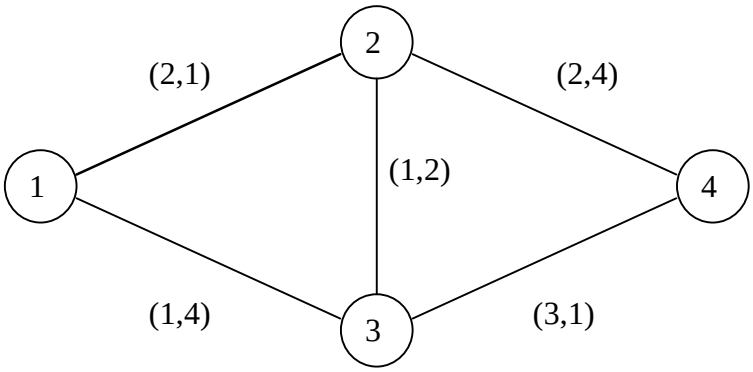
Bicriterial routing

The network of pay highways in Byteland is growing up very rapidly. It has become so dense, that the choice of the best route is a real problem. The network of highways consists of bidirectional roads connecting cities. Each such road is characterized by the traveling time and the toll to be paid.

The route is composed of consecutive roads to be traveled. The total time needed to travel the route is equal to the sum of traveling times of the roads constituting the route. The total fee for the route is equal to the sum of tolls for the roads of which the route consists. The faster one can travel the route and the lower the fee, the better the route. Strictly speaking, one route is better than the other if one can travel it faster and does not have to pay more, or vice versa: one can pay less and can travel it not slower than the other one. We will call a route connecting two cities minimal if there is no better route connecting these cities. Unfortunately, not always exists one minimal route – there can be several incomparable routes or there can be no route at all.

Example

The picture below presents an example network of highways. Each road is accompanied by a pair of numbers: the toll and the traveling time.



Let us consider four different routes from city 1 to city 4, together with their fees and traveling times: 1-2-4 (fee 4, time 5), 1-3-4 (fee 4, time 5), 1-2-3-4 (fee 6, time 4) and 1-3-2-4 (fee 4, time 10).

Routes 1-3-4 and 1-2-4 are better than 1-3-2-4. There are two minimal pairs fee-time: fee 4, time 5 (roots 1-2-4 and 1-3-4) and fee 6, time 4 (root 1-2-3-4). When choosing the route we have to decide whether we prefer to travel faster but we must pay more (route 1-2-3-4), or we would rather travel slower but cheaper (route 1-3-4 or 1-2-4).

Task

Your task is to write a program, which:

- Reads the description of the highway network and starting and ending cities of the route form the text file `bic.in`.
- Computes the number of different minimal routes connecting the starting and ending city, however all the routes characterized by the same fee and traveling time count as just one route; we are interested just in the number of different minimal pairs fee-time.
- Writes the result to the output file `bic.out`.

Input data

There are four integers, separated by single spaces, in the first line of the text file `bic.in`: number of cities n (they are numbered from 1 to n), $1 \le n \le 100$, number of roads m, $0 \le m \le 300$, starting city s and ending city e of the route, $1 \le s, e \le n$, $s \ne e$. The consecutive m lines describe the roads, one road per line. Each of these lines contains four integers separated by single spaces: two ends of a road p and r, $1 \le p, r \le n$, $p \ne r$, the toll c, $0 \le c \le 100$, and the traveling time t, $0 \le t \le 100$. Two cities can be connected by more than one road.

Output data

Your program should write one integer, number of different minimal pairs fee-time for routes from s to e, in the first and only line of the text file `bic.out`.

Example

bic.in

```
4 5 1 4
2 1 2 1
3 4 3 1
2 3 1 2
3 1 1 4
2 4 2 4
```

bic.out

```
2
```

Comments

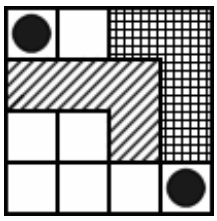
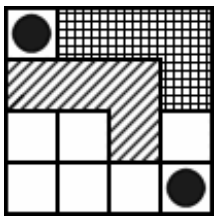
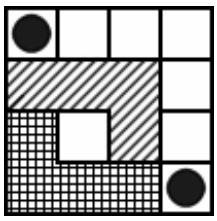
This is the case shown on the picture above.

L Game © Edward de Bono

The L Game was designed by Edward de Bono who enjoys playing games and yet hates to concentrate on a large number of pieces. The intention was to produce the simplest possible game that could be played with a high degree of skill. The L game was the result.

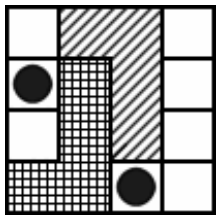
Each player has only one piece, an 'L piece'. There are also two neutral square pieces. The board is four squares by four squares. The object of the game is to manoeuvre the other player into a position on the board where he cannot move his L piece.

Proceeding from the starting position, the first player (and each player on each move thereafter) must move the L piece first. When moving, a player may slide, turn or pick up and flip the L piece into any open position other than the one it occupied prior to the move. When the L piece has been moved, a player may move either one (but only one) of the neutral square pieces to any open square on the board. It is not required that a neutral piece be moved, this is up to the player!

From any of the three positions above, the player with the checkered L can move his L piece to two new locations (the other two of the three positions above). After moving the L piece, he can move one of the neutral pieces to any of the 6 remaining empty squares or decide not to move any of the neutral pieces. All in all, there are 2 * (6 + 6 + 1) possible moves.

A player wins the game when his opponent cannot move his L piece. In the position below, if the player with the checkered L is to move, he loses, as he cannot move his L to a new, unoccupied position on the board.



The game is simple, yet hard as there are so many moves. There are over 18,000 positions for the pieces on the small board and at any moment there may be as many as 195 different moves of which only one is successful.

Write a program that given the position of the pieces and which player is to move, decides if the player has a winning move and output such a move if it exist. If there exist several winning moves, output any one of them. If no winning move exists, you should decide whether the game will end in a draw (assuming perfect play from both players), or if the player to move will in fact lose.

A winning move is defined as a move that, no matter how the opponent plays, he cannot avoid losing the game if the player who is about to move continues the game playing perfectly.

A player is losing the game if, no matter which move he plays, he cannot avoid losing if the opponent continues to play the game perfectly.

If neither of these conditions hold (that is, none of the players can force a win), we say the game position is a draw.

Input data

The input file `lgame.in` consists of four lines describing the position of a game in progress. A dot (‘`.`’) represents an empty square, an ‘`x`’ (lowercase X) represents a neutral square piece, ‘`#`’ represents the L piece of the player A and ‘`*`’ represents the L piece of the player B. The player A is about to move. You may assume that the position is legal and that the player to move has at least one legal move from the current position.

Output data

Output data should be written to the file `lgame.out`. If a winning move exists, output the position after the winning move, using the same format as the input. Otherwise output `No winning move` on a line by itself, and on the next line either `Draw` if the game will end in a draw (assuming perfect play) or `Losing` if the player to move will lose the game.

Examples

lgame.in	**lgame.in**	**lgame.in**

`.***`
`#*.x`
`###.`
`x...`
	`...x`
	`###.`
	`#***`
	`x..*`
		`.###`
		`x#*x`
		`***.`
		`....`

lgame.out.	**lgame.out.**	**lgame.out.**

`.***`
`x*#x`
`###.`
`....`
	`No winning move`
	`Draw`
		`No winning move`
		`Losing`

Grading

In this task, a program receives points for tests with no winning move only if it solves correctly at least one of the tests with a winning move.

Moving Robots

There are several robots moving on a 2-d whole grid plane. The state of every robot is defined by the current position and direction and every robot moves according to its own finite sequence of commands. Position is described by a pair of integers (x, y). There are four possible robot's directions that are described by the number of degrees: 0, 90, 180 or 270. The commands are of two kinds – turn and move. Turn command has one parameter D which can be one of 90, 180 or 270. The command changes the robot's current direction in D degrees: direction C is changed to direction (C + D) mod 360. Step command has no parameters and makes the robot move one step in its current direction. One step in direction 0 changes robot's current position by (1, 0), in direction 90 – by (0, 1), in direction 180 – by (-1, 0) and in direction 270 – by (0, -1).

A robot performs the commands of its sequence one after another. When the whole sequence is performed the robot stops in the final position.

The movement of a robot is not affected by the other robots in any way. In particular, any number of robots can share the same position.

Before the robots start to move the control center can order some robots to remove some commands from their sequences. So the control center can influence the paths of the robots as well as their final positions. The control center wants all robots to gather in the common final position for inspection. It also wants to achieve this with the minimum possible total number of removed commands.

Task

There are R ($2 \le R \le 10$) robots. Each robot has its own initial position, initial direction and its own sequence of commands that contains no more than 50 commands. Write a program that determines (if possible):

- the minimum total number of commands to be removed from some sequences that all robots would stop in the common final position;
- the common final position in that case.

If there are several such positions then the program should output any one of them.

Input data

Input data should be read from the file `robots.in`. The first line of the file contains integer R ($2 \le R \le 10$) – the number of robots. Then R sections of lines follow. Each section describes one robot. The first line of the section contains four integers, separated by single spaces: x, y – initial robot position (x, y), initial robot direction C (C = 0, 90, 180, or 270) and the length of robot's command sequence n ($1 \le n \le 50$). Then the section contains n lines describing the sequence of commands, one command per line. The line of the step command contains single character `S` in the first position and the line of the turn command contains character `T` in the first position that is followed by turn parameter – integer D (D = 90, 180 or 270) and is separated by a single space.

Output data

Output data should be written to the file `robots.out` . If it is not possible to make the robots stop in the common final position by removing any set of commands then the program should write integer –1 (minus one) to the first (and single) line of the file. Otherwise, the minimum total number of commands to be removed should be written to the first line. The second line should contain 2 integers separated by single space: the first and the second coordinates of the common final position.

Example

robots.in	**robots.out**	**Comment**
2 2 0 270 5 S T 180 S S S 1 -1 0 8 S S T 90 S T 270 S T 90 S	2 2 1	There are 2 moving robots. The first robot has initial position (2, 0), direction 270 and a sequence of 5 commands. The second one has initial position (1, -1), direction 0 and a sequence of 8 commands. The minimum total number of commands to be removed that makes robots share final position is 2: for example, remove the 3-rd command of the first robot and the 5-th command of the second robot. The common final position in that case is (2, 1).