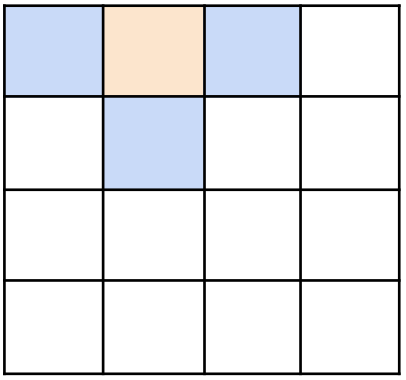
문제. 여행가의 등산

- 시간 제한: 1초
- 메모리 제한: 256MB
- 난이도: ★★★☆☆
- 문제 유형: 최단 경로, 다익스트라 최단 경로 알고리즘

$N \times N$ 보드 판 형태로 산이 존재하며, 여행가는 등산을 하고자 한다. 여행가의 위치는 초기에 (1, 1)로 가장 왼쪽 위의 칸에 서 있으며, 결과적으로 가장 오른쪽 아래의 위치인 (N, N)의 위치로 이동하는 것이 목표다.

보드 판의 각 위치는 0 이상 100 이하의 정수로 표현되는데, 이것은 해당 위치의 고도(높이)를 의미한다. 시작 위치에 해당하는 (1, 1) 위치의 고도(높이) 값은 항상 0이며, 도착 위치인 (N, N)까지 최소한의 체력을 소모하여 이동하는 것이 목표다.

여행가는 자신의 위치에서 인접한 상, 하, 좌, 우 위치로 이동이 가능하다. 예를 들어 $N = 4$인 상황을 가정하자. 이때, 여행가가 (1, 2)의 위치에 있다면 보드 판의 형태를 다음의 그림처럼 표현할 수 있을 것이다. 그림에서 여행가의 위치는 빨간색 음영으로 색칠하였고, 여행가가 이동 가능한 위치는 파란색 음영으로 색칠하였다.



여행가는 산을 벗어나지만 않는다면 현재 위치에서 인접한 위치(상, 하, 좌, 우)로 이동이 가능하다. 이때 현재 위치에서 인접한 위치 중 하나를 골라 다음 위치로 이동할 때, 고도가 다르더라도 이동이 가능하다. 단, 현재 위치와 이동할 인접 위치의 높이(고도)가 다르다면, 이동하기 위해 고도의 차이만큼 체력이 소모된다. 반면에 고도가 동일하다면 체력은 소모되지 않는다.

여행가가 (1, 1)의 위치에서 (N, N)의 위치로 이동하기 위한 체력 소모량의 최소 값을 계산하는 프로그램을 작성하여라.

$N = 5$일 때의 한 예시를 확인해 보자. 보드 판에서 각 위치에 쓰인 값은 해당 위치의 고도(높이)를 의미한다. 시작 위치에 해당하는 (1, 1) 위치의 고도(높이) 값은 항상 0이다.

0	0	0	0	1
1	1	1	0	1
1	4	4	4	4
1	1	4	3	3
0	1	3	1	1

각 위치는 (행, 열)을 의미한다고 가정하자. 현재 예시에서는 (1, 1) → (2, 1) → (3, 1) → (4, 1) → (4, 2) → (5, 2) → (5, 3) → (5, 4) → (5, 5) 순서대로 이동하면, 총 체력 소모량은 5이다. 그림으로 표현하면, (1, 1)에서 (5, 5)의 위치까지 이동하는 최단 경로에 대하여 음영으로 칠하였고, 이동을 수행했을 때 체력 소모가 발생하는 칸에 대하여 파란색으로 칠하였다.

0	0	0	0	1
1	1	1	0	1
1	4	4	4	4
1	1	4	3	3
0	1	3	1	1

이와 같이 움직일 때 체력 소모량은 5이며, 이것보다 적게 체력을 소모할 수 있는 경우는 존재하지 않는다. (최소 체력 소모량) 따라서 본 예시에서의 정답은 5이다.

입력 조건

가장 먼저 전체 공간의 크기 N이 주어진다. N은 300보다 작거나 같은 자연수이다.
이어서 보드 판 형태의 산의 각 위치에 대한 고도(높이) 정보가 담긴 $N \times N$ 크기의 배열 arr가 주어진다. 각 위치의 고도 값은 0 이상 100이하의 정수이다. 시작 위치에 해당하는 (1, 1) 위치의 고도(높이) 값은 항상 0이다.

출력 조건

여행가가 (1, 1)의 위치에서 (N, N)의 위치로 이동하기 위한 최소 체력 소모량을 반환한다.

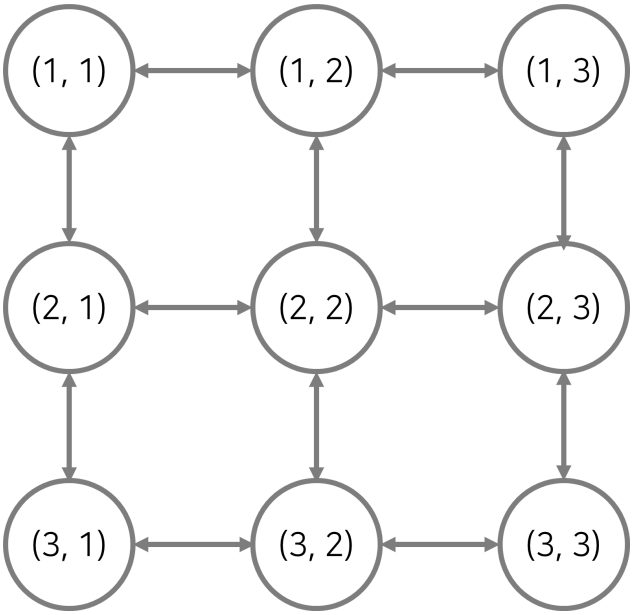
입출력 예시

N	arr	정답
5	[[0, 0, 0, 0, 1], [1, 1, 1, 0, 1], [1, 4, 4, 4, 4], [1, 1, 4, 3, 3], [0, 1, 3, 1, 1]]	5
6	[[0, 5, 6, 2, 3, 8], [1, 1, 1, 1, 1, 1], [5, 6, 5, 3, 2, 3], [2, 6, 5, 2, 2, 4], [7, 7, 7, 3, 5, 6], [6, 7, 8, 4, 4, 3]]	5

해설 3. 여행가의 등산

이 문제는 전체 산에 존재하는 $N \times N$개의 각 칸을 모두 개별 노드로 판단하여 그래프로 고려하면 보다 직관적으로 문제의 요구사항을 이해할 수 있다. 각 칸을 모두 개별적인 노드로 본다면, 상, 하, 좌, 우로 인접한 영역을 **인접 노드**로 판단할 수 있다.

$N = 3$인 상황을 가정하자. 이를 그래프로 표현하면 다음과 같이 나타낼 수 있다. 총 9개의 노드가 존재하며, 각 노드는 양방향 간선을 통해 인접한 노드와 연결되어 있는 것으로 이해할 수 있다. 따라서 이러한 아이디어로, **다익스트라 최단 경로 알고리즘**을 그대로 사용하여 문제를 해결할 수 있다.



만약 모든 위치에 대하여 인접한 위치와의 고도 차이가 동일하다는 가정이 있다면 BFS 기반의 접근 방법을 사용할 수도 있지만, 현재 문제에서는 각 칸의 고도(높이)가 다양한 값으로 구성된다. 따라서 본 문제는 (1, 1)의 위치에서 (N, N) 위치까지 도달하기 위한 최단 거리를 계산하는 문제로, 인접한 칸 사이의 **고도 차이**를 간선의 비용으로 간주한 뒤에 다익스트라 최단 경로 알고리즘을 사용하여 해결할 수 있다. 구체적으로 우선순위 큐를 활용한 다익스트라 알고리즘을 통해 문제를 해결할 수 있다.

N이 최대 300이므로, 가능한 전체 칸의 개수는 최대 90,000개로 이해할 수 있다. 각 노드에 대하여 인접한 노드는 약 4개로 볼 수 있으므로, 엄밀하지는 않지만 간선의 개수는 직관적으로 최대 약 360,000개 정도로 간주할 수 있을 것이다. 따라서, **다익스트라 최단 경로 알고리즘**을 이용하여 본 문제를 해결할 수 있다.

- **Python3 정답 코드 예시**

```
import heapq
```

```
INF = int(1e9)  # 무한을 의미하는 값으로 10억을 설정

# 인접한 상, 하, 좌, 우 방향 정보
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

# 전체 공간의 크기(N), 고도의 높이 정보 배열(arr)
def solution(N, arr):
    graph = [[None] * N for _ in range(N)]
    # 가능한 모든 위치 (x, y)에 대하여 확인
    for x in range(N):
        for y in range(N):
            # (x, y) 위치에서의 높이 값
            height = arr[x][y]
            # (x, y) 위치에서 인접한 4가지 위치 확인
            graph[x][y] = []  # 리스트 초기화
            for i in range(4):
                nx = x + dx[i]
                ny = y + dy[i]
                # 전체 공간을 벗어나는 경우 무시
                if nx < 0 or nx >= N or ny < 0 or ny >= N:
                    continue
                # 높이(고도) 차이 계산
                diff = abs(height - arr[nx][ny])
                # (nx, ny) 위치로 가려면 diff만큼 비용 필요
                graph[x][y].append(((nx, ny), diff))
    # 출발 위치
    start = (0, 0)
    # 각 위치까지의 거리 테이블
    distance = [[INF] * N for _ in range(N)]
    # 다익스트라 최단 경로 알고리즘 수행
    q = []  # 우선순위 큐 초기화
    # 시작 노드로 가기 위한 최단 경로는 0으로 설정하여, 큐에 삽입
    heapq.heappush(q, (0, start))
    x, y = start
    distance[x][y] = 0
    while q:  # 큐가 비어있지 않다면
        # 가장 최단 거리가 짧은 노드에 대한 정보 꺼내기
        dist, now = heapq.heappop(q)
        x, y = now
        # 현재 노드가 이미 처리된 적이 있는 노드라면 무시
```

```
        if distance[x][y] < dist:
            continue
        # 현재 노드와 연결된 다른 인접한 노드들을 확인
        for i in graph[x][y]:
            nx, ny = i[0]
            cost = dist + i[1]
            # 현재 노드를 거쳐 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[nx][ny]:
                distance[nx][ny] = cost
                heapq.heappush(q, (cost, (nx, ny)))
    # 가장 오른쪽 아래 목표 지점까지의 최단 거리(항상 도달 가능)
    answer = distance[N - 1][N - 1]
    return answer
```

- **Java 정답 코드 예시**

```
import java.util.*;

// 특정한 위치를 의미하는 Point 클래스
class Point {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

// 우선순위 큐에 들어가기 위한 노드(Node) 클래스
class Node implements Comparable<Node> {
    public Point point;
    public int distance;

    public Node(Point point, int distance) {
        this.point = point;
        this.distance = distance;
    }

    // 거리(비용)가 짧은 것이 높은 우선순위를 가지도록 설정
    @Override
```

```
    public int compareTo(Node other) {
        if (this.distance < other.distance) {
            return -1;
        }
        return 1;
    }
}

class Solution {
    // 무한을 의미하는 값으로 10억을 설정
    public static final int INF = (int) 1e9;

    // 인접한 상, 하, 좌, 우 방향 정보
    public static int[] dx = {-1, 1, 0, 0};
    public static int[] dy = {0, 0, -1, 1};

    // 전체 공간의 크기(N), 고도의 높이 정보 배열(arr)
    public static int solution(int N, int[][] arr) {
        // 각 노드에 연결되어 있는 노드에 대한 정보를 담는 배열
        ArrayList<Node>[][] graph = new ArrayList[N][N];

        // 가능한 모든 위치 (x, y)에 대하여 확인
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                // (x, y) 위치에서의 높이 값
                int height = arr[x][y];
                // 리스트 초기화
                graph[x][y] = new ArrayList<Node>();
                // (x, y) 위치에서의 인접한 4가지 위치 확인
                for (int i = 0; i < 4; i++) {
                    int nx = x + dx[i];
                    int ny = y + dy[i];
                    // 전체 공간을 벗어나는 경우 무시
                    if (nx < 0 || nx >= N || ny < 0 || ny >= N) {
                        continue;
                    }
                    // 높이(고도) 차이 계산
                    int diff = Math.abs(height - arr[nx][ny]);
                    // (nx, ny) 위치로 가려면 diff만큼 비용 필요
                    graph[x][y].add(new Node(new Point(nx, ny), diff));
                }
            }
```

```
}

// 다익스트라 최단 경로 알고리즘 수행
Point start = new Point(0, 0); // 출발 위치
// 각 위치까지의 거리 테이블
int[][] distance = new int[N][N];
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        // 초기 거리 값을 무한으로 초기화
        distance[i][j] = INF;
    }
}
// 우선순위 큐 초기화
PriorityQueue<Node> pq = new PriorityQueue<>();
// 시작 노드로 가기 위한 최단 경로는 0으로 설정하여, 큐에 삽입
pq.offer(new Node(start, 0));
int x = start.x;
int y = start.y;
distance[x][y] = 0;
// 큐가 비어있지 않다면
while(!pq.isEmpty()) {
    // 가장 최단 거리가 짧은 위치에 대한 정보 꺼내기
    Node node = pq.poll();
    int dist = node.distance; // 현재 위치까지의 비용
    Point now = node.point; // 현재 위치
    x = now.x;
    y = now.y;
    // 현재 위치가 이미 처리된 적이 있는 위치라면 무시
    if (distance[x][y] < dist) {
        continue;
    }
    // 현재 위치와 연결된 다른 인접한 위치들을 확인
    for (int i = 0; i < graph[x][y].size(); i++) {
        Point nextPoint = graph[x][y].get(i).point;
        int nx = nextPoint.x;
        int ny = nextPoint.y;
        int cost = distance[x][y] + graph[x][y].get(i).distance;
        // 현재 위치를 거쳐 다른 위치로 이동하는 거리가 더 짧은 경우
        if (cost < distance[nx][ny]) {
            distance[nx][ny] = cost;
            pq.offer(new Node(new Point(nx, ny), cost));
        }
```

```
            }
        }
        // 가장 오른쪽 아래 목표 지점까지의 최단 거리(항상 도달 가능)
        int answer = distance[N - 1][N - 1];
        return answer;
    }
}
```