**인공지능시스템 HW 1학번 : 214594 / 이름 : 이채은 / 제출일자 : 23-04-11**

|  |  |  |
| --- | --- | --- |
| **[1] 신입생에게 전대마을 소개하기**   |  | | --- | | **[코드]**  # <문제 풀이>  # 1. 방문하여 1인 곳은 방문표시하기  # 2. 1번을 기준으로 상하좌우, 8칸에 1이 있나 확인하기  # 3. 1이 있다면, 해당 집으로 방문하면서 all\_house 1 증가시키기  # 4. 1~3 과정을 반복하여 최종 all\_house 반환 -> 집과 창고 분리하는데 사용  # 5. 전체 행렬에 1~4 과정을 반복하여 전대마을 집/창고 분리하기  # 전대마을(행렬) 입력받기  n\_row, m\_col = map(int, input('마을의 행과 열 개수를 공백 기준으로 분리하여 입력: ').split())  village = [] # 빈 2차원 리스트를 생성한다  # 2차원 리스트의 전대마을 정보를 입력받는다  for i in range(n\_row):  col = list(map(int, input(f'{i} 행 입력: ').split()))  village.append(col)    all\_house = 0 # 몇개의 집이 인접해있는지 알기 위해 all\_house 변수를 초기화한다  sum\_house = [] # 결과값을 담을 빈 리스트 생성한다  # 상,하,좌,우와 같이 주변 8칸의 위치 재귀적으로 호출하기 위해서 미리 리스트를 생성한다  da = [-1, 1, 0, 0, -1, 1, 1, -1]  db = [0, 0, -1, 1, 1, -1, 1, -1]  # DFS 알고리즘!  # x, y 위치인 노드를 방문하는데, 이 노드와 연결된 모든 노드들을 전부 방문한다  def dfs(a,b): # dfs함수를 정의한다.  global all\_house # 글로벌 전역변수 cnt를 선언한다.  village[a][b] = 0 # 노드를 방문 처리한다. (0으로 값을 바꾼다)  all\_house += 1 # cnt를 1 증가시킨다.  for i in range(0, 8, 1) : # 상,하,좌,우 위치를 구한다.  na = a + da[i]  nb = b + db[i]  if(na <= -1 or na >= n\_row or nb <= -1 or nb >= m\_col):  continue # 만약 주어진 범위를 벗어나면 continue를 반환한다  if(village[na][nb] == 1):  # 움직인 좌표에 집이 있다면 다시 dfs탐색  dfs(na,nb) # 상,하,좌,우 위치를 재귀적으로 호출한다  home = 0 # 집의 개수를 초기화한다.  changgo = 0 # 창고의 개수를 초기화한다.  # 집 개수를 찾아내기  for i in range(n\_row) :  for j in range(m\_col) :  # 반복문 수행하면서 전대마을 행렬 전부를 dfs함수에 적용시킨다.  if village[i][j] == 1:  all\_house = 0 # all\_house를 초기화한다  dfs(i,j) # dfs 함수를 호출한다.  sum\_house.append(all\_house) # sum\_house리스트에 몇개의 1이 인접해있는 집인지 저장한다.    for i in range(len(sum\_house)): # 창고+집의 총 개수만큼 반복한다.  if sum\_house[i] == 1: # 만약 한 개의 1로 구성된 곳이라면 집이아니라 창고이다.  changgo += 1 # 창고 +1    home = len(sum\_house) - changgo # 창고+집의 총 개수에서 창고를 뺀 수는 집의 갯개수이다.  # 마을에 있는 집 개수 출력  print(f'전대마을에는 {home}개의 집, {changgo}개의 창고가 있습니다.') | | **[실행 결과]** | |

|  |  |  |  |
| --- | --- | --- | --- |
| **[2] 인공지능학부 디저트 나눔 사건**   |  | | --- | | **[코드]**  from collections import deque  def is\_ai\_student(x, y):  # 확인하려는 학생 좌표값을 각각 절대값으로 취한 후, 각 자릿수를 분리하여 합해줍니다.  abs\_sum = sum(map(int, str(abs(x) + abs(y))))  # 인공지능학부 학생 여부를 판별하였습니다. 총합이 16 이하이거나 짝수라면 인공지능학부 학생이고 True를 반환합니다.  # 아닌 경우 타 학과 학생이므로 False를 반환합니다.  return abs\_sum <= 16 or abs\_sum % 2 == 0  def count\_ai\_students(center\_x, center\_y, radius):  visited = set() # 방문한 위치를 저장하기 위한 집합입니다.  queue = deque([(center\_x, center\_y)]) # BFS 탐색을 위한 deque입니다. 처음에 중심 좌표를 큐에 넣습니다.  ans = 0 # 인공지능학부 학생 수를 카운트할 변수입니다.    while queue:  x, y = queue.popleft() # BFS 탐색을 위해 큐의 왼쪽에서 부터 좌표를 하나씩 가져옵니다.  # 이미 방문한 위치거나 반지름 radius 인 원 내부가 아니라면 continue  if (x, y) in visited or (x-center\_x)\*\*2 + (y-center\_y)\*\*2 > radius\*\*2:  continue  visited.add((x, y)) # 방문 처리합니다.    if is\_ai\_student(x, y): # 만약 인공지능학부 학생이라면 디저트를 전달하고, 다시 네 방향을 탐색하여 인공지능학부 학생인지 확인합니다.  ans += 1 # 인공지능학부 학생 수를 1 증가시킵니다.  for dx in range(-radius, radius+1): # x축 방향 반지름 내부 좌표를 모두 확인합니다.  for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]: # x축 방향 반지름 내부 좌표를 모두 확인합니다.  nx, ny = x + dx, y + dy  if (nx, ny) not in visited and abs(nx - center\_x) \*\* 2 + abs(ny - center\_y) \*\* 2 <= radius \*\* 2:  queue.append((nx, ny))    return ans  # 원의 중심좌표와 반지름 입력  cx, cy = map(int, input('중심 좌표 입력: ').split())  radius = int(input('원의 반지름 입력: '))  answer = count\_ai\_students(cx, cy, radius)  print(f'디저트를 받은 인공지능학부 학생은 총 {answer-81}명입니다.') | | **[실행 결과]** | |  | |

|  |  |  |
| --- | --- | --- |
| **[3] 감시 시스템을 피해 탈출하기**   |  | | --- | | **[코드]**  # 문제의 목적 : (0,0)의 출발지에 존재하는 로봇을 (N-1,M-1)의 도착지로 옮기는 최단거리를 계산하는 것이다.  # -> 전형적인 BFS 문제 유형이다.  # 1. 우주선이 차지하고 있는 위치가 두칸이다.  # -> 방문 여부를 관리하기 위해서는 우주선의 위치 정보를 튜플로 처리하면 된다. -> 집합 자료형 set을 사용한다.  # 2. 우주선이 이동하는 경우.  # -> 상,하,좌,우로 이동하는 경우이다.  # 3. 우주선이 회전하는 경우.  # -> 먼저 첫번째, 우주선이 가로로 놓인 상태에서 아래쪽으로 회전하는 경우가 있다.(아래쪽에 감시 시스템이 존재하지 않아야한다.)  # -> 두번째, 우주선이 가로로 놓인 상태에서 위쪽으로 회전하는 경우가 있다.(위쪽에 감시 시스템이 존재하지 않아야한다.)  # -> 세번째, 우주선이 세로로 놓인 상태에서 오른쪽으로 회전하는 경우가 있다.(오른쪽에 감시 시스템이 존재하지 않아야한다.)  # -> 네번째, 우주선이 세로로 놓인 상태에서 왼쪽으로 회전하는 경우가 있다.(왼쪽에 감시 시스템이 존재하지 않아야한다.)  # 4. 초기에 주어진 맵을 변형한다.  # -> 주변 외곽을 감시시스템으로 전부 둘러싸게 한다.  # 데큐 라이브러리 사용  from collections import deque  # 특정한 위치에서 이동 가능한 다음 위치를 반환하는 함수이다.  def possible\_position(position\_, board):    # 상하좌우로 이동하는 경우 4가지 경우를 미리 지정합니다.  dx = [-1, 1, 0, 0]  dy = [0, 0, -1, 1]    position\_ = list(position\_) # position을 집합에서 리스트로 변환시킵니다.  result\_position\_ = [] # 반환 결과를 저장할 리스트를 생성한다. (이 위치들은 이동이 가능하다)  # 00, 01, 10, 11인덱스의 값을 각각 4개의 값에 저장합니다.  p1\_a, p1\_b, p2\_a, p2\_b = position\_[0][0], position\_[0][1], position\_[1][0], position\_[1][1]  # 현재 우주선이 위, 아래, 왼쪽, 아래쪽으로 이동할 수 있는지 확인합니다.  for i in range(4):  pp1\_a, pp1\_b, pp2\_a, pp2\_b = p1\_a + dx[i], p1\_b + dy[i], p2\_a + dx[i], p2\_b + dy[i]  # 이동하려는 두 칸에 감시 시스템이 존재하는지 확인합니다.  if board[pp1\_a][pp1\_b] == 0 and board[pp2\_a][pp2\_b] == 0:  result\_position\_.append({(pp1\_a, pp1\_b), (pp2\_a, pp2\_b)})    if p1\_b == p2\_b: # 현재 우주선이 세로일 때, 오른쪽/왼쪽으로 회전할 수 있는지 확인합니다.  for i in [-1, 1]: # 왼쪽으로 회전하거나, 오른쪽으로 회전시킵니다.  if board[p1\_a][p1\_b + i] == 0 and board[p2\_a][p2\_b + i] == 0: # 왼쪽 or 오른쪽 두 칸이 모두 비어 있을 때,  # 위 칸을 통해 왼쪽/오른쪽으로 회전합니다.  result\_position\_.append({(p1\_a, p1\_b), (p1\_a, p1\_b + i)})  # 아래 칸을 통해 왼쪽/오른쪽으로 회전합니다.  result\_position\_.append({(p2\_a, p2\_b), (p2\_a, p2\_b + i)})  elif p1\_a == p2\_a: # 현재 우주선이 가로일 때, 위/아래로 회전할 수 있는지 확인합니다.  for i in [-1, 1]: # 위쪽으로 회전하거나, 아래쪽으로 회전시킵니다.  if board[p1\_a + i][p1\_b] == 0 and board[p2\_a + i][p2\_b] == 0: # 위쪽 or 아래쪽 두 칸이 모두 비어 있을 때,  # 왼쪽 칸을 통해 위/아래로 회전합니다.  result\_position\_.append({(p1\_a, p1\_b), (p1\_a + i, p1\_b)})  # 오른쪽 칸을 통해 위/아래로 회전합니다.  result\_position\_.append({(p2\_a, p2\_b), (p2\_a + i, p2\_b)})  # 상하좌우, 회전으로 이동할 수 있는 모든 위치를 반환합니다.  return result\_position\_  # 이 함수는 특정한 출발지에 존재하는 우주선을 도착지로 옮기는 최단시간을 계산하여 반환합니다.  def min\_time(n, m, board): # n행, m열, 지도보드를 인자로 갖는 함수입니다.  # 먼저 지도보드를 1로 둘러쌉니다.  new\_board = [[1] \* (n + 2) for \_ in range(m + 2)]  for i in range(n): # n행  for j in range(m): # m열  new\_board[i + 1][j + 1] = board[i][j] # 지도보드를 1로 둘러싼 새로운 지도보드를 만듭니다.  # BFS 알고리즘 !  queue = deque() # queue 구현을 위해 deque라이브러리를 사용합니다.  visited = [] # 방문처리를 위해 빈 리스트를 생성합니다.  position = {(1, 1), (1, 2)} # 시작 위치입니다. 지도보드의 테두리를 1로 둘러 쌌으므로, 0,0 / 0,1 이 아니라 1씩 더한 1,1 / 1,2 위치입니다.  queue.append((position, 0)) # 큐에 시작 위치를 추가합니다.  visited.append(position) # 시작 위치를 방문 처리합니다.    while queue: # 이 while문은 큐가 빌 때까지 반복합니다.  position, time = queue.popleft() # 현재 위치와 현재 시간을 꺼냅니다.  if (n, m) in position: # (n, m)에 우주선이 도착하면,  return time # 최단 시간을 return합니다.    for result\_position in possible\_position(position, new\_board): # 이동할 수 있는 위치를 반환하는 함수를 for문 안에 호출합니다.  if result\_position not in visited: # 만약 방문하지 않은 위치라면,  queue.append((result\_position, time + 1)) # 큐에 위치와 1초를 더한 시간을 추가합니다.  visited.append(result\_position) # 위치를 방문 처리합니다.  return 0  # 감시 공간 크기를 입력받습니다.  n, m = map(int, input('감시 공간 크기 입력: ').split())  board\_ = [] # 빈 리스트를 생성합니다.  for s in range(n): # n행만큼 반복합니다.  tmp = list(map(int, input(f'{s}행 입력: ').split())) # 행을 입력받습니다.  board\_.append(tmp) # 보드에 행을 추가하여 2차원 리스트를 생성합니다.  answer = min\_time(n, m, board\_) # 최단 시간을 반환하는 함수를 호출합니다.  print(f'감시 공간을 탈출할 수 있는 최단 시간은 {answer}초입니다.') # 정답을 출력합니다. | | **[실행 결과]** | |

|  |  |  |
| --- | --- | --- |
| **[4] 틱택토: 절대 지지 않는 (항상 비기거나 승리하는) 틱택토 인공지능 만들기**   |  | | --- | | **[코드]**  def board\_visualization(board):  ### 틱택토 게임 출력 함수  for i\_idx in range(3):  for j\_idx in range(3):  if board[i\_idx][j\_idx] == 0: # not occupied  print("▨", end='')  elif board[i\_idx][j\_idx] == 1: # user  print("○", end='')  elif board[i\_idx][j\_idx] == 2: # computer  print("×", end='')  else:  pass  print()    def board\_turn(board, row, col, val):  if 0 <= row < 3 and 0 <= col < 3:  if board[row][col] == 0:  board[row][col] = val  return 1  else:  print("그곳에 돌을 둘 수 없습니다. 다시 입력하세요.")  return 0  else:  print("그곳에 돌을 둘 수 없습니다. 다시 입력하세요.")  return 0      def win\_condition(v1,v2,v3): # 사용자 승리 return 100, 컴퓨터 승리 return -100, 무승부 return 0  if v1 == v2 == v3:  if v1 == 1: return 100  elif v2 == 2: return -100  else: return 0  else:  return 0      def game\_result(board): # 사용자 승리 return 100, 컴퓨터 승리 return -100, 무승부 return 0  result = 0  # check rows  for i\_idx in range(3):  result = win\_condition(board[i\_idx][0],board[i\_idx][1],board[i\_idx][2])  if result != 0: return result    # check cols  for i\_idx in range(3):  result = win\_condition(board[0][i\_idx],board[1][i\_idx],board[2][i\_idx])  if result != 0: return result    #check diagonals  result = win\_condition(board[0][0],board[1][1],board[2][2])  if result != 0: return result  result = win\_condition(board[0][2],board[1][1],board[2][0])  if result != 0: return result    return result  def position():  row = int(input("돌을 놓을 행을 입력하세요 (0~2):"))  col = int(input("돌을 놓을 열을 입력하세요 (0~2):"))    return row, col  # -----------------------------------------------------------------------------  # 미니맥스 알고리즘이란?  # 3번으로 제한을 둔 예시를 계산해보겠습니다.  # 마지막 최종 score가 정해질 때에는 한 가지 밖에 없기 때문에 점수를 그대로 올려줍니다.  # 다음은 플레이어의 턴이기 때문에 '미니'전략으로 가장 점수가 낮은 경우를 올려줍니다.  # 다음은 컴퓨터 턴이기 때문에, 컴퓨터에게 가장 유리한 '맥스'점수를 올려줍니다.  # 결국 컴퓨터는 최적의 수를 알게 됩니다.(비기거나 이길 수 있는 수를 찾을 수 있습니다.)  # 이렇게 3번을 계산하는 것은 간단하지만, 처음부터라면 9! 약 36만번의 계산량을 가지는 것을 알 수 있습니다.  # 따라서 깊이를 정해서, 일정 수준까지의 깊이만 경우의 수를 확인하는 방법이 필요합니다.  # 하지만, 9번이므로 그냥 구현해보았습니다.  # 이 함수는 게임판의 한 줄을 인자로 받아 그 줄에서 컴퓨터가 선택할 최선의 수를 계산합니다.  # empty는 해당 줄에 돌이 하나도 없는지, user는 해당 줄에 유저의 돌이 몇 개 있는지,  # com은 해당 줄에 컴퓨터의 돌이 몇 개 있는지를 나타냅니다.  # 현재 컴퓨터가 유리한 상태인지, 불리한 상태인지를 확인하고 중첩 if문으로 들어가 임의로 가치를 매깁니다.  def max\_value(none\_,user,computer,min\_max\_):  if(min\_max\_=='min'): # 플레이어 턴에는 컴퓨터에게 가장 불리한 방법인 미니 (유저에게는 유리합니다.)  if none\_: return 1 # 라인이 전부 비어있다면? + 임의로 1의 가치를 매겼습니다.  if computer: return 0 # 라인에 x가 하나라도 있다면?(User 돌 1개 존재) + 임의로 0의 가치를 매겼습니다.  if user == 1: return 3 # 라인에 o가 1개 있다면?(Computer 돌 2개 존재) + 임의로 3 의 가치를 매겼습니다.  if user == 2: return 7 # 라인에 o가 2개 있다면?(Computer 돌 2개 존재) + 임의로 7의 가치를 매겼습니다.    if(min\_max\_ =='max'): # 컴퓨터 턴에는 컴퓨터에게 가장 유리한 맥스  if none\_: return 1 # 라인이 전부 비어있다면? + 임의로 1의 가치를 매겼습니다.  if user: return 0 # 라인에 o가 하나라도 있다면?(User 돌 1개 존재) + 임의로 0의 가치를 매겼습니다.  if computer == 1: return 3 # 라인에 x가 1개 있다면?(Computer 돌 2개 존재) + 임의로 3의 가치를 매겼습니다.  if computer == 2: return 7 # 라인에 x가 2개 있다면?(Computer 돌 2개 존재) + 임의로 7의 가치를 매겼습니다.    # 보드 위치에서, 더해줘야할 줄의 개수를 저장한 리스트입니다.  pos\_list = {'00':3,'01':2,'02':3,'10':2,'11':4,'12':2,'20':3,'21':2,'22':3}  # minimax함수에서는 사용자가 둔 수 이후에 대한 탐색과 컴퓨터가 둔 수 이후에 대한 탐색을 재귀적’으로 수행됩니다.  # 이 때, 사용자는 값을 최대화 하는쪽으로 / 컴퓨터는 값을 최소화 하는 쪽으로 탐색합니다.  # 재귀적으로 탐색을 수행하다보면 승패가 결정된 값이 반환됩니다.  def minimax(board,i\_idx,j\_idx):  min\_ = 0  max\_ = 0  sum\_list = pos\_list[str(i\_idx) + str(j\_idx)] #행렬에 따라 사전에서 맞는 줄의 개수를 불러와서 조건에 따라 더함    # 저는 처음으로 돌을 뒀을 때, 되는 경우의 수를 전부 나누었습니다.  # 따라서 아래 코드부터는 반복되는 부분이 존재합니다.  # 먼저 가로 라인, 세로 라인, 대각 라인입니다.  if sum\_list==3:  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for a in range(3): # 가로 라인  if board[a][j\_idx]==1:  none\_=False  user+=1  elif board[a][j\_idx]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.    none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for b in range(3): # 세로 라인  if board[i\_idx][b]==1:  none\_=False  user+=1  elif board[i\_idx][b]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  if j\_idx==i\_idx: # 오른쪽 대각선 라인  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for d in range(3): # 대각 라인  if board[d][d]==1:  none\_=False  user+=1  elif board[d][d]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  else: # 왼쪽 대각선 라인  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for d in range(3):  if board[2-d][d]==1:  none\_=False  user+=1  elif board[2-d][d]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  elif sum\_list==2:  # 가로 라인, 세로 라인입니다.  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for b in range(3): # 가로 라인  if board[i\_idx][b]==1:  none\_=False  user+=1    elif board[i\_idx][b]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for a in range(3): # 세로 라인  if board[a][j\_idx]==1:  none\_=False  user+=1  elif board[a][j\_idx]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.    else:  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for d in range(3): # 오른쪽 대각선 라인  if board[d][d]==1:  none\_=False  user+=1  elif board[d][d]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for d in range(3): # 왼쪽 대각선 라인  if board[2-d][d]==1:  none\_=False  user+=1  elif board[2-d][d]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.    none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for b in range(3): # 가로 라인  if board[i\_idx][b]==1:  none\_=False  user+=1  elif board[i\_idx][b]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.  none\_=True # 빈 곳 초기화  computer=0 # computer 초기화  user=0 # user 초기화  for a in range(3): # 세로 라인  if board[a][j\_idx]==1:  none\_=False  user+=1  elif board[a][j\_idx]==2:  none\_=False  computer+=1  min\_+=max\_value(none\_,user,computer,'min') #min 값을 min\_에 넣습니다.  max\_+=max\_value(none\_,user,computer,'max') #max 값을 max\_에 넣습니다.    minimax\_val=max(min\_,max\_) # 최적의값을 찾아 minimax\_val에 넣는다.  return (i\_idx,j\_idx,minimax\_val) # i\_idx,j\_idx,minimax\_val 값을 반환한다.  score=[]  # row, col = computer(board) # computer 함수를 다시 코딩 하시오  def computer(board):  ### 현재 computer()함수는 빈곳에 랜덤하게 돌을 두도록 프로그래밍 되어있음  ### 무조건 user를 상대로 [비기거나/이기도록] 미니맥스 알고리즘을 이용하여 다시 프로그래밍 하시오  ### computer() 함수와 minimax() 함수만 새로 작성하여 문제를 해결하시오  row = 0  col = 0    # 둘 수 있는 곳이 몇 곳인지 확인한다.  for i\_idx in range(3):  for j\_idx in range(3):  if board[i\_idx][j\_idx] == 0:  score.append(minimax(board, i\_idx, j\_idx)) # 미니맥스 알고리즘으로 얻은 값을 score 리스트에 추가합니다.    score.sort(reverse=True, key=lambda x: x[2]) # 내림차순을 실행합니다.  row,col,\_=score.pop(0) # 맨 앞의 것을 pop()함수로 불러옵니다.  return row, col # 미니맥스 알고리즘을 통해 얻은 행, 열 위치를 반환합니다.  # -----------------------------------------------------------------------------  ### 게임 시작  print("게임 시작")  board = [[0,0,0],[0,0,0],[0,0,0]]  board\_visualization(board)  for i\_idx in range(5): # 플레이어는 최대 5번 돌을 둔다  ###### User ######  print("============= Your turn =============")  row, col = position()  while board\_turn(board, row, col,1) == 0:  row, col = position()    board\_visualization(board)  if game\_result(board) == 100:  print("사용자가 승리하였습니다.")  break  elif game\_result(board) == -100:  print("컴퓨터가 승리하였습니다.")  break  ########################        ###### Computer ######  print("============= Computer =============")  row, col = computer(board) # computer 함수를 다시 코딩 하시오  board\_turn(board, row, col, 2)  board\_visualization(board)  if game\_result(board) == 100:  print("사용자가 승리하였습니다.")  break  elif game\_result(board) == -100:  print("컴퓨터가 승리하였습니다.")  break  ########################    if game\_result(board) == 0:  print("비겼습니다.") | | **[실행 결과]** | |

|  |  |  |
| --- | --- | --- |
| **[5] 구름게임 : 컴퓨터는 항상 최선을 다한다.**   |  | | --- | | **[코드]**  import math  class Game():  def \_\_init\_\_(self, size, board, player, computer):  self.size = size # 구름게임 맵 크기 (정방행렬)  self.board = board # 구름게임 맵  self.player = player # 플레이어 위치  self.computer = computer # 컴퓨터 위치  # 구름게임 출력 함수  def board\_visualization(self):  for idx in range(self.size):  for jdx in range(self.size):  if self.board[idx][jdx] == 0: # 낭떠러지  print("□ ", end='')  elif self.board[idx][jdx] == 1: # 구름  print('▨ ', end='')  elif self.board[idx][jdx] == 2: # 플레이어  print('◆ ', end='')  elif self.board[idx][jdx] == 3: # 컴퓨터  print('● ', end='')  else:  pass  print()  # 플레이어 이동 위치 함수  def position(self):  player = list(map(int, input(f'플레이어가 이동할 위치 입력: ').split()))  # 맵을 벗어나지 않도록 설정  if player[0] >= self.size or player[1] >= self.size:  self.position()    # 이동할 위치가 구름이고 상하좌우 방향이라면 플레이어 위치로 변환 / 기존 위치는 낭떠러지로 변환  if self.board[player[0]][player[1]] == 1 and (abs(self.player[0]-player[0]) + abs(self.player[1]-player[1])) == 1:  self.board[self.player[0]][self.player[1]] = 0  self.player = player  self.board[player[0]][player[1]] = 2  else:  self.position()  # 게임이 끝났는지 확인하는 함수  def win\_condition(self, board):  # 리턴값 0: 컴퓨터 승리 / 1: 플레이어 승리 / 2: 승패 결정 X    for player in ['X', 'O']:  # 가로줄 검사  for i in range(size):  if board[i][0] == player and board[i][1] == player and board[i][2] == player:  if player == self.computer:  return 0  else:  return 1  # 세로줄 검사  for i in range(size):  if board[0][i] == player and board[1][i] == player and board[2][i] == player:  if player == self.computer:  return 0  else:  return 1  # 대각선 검사  if board[0][0] == player and board[1][1] == player and board[2][2] == player:  if player == self.computer:  return 0  else:  return 1  if board[0][2] == player and board[1][1] == player and board[2][0] == player:  if player == self.computer:  return 0  else:  return 1  # 게임 결과에 따른 minimax 알고리즘 점수를 리턴하는 함수  # 게임판(board)의 상태를 받아와서 게임 결과(result)에 따른 minimax 알고리즘에서의 점수를 계산하여 리턴하는 함수입니다.  def evaluate(self, board):  result = self.win\_condition(board) # 게임 결과를 받아옴  if result == 0: # 컴퓨터 승리  return 10  elif result == 1: # 플레이어 승리  return -10  elif result == 2: # 승패 결정 X  return 0  else: # 무승부  return 0    # 타겟이 이동할 수 있는 위치를 찾는 함수  # ismove 함수는 반환값으로 비어있는 모든 위치의 좌표를 담은 리스트를 반환합니다.  def ismove(self, board):  moves = []  for i in range(size):  for j in range(size):  if board[i][j] == '':  moves.append((i, j))  return moves  # MiniMax 알고리즘 수행 함수  # 현재 차례가 컴퓨터인지, 플레이어인지를 구분하여 최적의 수를 찾아야합니다.  # 현재 상태에서 가장 높은 점수를 얻을 수 있는 수를 찾는 경우. 컴퓨터가 수를 선택하고, 반대의 상황에서는 상대방이  # 가장 낮은 점수를 얻을 수 있는 수를 선택해야 하므로 플레이어가 수를 선택합니다.  # 최적의 수를 찾을 때마다 이에 따른 점수와 해당 수를 함께 저장하여 리턴합니다.    def minimax(self, board, depth, is\_maximizing):  # 리턴값 : best\_score, best\_move (플레이어가 갈 수 있는 최적의 위치와 이에 따른 점수 리턴)  # 게임이 끝났으면 현재 보드판 상태를 평가하고 점수를 리턴  result = self.win\_condition(board)  if result != 2:  return self.evaluate(board), None  if is\_maximizing:  # 컴퓨터 차례인 경우, 최고의 점수를 찾기 위해 가능한 모든 수를 시도  best\_score = -math.inf  best\_move = None  for move in self.ismove(board):  board[move[0]][move[1]] = self.computer  score, \_ = self.minimax(board, depth+1, False)  board[move[0]][move[1]] = ''  if score > best\_score:  best\_score = score  best\_move = move  return best\_score, best\_move  else:  # 플레이어 차례인 경우, 상대방이 최소한의 점수를 얻을 수 있도록 최적의 수를 선택  best\_score = math.inf  best\_move = None  for move in self.ismove(board):  board[move[0]][move[1]] = self.player  score, \_ = self.minimax(board, depth+1, True)  board[move[0]][move[1]] = ''  if score < best\_score:  best\_score = score  best\_move = move  return best\_score, best\_move  # 컴퓨터가 이동할 최적의 위치를 받는 함수  def best\_pos(self):  # 컴퓨터가 이동할 최적의 위치  position = self.minimax(self.board)[1]  # 위치를 받아와서 구름게임 맵과 컴퓨터 위치 변경  self.board[self.computer[0]][self.computer[1]] = 0  self.computer = [position[0], position[1]]  self.board[position[0]][position[1]] = 3  # 구름게임 맵 입력  board = []  size = int(input('구름게임 맵 크기(정방행렬): '))  print('낭떠러지 : 0 구름 : 1')  for s in range(size):  tmp = list(map(int, input(f'{s}행 입력: ').split()))  while True: # 플레이어 시작 위치 입력  player = list(map(int, input(f'플레이어 시작 위치 입력: ').split()))  if player[0] >= 0 and player[0] < size and player[1] >= 0 and player[1] < size:  if board[player[0]][player[1]] == 1:  break  board[player[0]][player[1]] = 2  while True: # 컴퓨터 시작 위치 입력  computer = list(map(int, input(f'컴퓨터 시작 위치 입력: ').split()))  if computer[0] >= 0 and computer[0] < size and computer[1] >= 0 and computer[1] < size:  if board[computer[0]][computer[1]] == 1:  break  board[computer[0]][computer[1]] = 3  # 구름게임 세팅  game = Game(size, board, player, computer)  print("============= Initial State =============")  game.board\_visualization()  # 구름게임 시작  while True:  print("=============== Computer ================")  result = game.win\_condition(game.board) # 컴퓨터가 진행하기 전 게임 상황 확인  if result == 0: # 컴퓨터가 이긴 경우 게임 종료  game.board\_visualization()  print('컴퓨터가 승리하였습니다.')  break  elif result == 1: # 플레이어가 이긴 경우 게임 종료  game.board\_visualization()  print('플레이어가 승리하였습니다.')  break  game.best\_pos()  game.board\_visualization()  print("=============== Your turn ===============")  result = game.win\_condition(game.board) # 플레이어가 진행하기 전 게임 상황 확인  if result == 0:  game.board\_visualization()  print('컴퓨터가 승리하였습니다.')  break  elif result == 1:  game.board\_visualization()  print('플레이어가 승리하였습니다.')  break  game.position()  game.board\_visualization() | | **[실행 결과]** | |