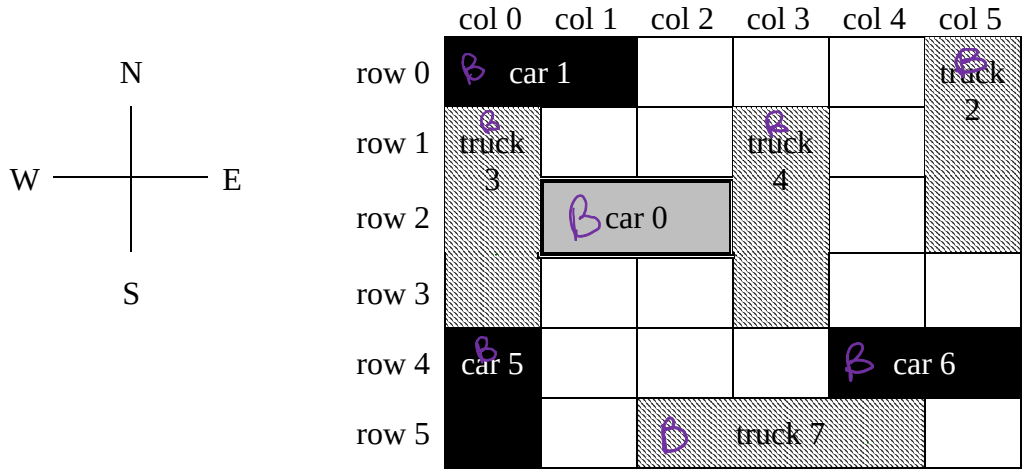
Rush Hour

input file: `rush.in` output file: `rush.out`

There is a commercially available puzzle called "Rush Hour" in which your goal is to unstick a complicated traffic jam. The puzzle is played on a 6x6 grid of squares. Vehicles (cars and trucks) are scattered over the grid at integer locations, as shown below. Both types of vehicles are 1 square wide. Cars are 2 squares long, and trucks are 3 squares long. Vehicles may be oriented either horizontally (East-West) or vertically (North-South) relative to the grid.



Vehicles cannot move through each other, cannot turn, and cannot move off the edge of the grid. They can move straight forwards and backwards along their direction of orientation, as long as they are not blocked by another vehicle or the edge of the grid. Only one vehicle may move at a time, and it may move by only one square at a time.

The goal of the puzzle is to move vehicles back and forth until a particular horizontally-oriented vehicle - your own car - arrives at the rightmost (eastern-most) edge of the grid, where it is considered to have escaped the traffic jam.

The goal of this problem is to write a program that will not only solve any solvable Rush Hour puzzle instance, but will give a solution requiring the *minimum* possible number of (1-square-at-a-time) moves.

Input

The input file will consist of one or more input scenarios. Each scenario begins with a single integer n, $0 \le n \le 10$, giving the number of vehicles in the scenario. There will then be n lines of input, each representing 1 vehicle. Each vehicle consists of a space-separated list of:

(a) a decimal number, 2 or 3, representing the length of that vehicle in grid squares,
(b) an uppercase letter H or V, indicating horizontal or vertical orientation for that vehicle,
(c) a decimal number 0-5 indicating the row number (numbered North to South) of the vehicle (or of its northernmost square, if it is oriented vertically) in the initial position.
(d) a decimal number 0-5 indicating the column number (numbered West to East) of the vehicle (or of its westernmost square, if it is oriented horizontally) in the initial position.

The first vehicle in the sequence will always be your horizontally-oriented car which needs to get to the right edge of the grid. You may assume your vehicle can always escape.

The last scenario will be indicated by a scenario with 0 vehicles. This scenario should not be processed.

Output

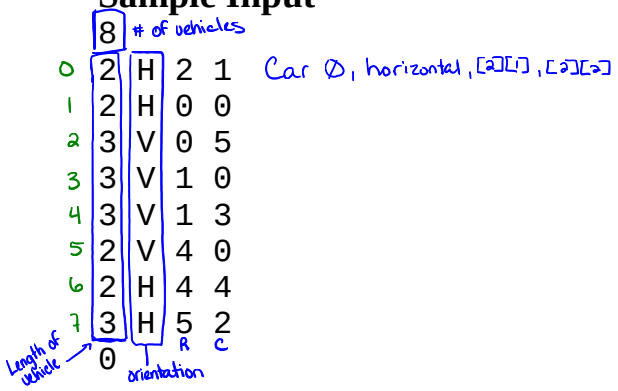
The statement "Scenario *k* requires *x* moves." on a line by itself, where *k* is the number of the scenario, starting with 1, and *x* is the (decimal integer) minimum number of moves required to solve the puzzle. This is immediately followed by a sequence of that many moves, one per line, that solves the puzzle. Each move consists of

(a) A decimal number designating the vehicle to be moved, from the vehicles in the input list. The first vehicle in the list (your car) is vehicle number 0; subsequent vehicles are numbered sequentially from 1.

(b) An uppercase letter F or B, indicating whether the given vehicle should move 1 square Forward, or 1 square Back. Forward means movement in the direction of increasing row or column number. (For horizontal vehicles, Forward means East; for vertical vehicles, Forward means South.)

If there is more than one solution with a minimal number of moves, give any minimal solution.

Sample Input



Sample Output (corresponding to sample input)

```
Scenario 1 requires 16 moves.
1 F
3 B
5 B
6 B
6 B
6 B
7 B
7 B
4 F
4 F
2 F
2 F
2 F
0 F
0 F
0 F
```

carNum carDir