로봇 대회

Szeged 대학의 로봇 연구자들이 로봇 프로그래밍 대회를 열고 있다. 여러분의 친구 은수가 대회에 참가하기로 했다. 이 대회의 목적은 영리하기로 유명한 헝가리 양치기 개 Puli의 이름을 딴 *Pulibot*을 프로그램하는 것이다.

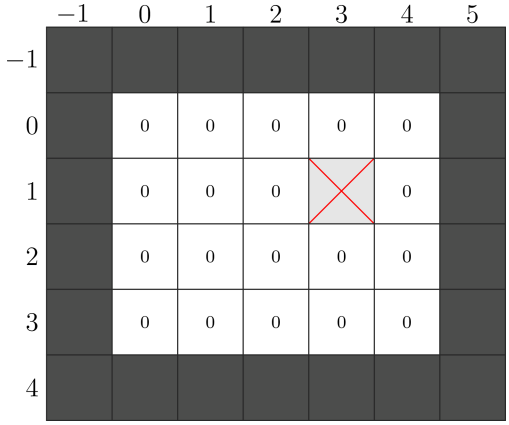
Pulibot을 $(H+2) \times (W+2)$ 크기 격자의 미로에서 테스트한다. 격자의 행은 북쪽에서 남쪽으로 -1부터 H로, 열은 서쪽에서 동쪽으로 -1부터 W로 차례대로 번호가 매겨져 있다. 격자의 r행 c열에 있는 칸을 (r, c)라고 하자.

$0 \le r < H$이고 $0 \le c < W$인 칸 (r, c)을 생각해보자. (r, c)에는 4개의 **인접한** 칸이 있다.

- 칸 $(r, c-1)$은 칸 (r, c)의 **서쪽** 칸이다.
- 칸 $(r+1, c)$은 칸 (r, c)의 **남쪽** 칸이다.
- 칸 $(r, c+1)$은 칸 (r, c)의 **동쪽** 칸이다.
- 칸 $(r-1, c)$은 칸 (r, c)의 **북쪽** 칸이다.

만약 $r = -1, r = H, c = -1, c = W$ 넷 중 하나 이상을 만족한다면 칸 (r, c)는 미로의 **경계**라고 한다. 미로에서 경계가 아닌 칸은 **장애물 칸**이거나 **빈 칸** 둘 중 하나이다. 또, 각 빈 칸에는 **색**이 있는데, 0 이상 Z_{MAX} 이하인 음이 아닌 정수로 표현된다. 처음에 모든 빈 칸의 색은 0이다.

예를 들어, $H = 4$이고 $W = 5$인 미로를 생각해보자. 장애물 칸은 칸 $(1, 3)$ 하나이다.



장애물 칸은 X자로 표시했다. 경계인 칸은 검게 칠해져 있다. 각 빈 칸에 쓰인 숫자는 그 칸의 색을 나타낸다.

칸 (r_0, c_0)에서 칸 (r_ℓ, c_ℓ)으로 가는 길이 ℓ인 $(\ell > 0)$ **경로**는 서로 다른 빈 칸들 $(r_0, c_0), (r_1, c_1), \ldots, (r_\ell, c_\ell)$ 인데, 각 i $(0 \le i < \ell)$에 대해 칸 (r_i, c_i)과 칸 (r_{i+1}, c_{i+1})은 인접해 있다.

길이 ℓ인 경로에는 정확히 $\ell + 1$개의 칸이 있음에 유의하라.

대회에 사용하는 미로에는 칸 $(0, 0)$에서 칸 $(H-1, W-1)$으로 가는 경로가 최소한 하나는 있다. 칸 $(0, 0)$과 칸 $(H-1, W-1)$은 반드시 빈 칸임에 유의하자.

은수는 미로의 어느 칸이 빈 칸이고 어느 칸이 장애물 칸인지 알지 못한다.

여러분이 할 일은 은수를 도와서 Pulibot을 프로그래밍해야 한다. 여러분이 프로그래밍한 Pulibot은 미지의 미로에 있는 칸 $(0, 0)$에서 칸 $(H-1, W-1)$으로 가는 **최단 경로** (즉, 길이가 가장 짧은 경로) 중 하나를 찾을 수 있어야 한다.
Pulibot의 명세와 로봇 대회 규칙은 아래에 설명한다.

이 문제의 제일 마지막에, Pulibot의 동작을 시각화하는데 도움을 줄 수 있는 시각화 도구에 대한 설명이 있다는데 유의하라.

Pulibot's Specification

$-1 \le r \le H$이고 $-1 \le c \le W$인 모든 칸 (r, c)에 대해서 **상태**를 다음과 같이 정의한다.

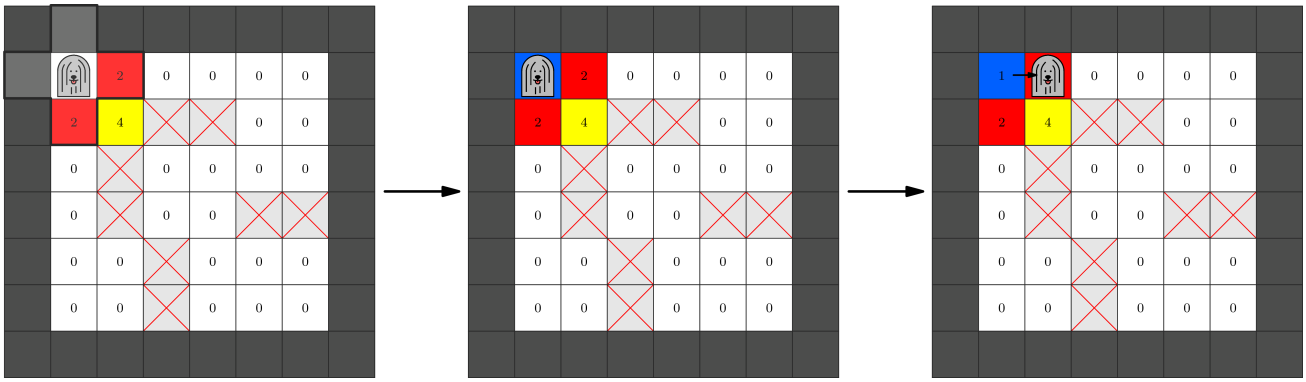
- 칸 (r, c)이 경계인 칸이라면 상태는 -2이다.
- 칸 (r, c)이 장애물 칸이라면 상태는 -1이다.
- 칸 (r, c)이 빈 칸이라면 상태는 이 칸의 색과 같다.

Pulibot의 프로그램은 여러 단계의 스텝으로 이루어져 있다. 각각 스텝에서, Pulibot은 인접한 칸의 상태를 알아낸 후 명령을 수행한다. 수행할 명령은 알아낸 상태들에 따라 결정된다. 이를 보다 정확히 설명하면 다음과 같다.

Pulibot이 빈 칸 (r, c)에 있고 한 스텝을 실행하려고 한다. 이 스텝은 다음과 같이 진행된다.

1. 먼저 Pulibot은 현재 **상태 배열**을 알아낸다. 상태 배열 $S = [S[0], S[1], S[2], S[3], S[4]]$은 칸 (r, c)과 인접한 칸들의 상태값들이다.
    - $S[0]$은 칸 (r, c)의 상태이다.
    - $S[1]$은 칸 (r, c)의 서쪽 칸의 상태이다.
    - $S[2]$은 칸 (r, c)의 남쪽 칸의 상태이다.
    - $S[3]$은 칸 (r, c)의 동쪽 칸의 상태이다.
    - $S[4]$은 칸 (r, c)의 북쪽 칸의 상태이다.
2. 다음 Pulibot은 상태 배열에 따라 결정되는 **명령** (Z, A)를 정한다.
3. 마지막으로 Pulibot은 명령을 수행한다. 칸 (r, c)의 색을 Z로 바꾼 뒤 행동 A를 수행하는데, 이는 다음 중 하나이다.
    - 칸 (r, c)에 **머뭄**
    - 4개의 인접한 칸 중 하나로 **이동**
    - **프로그램 종료**

예를 들어, 다음 그림 왼쪽에 있는 시나리오를 생각해보자. Pulibot은 칸 $(0, 0)$에 있고 이 칸의 색은 0이다. Pulibot은 상태 배열 $S = [0, -2, 2, 2, -2]$을 알아낸다. Pulibot의 프로그램이 만약 이 상태 배열이 주어졌을 때 칸의 색을 $Z = 1$로 바꾸고 동쪽 칸으로 이동하게 한다면, 이 과정은 그림의 가운데와 오른쪽 그림에서 보여주는 것과 같다.



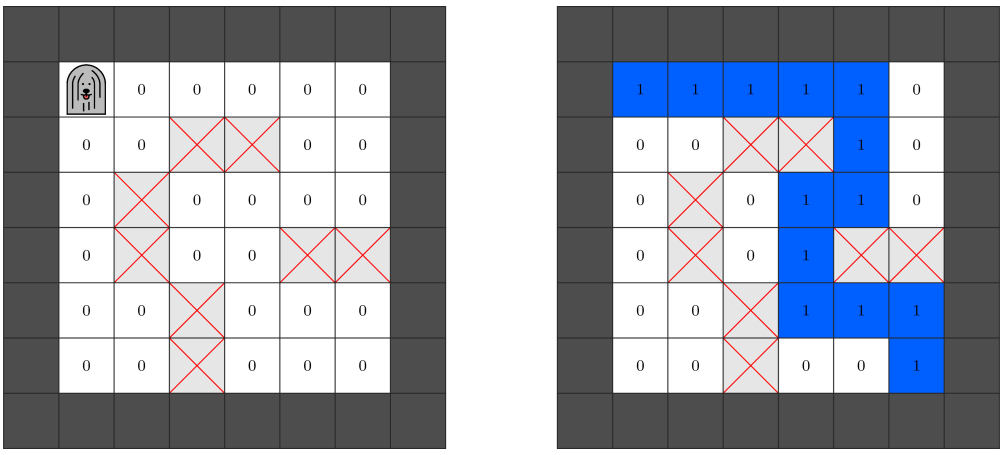
Robot Contest Rules

- 처음 Pulibot은 칸 $(0, 0)$에서 프로그램을 시작한다.
- Pulibot은 비어있지 않은 칸으로 이동할 수 없다.
- Pulibot의 프로그램은 최대 $500\,000$ 스텝까지 수행할 수 있다. 즉, $500\,000$ 스텝을 수행한 후에는 종료해야 한다.

- Pulibot의 프로그램을 종료한 후에는, 미로의 빈 칸들의 색은 다음 규칙을 따라야 한다.
    - 칸 $(0, 0)$에서 칸 $(H - 1, W - 1)$으로 가는 최단 경로 중 하나에 대해서, 이 경로에 포함된 칸의 색은 1이다.
    - 이 경로에 포함되지 않은 다른 모든 빈 칸의 색은 0이다.

- Pulibot은 어떤 빈 칸에서든지 프로그램을 종료할 수 있다.

예를 들어, 다음 그림은 $H = W = 6$인 미로 중 하나를 나타낸다. 처음 미로의 상태는 왼쪽 그림에 있고, 타당한 종료 후 미로의 상태 중 하나는 오른쪽 그림과 같다.



Implementation Details

다음 함수를 구현해야 한다.

```
void program_pulibot()
```

- 이 함수는 Pulibot의 프로그램을 만들어야 한다.이 함수가 만든 프로그램은 제약 조건을 만족하는 모든 미로에 대해서 정확히 동작해야 한다.
- 이 함수는 각 테스트 케이스마다 정확히 한 번 호출된다.

이 함수는 Pulibot의 프로그램을 만들기 위해서 다음 함수를 호출할 수 있다.

```
void set_instruction(int[] S, int Z, char A)
```

- S: 상태 배열을 나타내는 길이 5인 배열.
- Z: 색을 나타내는 음이 아닌 정수
- A: Pulibot의 행동을 나타내는 한 글자. 각 글자의 뜻은 다음과 같다.
  - `H`: 머뭄
  - `W`: 서쪽 칸으로 이동
  - `S`: 남쪽 칸으로 이동
  - `E`: 동쪽 칸으로 이동
  - `N`: 북쪽 칸으로 이동
  - `T`: 프로그램 종료
- 이 함수를 호출하면 Pulibot이 상태 배열 S에 대해서 명령 (Z, A)을 수행하게 된다.

동일한 상태 배열 S에 대해서 이 함수를 여러번 호출하면 `Output isn't correct`를 받게 된다.

모든 가능한 상태 배열 S에 대해서 `set_instruction`을 호출할 필요는 없다. 그러나 Pulibot이 알아낸 상태 배열에 대해서 이에 대한 명령이 정해져 있지 않으면, `Output isn't correct`를 받게 된다.

`program_pulibot` 호출이 종료하면 그레이더는 하나 이상의 미로에 대해서 Pulibot의 프로그램을 실행시킨다. 함수를 호출하는데 필요한 시간은 여러분의 프로그램의 시간 제한에 포함되지 *않는다.* 그레이더는 *적응적*이지 않다. 즉, 각 테스트 케이스의 미로들은 이미 정해져 있다.

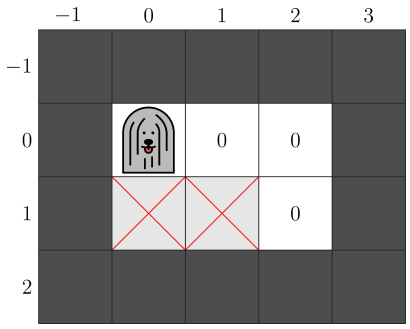
Pulibot의 프로그램이 종료하기 전에 로봇 대회 규칙을 어기면, `Output isn't correct`를 받게 된다.

Example

`program_pulibot` 함수가 `set_instruction`을 다음과 같이 호출했다고 하자.

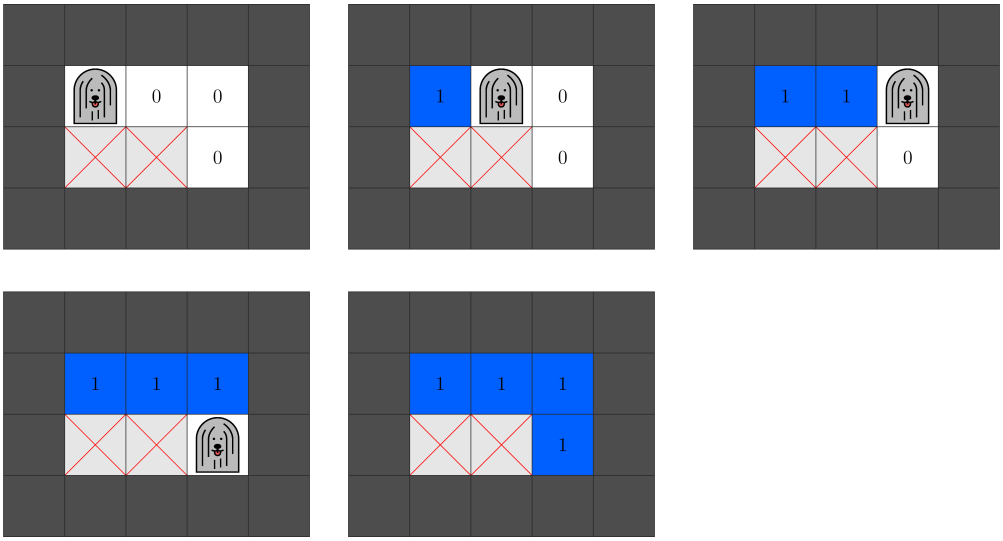
호출	상태 배열 S에 대한 명령
`set_instruction([0, -2, -1, 0, -2], 1, E)`	색을 1로 바꾸고 동쪽으로 이동
`set_instruction([0, 1, -1, 0, -2], 1, E)`	색을 1로 바꾸고 동쪽으로 이동
`set_instruction([0, 1, 0, -2, -2], 1, S)`	색을 1로 바꾸고 남쪽으로 이동
`set_instruction([0, -1, -2, -2, 1], 1, T)`	색을 1로 바꾸고 프로그램 종료

$H = 2, W = 3$이고 미로가 다음 그림과 같다고 하자.



이 미로에 대해서 Pulibot 의 프로그램의 동작은 네 스텝이다. Pulibot이 알아낸 상태 배열과 그에 대한 동작은 정확히 위의 표에서 차례대로 `set_instruction`에 주어진 파라미터와 같다. 마지막 명령에 의해서 프로그램이 종료한다.

다음 그림은 네 스텝 각각을 수행하기 전 미로의 상태와, 프로그램이 종료한 후 미로의 최종 상태를 나타낸다.



위 4개의 명령으로 된 프로그램이 다른 타당한 미로에 대해서 최단 경로를 찾지 못할 수 있다는데 주의하라. 따라서 그대로 제출한다면, `Output isn't correct`를 받을 수 있다.

Constraints

$Z_{MAX} = 19$. 따라서, Pulibot은 0 이상 19 이하의 색을 사용할 수 있다.

Pulibot을 테스트하는 모든 미로에 대해서

- $2 \leq H, W \leq 15$
- 칸 $(0, 0)$에서 칸 $(H-1, W-1)$으로 가는 경로가 최소한 하나는 있다.

Subtasks

1. (6 points) 미로에는 장애물 칸이 없다.
2. (10 points) $H = 2$
3. (18 points) 어떤 두 빈 칸에 대해서도 이 둘을 잇는 경로는 정확히 하나 있다.
4. (20 points) 칸 $(0, 0)$에서 칸 $(H-1, W-1)$으로 가는 최단 경로는 길이가 $H + W - 2$이다.
5. (46 points) 추가적인 제약 조건이 없다.

어떤 테스트 케이스에서 `set_instruction` 호출이나 Pulibot의 프로그램이 Implementation Details에서 설명한 제약 조건을 어긴다면, 해당 서브태스크에 대한 점수는 0점이다.

각 서브태스크에서, 거의 정확하게 색을 칠했다면 부분 점수를 받을 수 있다.

구체적으로는

- 만약 프로그램이 종료했을 때 빈 칸들의 색이 로봇 대회 규칙을 만족한다면 **정답**이다.
- 만약 프로그램이 종료했을 때 정답이 아니면서 빈 칸들의 색이 다음과 같다면 **부분 정답**이다
  - 칸 $(0, 0)$에서 칸 $(H-1, W-1)$으로 가는 최단 경로 중 하나에 대해, 이 경로에 포함된 칸의 색은 1이다.
  - 이 경로에 포함되지 않은 다른 모든 빈 칸의 색은 1이 아니다.

만약 여러분의 답이 정답이나 부분 정답이 아니라면, 해당 테스트 케이스에 대한 점수는 0이다.

서브태스크 1-4에서, 답이 정답이면 만점을, 답이 부분 정답이라면 만점의 50%를 받는다.

서브태스크 5에서, Pulibot의 프로그램이 사용한 색의 가짓수에 따라 점수가 정해진다. 보다 정확히는, $Z^\star$가 `set_instruction`의 모든 호출에서 사용한 Z의 최대값이라고 하자. 테스트 케이스에 대한 점수는 다음 표와 같다.

조건	점수(정답)	점수(부분 정답)
$11 \le Z^\star \le 19$	$20 + (19 - Z^\star)$	$12 + (19 - Z^\star)$
$Z^\star = 10$	31	23
$Z^\star = 9$	34	26
$Z^\star = 8$	38	29
$Z^\star = 7$	42	32
$Z^\star \le 6$	46	36

각 서브태스크의 점수는 해당하는 서브태스크에 포함된 테스트 케이스들의 점수 중 최소값이다.

Sample Grader

샘플 그레이더는 다음 양식으로 입력을 읽는다.

- line 1: H W
- line $2 + r$ ($0 \le r < H$): $m[r][0]$ $m[r][1]$ $\ldots$ $m[r][W-1]$

여기에서 m은 길이 W의 정수 배열이 H개 있는 배열로, 미로에서 경계가 아닌 칸들에 대한 정보를 나타낸다.
칸 (r, c)가 빈 칸이면 $m[r][c] = 0$이고, 칸 (r, c)가 장애물 칸이면 $m[r][c] = 1$이다.

샘플 그레이더는 먼저 `program_pulibot()`를 호출한다. 샘플 그레이더가 규칙 위반을 탐지하면 `Protocol Violation: <MSG>`를 출력하고 종료하는데, `<MSG>`는 다음 에러 메시지 중 하나이다.

- `Invalid array`: 어떤 i에 대해 $-2 \le S[i] \le Z_{MAX}$가 아니거나, S의 길이가 5가 아니다.
- `Invalid color`: $0 \le Z \le Z_{MAX}$ 조건이 만족되지 않음.
- `Invalid action`: 글자 A가 `H`, `W`, `S`, `E`, `N`, `T` 중 하나가 아님.
- `Same state array`: 동일한 배열 S에 대해 `set_instruction`가 두 번 이상 호출됨

그 외의 경우에 `program_pulibot`가 종료하면, 샘플 그레이더는 입력된 미로에 대해 Pulibot의 프로그램을 수행한다.

샘플 그레이더는 두 가지를 출력한다.

먼저, 샘플 그레이더는 Pulibot의 동작에 대한 로그를 작업 디렉토리에 있는 `robot.bin`에 쓴다. 이 파일은 다음에 설명하는 시각화 툴의 입력이 된다.

다음, Pulibot의 프로그램이 성공적으로 종료하지 않을 경우, 샘플 그레이더는 다음 에러 메시지 중 하나를 출력한다.

- `Unexpected state`: Pulibot이 알아낸 상태 배열에 대해 `set_instruction`가 호출된 적이 없다.
- `Invalid move`: 프로그램에 의해 Pulibot이 비어있지 않은 칸으로 이동했다.
- `Too many steps`: Pulibot이 $500\,000$개의 스텝을 수행했지만 프로그램이 종료하지 않았다.

그 외의 경우에, $e[r][c]$이 Pulibot의 프로그램이 종료한 후 칸 (r, c)의 색이라고 하자. 샘플 그레이더는 H 줄을 다음 양식에 따라 찍는다.

- Line $1 + r$ ($0 \le r < H$): $e[r][0]$ $e[r][1]$ $\ldots$ $e[r][W-1]$

Display Tool

이 문제에 대한 패키지에는 `display.py`라는 이름의 파일이 있다. 실행시키면, 이 Python 스크립트는 샘플 그레이더의 입력에 기술된 미로에 대해서 Pulibot의 동작을 보여준다. 바이너리 파일 `robot.bin`은 작업 디렉토리에 있어야 한다.

이 스크립트를 실행시키려면 다음 명령을 실행하면 된다.

```
python3 display.py
```

간단한 그래픽 인터페이스가 나타난다. 주 기능은 다음과 같다.

- 전체 미로의 상태를 볼 수 있다. Pulibot의 현재 위치는 직사각형에 의해 강조된다.
- 화살표 버튼을 누르거나 해당하는 핫키를 눌러서 각 단계 Pulibot의 동작을 볼 수 있다.
- Pulibot 프로그램의 다음 스텝이 맨 아래에 있다. 현재의 상태 배열과 앞으로 수행할 명령을 보여준다. 마지막 스텝이 끝난 뒤에 프로그램이 잘못되었다면 그레이더가 출력하는 에러 메시지를 보여주고, 성공적으로 종료했다면 `Terminated`를 출력한다.
- 각 색을 나타내는 숫자마다 텍스트를 보여주는 것 뿐 만 아니라 배경색을 설정할 수 있다. 텍스트는 같은 색인 칸마다 출력될 짧은 문자열이다. 배경색과 텍스트는 다음 방법으로 정할 수 있다.
  - `Colors` 버튼을 누른 후 대화 윈도우에서 설정한다.
  - `colors.txt` 파일을 수정한다.
- `robot.bin`를 다시 로드하려면 `Reload` 버튼을 사용한다. 만약 `robot.bin` 파일의 내용이 바뀌었다면 이 기능을 사용하면 편리하다.