```java
import java.util.*;

public class Lost {
    //Types of places in the island
    public final static int GRASS = 0;
    public final static int OBSTACLE = 1;
    public final static int WATER = 2;
    public final static int MAGIC_WHEEL = 3;
    public final static int EXIT = 4;
    //No paths constants
    public static final int UNREACHABLE = Integer.MAX_VALUE;
    public static final int LOST_IN_TIME = Integer.MIN_VALUE;
    //Type of players
    public static final int CAN_SWIM = 0;
    public static final int CAN_USE_WHEEL = 1;
    //Cost of paths from cells in island
    private final static int WATER_COST = 2;
    private final static int GRASS_COST = 1;
    //Paths structure - way the edge is represented
    private static final int START_PLACE = 0;
    private static final int END_PLACE = 1;
    private static final int PLACE_COST = 2;
    /**
     * Types of cells in which island's positions
     */
    private final int[][] island;
    /**
     * Paths from and to a Grass cell
     */
    private final List<int[]> normalPaths;
    /**
     * Paths to or from a water cell
     */
    private final List<int[]> waterPaths;
    /**
     * Paths from the magic wheel
     */
    private final List<int[]> magicWheelPaths;

    /**
     * All magic wheels. Key is the number presented and value is the magic wheel codification
     * position (from 0 to numRows*numCols-1, the total number of cells)
     */
    private final Map<Integer, Integer> magicWheels;
    /**
     * All cells of the island. key is the (x,y) value, in integer form, and the value is the
     * codification position (from 0 to numRows*numCols-1, the total number of cells)
     */
    private final Map<Integer, Integer> places;
    private final int numRows;
    private final int numCols;
    private int numPlaces;
    /**
     * Codification of the exit cell
     */
    private int EXIT_POS;

    public Lost(int numRows, int numCols, int numMagicWheels) {
        this.numRows = numRows;
        this.numCols = numCols;

        normalPaths = new LinkedList<>();
        waterPaths = new LinkedList<>();
        magicWheelPaths = new LinkedList<>();

        island = new int[numRows][numCols];
        magicWheels = new HashMap<>(numMagicWheels);
        places = new HashMap<>(numRows * numCols);
        numPlaces = 0;

    }

```

```java
    /**
     * Adds the edges associated with a given position. The graph edges are separated by type,
     * the edges will the added to the correspondent Type List.
     *
     * @param x          x position to the island's cell to evaluate
     * @param y          y position to the island's cell to evaluate
     * @param type       type of island's cell to evaluate
     * @param magicWheel if the cell is a magic island this is the number presented in the grid
     */
    public void addIslandPosition(int x, int y, int type, int magicWheel) {
        island[y][x] = type;
        int start = posToInt(x, y);
        int pos = numPlaces++;
        places.put(start, pos);
        if (y > 0 && type != OBSTACLE) { //has up
            //if the current cell has an upper cell, add the 2 edges to and from the current cell
            int upType = island[y - 1][x];
            int upPos = places.get(posToInt(x, y - 1));
            addPaths(type, pos, upType, upPos);
        }
        if (x > 0 && type != OBSTACLE) { //has left
            //if the current cell has a left cell, add the 2 edges to and from the current cell
            int leftType = island[y][x - 1];
            int leftPos = places.get(posToInt(x - 1, y));
            addPaths(type, pos, leftType, leftPos);
        }

        if (type == MAGIC_WHEEL) {
            //if it is a magic wheel store the number presented in the grid and its codification
            // position
            magicWheels.put(magicWheel, pos);
        }
        if (type == EXIT) {
            //store the codification position of the exit
            EXIT_POS = pos;
        }
    }

    /**
     * Converts an (x,y) position to an int
     *
     * @param x x position
     * @param y y  position
     * @return transformed (x,y)  position to an int
     */
    private int posToInt(int x, int y) {
        return x * 100 + y;
    }

    /**
     * Adds the edges between two vertices to the correspondent types list
     *
     * @param startType type of the start position
     * @param startPos  codified start position
     * @param endType   type of the end position
     * @param endPos    codified end position
     */
    private void addPaths(int startType, int startPos, int endType, int endPos) {
        int[] edgeFrom = new int[]{startPos, endPos, cost(startType)};
        int[] edgeTo = new int[]{endPos, startPos, cost(endType)};

        if (startType == WATER || endType == WATER) {
            //if is a path connected to a water cell and is not from the exit
            if (startType != EXIT) {
                waterPaths.add(edgeFrom);
            }
            if (endType != EXIT) {
                waterPaths.add(edgeTo);
            }
        } else if (endType != OBSTACLE) {
            //if is not an obstacle add path to normal paths, bidirectional
            if (startType != EXIT) {
```

```
                normalPaths.add(edgeFrom);
            }
            if (endType != EXIT) {
                normalPaths.add(edgeTo);
            }
        }
    }

    /**
     * Computes the cost of exiting a cell
     *
     * @param type type of the cell to exit
     * @return the cost
     */
    private int cost(int type) {
        return type == GRASS || type == MAGIC_WHEEL ? GRASS_COST : WATER_COST;
    }

    /**
     * Adds the edges from a magic wheel
     *
     * @param i    magic wheel that is the start of the edge
     * @param x    x position of the end of the edge
     * @param y    y position of the end of the edge
     * @param cost cost of the edge of the magic wheel
     */
    public void addMagicWheel(int i, int x, int y, int cost) {
        int start = magicWheels.get(i);
        int end = places.get(posToInt(x, y));
        int[] edge = new int[]{start, end, cost};
        magicWheelPaths.add(edge);

    }

    /**
     * Computes the length of the path between a player's position to the exit.
     * Considers the type of player, if he can swim or use the wheel
     *
     * @param originX    x start position of the player
     * @param originY    y start position of the player
     * @param playerType boolean array with the type player. If he can swim in the first position
     *                   and if he can use the magic wheel in the second position
     * @return the length of the path from the player's initial position to the exit. If the exit
     * is unreachable returns INTEGER.MAX_VALUE. If the graph has a negative weight cycle
     * reachable by the player returns INTEGER.MIN_VALUE
     */
    public int solution(int originX, int originY, boolean[] playerType) {
        int[] lengths = new int[numRows * numCols];

        Arrays.fill(lengths, UNREACHABLE);

        int origin = places.get(posToInt(originX, originY));
        lengths[origin] = 0;
        boolean changes = false;
        for (int i = 1; i < lengths.length; i++) {
            changes = updateLength(lengths, playerType);
            if (!changes) {
                // length vector stabilized, end cycle
                break;
            }
        }

        //Detect negative-weight cycles
        if (changes && updateLength(lengths, playerType)) {
            lengths[EXIT_POS] = LOST_IN_TIME;
        }

        return lengths[EXIT_POS];
    }

    private boolean updateLength(int[] lengths, boolean[] playerType) {
        //Iterates all edges in the graph by types. The normal edges are always considered
```

```java
        boolean changes = updateLengthsInSubPaths(lengths, normalPaths);
        if (playerType[CAN_SWIM]) {
            //iterated only if the player can swim
            changes = updateLengthsInSubPaths(lengths, waterPaths) || changes;
        }
        if (playerType[CAN_USE_WHEEL] && magicWheels.size() > 0) {
            //iterated only if the player can use the magic wheels
            changes = updateLengthsInSubPaths(lengths, magicWheelPaths) || changes;
        }
        return changes;
    }

    /**
     * Performs the update length of the algorithm
     * @param lengths array of lengths used by Bellman-Ford algorithm
     * @param paths list of the paths to consider
     * @return if there are any changes in the vector lengths
     */
    private boolean updateLengthsInSubPaths(int[] lengths, List<int[]> paths) {
        boolean changes = false;
        for (int[] path : paths) {
            int tail = path[START_PLACE];
            int head = path[END_PLACE];
            int cost = path[PLACE_COST];
            if (lengths[tail] < Integer.MAX_VALUE) {
                int newCost = lengths[tail] + cost;
                if (newCost < lengths[head]) {
                    lengths[head] = newCost;
                    //continue cycle because there are changes in the length vector
                    changes = true;
                }
            }
        }
        return changes;
    }
}

```

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {

    public static void main(String[] args) throws IOException {

        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        int numTestCases = Integer.parseInt(input.readLine());
        for (int i = 0; i < numTestCases; i++) {
            Lost problem = processProblem(input);
            String[] playersPositions = input.readLine().split(" ");
            int yJ = Integer.parseInt(playersPositions[0]);
            int xJ = Integer.parseInt(playersPositions[1]);
            int yK = Integer.parseInt(playersPositions[2]);
            int xK = Integer.parseInt(playersPositions[3]);
            int solutionJohn = problem.solution(xJ, yJ, new boolean[]{false, true});
            int solutionKate = problem.solution(xK, yK, new boolean[]{true, false});
            System.out.println("Case #" + (i+1));
            String john = solutionString(solutionJohn);
            String kate = solutionString(solutionKate);
            System.out.println("John " + john);
            System.out.println("Kate " + kate);
        }


    }

    private static Lost processProblem(BufferedReader input) throws IOException {
        String[] problemInfo = input.readLine().split(" ");
        int numRows = Integer.parseInt(problemInfo[0]);
        int numCols = Integer.parseInt(problemInfo[1]);
        int numMagicWheels = Integer.parseInt(problemInfo[2]);
        Lost problem = new Lost(numRows, numCols, numMagicWheels);
        for (int y = 0; y < numRows; y++) {
            String row = input.readLine();
            for (int x = 0; x < numCols; x++) {
                char charAt = row.charAt(x);
                int type = positionType(charAt);
                int w = -1;
                if(type == Lost.MAGIC_WHEEL){
                    w = Integer.parseInt(String.valueOf(charAt));
                }
                problem.addIslandPosition(x, y, type, w);
            }
        }
        for (int i = 1; i <= numMagicWheels; i++) {
            String[] magicWheel = input.readLine().split(" ");
            int y = Integer.parseInt(magicWheel[0]);
            int x = Integer.parseInt(magicWheel[1]);
            int cost = Integer.parseInt(magicWheel[2]);
            problem.addMagicWheel(i, x, y, cost);
        }

        return problem;
    }

    private static String solutionString(int solutionJohn) {
        String string;
        if (solutionJohn == Lost.UNREACHABLE) {
            string = "Unreachable";

        } else if (solutionJohn == Lost.LOST_IN_TIME) {
            string = "Lost in Time";
        } else {
            string = String.valueOf(solutionJohn);
        }
        return string;
    }

    private static int positionType(char charAt) {
```

```
        int type;
        if (charAt == 'G') {
            type = Lost.GRASS;
        } else if (charAt == 'O') {
            type = Lost.OBSTACLE;
        } else if (charAt == 'W') {
            type = Lost.WATER;
        } else if (charAt == 'X') {
            type = Lost.EXIT;
        } else {
            type = Lost.MAGIC_WHEEL;
        }
        return type;
    }
}

```