Nokia Snake

Nokia is re-releasing its classic Nokia 3310 phone, which some of you might remember as the nostalgic brick phone. Not only did the old brick phone last very long in terms of battery life and the actual lifespan of the phone, it also included the classic snake game which many people were obsessed with.

You have recently been hired into Nokia and you have been given the task of coding a new and upgraded version of the snake game to be released with the new Nokia 3310 phones. In this new version, the game will still be played on a grid of R x C squares. However, the board will start with multiple pieces of food which can only be eaten once, and new food will not spawn when an existing food is eaten.

The player will start at a square as a snake of length 1. In every move, the snake will move forward one square depending on where the snake is facing. Initially the snake is facing up. The player can change the direction of facing to one of 4 directions: up, down, left or right. The player can also change the direction multiple times between 2 moves. When the snake eats a piece of food, the snake will increase in length by 1. Refer to the sample for more details.

The snake can warp through the grid's edges. For example, if it tries to go into a square above the grid, it will warp to the bottom square in the same column. If the snake tries to move into itself, it will die and the game ends. If the snake dies, your program should end and ignore the rest of the queries. As a programmer and a brilliant thinker, you now need to write a program to simulate the snake game.

Input

The first line of the input consists of two integers, **R, C (1 <= R, C <= 1000)**, the number of rows and the number of columns on the board respectively.

The next line of the input consists of a single integer, **N (0 <= N <= 500000)**, the number of pieces of food on the board.

It is then followed by **N** lines, each consisting of two integers **X, Y**, representing the coordinate of a piece of food at **(X,Y) (0 <= X < C, 0 <= Y < R)** where the top left hand corner of the grid has coordinate **(0,0)**, and bottom right hand corner has coordinate **(C-1, R-1)**. No two pieces of food will be at the same coordinate and there will not be a piece of food at the starting square of the snake.

The next line of input consists of two space-separated integers **X and Y** representing the coordinate of the starting square of the snake at **(X,Y)**, The next line of input consists of a single integer, **Q (1 <= Q <= 100000)**, the number of queries to answer. Q rows follow, containing one query each. The queries follow the following specification:

Query Type Input Format: `<QUERY_TYPE> <APPROPRIATE_PARAMETERS>`

1. `move STEPS`
   Move the snake STEPS number of steps in the current direction of the snake. If the snake dies at any of the steps by moving into itself, print "**Game Over! Score: SCORE**" where score is the number of pieces of food eaten, and then ignore the rest of the input. Otherwise, print "Snake is of length **LENGTH** with head **(X,Y)** and tail **(X,Y)**". Replace the bolded parts accordingly. It is possible that the snake does not die until the end of the game. In that case, "Game Over!" will **NOT** be printed. However, the total number of steps made by the snake will not exceed **500000**.

2. `change DIRECTION`
   The player changes the direction of the snake to DIRECTION. DIRECTION will either be "up", "down", "left" or "right". Print "Direction changed to **DIRECTION**" replacing with the new direction accordingly. Take note that the snake **cannot go backwards** on itself if it has grown in length, so **if the snake's length is more than 1, ignore the change** in direction if it is opposite to the direction of the last movement and **don't print anything.** Also, ignore the change if it is the same as the current direction.

Output

Print the result of the query as described in the input format above. The last line of the output **must contain a newline character**. In the sample output below, some lines are left empty for better clarity of the sample. **No blank lines are to be printed in the actual output.**

Sample Input	Sample Output

`3 3`
`4`
`0 2`
`2 0`
`0 0`
`1 1`
`2 2`
`20`	

`move 1`	`Snake is of length 1 with head (2,1) and tail (2,1)`
`change up`	
`change down`	`Direction changed to down`
`move 2`	`Snake is of length 2 with head (2,0) and tail (2,2)`
`change right`	`Direction changed to right`
`move 1`	`Snake is of length 3 with head (0,0) and tail (2,2)`
`change down`	`Direction changed to down`
`change left`	
`change up`	`Direction changed to up`
`move 1`	`Snake is of length 4 with head (0,2) and tail (2,2)`
`change right`	`Direction changed to right`
`change up`	`Direction changed to up`
`change left`	`Direction changed to left`
`move 2`	`Snake is of length 4 with head (1,2) and tail (0,0)`
`change left`	
`change right`	
`move 1`	`Game Over! Score: 3`
`change down`	
`move 10`	
`move 20`	

Explanation

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

The next set of grids will illustrate the moves according to the sample input shown above. The coordinate system is as shown on the left. In the grids below, 'F' will represent a piece of food and the snake will be represented by numbers from 1 to L where 1 is the head and L is the tail where L is the length of the snake.

F		F
	F	
F		1

This is the starting configuration of the grid. There are 4 pieces of food and the snake starts at square (2,2), initially facing upwards.

F		F
	F	1
F		

The snake moves upwards by 1 square so its head and tail are at (2,1). The change of direction to up is ignored since the snake is already facing upwards, but the change of direction to down is made since the snake is only of length 1 so it can change to face downwards.

F		1
	F	
F		2

The snake now moves 2 squares down. After the first move, its head and tail are both at (2,2). In the second move, the head warps to (2,0) and since a food is eaten, the snake extends to a length of 2 and its tail is left at (2,2). It then changes its direction to face right.

1		2
	F	
F		3

The snake now moves right and warps to (0,0) and extends to a length of 3 since it eats another piece of food. So its head is at (0,0) but tail is still at (2,2). It then makes 3 directional changes. The second change is ignored since the previous movement was right so the snake can't move left or else the head will go directly back into its body. The final change makes it face up.

2		3
	F	
1		4

The snake now moves up and warps to (0,2), extending again to length 4. It then makes 3 directional changes, all of which are valid since the direction changes from originally up to right then to up then to left and there are no consecutive repeats. The final direction is left.

4		
	F	
3	1	2

The snake now moves two spaces and the tail follows along. In the first move, the snake does not die because the tail moves away from (2,2) at the same time the head moves into (2,2) so it's not considered going into itself. In the second move, the head moves into (1,2) and the tail moves from (2,0) to (0,0). Thus, the output is as shown above. The next two changes are ignored since the snake is already moving left so it ignores a change to moving left, and it ignores a change to moving right since it cannot move right. The snake then dies on the next turn since it will move into its body at (0,2). Thus, we print "Game Over! Score: 3" since it has eaten 3 pieces of food.

Skeleton

You are given the skeleton file **NokiaSnake.java**. You should see a non-empty file when you open the skeleton, otherwise you might be in the wrong directory.

```
/**
 * Name          :
 * Matric No.    :
 * PLab Acct.    :
 */

public class NokiaSnake {

    private void run() {
         // treat this as your "main" method
    }

    public static void main(String[] args) {
        NokiaSnake myNokiaSnake = new NokiaSnake();
        myNokiaSnake.run();
    }
}
```

Notes

1. You should develop your program in the subdirectory **ex1** and use the skeleton file provided.
2. You are free to use anything to solve this problem.
3. This problem is worth **30%** of the total PE marks.
4. Let **M** be the total number of moves the snake makes until it dies or until the end of the game. You will get **at most**:
   - **80%** of the total marks you receive if your algorithm runs worse than **O(NM).**
   - **90%** of the total marks you receive if your algorithm runs in **O(NM).**
   - **100%** of the total marks you receive if your algorithm runs in **O(M).**
5. Please be reminded that the marking scheme is:

   Input : 10%
   Output : 10%
   Correctness : 50%
   Programming Style : 30% (awarded if you score **at least 20% from the above**):
   - Meaningful comments (pre- and post- conditions, comments inside the code): 10%
   - Modularity (modular programming, proper modifiers [public / private]): 10%
   - Proper Indentation: 5%
   - Meaningful Identifiers (for both method and variable names): 5%

   Compilation Error : Deduction of **50% of the total marks obtained**.