\mathcal{O}(1)$. Total time complexity: $\mathcal{O}((n + m) \cdot m + q)$.

Block 3: $q \leq 100$:

For each block, we use binary search for each query, in which we compute SCCs $\log n$ times in $\mathcal{O}(n + m)$ time.

This results in a total time complexity of $\mathcal{O}((n + m) \cdot q \cdot \log n)$.

Now we introduce offline incremental SCC algorithms which work in $\mathcal{O}(m \log m)$ time.

Let's use the divide and conquer technique on the list of edges. We can also think about it as divide and conquer over the timeline. So, let's assume we have a function `rec(a, b, edges)`, where a is the start of the time interval, b is the end (both inclusive), and `edges` is the list of edges along with the times when they were added. We start the process with `rec(0, m-1, everything)`.

Each time we look at the middle of the time interval, we compute $s = \lfloor \frac{a+b}{2} \rfloor$. We use a standard offline algorithm to calculate SCCs in the graph consisting only of edges from `edges` which were added at or before s. We want to split recursively while keeping good complexity.

It turns out that in the moments after s, all the SCCs we just computed will stay connected. So, we can merge whole components into single vertices and only pass to the right the edges connecting different components. The edges connecting different components were useless in the past, so we pass the rest to the left. This way, each edge appears once at every level of the recurrence. There are $\mathcal{O}(\log m)$ levels, and in each level the overall complexity is linear. Hence, we get $\mathcal{O}(m \log m)$ complexity (assuming a careful implementation with no extra logarithmic factors).

Block 4: Full solution, no tests with $t = 1$:

First, we create an additional $2 \cdot q$ edges and add them to the graph. These represent the queries. We then use the offline incremental SCC algorithm to get information about every edge and whether its endpoints are in the same SCC at any moment in time. After that, we extract the answers from the q additional edges that we added.

Time complexity: $\mathcal{O}((m + q) \log(m + q))$.

Block 5: Full online query solution:

We label each edge with the time when its endpoints are in the same SCC. (Cost: $\mathcal{O}(m \log m)$). After that, we create an undirected weighted copy of the graph, where the weights correspond to the time labels of the original edges. Then, we build a Minimum Spanning Tree (MST) using the edges of the copy. (Cost: $\mathcal{O}(m \log m)$)

Finally, we use binary lifting (binary jumps) + LCA (Lowest Common Ancestor) to find the maximum edge weight on the path from a to b in the MST. (Cost: $\mathcal{O}(n \log n)$)

Total time complexity: $\mathcal{O}(m \log m + m \log m + n \log n)$