Pebbling odometer

Leonardo invented the original *odometer*: a cart which could measure distances by dropping pebbles as the cart's wheels turned. Counting the pebbles gave the number of turns of the wheel, which allowed the user to compute the distance travelled by the odometer. As computer scientists, we have added software control to the odometer, extending its functionalities. Your task is to program the odometer under the rules specified below.

Operation grid

The odometer moves on an imaginary square grid of 256 × 256 unit cells. Each cell can contain at most 15 pebbles and is identified by a pair of coordinates (row, column), where each coordinate is in the range 0, …, 255. Given a cell (i, j), the cells adjacent to it are (if they exist) (i - 1, j), (i + 1, j), (i, j - 1) and (i, j + 1). Any cell laying on the first or last row, or on the first or last column, is called a *border*. The odometer always starts at cell (0, 0) (the north-west corner), facing north.

Basic commands

The odometer can be programmed using the following commands.

- `left` — turn 90 degrees to the left (counter clockwise) and remain in the current cell (e.g. if it was facing south before, then it will face east after the command).
- `right` — turn 90 degrees to the right (clockwise) and remain in the current cell (e.g. if it was facing west before, then it will face north after the command).
- `move` — move one unit forwards (in the direction the odometer is facing) into an adjacent cell. If no such cell exists (i.e. the border in that direction has been already reached) then this command has no effect.
- `get` — remove one pebble from the current cell. If the current cell has no pebbles, then the command has no effect.
- `put` — add one pebble to the current cell. If the current cell already contains 15 pebbles, then the command has no effect. The odometer never runs out of pebbles.
- `halt` — terminate the execution.

The odometer executes the commands in the order they are given in the program. The program must contain at most one command per line. Empty lines are ignored. The symbol # indicates a comment; any text that follows, up to the end of the line, is ignored. If the odometer reaches the end of the program, execution is terminated.

Example 1

Consider the following program for the odometer. It takes the odometer to the cell (0, 2), facing east. (Note that the first `move` is ignored, because the odometer is on the north-west corner facing north.)

```
move  # no effect
right
# now the odometer is facing east
move
move
```

Labels, borders and pebbles

To alter the flow of the program depending on the current status, you can use labels, which are case-sensitive strings of at most 128 symbols chosen from `a`, …, `z`, `A`, …, `Z`, `0`, …, `9`. The new commands concerning labels are listed below. In the descriptions below, *`L`* denotes any valid label.

- *`L`**`:`** (i.e. *`L`* followed by a colon '`:`') — declares the location within a program of a label *`L`*. All declared labels must be unique. Declaring a label has no effect on the odometer.
- `jump` *`L`* — continue the execution by unconditionally jumping to the line with label *`L`*.
- `border` *`L`* — continue the execution jumping to the line with label *`L`*, if the odometer is on a border facing the edge of the grid (i.e. a `move` instruction would have no effect); otherwise, the execution continues normally and this command has no effect.
- `pebble` *`L`* — continue the execution jumping to the line with label *`L`*, if the current cell contains at least one pebble; otherwise, the execution continues normally and this command has no effect.

Example 2

The following program locates the first (westmost) pebble in row 0 and stops there; if there are no pebbles in row 0, it stops on the border at the end of the row. It uses two labels `leonardo` and `davinci`.

```
right
leonardo:
pebble davinci  # pebble found
border davinci  # end of the row
move
jump leonardo
davinci:
halt
```

The odometer starts by turning to its right. The loop begins with the label declaration `leonardo:` and ends with the `jump leonardo` command. In the loop, the odometer checks for the presence of a pebble or the border at the end of the row; if not so, the odometer makes a `move` from the current cell (0, j) to the adjacent cell (0, j + 1) since the latter exists. (The `halt` command is not strictly necessary here as the program terminates anyway.)

Statement

You should submit a program in the odometer's own language, as described above, that makes the odometer behave as expected. Each subtask (see below) specifies a behavior the odometer is required to fulfill and the constraints the submitted solution must satisfy. The constraints concern the two following matters.

- *Program size* — the program must be short enough. The size of a program is the number of commands in it. Label declarations, comments and blank lines *are not counted* in the size.
- *Execution length* — the program must terminate fast enough. The execution length is the number of performed *steps*: every single command execution counts as a step, regardless of whether the command had an effect or not; label declarations, comments and blank lines do not count as a step.

In Example 1, the program size is 4 and the execution length is 4. In Example 2, the program size is 6 and, when executed on a grid with a single pebble in cell (0, 10), the execution length is 43 steps: `right`, 10 iterations of the loop, each iteration taking 4 steps (`pebble davinci`; `border davinci`; `move`; `jump leonardo`), and finally, `pebble davinci` and `halt`.

Subtask 1 [9 points]

At the beginning there are x pebbles in cell (0, 0) and y in cell (0, 1), whereas all the other cells are empty. Remember that there can be at most 15 pebbles in any cell. Write a program that terminates with the odometer in cell (0, 0) if $x \leq y$, and in cell (0, 1) otherwise. (We do not care about the direction the odometer is facing at the end; we also do not care about how many pebbles are present at the end on the grid, or where they are located.)

Limits: program size ≤ 100, execution length $\leq 1\,000$.

Subtask 2 [12 points]

Same task as above but when the program ends, the cell (0, 0) must contain exactly x pebbles and cell (0, 1) must contain exactly y pebbles.

Limits: program size ≤ 200, execution length $\leq 2\,000$.

Subtask 3 [19 points]

There are exactly two pebbles somewhere in row 0: one is in cell (0, x), the other in cell (0, y); x and y are distinct, and x + y is even. Write a program that leaves the odometer in cell (0, (x + y) / 2), i.e., exactly in the midpoint between the two cells containing the pebbles. The final state of the grid is not relevant.

Limits: program size ≤ 100, execution length $\leq 200\,000$.

Subtask 4 [up to 32 points]

There are at most 15 pebbles in the grid, no two of them in the same cell. Write a program that collects them all in the north-west corner; more precisely, if there were x pebbles in the grid at the beginning, at the end there must be exactly x pebbles in cell (0, 0) and no pebbles elsewhere.

The score for this subtask depends on the execution length of the submitted program. More precisely, if L is the maximum of the execution lengths on the various test cases, your score will be:

- 32 points if $L \leq 200\,000$;
- $32 - 32 \log_{10} (L / 200\,000)$ points if $200\,000 < L < 2\,000\,000$;
- 0 points if $L \geq 2\,000\,000$.

Limits: program size ≤ 200.

Subtask 5 [up to 28 points]

There may be any number of pebbles in each cell of the grid (of course, between 0 and 15). Write a program that finds the minimum, i.e., that terminates with the odometer in a cell (i, j) such that every other cell contains at least as many pebbles as (i, j). After running the program, the number of pebbles in each cell must be the same as before running the program.

The score for this subtask depends on the program size P of the submitted program. More precisely, your score will be:

- 28 points if $P \leq 444$;
- $28 - 28 \log_{10} (P / 444)$ points if $444 < P < 4\,440$;
- 0 points if $P \geq 4\,440$.

Limits: execution length ≤ 44 400 000.

Implementation details

You have to submit exactly one file per subtask, written according to the syntax rules specified above. Each submitted file can have a maximum size of 5 MiB. For each subtask, your odometer code will be tested on a few test cases, and you will receive some feedback on the resources used by your code. In the case the code is not syntactically correct and thus impossible to test, you will receive information on the specific syntax error.

It is not necessary that your submissions contain odometer programs for all the subtasks. If your current submission does not contain the odometer program for subtask X, your most recent submission for subtask X is automatically included; if there is no such program, the subtask will score zero for that submission.

As usual, the score of a submission is the sum of the scores obtained in each subtask, and the final score of the task is the maximum score among the release-tested submissions and the last submission.

Simulator

For testing purposes, you are provided with an odometer simulator, which you can feed with your programs and input grids. Odometer programs will be written in the same format used for submission (i.e., the one described above).

Grid descriptions will be given using the following format: each line of the file must contain three numbers, R, C and P, meaning that the cell at row R and column C contains P pebbles. All cells not specified in the grid description are assumed to contain no pebbles. For example, consider the file:

```
0 10 3
4 5 12
```

The grid described by this file would contain 15 pebbles: 3 in the cell (0, 10) and 12 in the cell (4, 5).

You can invoke the test simulator by calling the program `simulator.py` in your task directory, passing the program file name as argument. The simulator program will accept the following command line options:

- `-h` will give a brief overview of the available options;
- `-g GRID_FILE` loads the grid description from file `GRID_FILE` (default: empty grid);
- `-s GRID_SIDE` sets the size of the grid to `GRID_SIDE x GRID_SIDE` (default: 256, as used in the problem specification); usage of smaller grids can be useful for program debugging;
- `-m STEPS` limits the number of execution steps in the simulation to at most `STEPS`;
- `-c` enters compilation mode; in compilation mode, the simulator returns exactly the same output, but instead of doing the simulation with Python, it generates and compiles a small C program. This causes a larger overhead when starting, but then gives significantly faster results; you are advised to use it when your program is expected to run for more than about 10 000 000 steps.

Number of submissions

The maximum number of submissions allowed for this task is 128.