Solution — The Great Game

1 Modelling

We are given a labelled *directed acyclic graph* (DAG for short) G on the vertex set $[n] = \{1, \dots, n\}$ with m directed edges (from now on just edges). Moreover, the graph is *topologically sorted*, that is every edge (i, j) is such that $i < j$. In addition, we are given a red and a black meeple, initially lying at vertices r_{start} and b_{start} of the graph, respectively.

Two players, Sherlock and Moriarty, play a game on such a graph as follows. Every time it is Sherlock's turn to play, he moves the red meeple if he has played an even number of times so far, and otherwise moves the black meeple. Similarly, every time it is Moriarty's turn to play, he moves the black meeple if he has played an even number of times so far, and otherwise moves the red meeple. All movements of any of the meeples must go along one of the edges (in the direction the edge is oriented!) of the graph G. The game ends as soon as either of the meeples reaches the *target position*, i.e. the vertex n. If this happens to be the red meeple, then Sherlock wins; otherwise Moriarty wins. We say that a player has a *winning strategy* if he wins the game regardless of the other player's strategy. The problem asks for you to decide which player has a winning strategy.

More formally, Let S and M be two functions $S, M \colon \{0, 1\} \to [n]$ defined as

$$S(t) = \begin{cases} r, & \text{if } t = 0, \\ b, & \text{otherwise,} \end{cases} \qquad \text{and} \qquad M(t) = \begin{cases} b, & \text{if } t = 0, \\ r, & \text{otherwise,} \end{cases}$$

where r and b denote the current position of the red and black meeple, respectively. Sherlock (resp. Moriarty) on his turn moves the meeple at vertex $S(t)$ (resp. $M(t)$) to one of the possible vertices $N(S(t))$ (resp. $N(M(t))$), where for a vertex $i \in V(G)$ we let $N(i)$ be the set of all vertices j such that $(i, j) \in E(G)$. The game ends as soon as any of the players makes either one of the moves $r \to n$ or $b \to n$. With the former Sherlock wins, while with the latter Moriarty wins.

Before we continue, let us introduce some game theoretic concepts which prove to be useful in the analysis of the game. A *game tree* is a directed graph whose vertices are positions in a game and edges are legal moves. In our case, the vertices of the game tree are the possible positions of the red and black meeple, i.e. all pairs $(i, j) \in V(G) \times V(G)$, and the edges are the possible 'relocations' of the meeples, for example, $(i, j_1) \to (i, j_2)$ is an edge of the game tree if and only if $(j_1, j_2) \in E(G)$ and the black meeple should be relocated in the next move (i.e. it is either Sherlock's even turn or Moriarty's odd turn). Let (r, b, p, t) be a 4-tuple defining a *game state* as follows:

- $r \in [n]$ denotes the current position of the red meeple,
- $b \in [n]$ denotes the current position of the black meeple,
- $p \in \{0, 1\}$ is a binary variable denoting whose turn it is to play (say, 0 for Sherlock and 1 for Moriarty), and

- $t \in \{0, 1\}$ denotes the parity of the number of moves the current player has made up until this turn.

Clearly, the leaves in such a tree are the ending positions and the number of leaves represents the number of possible ways the game can be played out.

2 Algorithm Design

By looking at the constraints we see that the number of vertices is at most 50,000 and the number of edges is at most 50,000. This suggests that in order to get 100 points, an $O(n \cdot m)$ solution is certainly too slow. In the first two test sets we are guaranteed that n and m are at most 50 and 100, respectively. Furthermore, we know that there is a *unique* path from r_{start} to n and at most *two* or *six* paths from b_{start} to n, respectively. This means that we might want to try an approach which in general would not be feasible, but makes use of the above mentioned constraints in order to get at least 50 points.

$O(n!)$ solution (50 points). *This is an $O(n)$ solution for the first two test sets! What makes this solution 'efficient' is the fact that there is a unique path from r_{start} to n and at most two (six) different paths from b_{start} to n.*

The first and simplest idea is a brute-force algorithm—evaluate the whole game tree. Let us briefly describe the strategy. Assume for simplicity it is Sherlock's turn to play next (Moriarty's turn proceeds analogously). Depending on the value of t, where t is the parity of the number of times Sherlock has played up until this turn (not including this turn!), we iterate over all vertices $j \in N(S(t))$ and move the meeple located at the vertex $S(t)$ to the vertex j. For every such move $S(t) \to j$ of Sherlock, we try all possible 'answers' of Moriarty. If there exists one such move with the property that regardless of what Moriarty plays Sherlock eventually wins, then Sherlock has a *winning strategy*. The algorithm is given by the following pseudo-code.

```
sherlock(int r, int b, int t) {
    if r == n return "Sherlock"
    if b == n return "Moriarty"

    for every j in N(S(t)):
        make the move S(t) → j
        if moriarty(new_r, new_b, t) == "Sherlock":
            return "Sherlock"

    return "Moriarty"
}

moriarty(int r, int b, int t) {
    if r == n return "Sherlock"
    if b == n return "Moriarty"

    for every j in N(M(t)):
        make the move M(t) → j
        if sherlock(new_r, new_b, 1 - t) == "Moriarty":
            return "Moriarty"

    return "Sherlock"
}
```

The correctness of the algorithm is immediate, as we evaluate all the possible moves of both players for every state of the game. It remains to see why it meets the time-complexity requirement as well. As we evaluate the whole game tree, in order to give an upper bound on the running time of the algorithm it suffices to bound its size. Note that from any starting position $(r_{\text{start}}, b_{\text{start}})$ it takes at most $2n$ steps to reach the target vertex n. Hence, every path from the root of the game tree to any of the leaves is of length $O(n)$. Moreover, since there is a unique path from r_{start} to n, at most six paths from b_{start} to n, and the target position is reachable from every position via a sequence of transitions, it follows that there are at most six different ways the game can be played out. Recall, this implies that there are at most six leaves in the game tree. In conclusion, there are at most $O(n)$ vertices in the game tree and thus the algorithm from above runs in time $O(n)$ for the first two test sets.

The bottleneck of this approach is that $N(r)$ or $N(b)$ may be as large as n in the general case. More precisely, in the i-th move of a player $|N(r)|, |N(b)| \leqslant n - i$, which follows from the fact that the graph is topologically sorted. In the i-th call of a player's function, this results in possible $O(n - i)$ calls of the other player's function inside of the `for` loop, leading to an $O(n!)$ algorithm.

$O(n^2)$ solution (75 points). Observe that for a fixed game state (r, b, p, t) as defined in Section 1, we evaluate the winner starting from that state potentially many times. Therefore, we have an *overlapping subproblem structure*, which naturally leads to a *dynamic programming* approach. In order to improve the above brute-force method into a solution which achieves 75 points, we make use of dynamic programming with *memoization* as presented in the tutorial (see the slides from Week 2).

Recall that for the first three test sets we have $n \leqslant 2{,}000$, and thus an $O(n^2)$ solution seems to be reasonable. Let `DP[MaxN][MaxN][2][2]` be a four-dimensional array which we use for memoization. Namely, the value `DP[r][b][p][t]` represents the game state (r, b, p, t) as from above. With this notion at hand, the previous solution can straightforwardly be modified in order to accomplish our goal. All values of the array `DP` are initially set to -1. The algorithm is given by the following pseudo-code.

```
sherlock(int r, int b, int t) {
    if r == n return "Sherlock"
    if b == n return "Moriarty"
    if DP[r][b][0][t] ≠ -1 return DP[r][b][0][t]

    DP[r][b][0][t] = "Moriarty"
    for every j in N(S(t)):
        make the move S(t) → j
        if moriarty(new_r, new_b, t) == "Sherlock":
            DP[r][b][0][t] = "Sherlock"

    return DP[r][b][0][t]
}

moriarty(int r, int b, int t) {
    if r == n return "Sherlock"
    if b == n return "Moriarty"
    if DP[r][b][1][t] ≠ -1 return DP[r][b][1][t]

    DP[r][b][1][t] = "Sherlock"
    for every j in N(M(t)):
        make the move M(t) → j
        if sherlock(new_r, new_b, 1 - t) == "Moriarty":
            DP[r][b][1][t] = "Moriarty"

    return DP[r][b][1][t]
}
```

Since there are at most $n \cdot n \cdot 2 \cdot 2$ game states and for every state we determine the winner at most once, in constant time per state, it follows that the algorithm runs in time $O(n^2)$.

Remark: Note that this approach is not feasible for the general case not only from a time-complexity perspective, but also due to its space-complexity. In order to store all the states of the game we potentially need to allocate

$$\underbrace{4}_{p,t\in\{0,1\}} \quad \cdot \quad \underbrace{25 \cdot 10^8}_{n \leqslant 5\cdot 10^4} \text{ bits} \approx 1.25 \text{ gigabytes of memory.}$$

$O(n)$ solution (100 points). In order to get the full 100 points we require a new strategy. The first observation is that the meeples are *independent of each other*, that is the current position of one meeple does not affect in any way the number of moves it takes to reach the target position by playing only with the other meeple. This leads to the second and crucial observation: for a fixed meeple the game can be converted into the following Mini-Maxi game. There is *one meeple* and *two players*, Mini and Maxi, who take turns in moving the meeple. The goal of Mini is to reach the target position as quickly as possible, while Maxi wants to prolong the game for as long as possible. With this in mind, we are ready to derive the recursive formula for the minimum number of turns it takes to end the game.

Let $\text{mini}(i)$ and $\text{maxi}(i)$ denote the number of moves it takes for Mini and Maxi to finish the game starting at position i, respectively. Clearly, both $\text{mini}(n)$ and $\text{maxi}(n)$ are equal to 0 as the game is over once any of them reaches the position n. This provides us with a base case. On the other hand, for any $0 \leqslant i \leqslant n-1$, Mini needs a minimum over all possible moves $i \to j$ of $\text{maxi}(j)+1$ (where the $+1$ comes from making this particular $i \to j$ move), and Maxi likewise. Therefore, the recursion can be set up as follows:

$$\begin{aligned} \text{mini}(n) &= 0, \\ \text{mini}(i) &= \min_{j\in N(i)} \text{maxi}(j)+1, \end{aligned} \qquad \text{and} \qquad \begin{aligned} \text{maxi}(n) &= 0, \\ \text{maxi}(i) &= \max_{j\in N(i)} \text{mini}(j)+1. \end{aligned}$$

Observe that the recursion is well-defined as all $j \in N(i)$ are such that $j > i$ by the assumptions of the game, and have thus already been computed.

Let us now look back at our original game. By the observations from above that the two meeples are independent, we essentially play two parallel instances of the Mini-Maxi game:

(1) in the first we play with the red meeple where Sherlock is Mini and Moriarty is Maxi;
(2) in the second we play with the black meeple where Sherlock is Maxi and Moriarty is Mini.

By precomputing how many turns it takes in the Mini-Maxi game from every starting position i, we can now decide the winner of our original game by comparing the values of $\text{mini}(r)$ and $\text{mini}(b)$. If $\text{mini}(r) < \text{mini}(b)$ then Sherlock has a winning strategy, while if $\text{mini}(b) < \text{mini}(r)$ then Moriarty has a winning strategy. However, in case $\text{mini}(r) = \text{mini}(b)$, Sherlock wins if and only if $\text{mini}(r) \equiv 1 \pmod 2$. Note that this holds as by the definition of the game, the sequence of meeple moves is: red $\rightarrow$ black $\rightarrow$ black $\rightarrow$ red $\rightarrow$ red $\rightarrow \ldots$

It remains to determine how long it takes to compute the number of turns it takes to finish the Mini-Maxi game from every starting position $i \in [n]$. However, this is easily seen to be $O(n + m)$: for every starting position $i \in [n]$, and every edge (i, j) for all $j \in N(i)$, we can in constant time update the values $\text{mini}(i)$ and $\text{maxi}(i)$ for the move $i \rightarrow j$; since there are m edges in the graph and each edge is used at most once it follows that the number of steps it takes to compute the winner of the Mini-Maxi game is $O(n + m)$.

3 Implementation

The implementation is straightforward and follows the bottom-up dynamic programming approach presented in the tutorial (see the slides from Week 2). Let us remark that a recursive top-down approach is also a viable option. Nonetheless, as we presented such an algorithm in pseudo-code in the previous two sections, we resort to a bottom-up approach here for the sake of diversity.

The game graph is stored as an adjacency list, which is a two-dimensional vector of length $n + 1$. Furthermore, we use two vectors of length $n + 1$ each, denoted by `mini` and `maxi`, respectively, where the i-th index denotes the number of moves it takes to finish the game after Mini (and Maxi, respectively) makes a move starting at position i. The vector `mini` is initialised to `Max_N + 1`, where `Max_N` is the maximum possible number of vertices. The vector `maxi` is initialised to -1. We set both `mini(n) = 0` and `maxi(n) = 0` as the game ends once a player is at position n. To compute the remaining values in a recursive fashion, we use a `for` loop from $n - 1$ down to $\min(r, b)$.

4 Appendix

The following code is an implementation of the $O(n+m)$ solution using the bottom-up dynamic programming approach.

```cpp
#include <iostream>
#include <vector>

const int MAX_N = 50001;

typedef std::vector<int> VI;

int testcase() {
    int n, m; std::cin >> n >> m; // Reading the number of vertices and edges
    int r, b; std::cin >> r >> b; // Reading the starting positions of the meeples
    std::vector<VI> graph(n + 1, VI()); // Adjacency list
    for (int i = 0; i < m; ++i) {
        int u, v; std::cin >> u >> v; // Reading an edge
        graph[u].push_back(v); // Store the edge
    }

    VI mini(n + 1, MAX_N);
    VI maxi(n + 1, -1);

    // Set up the starting values of the recursion
    mini[n] = 0;
    maxi[n] = 0;

    // Do the recursive calculations
    for (int i = n - 1; i >= 1; --i) {
        // For all the possible moves from the current position
        for (int v : graph[i]) {
            mini[i] = std::min(mini[i], maxi[v] + 1); // Mini makes the move
            maxi[i] = std::max(maxi[i], mini[v] + 1); // Maxi makes the move
        }
    }

    int sherlock = mini[r]; // Sherlock needs mini[r] moves to finish the game
    int moriarty = mini[b]; // Moriarty needs mini[b] moves to finish the game

    // Sherlock needs fewer moves
    if (sherlock < moriarty) return 0;
    // Moriarty needs fewer moves
    if (moriarty < sherlock) return 1;
    // Both need the same number of moves but Sherlock wins if and only if
    // the number of moves is not divisible by 2, as the sequence of moves is:
    // r --> b --> b --> r --> r --> b --> ...
    // s --> m --> s --> m --> s --> m --> ...
    if (sherlock % 2 == 1) return 0;
    return 1;
}

int main() {
    std::ios_base::sync_with_stdio(false);

    int t; std::cin >> t;
    for (int i = 0; i < t; ++i)
        std::cout << testcase() << "\n";
}
```