Task idea: *Erik Sünderhauf, Marek Sokołowski*
Solutions, tests: *Jevgēnijs Vihrovs, Normunds Vilciņš*
Text: *Erik Sünderhauf, Marek Sokołowski, Jevgēnijs Vihrovs*

Joker

In this task, you are given a graph with N nodes and M edges. Furthermore, you are required to answer Q queries. In every query, all the edges from the interval $[l_i, r_i]$ are temporarily removed and you should check whether the graph contains an odd cycle or not.

Subtask 1 ($N, M, Q \leq 200$)

For every query do a DFS from every node and check if an odd cycle is formed.

Complexity: $O(QNM)$.

Subtask 2 ($N, M, Q \leq 2000$)

The graph is bipartite if and only if it contains no odd cycle. Therefore we can color the nodes with two colors (with DFS or BFS) and check if two adjacent nodes share the same color or not.

Complexity: $O(QM)$.

Subtask 3 ($l_i = 1$ for all queries)

We can sort the queries by their right endpoints (r_i) and answer them offline. Therefore we insert all the edges in the decreasing order of their indices into a DSU structure.

Complexity: $O(Q \log Q + M\alpha(N))$.

Subtask 4 ($l_i \leq 200$ for all queries)

We can sort the queries by their left endpoint (l_i) and apply our solution from Subtask 3 to all queries with the same l_i.

Complexity: $O(Q \log Q + 200M\alpha(N))$.

Subtask 5 ($Q \leq 2000$)

We use the "Mo's algorithm" technique. Split the range $[1, M]$ of edge indices into blocks of size B and sort the queries by l_i/B or by r_i if their left endpoint is in the same block. We can now keep two pointers to hold the current range of removed edges. If we answer all queries in the current block, the right pointer moves at most M steps. The left pointer moves at most QB steps in total. Since the left pointer may move to the left and the right inside the current block we need to modify our DSU to allow rollbacks. If we choose $B = M/\sqrt{Q}$ we get the following runtime:

Complexity: $O(Q \log Q + M\sqrt{Q} \log N)$.

Subtask 6 (no further constraints)

For any $1 \leq l \leq M$, let last$[l]$ be the last index r such that the answer for the query $[l, r]$ is positive (or $M+1$ if no such index exists). That is, the graph exluding the edges $[l, r]$ is bipartite, but the graph excluding the edges $[l, r-1]$ is not. We can prove that if $l_1 < l_2$, then last$[l_1] \leq$ last$[l_2]$. We will exploit this fact in order to compute the array last using a divide & conquer algorithm.

We write a recursive function rec, which takes two intervals: $[l_1, l_2], [r_1, r_2]$ ($1 \leq l_1 \leq l_2 \leq M$, $1 \leq r_1 \leq r_2 \leq M+1$), possibly intersecting, but $l_1 \leq r_1$ and $l_2 \leq r_2$. This function will compute last$[l]$

for each $l \in [l_1, l_2]$, assuming that for these values of l, we have $\text{last}[l] \in [r_1, r_2]$. We will initially call $\text{rec}([1, M], [1, M+1])$.

We assume that when $\text{rec}([l_1, l_2], [r_1, r_2])$ is called, then our DSU contains all edges with indices to the left of l_1 and to the right of r_2. For instance, when $M = 9$ and $\text{rec}([2, 5], [3, 7])$ is called, then edges with indices 1, 8, and 9 should be in the DSU. We also assume that the graph in the DSU is bipartite.

We take $l_{\text{mid}} := (l_1 + l_2)/2$ and compute $\text{last}[l_{\text{mid}}]$; this can be done by adding all edges $[l_1, l_{\text{mid}} - 1]$ to the DSU, and then trying to add all edges with indices $r_2, r_2 - 1, r_2 - 2, \ldots$, until we try to add an edge breaking the bipartiteness. The index r_{mid} of such an edge is exactly $\text{last}[l_{\text{mid}}]$. We then rollback all edges added so far in our recursive call.

Now that we know that $\text{last}[l_{\text{mid}}] = r_{\text{mid}}$, we will run two recursive calls:

- For each $l \in [l_1, l_{\text{mid}} - 1]$, we know that $\text{last}[l] \in [r_1, r_{\text{mid}}]$. We run $\text{rec}([l_1, l_{\text{mid}} - 1], [r_1, r_{\text{mid}}])$, remembering to add all necessary edges to the right of r_{mid}. After the recursive call, we rollback them.
- For each $l \in [l_{\text{mid}} + 1, l_2]$, we know that $\text{last}[l] \in [r_{\text{mid}}, r_2]$. We run $\text{rec}([l_{\text{mid}} + 1, l_2], [\max(l_{\text{mid}} + 1, r_{\text{mid}}), r_2])$, now adding all necessary edges to the left of $l_{\text{mid}} + 1$. We rollback them after the recursive call.

We can see that each recursive call uses at most $O((l_2 - l_1) + (r_2 - r_1))$ edge additions and rollbacks in DSU, each taking $O(\log N)$ time pessimistically. Also, on each level of recursion, the total length of all intervals is bounded by $2M$. Hence, all intervals in all recursive calls have total length bounded by $O(M \log M)$. Hence, the total time of the preprocessing is $O(M \log M \log N)$.

With our preprocessing, each query $[l, r]$ reduces to simply verifying $r \leq \text{last}[l]$, which can be done in constant time.

Complexity: $O(M \log M \log N + Q)$.