Project 3 – Jumping Jim **Name: Diya Jain U81673988**

Problem Description and Solution Approach

A. Workflow of the Problem

To begin with, we'll read the maze information from the input file, which includes the maze's dimensions (number of rows and columns) and the trampoline values for each cell in the maze. Each trampoline value indicates the number of spaces Jim will move when landing on that trampoline.

Next, we'll model the maze as a graph. In this graph representation, each cell in the maze corresponds to a vertex, and the edges between vertices represent movement through trampolines. We'll use an adjacency list to store this graph, where each vertex has a list of adjacent vertices it can move to based on the trampoline values.

After modeling the maze as a graph, we'll build the graph using a function specifically designed for this purpose. This 'buildGraph' function will iterate through each cell in the maze, calculate the index of the cell in the graph, and connect it to adjacent cells based on the trampoline values.

With the graph constructed, we'll employ the Breadth-First Search (BFS) algorithm to find the shortest path from the starting vertex (top left cell) to the ending vertex (bottom right cell) in the maze graph. BFS ensures that we explore all possible paths in a systematic manner, guaranteeing the shortest path is found.

During BFS traversal, we'll maintain a queue to keep track of vertices to visit, a visited array to mark visited vertices, and a parent array to track the path taken. Once we reach the ending vertex, we'll backtrack through the parent array to generate the path from start to end.

Finally, we'll format the path as a sequence of moves (N, E, S, W) representing North, East, South, and West directions, respectively. This sequence of moves will be written to the output file ('output.txt') in the specified format, providing a clear solution to the "Jumping Jim" maze problem that can be easily followed and executed.

B. Graph Representation:

To model the problem input, a grid-based graph structure is suitable. Each cell in the maze grid corresponds to a vertex in the graph, and the edges represent possible movements between adjacent cells. Specifically:
- Vertices: Each cell in the maze grid is a vertex in the graph.
- Edges: Edges connect vertices based on valid movements within the maze (north, south, east, west). An edge exists between two vertices if the movement between the corresponding cells is allowed.

C. Graph Construction:

The 'buildGraph' function constructs the graph based on the maze and the given cells. It iterates through each cell and checks valid movements (north, south, east, west) based on the cell's value. If a movement is valid, it adds an edge between the current cell and the destination cell in the graph.

Pseudocode:

```
function buildGraph(graph, maze, cells):
  dimension = maze.dimension
  for each cell in cells:
    index = cell.row * dimension + cell.col + 1

    if cell.row - cell.value >= 0:
      destIndex = (cell.row - cell.value) * dimension + cell.col + 1
      addEdge(graph, index, destIndex)

    if cell.row + cell.value < dimension:
      destIndex = (cell.row + cell.value) * dimension + cell.col + 1
      addEdge(graph, index, destIndex)

    if cell.col + cell.value < dimension:
      destIndex = cell.row * dimension + cell.col + cell.value + 1
      addEdge(graph, index, destIndex)

    if cell.col - cell.value >= 0:
      destIndex = cell.row * dimension + cell.col - cell.value + 1
      addEdge(graph, index, destIndex)
```

D. Algorithm for Finding Path:

The algorithm used to solve the problem is a graph traversal approach, specifically BFS (Breadth-First Search). It starts from the starting vertex (1-indexed) and explores adjacent vertices level by level until reaching the destination vertex (last cell in the maze). This algorithm identifies the sequence of moves Jim must take to reach the goal by maintaining a parent array to backtrack the path from the destination to the start.

Pseudocode:

```
function findPath(graph, start, end, path):
  visited = array of size graph.size() initialized to false
  queue = empty queue
  parent = array of size graph.size() initialized to -1

  queue.enqueue(start)
  visited[start] = true

  foundPath = false

  while queue is not empty:
    current = queue.dequeue()

    if current == end:
      foundPath = true
      break

    for each neighbor of current in graph:
```

```
        if neighbor is not visited:
            queue.enqueue(neighbor)
            visited[neighbor] = true
            parent[neighbor] = current

    if foundPath:
        current = end
        while current != start:
            path.append(current)
            current = parent[current]
        path.append(start)
        path.reverse()

    return foundPath
```

This algorithm ensures that Jim explores the maze in a systematic manner, finding the shortest path from the starting cell to the destination cell.
The main function reads input from a file, constructs the maze and graph, finds the path using the BFS algorithm, and writes the sequence of moves to an output file.
This report outlines the approach to solving the problem using a graph-based representation and BFS algorithm, along with corresponding pseudocode for clarity.