Summary on methods, algorithms, and data structures: For assignment 2

ASSIGNMENT NUMBER : 02
NAME : Dinesh Raj Pampa
ULID : dpampat
NAME OF SECRET DIRECTORY : namo002

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.

2. Algorithm:

- **Breadth-First Search (BFS)**:
    1. Start from an unvisited node and mark it as visited.
    2. Explore its adjacent nodes and mark them as visited.
    3. Repeat the process for all adjacent nodes, exploring their adjacent nodes until all nodes are visited.
    4. If there exist any unvisited nodes, repeat the process starting from an unvisited node.
    5. Print each connected component.
- **Depth-First Search (DFS)**:
    1. Start from an unvisited node, mark it as visited and push it to the stack.
    2. Explore its adjacent nodes. For each adjacent node, mark it as visited and push it to the stack.
    3. Repeat the process for the node on the top of the stack until the stack is empty.
    4. If there exist any unvisited nodes, repeat the process starting from an unvisited node.
    5. Print each connected component.

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.

Time Complexity

The overall time complexity of the code is determined primarily by the time complexities of the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms used, since these are the core computational parts of the program.

1. **BFS Time Complexity**: In BFS, each vertex is visited once, and each edge is also considered once when exploring the adjacent vertices. This leads to a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. The inner and outer while loops in the BFS part of the code ensure that all vertices and edges are considered, leading to this complexity.

```
while (checkFlags(bfsFlags, lastVertex)) {
  //...
  while (!q.isEmpty()) {
    //...
    while (!bfsList.get(i).isEmpty()) {
      //...operations within are O(1)
    }
  }
}
```

2. **DFS Time Complexity**: Similarly, in DFS, each vertex is visited once, and each edge is considered once. This results in the same time complexity of O(V + E). The nested while loops in the DFS section iterate over each vertex and its adjacent vertices, ensuring each vertex and edge is accounted for.

```
while (checkFlags(dfsFlags, lastVertex)) {
  //...
  while (!st.empty()) {
    //...
    while (!dfsList.get(j).isEmpty()) {
      //...operations within are O(1)
    }
  }
```

}Since BFS and DFS are performed sequentially and not nested, the overall time complexity of the program is O(V + E), rather than O(2 * (V + E)), because constant factors are dropped in Big O notation.

Space Complexity

The space complexity of the code is determined by the amount of additional memory used by the program relative to the input size. Several data structures contribute to the space complexity:

1. **Adjacency Lists**: The **bfsList** and **dfsList** are adjacency lists used to represent the graph. Each of these lists has a length of V (number of vertices), and each entry in the list has a LinkedList containing the adjacent vertices, contributing to a total of E (number of edges) entries across all lists. Therefore, the space complexity contributed by these lists is O(V + E).

```
ArrayList<LinkedList<Integer>> bfsList = new ArrayList<LinkedList<Integer>>(lastVertex);

ArrayList<LinkedList<Integer>> dfsList = new ArrayList<LinkedList<Integer>>(lastVertex);
```

2. **Visited Flags**: The **bfsFlags** and **dfsFlags** arrays are used to keep track of whether each vertex has been visited during the BFS and DFS traversals, respectively. These arrays each have a length of V, contributing O(V) to the space complexity.

```
boolean bfsFlags[] = new boolean[bfsList.size()];

boolean dfsFlags[] = new boolean[dfsList.size()];
```

3. **Queue/Stack for Traversal**: The Queue (**q**) used in BFS and the Stack (**st**) used in DFS can, in the worst case, store all vertices, leading to an additional O(V) space complexity.

```
Queue<Integer> q = new LinkedList<>();

Stack<Integer> st = new Stack<Integer>();
```

All of these space complexities are additive, resulting in an overall space complexity of O(V + E) + O(V) + O(V) = O(V + E), as constants and lower-order terms are omitted in Big O notation.