터치 미션

- 시간 제한: 1초
- 메모리 제한: 256MB
- 난이도: ★★★☆☆
- 문제 유형: 완전 탐색, 너비 우선 탐색(BFS)

$N \times N$ 보드판 위에 2개 이상의 타겟 지점이 있다. 보드판 위에 참가자가 위치한 곳은 "P"로 표시된다. 벽은 "X"로, 비어있는 공간은 "B"로 표시된다. 각 타겟 지점은 "O"로 표시된다. 참가자는 1명이며, 이 참가자는 1초에 인접한 상, 하, 좌, 우의 위치 중에서 한 칸으로 이동할 수 있다. 벽이 위치한 곳으로는 이동할 수 없으며, 비어 있는 공간 혹은 타겟 지점으로는 이동할 수 있다.

이때, 참가자는 **터치 미션**이 시작되면 자신이 **도달할 수 있는 지점 중에서 가장 최단 거리가 먼 타겟 지점**에 도달해야 미션을 클리어할 수 있다. 즉, 참가자는 도달할 수 있는 지점 중에서 가장 먼 지점을 터치해야 클리어할 수 있다. 도달할 수 없는 지점은 터치해야 할 지점에서 배제한다.

예를 들어, 아래의 $N = 6$인 경우를 가정하자. (행, 열)의 형태로 위치를 나타낼 때, 현재 상황에서 (1, 2)가 가장 먼 타겟 지점이며 클리어에 7초가 필요하다. 행의 번호와 열의 번호는 1부터 시작한다고 가정하자.

B	O	B	B	B	B
X	X	X	X	X	B
B	B	O	B	P	B
X	X	B	X	B	B
B	B	O	X	B	B
B	B	B	B	B	B

참가자는 최소한의 시간으로 클리어하는 것이 목표다. 현재 예시에서는 참가자가 최적의 경로 (3, 5) → (3, 6) → (2, 6) → (1, 6) → (1, 5) → (1, 4) → (1, 3) → (1, 2)로 이동했을 때, 7초의 시간으로 클리어할 수 있다.

단, 참가자는 **두 가지 기술 중 하나를 사전에 사용**할 수 있다. 사전 사용 이후에 실질적으로 터치 미션이 진행되는 것이다. 두 가지 기술에 대한 구체적인 설명은 다음과 같다.

① **특정 타겟 제거**: 원하는 타겟 지점을 하나 선택하여 없앨 수 있다.
② **벽 없애기**: 벽을 하나 없앨 수 있다. 즉, 하나의 벽 X를 B로 바꾸는 것이다.

다시 말해, 참가자는 **터치 미션을 진행하기 전에 둘 중 하나의 기술을 사용**하여 결과적으로 터치 미션을 클리어하기 위한 시간을 줄일 수 있다. 우선 아래와 같이 "② **벽 없애기**" 기술을 사용하여 하나의 벽을 없애는 경우를 고려해 보자. 없어진 벽의 위치에 대해서는 음영 처리를 했다. 이 경우 답이 5로 줄어드는 것을 확인할 수 있다. 본 예시에서 (2, 5) 위치에 있는 벽이 아닌 다른 위치에 있는 벽을 제거하더라도 가장 먼 지점인 (1, 2)까지의 거리가 5보다 작아지지 않는다.

B	O	B	B	B	B
X	X	X	X	B	B
B	B	O	B	P	B
X	X	B	X	B	B
B	B	O	X	B	B
B	B	B	B	B	B

혹은 아래와 같이 "① **특정 타겟 제거**" 기술을 사용하여 (5, 3)에 존재하는 타겟 지점을 제거한다고 해보자. 이후에 터치 미션을 시작하면, 가장 먼 타겟 지점까지의 거리는 (1, 2)까지의 거리로 7이다. 따라서, 클리어를 위해 7초만큼의 시간이 소요된다.

B	O	B	B	B	B
X	X	X	X	X	B

B	B	O	B	P	B
X	X	B	X	B	B
B	B	B	X	B	B
B	B	B	B	B	B

반면에 아래와 같이 "① **특정 타겟 제거**" 기술을 사용하여 (1, 2)에 존재하는 타겟 지점을 제거한다고 해보자. 이후에 터치 미션을 시작하면, 가장 먼 타겟 지점까지의 거리는 (5, 3)까지의 거리로 4이다. 따라서, 클리어를 위해 4초만큼의 시간이 소요된다.

B	B	B	B	B	B
X	X	X	X	X	B
B	B	O	B	P	B
X	X	B	X	B	B
B	B	O	X	B	B
B	B	B	B	B	B

따라서 본 예시에서 최종적인 답은 4이다.

입력 조건

가장 먼저 보드의 크기 N이 자연수로 주어진다. N은 4 이상 40 이하의 자연수다.
이후에 $N \times N$ 크기의 보드의 정보가 담긴 배열 $board$가 주어진다. 참가자가 위치한 공간 "P"는 1개만 존재하며, 타겟 지점 "O"의 수는 최소 2개 이상 존재한다.

출력 조건

참가자가 **두 가지 기술 중 하나를 사전에 사용**할 수 있는 상황에서, 결과적으로 터치 미션을 클리어하는 최소 시간을 반환한다. 어떻게 하더라도 도달할 수 있는 타겟 지점이 없어 클리어할 수 없는 경우에는 -1을 반환한다.

입출력 예시

N	$board$	정답
6	[["B", "O", "B", "B", "B", "B"], ["X", "X", "X", "X", "X", "B"], ["B", "B", "O", "B", "P", "B"], ["X", "X", "B", "X", "B", "B"], ["B", "B", "O", "X", "B", "B"], ["B", "B", "B", "B", "B", "B"]]	4
6	[["P", "B", "B", "B", "B", "B"], ["B", "B", "B", "B", "B", "B"], ["B", "B", "X", "X", "X", "X"], ["B", "B", "X", "X", "X", "X"], ["B", "B", "X", "X", "X", "O"], ["B", "B", "X", "X", "O", "O"]]	-1
6	[["P", "B", "B", "B", "B", "B"], ["B", "B", "B", "B", "B", "B"], ["B", "B", "X", "X", "X", "X"], ["B", "B", "X", "X", "X", "X"], ["B", "B", "X", "X", "X", "O"], ["B", "B", "X", "B", "O", "O"]]	11

해설 3. 터치 미션

본 문제에서는 두 가지 기술을 사용할 수 있는 상황에서 클리어에 필요한 최소한의 시간을 계산하는 것이 목표다. 먼저, **첫 번째 기술**을 사용하는 경우 터치 미션이 시작되기 전에 "도달 가능한" 타겟 중에서 가장 먼 타겟 지점을 제거하는 것이 가장 큰 이득이 된다. 이어서 **두 번째 기술**의 경우, 각 벽("X")이 존재하는 위치마다 해당 벽을 빈 공간("B")으로 변경한 뒤에 매 경우의 수 마다 너비 우선 탐색(BFS)을 수행하는 방식으로 최적의 해를 계산할 수 있다. 결과적으로 도달할 수 있는 지점 중에서 **가장 최단 거리가 먼 타겟 지점**에 도달하기 위한 최소 시간을 반환하여 해결할 수 있다. 각 벽이 존재하는 위치의 개수를

$O(N^2)$로 간주한다면, 매 경우마다 너비 우선 탐색에 $O(N^2)$의 시간이 요구되므로, 결과적으로 최악의 경우 시간 복잡도가 $O(N^4)$인 정답 코드를 작성할 수 있다.

- **Python3 정답 코드 예시**

```
from collections import deque

dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

def bfs(N, temp, x, y):
    graph = [[-1] * N for _ in range(N)] # 최단 거리 맵
    q = deque()
    q.append((x, y))
    graph[x][y] = 0
    while q: # BFS 수행
        x, y = q.popleft()
        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            # 공간을 벗어난 경우 무시
            if nx < 0 or ny < 0 or nx >= N or ny >= N:
                continue
            # 벽인 경우 무시
            if temp[nx][ny] == "X":
                continue
            # 처음 방문하는 경우
            if graph[nx][ny] == -1:
                graph[nx][ny] = graph[x][y] + 1 # 최단 거리 기록
                q.append((nx, ny))
    return graph

# 전체 보드의 크기(N)와 각 보드 정보 배열(board)을 입력받기
def solution(N, board):
    answer = int(1e9)
    temp = [["B"] * N for _ in range(N)] # 2차원 배열 생성
    start_x = 0
    start_y = 0
    # 기술 ②: 두 번째로 먼 타켓 찾기
    targets = []
```

```
    for i in range(N):
        for j in range(N):
            temp[i][j] = board[i][j]
            if board[i][j] == "P":
                start_x = i
                start_y = j
            elif board[i][j] == "O":
                targets.append((i, j))
    graph = bfs(N, temp, start_x, start_y)
    result = []
    for (i, j) in targets:
        # 도달 가능한 타겟 지점들의 최단 거리 확인
        if graph[i][j] != -1:
            result.append(graph[i][j])
    result.sort() # 도달 가능한 타겟 지점들에 대한 최단 거리 순으로 정렬
    if len(result) >= 2: # 도달 가능한 타겟 지점이 두 개 이상이라면
        answer = result[len(result) - 2] # 두 번째로 먼 타켓까지의 최단 거리
    if len(result) == 1: # 도달 가능한 타겟 지점이 하나라면
        answer = result[0]
    # 기술 ①: 각 벽을 지워보며, BFS 수행하기
    for i in range(N):
        for j in range(N):
            if board[i][j] == "X":
                for x in range(N):
                    for y in range(N):
                        temp[x][y] = board[x][y]
                temp[i][j] = "B"  # 벽을 빈 공간으로 변경
                graph = bfs(N, temp, start_x, start_y)
                max_dist = -1 # 가장 먼 타겟 지점까지의 최단 거리
                for (x, y) in targets:
                    # 도달 가능한 타겟 지점들의 최단 거리 확인
                    if graph[x][y] != -1:
                        max_dist = max(max_dist, graph[x][y])
                if max_dist == -1: continue # 도달 가능한 곳이 없다면 무시
                answer = min(answer, max_dist)
    if answer == int(1e9): answer = -1
    return answer
```

- **Java 정답 코드 예시**

```
import java.util.*;
```

```
class Node {
    public int x;
    public int y;

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

class Solution {
    public static int[] dx = { -1, 1, 0, 0 };
    public static int[] dy = { 0, 0, -1, 1 };

    public static int[][] bfs(int N, String[][] temp, int x, int y) {
        int[][] graph = new int[N][N]; // 최단 거리 맵
        for (int i = 0; i < N; i++)
            Arrays.fill(graph[i], -1);
        Queue<Node> q = new LinkedList<>();
        q.offer(new Node(x, y));
        graph[x][y] = 0;
        while (!q.isEmpty()) { // BFS 수행
            Node cur = q.poll();
            x = cur.x;
            y = cur.y;
            for (int i = 0; i < 4; i++) {
                int nx = x + dx[i];
                int ny = y + dy[i];
                // 공간을 벗어난 경우 무시
                if (nx < 0 || ny < 0 || nx >= N || ny >= N)
                    continue;
                // 벽인 경우 무시
                if (temp[nx][ny].equals("X"))
                    continue;
                // 처음 방문하는 경우
                if (graph[nx][ny] == -1) {
                    // 최단 거리 기록
                    graph[nx][ny] = graph[x][y] + 1;
                    q.offer(new Node(nx, ny));
                }
```

```
        }
    }
    return graph;
}

// 전체 보드의 크기(N)와 각 보드 정보 배열(board)을 입력받기
public static int solution(int N, String[][] board) {
    int answer = (int) 1e9;
    String[][] temp = new String[N][N]; // 2차원 배열 생성
    for (int i = 0; i < N; i++)
        Arrays.fill(temp[i], "B");
    int startX = 0;
    int startY = 0;
    // 기술 ②: 두 번째로 먼 타켓 찾기
    ArrayList<Node> targets = new ArrayList<Node>();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            temp[i][j] = board[i][j];
            if (board[i][j].equals("P")) {
                startX = i;
                startY = j;
            } else if (board[i][j].equals("O")) {
                targets.add(new Node(i, j));
            }
        }
    }
    int[][] graph = bfs(N, temp, startX, startY);
    ArrayList<Integer> result = new ArrayList<Integer>();
    for (Node node : targets) {
        // 도달 가능한 타겟 지점들의 최단 거리 확인
        int i = node.x;
        int j = node.y;
        if (graph[i][j] != -1)
            result.add(graph[i][j]);
    }
    // 도달 가능한 타겟 지점들에 대한 최단 거리 순으로 정렬
    Collections.sort(result);
    if (result.size() >= 2) { // 도달 가능한 타겟 지점이 두 개 이상이라면
        // 두 번째로 먼 타켓까지의 최단 거리
        answer = result.get(result.size() - 2);
    }
    if (result.size() == 1) { // 도달 가능한 타겟 지점이 하나라면
```

```
            answer = result.get(0);
        }
        // 기술 ①: 각 벽을 지워보며, BFS 수행하기
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j].equals("X")) {
                    for (int x = 0; x < N; x++) {
                        for (int y = 0; y < N; y++) {
                            temp[x][y] = board[x][y];
                        }
                    }
                    temp[i][j] = "B"; // 벽을 빈 공간으로 변경
                    graph = bfs(N, temp, startX, startY);
                    int maxDist = -1; // 가장 먼 타겟 지점까지의 최단 거리
                    for (Node node : targets) {
                        int x = node.x;
                        int y = node.y;
                        // 도달 가능한 타겟 지점들의 최단 거리 확인
                        if (graph[x][y] != -1) {
                            maxDist = Math.max(maxDist, graph[x][y]);
                        }
                    }
                    // 도달 가능한 곳이 없다면 무시
                    if (maxDist == -1)
                        continue;
                    answer = Math.min(answer, maxDist);
                }
            }
        }
        if (answer == (int) 1e9)
            answer = -1;
        return answer;
    }
}
```