Summary on methods, algorithms, and data structures: For assignment 3

ASSIGNMENT NUMBER : 03
NAME : Dinesh Raj Pampati
ULID : dpampat
NAME OF SECRET DIRECTORY : namo

Summary on methods, algorithms, and data structures:

1. Methods Used:

- **checkFlags(boolean[] bfsFlags, int size)**: Returns true if there exists any unvisited node, false otherwise.
- **selectTheFirstNode(boolean[] flag)**: Returns the index of the first unvisited node, or -1 if all nodes are visited.
- **main(String[] args)**: The driver method, reads graph data from a file, processes it, and prints the connected components using both BFS and DFS.
- **findCycle(int[] vertices, int[] cycleDectecIntegers)**: Determines if adding an edge between two vertices would create a cycle.
- **findabsolute(int[] vertices, int[] cycleDectecIntegers)**: Finds the absolute root of a vertex in the union-find structure.

2. Algorithm:

- **Kruskal's Algorithm**:
    1. Initialize a structure to keep track of connected components.
    2. Sort all the edges in increasing order by weight.
    3. For each edge, in increasing weight:
        - Check if adding the edge would create a cycle using the union-find structure.
        - If it doesn't create a cycle, add the edge to the MST.
    4. Continue until the MST contains $V-1$ edges or all edges are considered.
- **Prim's Algorithm**:
    1. Start with an arbitrary vertex as the initial MST.
    2. Grow the MST by adding the minimum weight edge that has one endpoint in the MST and the other outside it.
    3. Repeat until the MST spans all the vertices.

3. Data Structures Used:

- **ArrayList<LinkedList<Integer>>**: Used to represent adjacency lists of the graph for both BFS and DFS.
- **boolean[]**: Used as flags to keep track of visited nodes during BFS and DFS.
- **Queue<Integer>**: Used to store nodes during BFS traversal.
- **Stack<Integer>**: Used to store nodes during DFS traversal.

- **Scanner**: Used to read input from the file.
- **File**: Used to specify the file path for the input.
- **Hashtable<Integer, PriorityQueue<VertexWeighted>>**: Used to store the edges associated with each vertex for Prim's algorithm.
- **PriorityQueue<VertexWeighted>**: Used to get the minimum weight edge for both Kruskal's and Prim's algorithms.

Time Complexity:

- **Kruskal's Algorithm**:
    - Initialization: $O(V)$, where V is the number of vertices.
    - Main Loop: $O(E)$, where E is the number of edges.
    - Finding Cycles: $O(V)$ for each edge.
    - Overall complexity: $O(V)+O(E \times V)$. In the worst case, this can be simplified to $O(E \times V)$. However, with path compression and union by rank optimizations in the union-find data structure, the complexity can be reduced $O(E \log V)$.
    - Inside the loop:
        - **values.poll()**: $O(\log E)$
        - **findCycle()**: $O(V)$
    - Overall complexity for Kruskal's: $O(E \times (\log E + V))$.
- **Prim's Algorithm**:
    - Main loop: $O(V)$
    - Nested loop: $O(V)$
    - Innermost loop: $O(E)$
    - Operations inside the innermost loop: $O(\log E)$
    - Rest of the operations inside the main loop: $O(\log E)$
    - Overall complexity: $(V \times V \times E \times \log E)$. This is not a strict upper-bound. The real-world scenario would likely be a lot better since not every vertex will connect to every other vertex. However, based on the code, this would be the theoretical upper bound.

Space Complexity:

- **Kruskal's Algorithm**:
    - Arrays: $O(V)$.
    - Priority Queue: $O(E)$.

    - Overall space complexity: $O(V+E)$.
- **Prim's Algorithm**:
    - The space complexity is primarily determined by the data structures used, which include the priority queue and the visited list. The priority queue can have at most E edges, and the visited list can have at most V vertices. Thus, the space complexity is $O(V+E)$.