```
#include <iostream>

#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

/** Helper function for debugging that prints the vector of ints. */
void print_vector(vector<int>& v) {
    for (int elem : v) {
        cout << elem << " ";
    }
    cout << endl;
}

/** Returns transitions read from the standard input. */
vector<vector<int>> read_transitions(int n, int m) {
    vector<vector<int>> transitions(n, vector<int>());
    for (int i = 0; i < m; i++) {
        int source, target;
        cin >> source;
        cin >> target;
        transitions[source - 1].push_back(target - 1);
    }

    return transitions;
}

void make_optimal_step(int, vector<int>&, vector<int>&,
vector<vector<int>>&);
void make_adversary_step(int, vector<int>&, vector<int>&,
vector<vector<int>>&);

void make_optimal_step(int position, vector<int>& optimal,
vector<int>& adversary, vector<vector<int>>& transitions) {
    int board_size = optimal.size();

    // We have reached target position.
    if (position == board_size - 1) {
        optimal[position] = 0;
        return;
    }

    // Initialize with value that is knows to be greater than eventual
min_.
    int min_ = board_size + 1;
    for (int target : transitions[position]) {
        if (adversary[target] == -1) {
            // The value is not yet computed.
```

```
            make_adversary_step(target, optimal, adversary,
transitions);
        }
        min_ = min(min_, adversary[target]);
    }

    optimal[position] = min_ + 1;
}

void make_adversary_step(int position, vector<int>& optimal,
vector<int>& adversary, vector<vector<int>>& transitions) {
    int board_size = optimal.size();

    // We have reached target position.
    if (position == board_size - 1) {
        adversary[position] = 0;
        return;
    }

    // Initialize with value that is known to be less than eventual
max_.
    int max_ = -1;
    for (int target : transitions[position]) {
        if (optimal[target] == -1) {
            make_optimal_step(target, optimal, adversary,
transitions);
        }
        max_ = max(max_, optimal[target]);
    }

    adversary[position] = max_ + 1;
}

void testcase() {
    // Size of the board and number of transitions.
    int board_size, transitions_num;
    cin >> board_size;
    cin >> transitions_num;
    // Initial position for red and black meeple.
    int red_position, black_position;
    cin >> red_position;
    cin >> black_position;
    // Make positions to be zero-based.
    red_position -= 1;
    black_position -= 1;
    vector<vector<int>> transitions = read_transitions(board_size,
transitions_num);

    // "optimal_{r, b}"/"adversary_{r, b}" is memoization vector that
contains
```

```
    // the minimum/maximum number of moves needed to reach the target
position
    // from the current index.

    vector<int> optimal_r(board_size, -1);
    vector<int> adversary_r(board_size, -1);
    make_optimal_step(red_position, optimal_r, adversary_r,
transitions);

    vector<int> optimal_b(board_size, -1);
    vector<int> adversary_b(board_size, -1);
    make_optimal_step(black_position, optimal_b, adversary_b,
transitions);

    if (optimal_r[red_position] < optimal_b[black_position]) {
        // Red meeple reaches the target in less moves, so Sherlock
wins.
        cout << 0 << endl;
    } else if (optimal_r[red_position] > optimal_b[black_position]) {
        // Black meeple reaches the target in less moves, so Moriarty
wins.
        cout << 1 << endl;
    } else {
        // Read and black meeple reach reach target in the same amount
of moves,
        // so however makes the last move first - wins.
        if (optimal_r[red_position] % 2 == 1) {
            cout << 0 << endl;
        } else {
            cout << 1 << endl;
        }
    }
}

int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        testcase();
    }
    return 0;
}
```