문제. 벽 피하기

벽 피하기 게임에서는 $N \times M$ 크기의 격자가 주어진다. 격자의 각 칸에 대하여 0은 빈 칸, 1은 벽, 2는 플레이어의 위치를 의미한다. 플레이어는 1명이다.

플레이어는 초기에 항상 가장 아래 행(줄)의 한 위치에 존재한다. 플레이어는 매 번 (1) 가만히 있거나, (2) 왼쪽으로 한 칸 이동하거나, 혹은 (3) 오른쪽으로 한 칸 이동할 수 있다. 플레이어가 이동한 뒤에는 위에서부터 존재하는 모든 벽이 한 칸씩 아래로 내려온다.

구체적으로 1초동안 ① 초기 플레이어가 먼저 행동하고, ② 이후에 모든 벽에 해당하는 칸이 한 칸씩 아래로 내려온다. 플레이어가 벽에 부딪힌다면 즉시 게임이 종료되며, 부딪히지 않은 경우 플레이어가 생존한 시기가 1초만큼 증가하는 것으로 간주한다.

벽이 내려온 직후에 플레이어가 살아남았을 경우 1초만큼 생존한 시간이 증가한 것으로 간주하기 때문에, 만약 처음부터 플레이어가 존재하는 위치의 윗줄의 칸들이 모두 벽으로 완전히 가로막혀있다면 정답으로는 0초를 출력해야 할 것이다. 예를 들어 다음과 같은 예시가 있다.

1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	2	0	0	0	0

플레이어는 벽 피하기 게임으로부터 최대한 오랜 시간을 버티는 것이 목표다. 전체 행(줄)의 수가 최대 N이라는 점에서 최대로 버틸 수 있는 시간은 $N - 1$초이다. 결과적으로 플레이어가 최적의 움직임을 보인다고 했을 때, **플레이어가 생존 가능한 최대 시간**을 계산하는 프로그램을 작성하여라.

예를 들어 $N = 9$, $M = 10$인 상황을 가정하자. $N \times M$ 크기의 각 공간에 대한 정보는 아래의 2차원 배열과 같다.

0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1

1	1	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	2	0	0	0	0	0	0

초기 3초 동안 플레이어가 "왼쪽 → 가만히 → 가만히" 순서대로 행동한다면, 게임 시작 이후 3초 뒤에 전체 공간의 형태는 다음과 같을 것이다.

0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	2	0	0	0	0	0	0	0

이후에 "오른쪽 → 오른쪽 → 가만히" 순서대로 행동한다면, 게임 시작 이후 6초 뒤에 전체 공간의 형태는 다음과 같을 것이다.

0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	0	2	1	1	1	1	1

현재 상황에서는 플레이어가 어떤 행동을 취해도 벽에 부딪혀 게임이 종료된다. 결과적으로 현재 예시에서는 플레이어가 최적의 행동을 취했을 때 최대 6초까지 생존이 가능한 것으로 이해할 수 있다.

초기 게임 보드 판의 정보가 2차원 배열 형태로 주어졌을 때, **플레이어가 생존 가능한 최대 시간**을 반환하는 프로그램을 작성하여라.

입력 조건

가장 먼저 벽 피하기 게임을 진행할 보드 판에 대한 크기 정보 N과 M이 주어진다. N과 M은 3보다 크거나 같고 100보다 작거나 같은 자연수다.
이어서 벽 피하기 게임을 진행할 보드 판에 대한 정보가 담긴 $N \times M$ 크기의 2차원 배열 $board$가 주어진다. 격자의 각 칸에 대하여 0은 빈 칸, 1은 벽, 2는 플레이어의 초기 위치를 의미한다. 플레이어는 항상 1명이며, 플레이어는 초기에 가장 아래 행(줄)의 한 위치에 존재한다.

출력 조건

플레이어가 최선을 선택을 함으로써 생존 가능한 최대 시간을 반환한다.

입출력 예시

N	M	$board$	정답
9	10	[[0, 0, 1, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [1, 1, 1, 1, 0, 0, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0, 0, 0, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 2, 0, 0, 0, 0, 0, 0]]	6

해설 2. 벽 피하기

본 문제는 가장 아래 행부터 시작하여 가장 위쪽 행까지, 차례대로 각 위치까지의 도달 가능 여부를 기록하여 문제를 해결할 수 있다. 문제에서 언급된 예시에서 초기에 플레이어가 존재하는 행을 포함하여 아래의 5가지 행까지만 고려해보자. 이는 아래 그림에서 빨간색 음영으로 처리된 영역과 같다.

0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	2	0	0	0	0	0	0

가장 아래 행부터 5번째 행까지 모든 빈 칸에 대하여, 각 위치까지의 도달 가능 여부를 **가능(T)**과 **불가능(F)**으로 표현하면 다음과 같다.

0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1
T	T	T	T	T	F	F	F	F	F
T	T	T	T	F	F	F	F	F	F
1	T	T	1	F	F	F	F	1	1
F	F	T	1	1	1	1	1	1	1

0	0	T	2	T	0	0	0	0	0

본 문제의 요구사항대로 가장 아래 행부터 시작하여 위쪽 방향으로 한 칸씩 이동한다고 가정하자. 플레이어가 도달 가능한 모든 위치에 대하여 (1) 바로 위, (2) 왼쪽 위, (3) 오른쪽 위 위치에 대하여 이동이 가능한지 확인하는 방법으로 문제를 해결할 수 있다. 현재 행의 특정한 위치에 도달 가능한지 결정하기 위해서는 이전 행(아래 행)의 각 위치에 대한 도달 가능 여부 정보가 활용된다. 본 문제는 **다이나믹 프로그래밍(DP)**를 활용하여 해결할 수 있는 문제로 이해할 수 있다.

다시 말해 가장 아랫줄의 초기 플레이어가 존재하는 위치부터 시작하여 한 줄(행)씩 윗줄로 이동하면서 점화식에 따라서 **DP 테이블**을 갱신하여 문제를 해결할 수 있다. 구체적으로 도달 가능한 모든 위치(T)에 대하여 세 가지 위치 (1) 바로 위, (2) 왼쪽 위, (3) 오른쪽 위 위치에 대하여 일일이 확인하여 DP 테이블을 업데이트한다.

문제를 해결하기 위해 다양한 구현 방식이 존재한다. 아래에서 위로 올라가므로, 바로 **윗칸에 대해서만 값을 업데이트** 하고, 옆칸에 대해서는 DP 테이블을 업데이트 하지 않는 방식으로 해결할 수 있다. 구체적인 구현 방법으로는, 1초씩 증가시키며 특정한 **도달 가능한 위치(T)**를 기준으로 다음의 공식에 따라서 DP 테이블을 갱신하면 된다.

1) 위쪽이 빈 칸이라면: 위쪽을 도달 가능(T)으로 처리
2) 왼쪽과 왼쪽 위가 모두 빈 칸이라면: 왼쪽 위를 도달 가능(T)으로 처리
3) 오른쪽과 오른쪽 위가 모두 빈 칸이라면: 오른쪽 위를 도달 가능(T)으로 처리

이러한 점화식에 따라서 DP 테이블을 갱신한다면, 결과적으로 $O(3 \times N \times N) = O(N^2)$의 시간 복잡도로 본 문제를 해결할 수 있다. 점화식에 따라서 DP 테이블을 계산한 뒤에는 **도달 가능(T)한 위치**가 존재하는 가장 높은 행(가장 아래 행부터 가장 멀리 떨어진 행)이 어디인지를 계산하여 문제를 해결할 수 있다. 결과적으로 본 문제는 **시뮬레이션 및 다이나믹 프로그래밍 유형**에 속한다.

- **Python3 정답 코드 예시**

```
# 보드 판의 크기(N, M)와 벽 피하기 게임을 진행할 보드 판 정보 배열(board) 입력받기
def solution(N, M, board):
```

```
    # N X M 크기의 배열 위치에 대하여 처음에는 모두 도달 불가능(F)하다고 가정
    dp = [[False] * M for i in range(N)]
    # 가장 아랫줄의 위치를 하나씩 확인하며
    for j in range(M):
        if board[N - 1][j] == 2:  # 초기 플레이어가 존재하는 위치라면
            dp[N - 1][j] = True  # 해당 위치에 도달 가능(T)한 것으로 수정
    # 가장 아랫줄부터 위에서 두 번째 줄까지 확인하며
    for i in range(N - 1, 0, -1):
        for j in range(M):
            if dp[i][j] == True:  # 현재 위치가 도달 가능한 위치인 경우
                # 바로 위쪽 위치가 빈 칸이라면
                if board[i - 1][j] == 0:
                    dp[i - 1][j] = True  # 바로 위쪽 위치에 도달 가능(T)
                # 왼쪽과 왼쪽 위 위치가 모두 빈 칸이라면
                if j > 0:
                    if board[i][j - 1] == 0 and board[i - 1][j - 1] == 0:
                        # 왼쪽 위 위치에 도달 가능(T)
                        dp[i - 1][j - 1] = True
                # 오른쪽과 오른쪽 위 위치가 모두 빈 칸이라면
                if j < M - 1:
                    if board[i][j + 1] == 0 and board[i - 1][j + 1] == 0:
                        # 오른쪽 위 위치에 도달 가능(T)
                        dp[i - 1][j + 1] = True
    answer = 0  # 주인공이 최대 몇 초까지 살아남을 수 있는지
    # 가장 아랫줄부터 가장 윗줄까지 확인하며
    for i in range(N - 1, -1, -1):
        # 현재 행에 도달 가능한 위치가 하나라도 있는 경우
        if dp[i].count(True) >= 1:
            answer = (N - i) - 1
    return answer
```

- **Java 정답 코드 예시**

```
import java.util.*;

class Solution {
    // 보드 판의 크기(N, M)와 보드 판 정보 배열(board) 입력받기
    public static int solution(int N, int M, int[][] board) {
        // N X M 크기의 배열 위치에 대하여 처음에는 모두 도달 불가능(F)하다고 가정
        boolean[][] dp = new boolean[N][M];
        // 가장 아랫줄의 위치를 하나씩 확인하며
```

```
for (int j = 0; j < M; j++) {
    if (board[N - 1][j] == 2) { // 초기 플레이어가 존재하는 위치라면
        dp[N - 1][j] = true; // 해당 위치에 도달 가능(T)한 것으로 수정
    }
}
// 가장 아랫줄부터 위에서 두 번째 줄까지 확인하며
for (int i = N - 1; i > 0; i--) {
    for (int j = 0; j < M; j++) {
        // 현재 위치가 도달 가능한 위치인 경우
        if (dp[i][j] == true) {
            // 바로 위쪽 위치가 빈 칸이라면
            if (board[i - 1][j] == 0) {
                // 바로 위쪽 위치에 도달 가능(T)
                dp[i - 1][j] = true;
            }
            // 왼쪽과 왼쪽 위 위치가 모두 빈 칸이라면
            if (j > 0) {
                if (board[i][j - 1] == 0 && board[i - 1][j - 1] == 0) {
                    // 왼쪽 위 위치에 도달 가능(T)
                    dp[i - 1][j - 1] = true;
                }
            }
            // 오른쪽과 오른쪽 위 위치가 모두 빈 칸이라면
            if (j < M - 1) {
                if (board[i][j + 1] == 0 && board[i - 1][j + 1] == 0) {
                    // 오른쪽 위 위치에 도달 가능(T)
                    dp[i - 1][j + 1] = true;
                }
            }
        }
    }
}
int answer = 0; // 주인공이 최대 몇 초까지 살아남을 수 있는지
// 가장 아랫줄부터 가장 윗줄까지 확인하며
for (int i = N - 1; i >= 0; i--) {
    // 현재 행에 도달 가능한 위치가 하나라도 있는 경우
    for (int j = 0; j < M; j++) {
        if (dp[i][j] == true) {
            answer = (N - i) - 1;
        }
    }
}
```

```
        return answer;
    }
}
```