ICS202 Project Report

Treasure Quest

Kaltham Alhashmi – 202371470

Sana Ammar – 202337790

F61

1. Data structure used for the layout chamber

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);


    int n = scanner.nextInt();
    int m = scanner.nextInt();
    int k = scanner.nextInt();
    scanner.nextLine();
    // Read map
    char[][] map = readMap(n, m);

    // Find S and E positions
    int[] startEnd = findSpecialTiles(map);
    int start = startEnd[0];
    int end = startEnd[1];
    System.out.println("Start: " + start + ", End: " + end);
    int[][] monsters = readMonsterPositions(k, n ,m);


    boolean[][] safeMap = markMonitoredTiles(map, n, m, monsters);
```

As shown in the figure we represent the layout of the chamber using:

- a 2D character array called “map” which can hold

  S → start point
  E → end point
  # → wall or blocked tile

  . → open path

- Then, a 2D boolean array called safeMap is created:
  to mark which tiles are safe for movement; where true refers to safe tiles, and false refers to blocked /monster range tiles.

2. Data structure used for the shortest path algorithm

```java
public class Graph {  5 usages
    int vertices;  6 usages
    LinkedList<Edge>[] adjacencylist;  6 usages

    public Graph(int vertices) {  1 usage
        this.vertices = vertices;
        adjacencylist = new LinkedList[vertices];
        for (int i = 0; i < vertices; i++) {
            adjacencylist[i] = new LinkedList<>();
        }
    }

    public void addEdge(int source, int destination, int weight) {
        Edge edge = new Edge(source, destination, weight);
        adjacencylist[source].addFirst(edge);
        edge = new Edge(destination, source, weight);
        adjacencylist[destination].addFirst(edge);
    }
```

As shown, we used an adjacency list to represent the graph; where each node has a linked list of its connected neighbors and each neighbor is represented by an Edge object.

We have selected this structure regarding its efficiency:

- We only store actual edges, not full VxV matrices.
- We can loop over neighbors quickly without searching the whole graph.

This design is critical for graphs where most nodes only have a few connections like a 2D grid map.

3. Shortest path algorithm and its analysis

To find the shortest path, we used Dijkstra's Algorithm with a custom MinHeap for priority queues operations and an array of HeapNode objects to track the shortest distances.

Flow:

- Initialize all nodes distances to be = infinity
- Putting all nodes into a min-heap priority queue
- Set the start node = 1
- While the heap is not empty, we extract the node with the smallest distance and for each neighbor, if a shorter path is found, update its distance and reorder the heap
- When the end node is finalized, the shortest path length is available.

Why?

1. Dijkstra Works well on graphs with non-negative weights (our map uses weight 1 for each move).
2. Dijkstra is efficient with a priority queue ($O((V + E) \log V)$).
3. Dijkstra guaranteed to find the shortest path if one exists.
4. The map is modeled as a weighted undirected graph.
5. We only care about minimal steps, not multiple paths or negative cycles.