```
import heapq
import math

# Directions: 8 possible moves
dirs = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]

# Heuristic function: Euclidean distance
def heuristic(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

# ------------------ Best First Search ------------------
def best_first_search(grid):
    n = len(grid)
    if grid[0][0] == 1 or grid[n-1][n-1] == 1:
        return -1, []

    start, goal = (0,0), (n-1,n-1)
    pq = [(heuristic(start, goal), start, [start])]
    visited = set([start])

    while pq:
        _, (x,y), path = heapq.heappop(pq)
        if (x,y) == goal:
            return len(path), path
        for dx,dy in dirs:
            nx, ny = x+dx, y+dy
            if 0<=nx<n and 0<=ny<n and grid[nx][ny]==0 and (nx,ny) not in visited:
                visited.add((nx,ny))
                heapq.heappush(pq, (heuristic((nx,ny), goal), (nx,ny), path+[(nx,ny)]))
    return -1, []

# ------------------ A* Search ------------------
def a_star_search(grid):
    n = len(grid)
    if grid[0][0] == 1 or grid[n-1][n-1] == 1:
        return -1, []

    start, goal = (0,0), (n-1,n-1)
    pq = [(heuristic(start, goal), 0, start, [start])]
    g_score = {start: 0}

    while pq:
        _, cost, (x,y), path = heapq.heappop(pq)
        if (x,y) == goal:
            return len(path), path
        for dx,dy in dirs:
            nx, ny = x+dx, y+dy
            if 0<=nx<n and 0<=ny<n and grid[nx][ny]==0:
                new_cost = cost + 1
                if (nx,ny) not in g_score or new_cost < g_score[(nx,ny)]:
                    g_score[(nx,ny)] = new_cost
                    f = new_cost + heuristic((nx,ny), goal)
```

```
                heapq.heappush(pq, (f, new_cost, (nx,ny), path+[(nx,ny)]))
    return -1, []

# ------------------ Test Cases ------------------
grids = [
    [[0,1],[1,0]],
    [[0,0,0],[1,1,0],[1,1,0]],
    [[1,0,0],[1,1,0],[1,1,0]]
]

for i,grid in enumerate(grids,1):
    bfs_len, bfs_path = best_first_search(grid)
    a_len, a_path = a_star_search(grid)
    print(f"Example {i}:")
    print(f"Best First Search → Path length: {bfs_len}, Path: {bfs_path}")
    print(f"A* Search         → Path length: {a_len}, Path: {a_path}\n")
```

```
user@rishi:~/Downloads$ python3 best.py
Example 1:
Best First Search →Path length: 2, Path: [(0, 0), (1, 1)]
A* Search         →Path length: 2, Path: [(0, 0), (1, 1)]

Example 2:
Best First Search →Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]
A* Search         →Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]

Example 3:
Best First Search →Path length: -1, Path: []
A* Search         →Path length: -1, Path: []
```

Comparison between Best First Search and A*

Best First Search (Greedy Search) relies only on the heuristic (distance to goal) to guide its search. This makes it faster and memory-efficient in many cases, since it always expands the node that seems closest to the destination. However, because it ignores the actual cost already traveled, it may fail to find the optimal path and can even get trapped exploring less promising routes.

On the other hand, A* Search balances both the cost so far (g) and the heuristic estimate (h). This ensures that it always finds the shortest path when one exists, making it more reliable and accurate than Best First Search. The trade-off is that A* often explores more nodes and requires more computation and memory compared to Greedy Best First Search. In practice, Best First can be faster but less accurate, while A* guarantees optimality with higher performance cost.