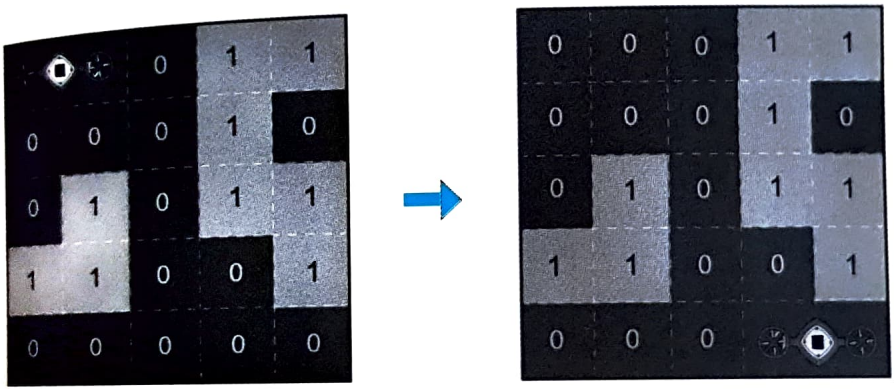
```
            index += 1
    # 모든 인구 이동이 끝난 경우
    if index == n * n:
        break
    total_count += 1

# 인구 이동 횟수 출력
print(total_count)
```

A 22 블록 이동하기

이 문제는 다소 복잡해 보이지만, 전형적인 BFS 문제 유형이다. 문제에서 로봇이 존재할 수 있는 각 위치(각 칸)를 노드로 보고, 인접한 위치와 비용이 1인 간선으로 연결되어 있다고 볼 수 있다. 간선의 비용이 모두 1로 동일하기 때문에 BFS를 이용하여 최적의 해를 구할 수 있다. 다시 말해 이 문제는 (1, 1)의 위치에 존재하는 로봇을 (N, N)의 위치로 옮기는 최단 거리를 계산하는 문제로 볼 수 있다.

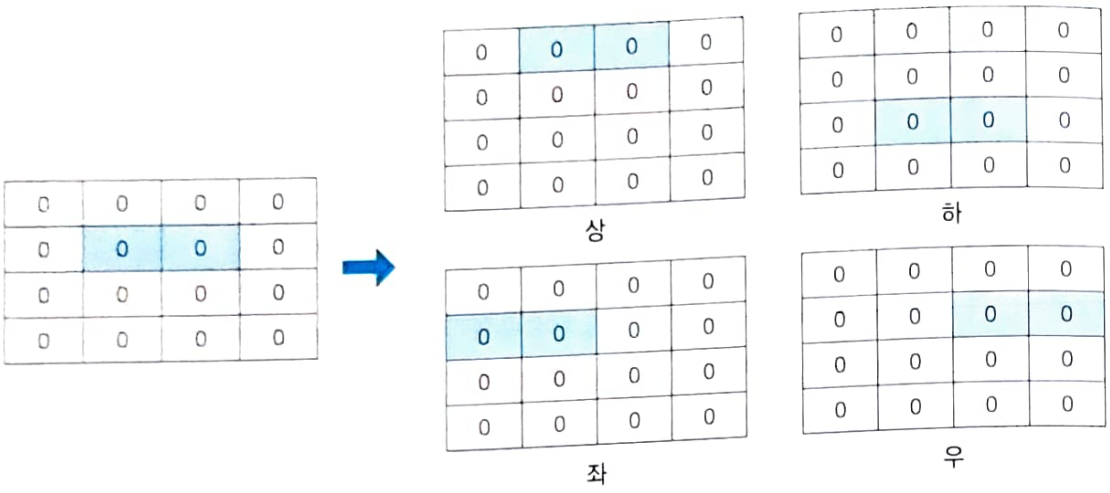


다만, 이 문제가 일반적인 BFS 문제와 다른 점은, 로봇이 차지하고 있는 위치가 두 칸이며 회전을 통해 이동할 수 있다는 점이다. 물론 로봇이 차지하고 있는 위치가 두 칸이라고 해도, 여전히 방문 여부를 관리할 수 있으니 걱정하지 말자. 바로 위치 정보를 튜플로 처리하면 된다. 로봇의 상태를 집합 자료형Set으로 관리한다고 가정해보자. 파이썬에서 {(1, 1), (1, 2)}와 {(1, 2), (1, 1)}은 같은 집합 객체로 처리된다. 따라서 이처럼 로봇의 상태를 집합 자료형을 이용하여 관리하면, 한 번 방문한(큐에 들어간) 자전거의 상태는 두 번 방문하지 않는다.

이제 로봇의 현재 상태가 주어졌을 때, 이동 가능한 다음 위치는 어떻게 계산할 수 있을지 고민해보자. 문제에서 로봇은 이동하거나 회전할 수 있다고 하였다.

step 1 이동

먼저 로봇이 단순히 '이동'하는 경우는 단순히 상, 하, 좌, 우로 이동하는 모든 경우를 계산하면 된다. 예를 들어 로봇이 가로로 놓인 상태에서 상, 하, 좌, 우로 이동하는 경우를 생각해보자. 아래 그림에서는 로봇이 존재하는 위치에 색을 칠하였다.

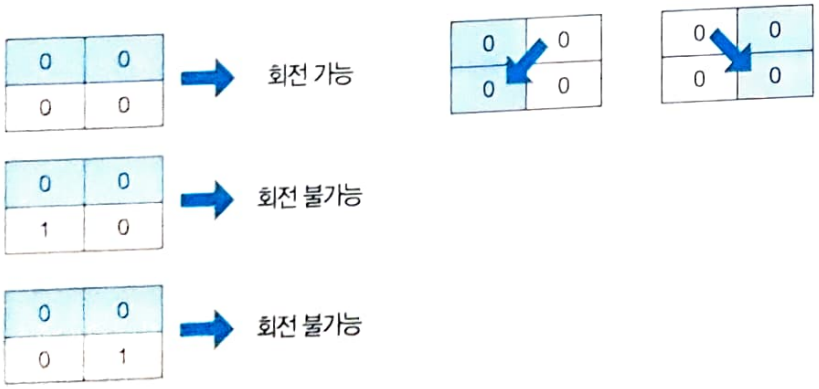


step 2 회전

회전의 경우 로봇이 가로로 놓여 있는 경우와 세로로 놓여 있는 경우를 모두 고려해야 한다.

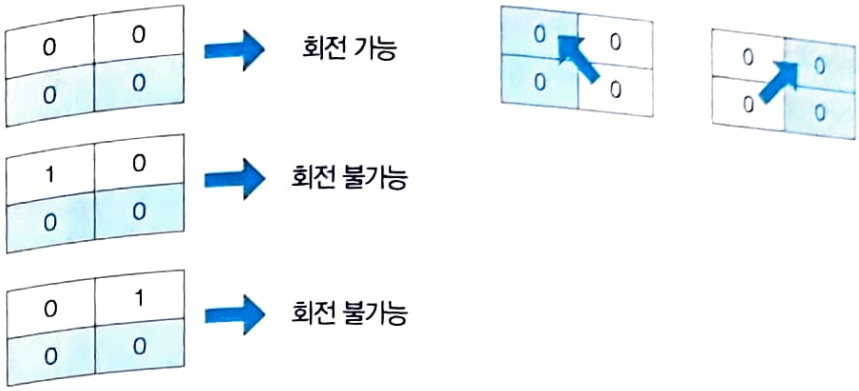
1 로봇이 가로로 놓인 상태에서 아래쪽으로 회전하는 경우

현재 로봇이 가로로 놓인 상태에서 아래쪽으로 회전하고자 한다면, 아래쪽에 벽이 없어야 한다. 따라서 아래쪽의 두 칸 중에서 하나라도 벽이 존재하는 경우(값이 1인 경우)를 제외하고, 회전을 수행할 수 있다.



2 로봇이 가로로 놓인 상태에서 위쪽으로 회전하는 경우

현재 로봇이 가로로 놓인 상태에서 위쪽으로 회전하고자 한다면, 위쪽에 벽이 없어야 한다. 따라서 위쪽의 두 칸 중에서 하나라도 벽이 존재하는 경우(값이 1인 경우)를 제외하고, 회전을 수행할 수 있다.



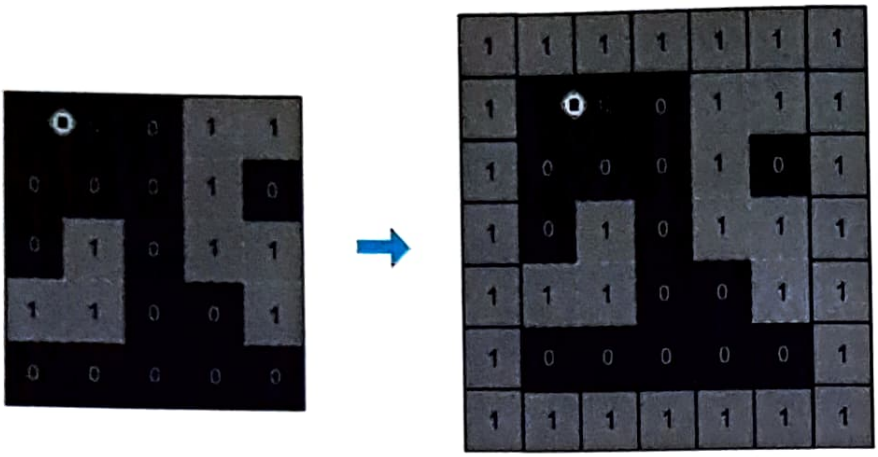
3 로봇이 세로로 놓인 상태에서 오른쪽으로 회전하는 경우

현재 로봇이 세로로 놓인 상태에서 오른쪽으로 회전하고자 한다면, 오른쪽에 벽이 없어야 한다. 따라서 오른쪽의 두 칸 중에서 하나라도 벽이 존재하는 경우(값이 1인 경우)를 제외하고, 회전을 수행할 수 있다.

4 로봇이 세로로 놓인 상태에서 왕쪽으로 회전하는 경우

현재 로봇이 세로로 놓인 상태에서 왼쪽으로 회전하고자 한다면, 왼쪽에 벽이 없어야 한다. 따라서 왼쪽의 두 칸 중에서 하나라도 벽이 존재하는 경우(값이 1인 경우)를 제외하고, 회전을 수행할 수 있다.

또한 소스코드를 간단하게 작성하기 위하여, 초기에 주어진 맵을 변형하여 외곽에 벽을 둘 수 있다. 이렇게 하면 로봇이 맵을 벗어나지 않는지, 그 범위 판정을 더 간단히 할 수 있다.



전체 소스코드는 다음과 같이 작성할 수 있다. 특정한 위치에서 이동 가능한 다음 위치를 반환하는 별도의 get_next_pos() 함수를 구현하여, 소스코드를 최대한 간결하게 작성하였다.

A22.py 답안 예시

```
# 이 코드는 다음 프로그래머스 사이트에서 테스트해야 정상 동작한다.
# https://programmers.co.kr/learn/courses/30/lessons/60063

from collections import deque

def get_next_pos(pos, board):
    next_pos = [] # 반환 결과(이동 가능한 위치들)
    pos = list(pos) # 현재 위치 정보를 리스트로 변환(집합 → 리스트)
    pos1_x, pos1_y, pos2_x, pos2_y = pos[0][0], pos[0][1], pos[1][0], pos[1][1]
    # (상, 하, 좌, 우)로 이동하는 경우에 대해서 처리
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    for i in range(4):
        pos1_next_x, pos1_next_y, pos2_next_x, pos2_next_y = pos1_x + dx[i], pos1_y +
dy[i], pos2_x + dx[i], pos2_y + dy[i]
        # 이동하고자 하는 두 칸이 모두 비어 있다면
        if board[pos1_next_x][pos1_next_y] == 0 and board[pos2_next_x][pos2_next_y] == 0:
            next_pos.append({(pos1_next_x, pos1_next_y), (pos2_next_x, pos2_next_y)})
    # 현재 로봇이 가로로 놓여 있는 경우
    if pos1_x == pos2_x:
        for i in [-1, 1]: # 위쪽으로 회전하거나, 아래쪽으로 회전
```

```
            if board[pos1_x + i][pos1_y] == 0 and board[pos2_x + i][pos2_y] == 0: # 위
쪽 혹은 아래쪽 두 칸이 모두 비어 있다면
                next_pos.append({(pos1_x, pos1_y), (pos1_x + i, pos1_y)})
                next_pos.append({(pos2_x, pos2_y), (pos2_x + i, pos2_y)})
    # 현재 로봇이 세로로 놓여 있는 경우
    elif pos1_y == pos2_y:
        for i in [-1, 1]: # 왼쪽으로 회전하거나, 오른쪽으로 회전
            if board[pos1_x][pos1_y + i] == 0 and board[pos2_x][pos2_y + i] == 0: # 왼
쪽 혹은 오른쪽 두 칸이 모두 비어 있다면
                next_pos.append({(pos1_x, pos1_y), (pos1_x, pos1_y + i)})
                next_pos.append({(pos2_x, pos2_y), (pos2_x, pos2_y + i)})
    # 현재 위치에서 이동할 수 있는 위치를 반환
    return next_pos

def solution(board):
    # 맵의 외곽에 벽을 두는 형태로 맵 변형
    n = len(board)
    new_board = [[1] * (n + 2) for _ in range(n + 2)]
    for i in range(n):
        for j in range(n):
            new_board[i + 1][j + 1] = board[i][j]
    # 너비 우선 탐색(BFS) 수행
    q = deque()
    visited = []
    pos = {(1, 1), (1, 2)} # 시작 위치 설정
    q.append((pos, 0)) # 큐에 삽입한 뒤에
    visited.append(pos) # 방문 처리
    # 큐가 빌 때까지 반복
    while q:
        pos, cost = q.popleft()
        # (n, n) 위치에 로봇이 도달했다면, 최단 거리이므로 반환
        if (n, n) in pos:
            return cost
        # 현재 위치에서 이동할 수 있는 위치 확인
        for next_pos in get_next_pos(pos, new_board):
            # 아직 방문하지 않은 위치라면 큐에 삽입하고 방문 처리
            if next_pos not in visited:
                q.append((next_pos, cost + 1))
                visited.append(next_pos)
    return 0
```

```
        # 큐의 원소가 2개 이상이라면 가능한 정렬 결과가 여러 개라는 의미
        if len(q) >= 2:
            certain = False
            break
        # 큐에서 원소 꺼내기
        now = q.popleft()
        result.append(now)
        # 해당 원소와 연결된 노드들의 진입차수에서 1 빼기
        for j in range(1, n + 1):
            if graph[now][j]:
                indegree[j] -= 1
                # 새롭게 진입차수가 0이 되는 노드를 큐에 삽입
                if indegree[j] == 0:
                    q.append(j)

    # 사이클이 발생하는 경우(일관성이 없는 경우)
    if cycle:
        print("IMPOSSIBLE")
    # 위상 정렬 결과가 여러 개인 경우
    elif not certain:
        print("?")
    # 위상 정렬을 수행한 결과 출력
    else:
        for i in result:
            print(i, end=' ')
        print()
```

A 46 아기 상어

아기 상어는 먹을 수 있는 물고기 중에서 가장 가까운 물고기를 먼저 먹는다고 했다. 가장 가까운 물고기는 최단 거리 알고리즘을 이용해서 찾을 수 있다. 현재 상어는 전체 N × N 공간에서 상, 하, 좌, 우 위치로 이동이 가능하므로, 5장 'DFS/BFS'에서 다룬 '미로 탈출' 문제와 유사하게 BFS를 이용하여 최단 거리를 찾으면 효과적이다.

따라서 매번 현재 위치에서 도달 가능한 다른 모든 위치까지의 최단 거리를 구한 뒤에, 먹을 물고기의 위치를 찾는 과정을 반복하도록 하자. 다만, 문제에서는 '자신의 크기보다 큰 물고기가 있는 칸은 지나갈 수 없다', '자신의 크기보다 작은 물고기만 먹을 수 있다'와 같은 실수하기 쉬운 세부 조건이

있으므로, 구현 과정에서 실수가 없도록 각별한 주의가 필요하다.

핵심 아이디어는 BFS를 이용하여 최단 거리를 구하는 것이고, 세부 조건을 잘 이해해 구현 실수만 피한다면 정답 판정을 받을 수 있다. 필자는 소스코드를 짧게 만드는 것보다는, 세부 기능을 함수 여러 개로 나누어서 가독성을 높이고자 하였다.

A46.py 답안 예시

```
from collections import deque
INF = 1e9 # 무한을 의미하는 값으로 10억을 설정

# 맵의 크기 N을 입력받기
n = int(input())

# 전체 모든 칸에 대한 정보 입력
array = []
for i in range(n):
    array.append(list(map(int, input().split())))

# 아기 상어의 현재 크기 변수와 현재 위치 변수
now_size = 2
now_x, now_y = 0, 0

# 아기 상어의 시작 위치를 찾은 뒤에 그 위치엔 아무것도 없다고 처리
for i in range(n):
    for j in range(n):
        if array[i][j] == 9:
            now_x, now_y = i, j
            array[now_x][now_y] = 0

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

# 모든 위치까지의 '최단 거리만' 계산하는 BFS 함수
def bfs():
    # 값이 -1이라면 도달할 수 없다는 의미(초기화)
    dist = [[-1] * n for _ in range(n)]
    # 시작 위치는 도달이 가능하다고 보며 거리는 0
    q = deque([(now_x, now_y)])
    dist[now_x][now_y] = 0
```

```
    while q:
        x, y = q.popleft()
        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            if 0 <= nx and nx < n and 0 <= ny and ny < n:
                # 자신의 크기보다 작거나 같은 경우에 지나갈 수 있음
                if dist[nx][ny] == -1 and array[nx][ny] <= now_size:
                    dist[nx][ny] = dist[x][y] + 1
                    q.append((nx, ny))
    # 모든 위치까지의 최단 거리 테이블 반환
    return dist

# 최단 거리 테이블이 주어졌을 때, 먹을 물고기를 찾는 함수
def find(dist):
    x, y = 0, 0
    min_dist = INF
    for i in range(n):
        for j in range(n):
            # 도달이 가능하면서 먹을 수 있는 물고기일 때
            if dist[i][j] != -1 and 1 <= array[i][j] and array[i][j] < now_size:
                # 가장 가까운 물고기 1마리만 선택
                if dist[i][j] < min_dist:
                    x, y = i, j
                    min_dist = dist[i][j]
    if min_dist == INF: # 먹을 수 있는 물고기가 없는 경우
        return None
    else:
        return x, y, min_dist # 먹을 물고기의 위치와 최단 거리

result = 0 # 최종 답안
ate = 0 # 현재 크기에서 먹은 양

while True:
    # 먹을 수 있는 물고기의 위치 찾기
    value = find(bfs())
    # 먹을 수 있는 물고기가 없는 경우, 현재까지 움직인 거리 출력
    if value == None:
        print(result)
        break
    else:
        # 현재 위치 갱신 및 이동 거리 변경
```

```
    now_x, now_y = value[0], value[1]
    result += value[2]
    # 먹은 위치에는 이제 아무것도 없도록 처리
    array[now_x][now_y] = 0
    ate += 1
    # 자신의 현재 크기 이상으로 먹은 경우, 크기 증가
    if ate >= now_size:
        now_size += 1
        ate = 0
```

A 47 청소년 상어

이 문제는 시뮬레이션과 완전 탐색을 함께 수행해야 하는 문제로, 소스코드가 길고 실수하기 쉬운 문제이다. 청소년 상어가 먹을 수 있는 물고기의 수가 여러 개인 경우가 존재하므로, 완전 탐색을 수행하여 청소년 상어가 물고기를 먹게 되는 모든 경우를 찾아야 한다. 또한 이 문제는 방향이 8가지로 정의된다는 점을 고려해서 코드를 작성해야 한다.

따라서 청소년 상어가 물고기를 먹는 모든 경우를 찾고, 그 경우 각각에 대하여 문제에서 요구하는 대로 시뮬레이션을 수행하면 된다. 필자는 완전 탐색을 위해 DFS를 사용했으며, 가독성을 위하여 모든 물고기가 움직이는 함수와 청소년 상어가 움직이는 함수를 분리하여 구현하였다.

A47.py 답안 예시

```
import copy

# 4 × 4 크기의 정사각형에 존재하는 각 물고기의 번호(없으면 -1)와 방향 값을 담는 테이블
array = [[None] * 4 for _ in range(4)]

for i in range(4):
    data = list(map(int, input().split()))
    # 매 줄마다 4마리의 물고기를 하나씩 확인하며
    for j in range(4):
        # 각 위치마다 [물고기의 번호, 방향]을 저장
        array[i][j] = [data[j * 2], data[j * 2 + 1] - 1]

# 8가지 방향에 대한 정의
dx = [-1, -1, 0, 1, 1, 1, 0, -1]
dy = [0, -1, -1, -1, 0, 1, 1, 1]
```

```
# 현재 위치에서 왼쪽으로 회전된 결과 반환
def turn_left(direction):
    return (direction + 1) % 8

result = 0 # 최종 결과

# 현재 배열에서 특정한 번호의 물고기 위치 찾기
def find_fish(array, index):
    for i in range(4):
        for j in range(4):
            if array[i][j][0] == index:
                return (i, j)
    return None

# 모든 물고기를 회전 및 이동시키는 함수
def move_all_fishes(array, now_x, now_y):
    # 1번부터 16번까지의 물고기를 차례대로 (낮은 번호부터) 확인
    for i in range(1, 17):
        # 해당 물고기의 위치 찾기
        position = find_fish(array, i)
        if position != None:
            x, y = position[0], position[1]
            direction = array[x][y][1]
            # 해당 물고기의 방향을 왼쪽으로 계속 회전시키며 이동이 가능한지 확인
            for j in range(8):
                nx = x + dx[direction]
                ny = y + dy[direction]
                # 해당 방향으로 이동이 가능하다면 이동시키기
                if 0 <= nx and nx < 4 and 0 <= ny and ny < 4:
                    if not (nx == now_x and ny == now_y):
                        array[x][y][1] = direction
                        array[x][y], array[nx][ny] = array[nx][ny], array[x][y]
                        break
                direction = turn_left(direction)

# 상어가 현재 위치에서 먹을 수 있는 모든 물고기의 위치 반환
def get_possible_positions(array, now_x, now_y):
    positions = []
    direction = array[now_x][now_y][1]
    # 현재의 방향으로 계속 이동시키기
    for i in range(4):
        now_x += dx[direction]
```

```
            now_y += dy[direction]
            # 범위를 벗어나지 않는지 확인하며
            if 0 <= now_x and now_x < 4 and 0 <= now_y and now_y < 4:
                # 물고기가 존재하는 경우
                if array[now_x][now_y][0] != -1:
                    positions.append((now_x, now_y))
        return positions

    # 모든 경우를 탐색하기 위한 DFS 함수
    def dfs(array, now_x, now_y, total):
        global result
        array = copy.deepcopy(array) # 리스트를 통째로 복사

        total += array[now_x][now_y][0] # 현재 위치의 물고기 먹기
        array[now_x][now_y][0] = -1 # 물고기를 먹었으므로 번호 값을 -1로 변환

        move_all_fishes(array, now_x, now_y) # 전체 물고기 이동시키기

        # 이제 다시 상어가 이동할 차례이므로, 이동 가능한 위치 찾기
        positions = get_possible_positions(array, now_x, now_y)
        # 이동할 수 있는 위치가 하나도 없다면 종료
        if len(positions) == 0:
            result = max(result, total) # 최댓값 저장
            return
        # 모든 이동할 수 있는 위치로 재귀적으로 수행
        for next_x, next_y in positions:
            dfs(array, next_x, next_y, total)

    # 청소년 상어의 시작 위치(0, 0)에서부터 재귀적으로 모든 경우 탐색
    dfs(array, 0, 0, 0)
    print(result)
```

A 48 어른 상어

이 문제는 매초마다 모든 상어를 이동시키며 요구하는 기능을 정해진 내용대로 처리하는 전형적인 시뮬레이션 문제이다. 다른 시뮬레이션 문제와의 차별점은 상어마다 방향 우선순위 정보가 주어진다는 점이다. 따라서 상어마다 서로 다른 방향으로 이동하기 때문에, 모든 방향 정보를 담을 리스트를 별도로 선언해야 한다. 답안 예시는 다음과 같으며, 시뮬레이션 유형이므로 문제에서 제시한

알고리즘을 실수 없이 그대로 구현한다면 정답 판정을 받을 수 있다.

A48.py 답안 예시

```
n, m, k = map(int, input().split())

# 모든 상어의 위치와 방향 정보를 포함하는 2차원 리스트
array = []
for i in range(n):
    array.append(list(map(int, input().split())))

# 모든 상어의 현재 방향 정보
directions = list(map(int, input().split()))

# 각 위치마다 [특정 냄새의 상어 번호, 특정 냄새의 남은 시간]을 저장하는 2차원 리스트
smell = [[[0, 0]] * n for _ in range(n)]

# 각 상어의 회전 방향 우선순위 정보
priorities = [[] for _ in range(m)]
for i in range(m):
    for j in range(4):
        priorities[i].append(list(map(int, input().split())))

# 특정 위치에서 이동 가능한 4가지 방향
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

# 모든 냄새 정보를 업데이트
def update_smell():
    # 각 위치를 하나씩 확인하며
    for i in range(n):
        for j in range(n):
            # 냄새가 존재하는 경우, 시간을 1만큼 감소시키기
            if smell[i][j][1] > 0:
                smell[i][j][1] -= 1
            # 상어가 존재하는 해당 위치의 냄새를 k로 설정
            if array[i][j] != 0:
                smell[i][j] = [array[i][j], k]

# 모든 상어를 이동시키는 함수
def move():
    # 이동 결과를 담기 위한 임시 결과 테이블 초기화
    new_array = [[0] * n for _ in range(n)]
```

```
    # 각 위치를 하나씩 확인하며
    for x in range(n):
        for y in range(n):
            # 상어가 존재하는 경우
            if array[x][y] != 0:
                direction = directions[array[x][y] - 1] # 현재 상어의 방향
                found = False
                # 일단 냄새가 존재하지 않는 곳이 있는지 확인
                for index in range(4):
                    nx = x + dx[priorities[array[x][y] - 1][direction - 1][index] - 1]
                    ny = y + dy[priorities[array[x][y] - 1][direction - 1][index] - 1]
                    if 0 <= nx and nx < n and 0 <= ny and ny < n:
                        if smell[nx][ny][1] == 0: # 냄새가 존재하지 않는 곳이면
                            # 해당 상어의 방향 이동시키기
                            directions[array[x][y] - 1] = priorities[array[x][y] - 1]
                                                          [direction - 1][index]
                            # (만약 이미 다른 상어가 있다면 번호가 낮은 상어가 들어가도록)
                            # 상어 이동시키기
                            if new_array[nx][ny] == 0:
                                new_array[nx][ny] = array[x][y]
                            else:
                                new_array[nx][ny] = min(new_array[nx][ny], array[x][y])
                            found = True
                            break
                if found:
                    continue
                # 주변에 모두 냄새가 남아 있다면, 자신의 냄새가 있는 곳으로 이동
                for index in range(4):
                    nx = x + dx[priorities[array[x][y] - 1][direction - 1][index] - 1]
                    ny = y + dy[priorities[array[x][y] - 1][direction - 1][index] - 1]
                    if 0 <= nx and nx < n and 0 <= ny and ny < n:
                        if smell[nx][ny][0] == array[x][y]: # 자신의 냄새가 있는 곳이면
                            # 해당 상어의 방향 이동시키기
                            directions[array[x][y] - 1] = priorities[array[x][y] - 1]
                                                          [direction - 1][index]
                            # 상어 이동시키기
                            new_array[nx][ny] = array[x][y]
                            break
    return new_array

time = 0
while True:
```

윗줄에 이어진 코드입니다.

윗줄에 이어진 코드입니다.

```
update_smell() # 모든 위치의 냄새를 업데이트
new_array = move() # 모든 상어를 이동시키기
array = new_array # 맵 업데이트
time += 1 # 시간 증가

# 1번 상어만 남았는지 체크
check = True
for i in range(n):
    for j in range(n):
        if array[i][j] > 1:
            check = False
if check:
    print(time)
    break

# 1,000초가 지날 때까지 끝나지 않았다면
if time >= 1000:
    print(-1)
    break
```