조약돌 주행거리계

다빈치는 최초의 *주행거리계*를 발명하였다. 이 기계는 수레를 이용하여 만든 것으로서 수레의 바퀴가 한바퀴 돌 때 마다 조약돌을 떨어트려서 거리를 잴 수 있게 한 것이다. 떨어트린 조약돌의 수를 세면 바퀴의 회전수를 알아낼 수 있고, 이에 따라 수레가 이동한 거리를 알아낼 수 있다. 컴퓨터 과학자인 우리는 주행거리계에 소프트웨어를 이용한 제어기를 도입하여 기능을 확장하였다. 이 문제의 목표는 주행거리계를 위해 만들어진 아래에 설명한 규칙과 같은 새로운 언어로 프로그램을 작성하여 제출하는 것이다.

사용하는 격자 (맵)

주행거리계는 가로와 세로가 각각 256개의 셀로 구성된 가상의 격자 위에서 움직인다. 각 셀은 2차원의 좌표로 표시되며, 각 좌표의 값은 0 이상 255 이하이다. 가장 북서쪽(왼쪽 위)의 셀이 (0, 0) 셀이다. (첫번째 좌표 값은 남북 방향에 대한 것이며, 두번째 좌표 값은 동서 방향에 대한 것이다. 즉 (0, 0) 셀의 오른쪽에 인접한 셀은 (0, 1) 셀이다.) 각 셀에는 최대 15개의 조약돌이 존재할 수 있다. 어떤 (i, j) 셀에 대해서 인접한 셀들은 (i - 1, j), (i + 1, j), (i, j - 1)과 (i, j + 1)이다. 물론 좌표 값이 지정된 범위 밖인 경우는 셀이 존재하지 않는다. 두 좌표 중 어느 하나라도 0이거나 255인 셀들은 *경계* 셀이라고 부른다. 주행거리계의 초기 상태는 (0, 0) 셀에서 북쪽을 향하고 있다.

기본 명령어들

주행거리계는 다음의 명령들을 이용하여 프로그램할 수 있다.

- `left` — 제자리에서 왼쪽으로 (시계 반대 방향으로) 90도 회전한다. (예를 들어, 주행거리계가 남쪽을 향하고 있었다면 이 명령 이후에는 동쪽을 향하게 된다.)
- `right` — 제자리에서 오른쪽으로 (시계 방향으로) 90도 회전한다. (예를 들어, 주행거리계가 서쪽을 향하고 있었다면 이 명령 이후에는 북쪽을 향하게 된다.)
- `move` — 현재 주행거리계가 향하고 있는 방향의 인접한 셀로 이동(전진)한다. 만약 그 방향에 셀이 존재하지 않는 경우이면 (현재 위치가 경계 셀이면 발생할 수 있는 경우임) 이 명령은 아무 효과도 없다.
- `get` — 현재 셀에서 하나의 조약돌을 제거한다. 만약 현재 셀에 조약돌이 없다면 이 명령은 아무 효과도 없다.
- `put` — 현재 셀에 하나의 조약돌을 추가한다. 만약 현재 셀에 이미 15개의 조약돌이 있는 경우라면 이 명령은 아무 효과가 없다. 주행거리계는 무한한 개수의 조약돌을 가지고 있어서 조약돌이 모자라는 경우는 없다.
- `halt` — 프로그램의 수행이 종료된다.

주행거리계의 프로그램은 한 줄에 최대 하나씩의 명령어를 가지도록 주어져야 한다. 주행거리계는 명령들을 주어진 순서대로 하나씩 수행한다. 빈 줄은 완전히 무시된다. 기호 `#`는 주석를 위한 것인데 어떤 줄에 이 기호가 나타나면 그 다음 글자 부터 그 줄의 끝까지는 모두 주석으로 간주되어 무시된다. 프로그램의 수행이 마지막 줄을 넘어서게 되면 프로그램은 자동으로 종료된다.

예제 1

주행거리계를 위한 다음 프로그램을 보자. 다음 프로그램은 주행거리계를 (0, 2) 셀로 보내고 마지막에 동쪽으로 향하도록 만든다. (제일 첫 `move` 명령은 무시됨에 주의하자. 그 이유는 초기에 주행거리계는 제일 북서쪽 셀에 위치해서 북쪽으로 향하고 있기 때문이다.)

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

Label, border, pebble 명령문

현재 상태에 따라 프로그램의 동작이 달라지도록 하기 위하여 레이블을 정의할 수 있다. 레이블은 `a`, …, `z`, `A`, …, `Z`, `0`, …, `9` 의 문자들로 만들어진 문자열이며, 대소문자를 구분하고 최대 128의 길이를 가질 수 있다. 레이블을 사용하는 명령들이 아래에 제시되어 있다. 아래 설명에서 *L*은 임의의 유효한 레이블을 의미한다.

- *L*`:` (즉, *L* 다음에 콜론 기호‘ `:`’가 있음) — 프로그램의 해당 위치를 *L* 레이블로 선언한다. 프로그램 내의 레이블들은 모두 달라야 한다. 레이블 선언 자체는 프로그램 내의 위치만을 의미하고 주행거리계가 특별히 수행하는 동작은 없다.
- `jump` *L* — 레이블 *L*의 위치로 무조건 점프를 수행하여 그 레이블 다음의 명령을 수행한다.
- `border` *L* — 현재 주행거리계가 경계 셀에 위치하고 전방에 셀이 존재하지 않는 경우(즉, `move` 명령이 아무 동작을 하지 않게 되는 경우)라면 *L*에 해당하는 레이블의 위치로 점프한다. 그와 다른 경우, 이 명령은 아무 동작을 수행하지 않고, 이어진 다음 명령이 수행된다.
- `pebble` *L* — 현재 셀에 조약돌이 하나 이상 존재하는 경우에는 레이블 *L*에 해당하는 위치로 점프한다. 그와 다른 경우에는 이 명령은 아무런 동작을 수행하지 않고, 이어진 다음 명령이 수행된다.

예제 2

다음 프로그램은 제일 위쪽(북쪽) 행(row)에서 조약돌이 있는 제일 왼쪽(서쪽) 셀을 찾아서, 그 셀에서 주행거리계가 멈추도록 하는 프로그램이다. 만약 제일 위쪽 행에 조약돌이 존재하는 셀이 없는 경우 주행거리계는 제일 오른쪽 셀에서 멈춘다. 프로그램에는 `leonardo`와 `davinci`의 두 개의 레이블이 사용된다.

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

주행거리계는 우선 오른쪽으로 회전한다. 프로그램의 루프는 `leonardo:`로 시작하여 `jump leonardo`로 끝나는 부분이다. 루프에서 수행하는 작업은 현재 셀에 조약돌이 존재하는 지의 여부를 검사하는 것과 열의 오른쪽 끝에 도달했는지를 검사하는 것이다. 두 경우 모두 아니라면, 주행거리계는 `move`를 수행하여 현재 셀 (0, j) 에서 인접한 셀 (0, j + 1)로 이동한다. 인접한 셀이 존재함은 반드시 보장된다. (마지막의 `halt` 명령이 없어도 프로그램은 정상적으로 종료되므로 마지막 명령은 반드시 필요한 것은 아니다.)

해야할 일

주행거리계를 기대한대로 움직이게 하도록 위에서 설명한 주행거리계 언어로 프로그램을 작성한다. 각 서브태스크(아래에 설명한다)에서 주행거리계가 수행할 행동과 제출할 프로그램이 만족해야하는 제약조건들을 설명한다. 제약조건들은 다음 두 가지 요소들과 관련이 있다.

- *프로그램 크기* - 프로그램은 충분히 짧아야 한다. 프로그램의 크기는 포함된 명령어들의 수이다. 레이블 선언들, 주석들, 빈 라인들은 *세지 않는다*.
- *실행 길이* - 프로그램은 충분히 빠르게 끝나야 한다. 실행 길이는 수행된 *스탭*들의 수이다: 모든 단일 명령어 실행은 그것이 효과가 있던 없던 간에 한 스탭으로 센다; 레이블 선언, 주석, 빈 라인은 한 스탭으로 세지 않는다.

예제 1에서 프로그램 크기는 4이고 실행 길이도 4이다. 예제 2에서 프로그램 크기는 6이고, (0, 10) 셀에 하나의 조약돌이 있는 격자의 경우에 실행 길이는 43이다. 즉, `right`, 각각 4 스탭인 (`pebble davinci`, `border davinci`, `move`, `jump leonardo`) 루프의 10번 반복 , 그리고 마지막으로, `pebble davinci` 와 `halt`가 있다.

서브태스크 1 [9점]

초기에 (0, 0) 셀에 x 개 조약돌들과 (0, 1) 셀에 y 개 조약돌들이 있고, 모든 다른 셀들은 비어있다. 만약 $x \le y$라면, 주행거리계는 (0, 0) 셀에서 종료하고, 그렇지 않으면, (0, 1) 셀에서 종료하게 되는 프로그램을 작성하시오. (주행거리계가 마지막에 어디를 향하고 있든지 상관없다. 또한, 마지막에 격자에 얼마나 많은 조약돌들이 존재하던지 또는 조약돌들이 어디에 있던지 상관없다.)

제한들: 프로그램 크기 ≤ 100, 실행 길이 $\le 1{,}000$.

서브태스크 2 [12 점]

서브태스크 1과 같지만 프로그램이 끝났을 때, (0, 0) 셀에 정확히 x 개 조약돌이, (0, 1) 셀에 정확히 y 개 조약돌이 존재해야 한다.

제한들: 프로그램 크기 ≤ 200, 실행 길이 ≤ 2,000.

서브태스크 3 [19점]

제일 북쪽 행 어딘가에 정확히 두 개의 조약돌들이 존재한다: 하나는 (0, x) 셀에 있고, 다른 하나는 (0, y) 셀에 있다. x 와 y 는 다르고, x + y 는 짝수이다. 주행거리계가 (0, (x + y) / 2) 셀에서, 다시 말해서, 조약돌을 포함하는 두 셀들의 정확히 중간지점에서, 종료하도록 하는 프로그램을 작성하시오. 종료 시 격자의 조약돌들의 개수와 위치는 상관이 없다.

제한들: 프로그램 크기 ≤ 100, 실행 길이 ≤ 200,000.

서브태스크 4 [최대 32점]

격자에는 최대 15개의 조약돌이 존재하며, 하나의 셀에는 최대 1개의 조약돌이 존재한다. 조약돌 모두를 제일 북서쪽 셀에 모으는 프로그램을 작성하시오. 정확히, 초기에 격자에 x 개의 조약돌들이 존재하면, 마지막에 (0, 0) 셀에 정확히 x 개의 조약돌들이 존재해야 하며, 그 밖의 셀에는 조약돌이 없어야 한다.

이 서브 태스크의 점수는 제출된 프로그램의 실행 길이에 따라 다르다. 정확히, L 이 여러 테스트 케이스들에 대한 실행 길이의 최대값이면, 점수는 다음과 같을 것이다:

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

제한들: 프로그램 크기 ≤ 200.

서브태스크 5 [최대 28점]

격자의 각 셀에는 임의의 수의 조약돌들이 존재할 수 있다 (물론, 0과 15 사이의 수이다). 그러면 최소 조약돌 수를 가진 셀을 찾는, 다시 말해서, 다른 모든 셀 보다 작거나 같은 수의 조약돌들이 존재하는 (i, j) 셀에서 주행 거리계가 종료하도록 하는 프로그램을 작성하시오. 프로그램을 실행한 후, 각 셀의 조약돌 수는 프로그램을 수행하기 전과 같아야 한다.

이 서브 태스크의 점수는 제출된 프로그램의 크기 P 에 따라 다르다. 정확히 점수는 다음과 같다:

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

제한들: 실행 길이 ≤ 44,400,000.

구현 세부 사항

위에서 기술된 규칙들에 따라서 작성된 프로그램을 서브 태스크마다 정확히 하나의 파일로 제출해야한다. 각 제출된 파일은 최대 5MB를 넘지 않아야 한다. 각 서브 태스크에 대해서, 코드는 몇 가지 테스트 케이스들로 테스트 될 것이고, 코드에서 사용된 자원들에 대한 피드백을 받게 될 것이다. 코드가 구문적으로 옳지 않고 테스트가 불가능한 경우에 특별한 구문 에러에 대한 정보를 받게 될 것이다.

제출할 때, 반드시 모든 서브 태스크들에 대한 프로그램들을 포함할 필요는 없다. 현재 제출이 서브 태스크 X 에 대한 프로그램을 포함하지 않는다면, 서브 태스크 X 에 대한 가장 최근 제출이 자동적으로 포함될 것이다; 만약 그런 프로그램이 없다면, 이 제출에서 그 서브 태스크는 점수 0 을 받을 것이다.

제출 결과 점수는 항상 각 서브 태스크에 포함된 점수들의 합이고 마지막 점수는 테스트된 제출들과 마지막 제출 중 최대값이다.

시뮬레이터

프로그램을 시험해 볼수 있도록 주행거리계 시뮬레이터가 여러분의 컴퓨터에 주어진다. 여러분은 시뮬레이터에 프로그램과 격자를 입력으로 주고 그 수행 결과를 볼수 있다. 주행거리계를 위한 프로그램의 문법은 제출을 위한 것과 동일하다. (즉, 위에서 설명한 것과 같다.)

격자의 초기 상태 표현 방법은 아래와 같다. 각 줄에는 빈칸으로 구분된 세 개의 자연수 R, C, P가 있어야 하는데, R은 행 번호, C는 열 번호이며, P는 (R, C) 셀에 P개의 조약돌이 있음을 의미한다. 조약돌이 존재하지 않는 셀들은 설명에 포함할 필요가 없다. 예를 들어, 다음 파일을 보자.

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

이 파일에서 설명하는 격자에는 15개의 조약돌이 존재하는데, (0, 10) 셀에 3개와 (4, 5) 셀에 12개가 존재한다.

시뮬레이터는 작업 디렉토리의 `simulator.py`에 프로그램 파일 이름을 인자로 주어 사용할 수 있다. 시뮬레이터가 받을 수 있는 옵션은 다음과 같다.

- `-h` 옵션은 가능한 옵션들에 대한 설명을 제공한다.
- `-g GRID_FILE` 옵션은 `GRID_FILE` 에서 격자를 로드한다. (디폴트는 조약돌이 하나도 없는 격자이다.)
- `-s GRID_SIDE` 옵션은 격자의 크기를 `GRID_SIDE x GRID_SIDE`로 잡는다. (디폴트 값은 256이다.) 작은 크기의 격자를 사용하면 디버깅에 도움이 될 수도 있다.
- `-m STEPS` 옵션은 프로그램의 수행 길이를 최대 `STEPS`로 제한한다.

- `-c` 옵션으로 컴파일 모드를 사용할 수 있다. 컴파일 모드에서 수행 결과 자체는 똑같지만 시뮬레이터는 Python을 이용하는 대신 C 프로그램을 자동적으로 생성하여 컴파일하고 그 결과로 생성되는 실행파일을 실행시킨다. 그 결과, 시뮬레이션의 시작은 조금 늦어질 수 있지만, 전체 시뮬레이션 시간은 (특히 수행 길이가 긴 경우에) 많이 줄어들 수 있다. 수행 길이가 10,000,000 단계 이상인 경우에는 이 옵션을 사용할 것을 권장한다.

제출 횟수

이 문제의 최대 제출 횟수는 128이다.