import heapq

class Node:

    def \_\_init\_\_(self, position, g\_cost, h\_cost, parent=None):

        self.position = position

        self.g\_cost = g\_cost  # start node cost

        self.h\_cost = h\_cost  # Heuristic cost

        self.f\_cost = g\_cost + h\_cost  # total cost

        self.parent = parent  # Parent node for path reconstruction

    def \_\_lt\_\_(self, other):

        return self.f\_cost < other.f\_cost

maze = [

    [0, 0, 1, 1, 0, 0],

    [1, 0, 1, 0, 1, 0],

    [1, 0, 1, 0, 1, 0],

    [0, 0, 0, 0, 1, 0],

    [0, 1, 1, 1, 0, 0],

    [0, 0, 0, 0, 0, 0]

]

start\_player1 = (0, 0)

start\_player2 = (5, 5)

goal = (5, 0)

def heuristic(position, goal): # also called Manhattan distance

    return abs(position[0] - goal[0]) + abs(position[1] - goal[1])

def get\_neighbors(position, maze):

    x, y = position

    neighbors = []

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Movement

    for dx, dy in directions:

        new\_x, new\_y = x + dx, y + dy

        if 0 <= new\_x < len(maze) and 0 <= new\_y < len(maze[0]) and maze[new\_x][new\_y] == 0:

            neighbors.append((new\_x, new\_y))

    return neighbors

def reconstruct\_path(node):

    path = []

    while node:

        path.append(node.position)

        node = node.parent

    return path[::-1]  # reversepath as path is stored from goal to strt pos, so it will reverse it to be from start to goal posi

def display\_maze(player1\_pos, player2\_pos):

    for x in range(len(maze)):

        for y in range(len(maze[0])):

            if (x, y) == player1\_pos:

                print("P1", end=" ")

            elif (x, y) == player2\_pos:

                print("P2", end=" ")

            elif maze[x][y] == 1:

                print("X", end=" ") #x means border,no move

            else:

                print("O", end=" ") #moveable neighbour

        print()

def a\_star(maze, start, goal):

    priority\_queue = []

    visited = set() #for unique data

    start\_node = Node(start, 0, heuristic(start, goal))

    heapq.heappush(priority\_queue, start\_node)

    while priority\_queue:#not empty

        current\_node = heapq.heappop(priority\_queue)

        if current\_node.position == goal:

            return reconstruct\_path(current\_node)

        visited.add(current\_node.position)

        for neighbor in get\_neighbors(current\_node.position, maze):

            if neighbor in visited:

                continue

            g\_cost = current\_node.g\_cost + 1  # movment cost=1

            h\_cost = heuristic(neighbor, goal)

            neighbor\_node = Node(neighbor, g\_cost, h\_cost, current\_node)

            #  neighbor is already in p.q with a lower f\_cost

            if any(open\_node.position == neighbor and open\_node.f\_cost <= neighbor\_node.f\_cost for open\_node in priority\_queue):

                continue

            heapq.heappush(priority\_queue, neighbor\_node)

    return None  # nosolutn

def game():

    player1\_pos = start\_player1

    player2\_pos = start\_player2

    while True:

        display\_maze(player1\_pos, player2\_pos) # display maze with currentpos

        if player1\_pos == goal:

            print("Player 1 wins!")

            break

        elif player2\_pos == goal:

            print("Player 2 wins!")

            break

        move = input("Your move (w=up, a=left, s=down, d=right): ")

        next\_pos = None

        if move == 'w':

            next\_pos = (player1\_pos[0] - 1, player1\_pos[1])

        elif move == 's':

            next\_pos = (player1\_pos[0] + 1, player1\_pos[1])

        elif move == 'a':

            next\_pos = (player1\_pos[0], player1\_pos[1] - 1)

        elif move == 'd':

            next\_pos = (player1\_pos[0], player1\_pos[1] + 1)

        else:

            print("Invalid input.")

            continue

        # Valid move

        if next\_pos and 0 <= next\_pos[0] < len(maze) and 0 <= next\_pos[1] < len(maze[0]) and maze[next\_pos[0]][next\_pos[1]] == 0:

            player1\_pos = next\_pos

        else:

            print("Invalid move. Try again.")

            continue

        pc\_path = a\_star(maze, player2\_pos, goal)

        if pc\_path and len(pc\_path) > 1:

            player2\_pos = pc\_path[1]

        else:

            print("Invalid")

            break

game()