Odometro de piedras

Leonardo inventó un *Odómetro* original: un carro que podía medir distancias dejando caer piedras mientras las rueddas del carro giraban. Contando las piedras obtenía el número de giros de la rueda, lo que le permitía al usuario computar la distancia recorrida por el odómetro. Como informáticos. hemos añadido software para controlar el odómetro, y aumentar sus funcionalidades. Tu tarea es programar el odómetro bajo las reglas especificadas a continuación.

Tablero de operaciones

El odómetro se mueve en una tablero cuadrado imaginario de 256x256 celdas. Cada celda contiene como maximo 15 piedras y se identifica por un par de coordenadas (fila, columna), donde cada coordenada se encuentra en el rango 0, ..., 255. Dada una celda (i, j), sus celdas adyacentes (si existen) son (i - 1, j), (i + 1, j), (i, j - 1) y (i, j + 1). Cualquier celda de la primera o última fila o de la primera o última columna se llama *border*. El odómetro siempre empieza en la celda (0, 0) (la esquina noroeste), mirando al norte.

Comandos básicos

El odometro puede ser programado usando los siguientes comandos

- `left` —gire 90 grados a la izquierda (sentido contrahorario) y permanezca en la celda corriente (e.g. si estaba mirando al sur antes, entonces mirará al este después del comando).
- `right` — gire 90 grados a la derecha (sentido horario) y permanezca en la celda corriente (e.g. si estaba mirando al oeste antes, entonces mirará al norte después del comando).
- `move` — mueva una unidad hacia adelante (en la dirección en la que apunta el odómetro) hacia la celda adyacente. Si no existe tal celda (i.e. el border en tal dirección ha sido ya alcanzado) entonces este comando no tiene efecto.
- `get` — remueva una piedra de la celda corriente. Si la celda corriente no tiene piedras, entonces el comando no tiene efecto.
- `put` — agregue una piedra a la celda corriente. Si la celda corriente ya contiene 15 piedras, entonces el comando no tiene efecto. El odómetro nunca agota sus piedras.

- `halt` — termina la ejecución.

El odómetro ejecuta los comandos en el orden en que están dados en el programa. El programa debe contener a lo sumo un comando por línea. Líneas vacías son ignoradas. El símbolo `#` indica un comentario; cualquier texto que sigue, hasta el fin de la línea, es ignorado. Si el odómetro llega al fin del programa, la ejecución es terminada.

Ejemplo 1

Considere el siguiente programa para el odómetro. Lleva al odómetro a la celda (0, 2), mirando al Este. (Observe que el primer `move`es ignorado, porque el odómetro está sobre la esquina NorOeste mirando al Norte.)

```
move  # no efecto
right
# ahora el odómetro esta mirando al Este
move
move
```

Rótulos, borders y piedras

Para alterar el flujo del programa dependiendo del estado corriente, puedes usar rótulos, que son cadenas sensibles a mayúscula de a lo más 128 símbolos elegidos de `a`, …, `z`, `A`, …, `Z`, `0`, …, `9`. Los nuevos comandos relativos al uso de rótulos están listados abajo. En las descripciones abajo, *`L`* denota cualquier rótulo válido.

- *`L`*`:` (i.e. *`L`* seguido por dos puntos '`:`') — declara la posición de un rótulo *`L`* dentro del programa. Todos los rótulos deben ser únicos. Declarar un rótulo no tiene efecto sobre el odómetro.

- `jump` *`L`* — continúe la ejecución tras saltar incondicionalmente a la línea con rótulo *`L`*.

- `border` *`L`* — si el odómetro está en un border mirando una arista de la grilla (i.e. una instrucción move no tendría efecto) continúe la ejecución saltando a la línea con rótulo *`L`*; de otra forma, la ejecución continúa normalmente y este comando no tiene efecto.

- `pebble` *`L`* — si la celda actual contiene al menos una piedra, continúe la ejecución saltando a la línea con rótulo *`L`*; de otro modo, la ejecución continúa normalmente y este comando no tiene efecto.

Ejemplo 2

El siguiente programa ubica la primera (la más al oeste) piedra en la fila 0 y allí se detiene; si no hay piedras en fila 0, para en el border al final de la fila. Usa dos rótulos `leonardo` y `davinci`.

```
right
leonardo:
pebble davinci  # piedra encontrada
border davinci  # fin de la fila
move
jump leonardo
davinci:
halt
```

El odómetro inicia girando a la derecha. El ciclo comienza con la declaración del rótulo `leonardo:` y termina con el comando `jump leonardo`. En el ciclo, el odómetro verifica la presencia de una piedra o del border al final de la fila; si no es así, el odómetro efectúa un move de la celda corriente (0, j) a la celda adyacente (0, j + 1) siempre que esta última exista. (El comando `halt` no es estrictamente necesario aquí ya que el programa termina de todos modos.)

Enunciado

Usted debe enviar un programa en el lenguage propio del odometro, como esta descripto arriba, que marca el comportamiento esperado del odometro. Cada subtarea (ver abajo) especifica un comportamiento que es requierido el odometro cumpla y las restricciones que el programa enviado debe satisfacer. Las restriciones corresponden a los siguientes dos topicos.

- *Tamaño del programa* —el programa debe ser lo suficientemente corto. El tamaño del programa es el número de comandos en él. Declaraciones de etiquetas, comentarios y líneas en blanco *no son contadas* en el tamaño.
- 'Longitud (duracion) de la ejecución — *el programa debe terminar lo suficientemente rápido. La longitud de la ejecución es el número de* pasos *realizados: cada una de las ejecuciones de un comando cuenta como un paso, independientemente de si el comando tuvo un efecto o no; declaraciones de etiquetas, comentarios y lineas en blanco no cuentan como pasos.*

En el ejemplo 1, el tamaño del programa es 4 y la longitud de ejecucion es 4. En el Ejemplo 2, el tamaño del programa es 6 y, cuando se ejecuta en una grilla con una piedra en la celda (0, 10), la lonfgitud de ejecucion es de 43 pasos: `right`, 10 iteraciones de el ciclo, cada iteracion toma 4 pasos (`pebble davinci`; `border davinci`; `move`; `jump leonardo`), y finalmente, `pebble davinci` y `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 ≤ 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 ≤ 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 ≤ 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.