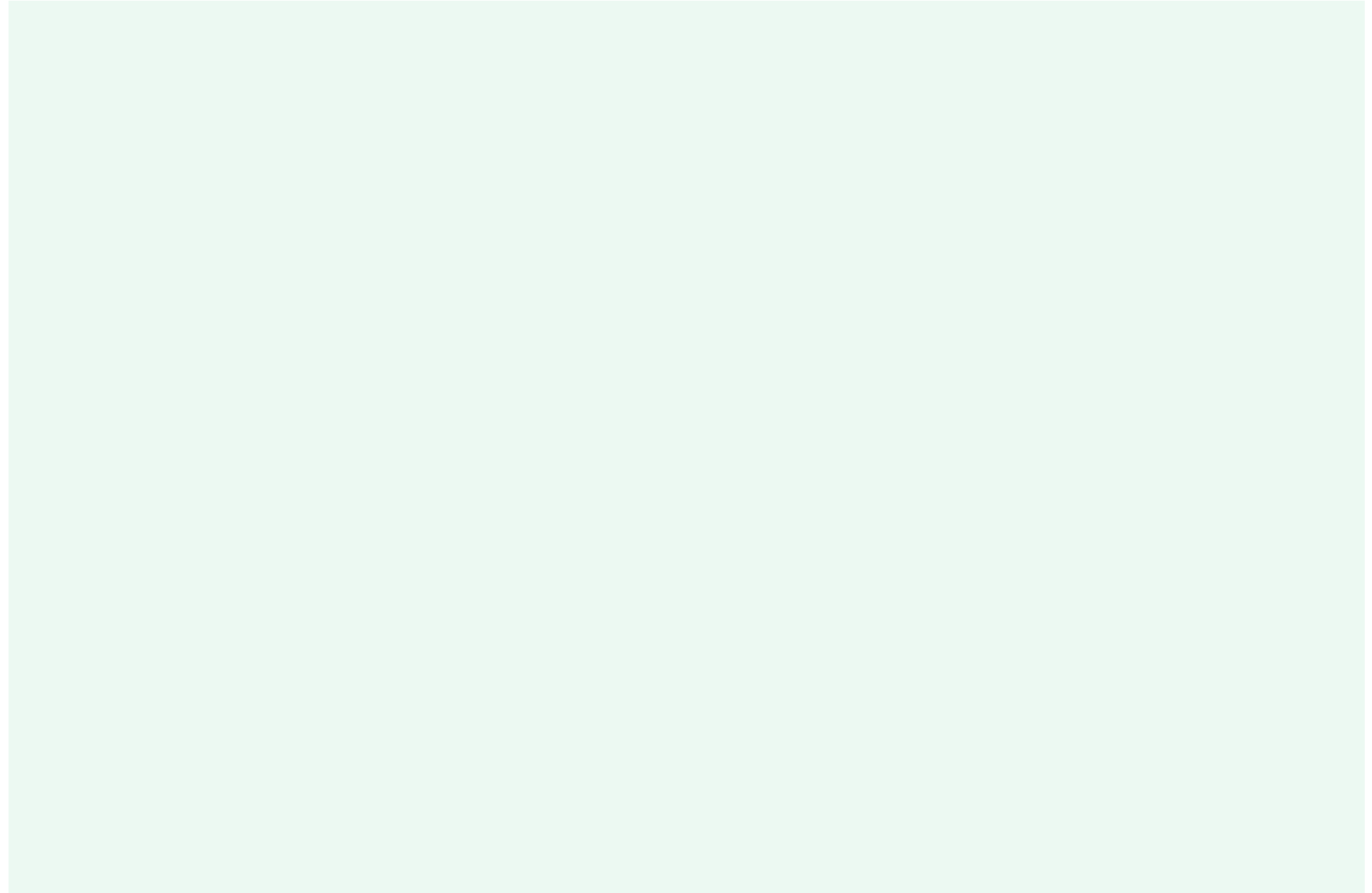


0/1 Matrix BFS Explanation





```

1  /*
2  Q. 0/1 Matrix (Bfs Problem)
3  Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.
4
5  The distance between two adjacent cells is 1.
6
7
8
9  Example 1:
10
11
12
13  Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
14
15
16  | 0, 0, 0 |
17  | 0, 1, 0 |
18  | 0, 1, 0 |
19  | 0, 0, 0 |
20
21  Output: [[0,0,0],[0,1,0],[0,0,0]]
22  Example 2:
23
24
25  Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
26
27
28  | 0, 0, 0 |
29  | 0, 1, 0 |
30  | 1, 1, 1 |
31  | 0, 0, 0 |
32
33  Output: [[0,0,0],[0,1,0],[1,2,1]]

```

```
34 |
35 */
40
41 class Pair{
42     int row;
43     int col;
44     int steps;
45     Pair(int row,int col, int steps){
46         this.row = row;
47         this.col = col;
48         this.steps = steps;
49     }
50 }
```

```

52 public class Q_01_Matrix {
53     /*
54     * Time Complexity: O(N*M) ----> For every node four directions
55     * Space Complexity: O(N*M)
56     */
57     Codeium: Refactor | Explain | X
58     public static int[][] updateMatrix(int[][] mat) {
59         int n = mat.length;
60         int m = mat[0].length;
61         int[][] vis = new int[n][m];
62         int[][] dist = new int[n][m];
63         Queue<Pair> q = new LinkedList<>();
64         for(int i = 0; i < n; i++){
65             for(int j = 0; j < m; j++){
66                 if(mat[i][j] == 0){
67                     q.add(new Pair(i, j, 0));
68                     vis[i][j] = 1;
69                 }
70             }
71         }
72         int[] delRow = {-1, 0, 1, 0};
73         int[] delCol = {0, 1, 0, -1};
74
75         while(!q.isEmpty()){
76             int row = q.peek().row;
77             int col = q.peek().col;
78             int steps = q.peek().steps;
79             dist[row][col] = steps;
80
81             q.remove();
82             for(int i = 0; i < 4; i++){
83                 int nrow = row + delRow[i];
84                 int ncol = col + delCol[i];
85
86                 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0){
87                     vis[nrow][ncol] = 1;
88                     q.add(new Pair(nrow, ncol, steps + 1));
89                 }
90             }
91         }
92     }
93 }

```

```

90     }
91 }
92 return dist;
93 }
}

Codeium: Refactor | Explain | Generate Javadoc | X
94 public static void main(String[] args) {
95     int[][] mat = {{0, 0, 0}, {0, 1, 0}, {1, 1, 1}};
96     int[][] dist = updateMatrix(mat);
97     for(int[] nums : dist){
98         System.out.println(Arrays.toString(nums));
99     }
100 }
101 }
102

```

Kindly explain the code along with example and output



The code provided solves the **0/1 Matrix Problem** using a Breadth-First Search (BFS) algorithm. Let's break it down step by step:

Problem Description

Given an $m \times n$ binary matrix mat , each cell is either 0 or 1. The task is to find the distance of the nearest 0 for every cell in the matrix. The distance between two adjacent cells is 1.

Example Input and Output

Example 1:

Input:

```
plaintext
```

```
mat = [[0, 0, 0],
       [0, 1, 0],
```

```
[0, 0, 0]]
```

Output:

```
plaintext
```

```
dist = [[0, 0, 0],  
        [0, 1, 0],  
        [0, 0, 0]]
```

Explanation:

- The `1` in the middle is 1 step away from the nearest `0`.

Example 2:**Input:**

```
plaintext
```

```
mat = [[0, 0, 0],  
       [0, 1, 0],  
       [1, 1, 1]]
```

Output:

```
plaintext
```

```
dist = [[0, 0, 0],  
        [0, 1, 0],  
        [1, 2, 1]]
```

Explanation:

- For the bottom-left and bottom-right cells, the distance to the nearest `0` is 1 and 2, respectively.

Code Explanation

The code is implemented in Java. Here's how it works:

1. Pair Class

The `Pair` class is a helper class to store the row, column, and distance from the nearest `0` for each cell.

2. updateMatrix Method

This method takes the matrix as input and returns a matrix of distances.

Steps:

1. Initialization:

- `n` and `m`: Dimensions of the matrix.
- `dist`: A 2D array to store the distances, initialized to zero.
- `vis`: A 2D array to mark visited cells (all initially 0).
- `Queue<Pair> q`: A queue for BFS.

2. Queue Initialization:

- Traverse the matrix.
- For every `0`, add its coordinates to the queue and mark it as visited.

3. BFS Logic:

- Use a queue to process cells level by level (BFS).
- For each cell in the queue:
 - Pop the cell and get its coordinates (`row`, `col`) and steps.
 - Update the distance for that cell in the `dist` array.
 - For all 4 possible directions (top, bottom, left, right):
 - Compute new coordinates (`nrow`, `ncol`).
 - If the new cell is within bounds and not visited:
 - Mark it visited.

- Add it to the queue with steps incremented by 1.

4. Return Result:

- The `dist` matrix is returned.

3. Main Method

- The matrix is defined as input.
- The `updateMatrix` method is called to compute distances.
- The result is printed row by row.

BFS Example Walkthrough

Input:

```
plaintext
```

```
mat = [[0, 0, 0],
        [0, 1, 0],
        [1, 1, 1]]
```

1. Initialization:

- Start with all `0`'s in the queue:

```
plaintext
```

```
Queue: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (1, 0, 0), (1, 2, 0)]
```

2. BFS Traversal:

- Pop `(0, 0, 0)` → Process neighbors, add `(1, 1, 1)` to the queue.
- Pop `(0, 1, 0)` → Add neighbors (if unvisited).
- Continue until all cells are visited.

Output:

```
plaintext
```

```
dist = [[0, 0, 0],  
        [0, 1, 0],  
        [1, 2, 1]]
```

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is processed once, and all 4 neighbors are checked.
2. **Space Complexity:** $O(n \times m)$
 - Due to the `queue` and `visited` matrix.

Output for Given Input

Input:

```
plaintext
```

```
mat = [[0, 0, 0],  
        [0, 1, 0],  
        [1, 1, 1]]
```

Output:

```
plaintext
```

```
dist = [[0, 0, 0],  
        [0, 1, 0],  
        [1, 2, 1]]
```



```
108  /*
109  Given a binary grid of n*m. Find the distance of the nearest 1 in the grid for each cell.
110  The distance is calculated as |i1 - i2| + |j1 - j2|, where i1, j1 are the row number and column number of
111  the current cell, and i2, j2 are the row number and column number of the nearest cell having value 1.
112
113
114  Example 1:
115
116  Input: grid = {{0,1,1,0},{1,1,0,0},{0,0,1,1}}
117  Output: {{1,0,0,1},{0,0,1,1},{1,1,0,0}}
118  Explanation: The grid is-
119  0 1 1 0
120  1 1 0 0
121  0 0 1 1
122  0's at (0,0), (0,3), (1,2), (1,3), (2,0) and
123  (2,1) are at a distance of 1 from 1's at (0,1),
124  (0,2), (0,2), (2,3), (1,0) and (1,1)
125  respectively.
126
127
128  Example 2:
129
130  Input: grid = {{1,0,1},{1,1,0},{1,0,0}}
131  Output: {{0,1,0},{0,0,1},{0,1,2}}
132  Explanation: The grid is-
133  1 0 1
134  1 1 0
135  1 0 0
136  0's at (0,1), (1,2), (2,1) and (2,2) are at a
137  distance of 1, 1, 1 and 2 from 1's at (0,0),
138  (0,2), (2,0) and (1,1) respectively.
139
140  */
```

```
143 class Pair{
144     int row;
145     int col;
146     int steps;
147     Pair(int row,int col, int steps){
148         this.row = row;
149         this.col = col;
150         this.steps = steps;
151     }
152 }
```

```

154 class Solution
155 {
156     //Function to find distance of nearest 1 in the grid for each cell.
157     public int[][] nearest(int[][] mat)
158     {
159         int n = mat.length;
160         int m = mat[0].length;
161         int[][] vis = new int[n][m];
162         int[][] dist = new int[n][m];
163         Queue<Pair> q = new LinkedList<>();
164         for(int i = 0; i < n; i++){
165             for(int j = 0; j < m; j++){
166                 if(mat[i][j] == 1){
167                     q.add(new Pair(i, j, 0));
168                     vis[i][j] = 1;
169                 }
170             }
171         }
172
173         int[] delRow = {-1, 0, 1, 0};
174         int[] delCol = {0, 1, 0, -1};
175
176         while(!q.isEmpty()){
177             int row = q.peek().row;
178             int col = q.peek().col;
179             int steps = q.peek().steps;
180             dist[row][col] = steps;
181
182             q.remove();
183             for(int i = 0; i < 4; i++){
184                 int nrow = row + delRow[i];
185                 int ncol = col + delCol[i];
186
187                 if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0){
188                     vis[nrow][ncol] = 1;
189                     q.add(new Pair(nrow, ncol, steps + 1));
190                 }
191             }
192         }
193     }

```

```
193     return dist;
194 }
195 }
196
```

Kindly explain the code along with example and output



The second set of code solves a problem similar to the first, but this time the goal is to find the distance of the nearest `1` for each cell in a binary grid. Here's a detailed explanation:

Problem Description

Given a binary grid of size $n \times m$, find the distance of the nearest `1` for each cell. The distance is calculated as:

$$|i_1 - i_2| + |j_1 - j_2|$$

Where:

- (i_1, j_1) are the row and column indices of the current cell.
- (i_2, j_2) are the row and column indices of the nearest cell containing `1`.

Example Input and Output

Example 1:

Input:

```
plaintext

grid = [[0, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1]]
```

Output:

```
plaintext

dist = [[1, 0, 0, 1],
        [0, 0, 1, 2],
        [1, 1, 2, 0]]
```

Explanation:

- The cells at `(0, 0)`, `(0, 3)`, `(1, 2)`, `(1, 3)`, `(2, 0)`, and `(2, 1)` have their nearest `1` at distances `1`, `1`, `1`, `2`, `1`, and `1`, respectively.

Example 2:

Input:

```
plaintext

grid = [[1, 0, 1],
        [1, 1, 0],
        [1, 0, 0]]
```

Output:

```
plaintext

dist = [[0, 1, 0],
        [0, 0, 1],
        [0, 1, 2]]
```

Explanation:

- The cells at `(0, 1)`, `(1, 2)`, and `(2, 2)` have their nearest `1` at distances `1`, `1`, and `2`, respectively.

Code Explanation

The code uses the **Breadth-First Search (BFS)** algorithm to calculate distances, where cells containing `1` are the starting points.

Step-by-Step Breakdown

1. Class Pair:

- The `Pair` class is used to store the row, column, and distance for each cell.

2. `nearest` Method:

- **Inputs:** A binary matrix *mat*.
- **Output:** A matrix *dist* of the same size, where each cell contains the distance to the nearest `1`.

3. Initialization:

- *n, m*: Dimensions of the matrix.
- *vis*: A `visited` array to mark cells already processed.
- *dist*: A result array initialized to `0`.
- *q*: A queue to perform BFS.

4. Step 1 - Queue Initialization:

- Traverse the grid. For every cell with `1`, add its coordinates to the queue with distance `0`.
- Mark these cells as visited.

5. Step 2 - BFS Traversal:

- While the queue is not empty:
 - Dequeue the current cell and retrieve its coordinates and distance.
 - For each of the 4 possible directions (top, bottom, left, right):
 - Compute the new coordinates.
 - If the new cell is within bounds and not visited:
 - Update its distance in the *dist* array.
 - Mark it as visited and add it to the queue with an incremented distance.

6. Return:

- Return the *dist* matrix.

BFS Example Walkthrough

Input:

```
plaintext
```

```
grid = [[0, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1]]
```

BFS Initialization:

- Queue:

```
plaintext
```

```
[(0, 1, 0), (0, 2, 0), (1, 0, 0), (1, 1, 0), (2, 3, 0)]
```

- *vis*:

```
plaintext
```

```
[[0, 1, 1, 0],
 [1, 1, 0, 0],
 [0, 0, 0, 1]]
```

BFS Execution:

1. Process `(0, 1, 0)`:
 - Neighbors: `(0, 0, 1), (1, 1, 1)`.
 - Update *dist*:

```
plaintext
```

```
[[1, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

2. Continue processing all queue elements until the queue is empty.

Complexity

1. **Time Complexity:** $O(n \times m)$

- Each cell is visited at most once, and its neighbors are processed.

2. **Space Complexity:** $O(n \times m)$

- Space for the queue and the `visited` array.

Output for Given Input

Input:

```
plaintext
```

```
grid = [[0, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1]]
```

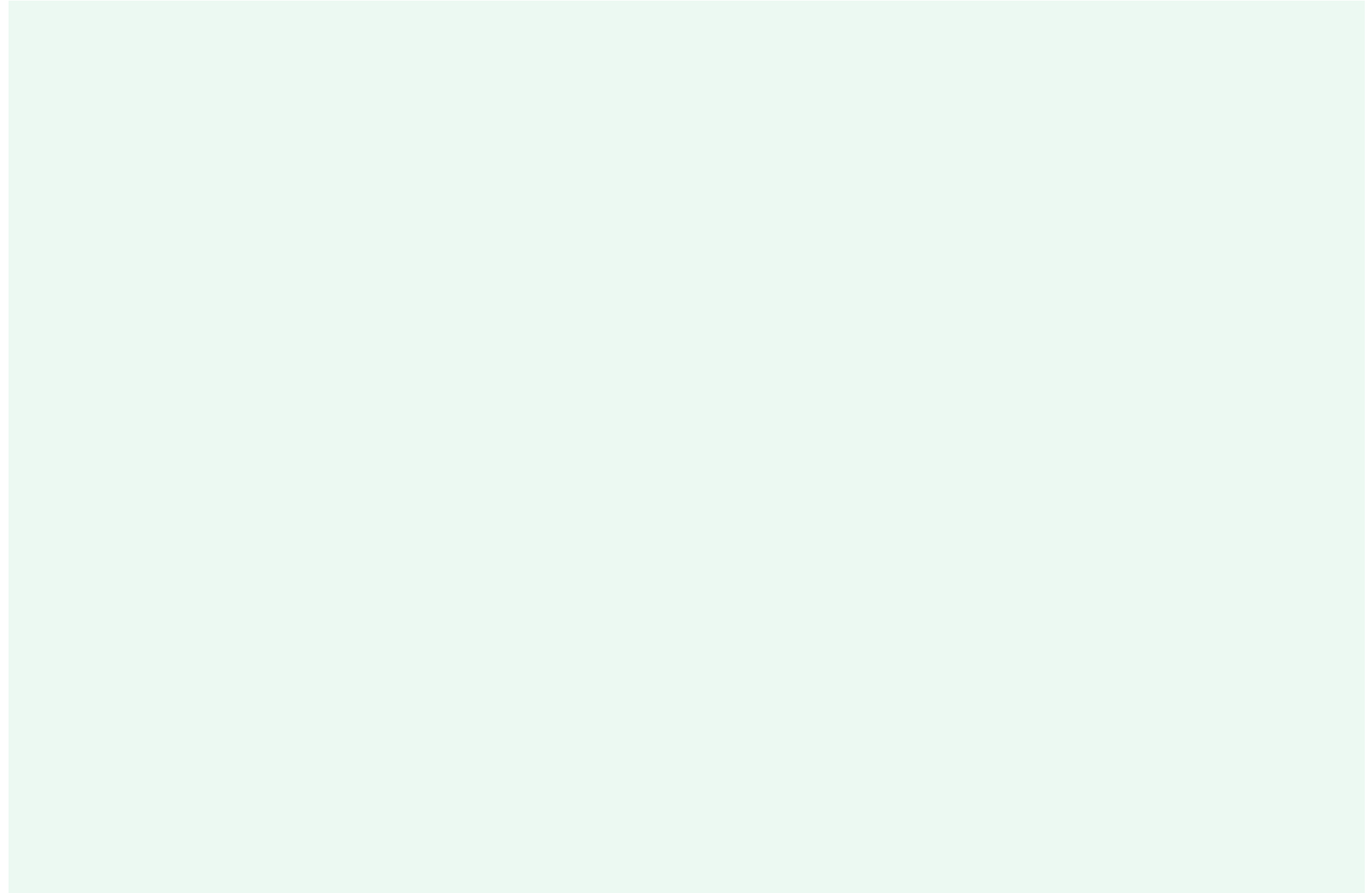
Output:

```
plaintext
```

```
dist = [[1, 0, 0, 1],
        [0, 0, 1, 2],
```

[1, 1, 2, 0]]





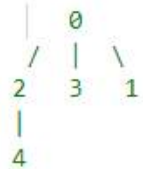
```

1  /*
2  Q. Bipartite Graph
3  Given an adjacency list of a graph adj of V no. of vertices having 0 based index.
4  Check whether the graph is bipartite or not.

```

Example 1:

Input:

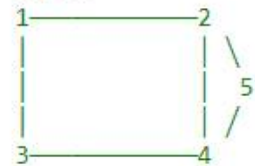


Output: true

Explanation: The given graph can be colored in two colors so, it is a bipartite graph.

Example 2:

Input:



Output: false

Explanation: The given graph cannot be colored in two colors such that color of adjacent vertices differs.

Example 3:



```
40 | 1 | 1
41 | 3 — 4
42 |
43 | output: true
44 | */
```

Codeium: Refactor | Explain

```

52 public class Bipartite_Graph {
53     /*
54     * Time Complexity: O(N + 2E)
55     * Space Complexity: O(N)
56     */
57
58
59
60     // DFS - Depth First Search
61
62     /*
63     public static boolean dfs(int start, int n, ArrayList<ArrayList<Integer>>adj, int[] color, int currCol){
64         color[start] = currCol;
65         for(int node : adj.get(start)){
66             if(color[node] == -1){
67                 if(dfs(node, n, adj, color, 1 - currCol) == false){
68                     return false;
69                 }
70             }
71             else if(color[node] == currCol) return false;
72         }
73         return true;
74     }
75     public static boolean isBipartite(int n, ArrayList<ArrayList<Integer>>adj) {
76         int[] color = new int[n];
77         Arrays.fill(color, -1);
78         for(int i = 0; i < n; i++){
79             if(color[i] == -1){
80                 if(dfs(i, n, adj, color, 0) == false){
81                     return false;
82                 }
83             }
84         }
85         return true;
86     }
87     */

```

Codeium: Refactor | Explain | Generate Javadoc | ✕

```

91 public static boolean check(int start, int n, ArrayList<ArrayList<Integer>>adj, int[] color){
92     Queue<Integer> q = new LinkedList<>();
93     q.add(start);
94     color[start] = 0;
95     while(!q.isEmpty()){
96         int node = q.remove();
97         for(int it : adj.get(node)){
98             if(color[it] == -1){
99                 color[it] = 1 - color[node];
100                 q.add(it);
101             }
102             else if(color[it] == color[node]){
103                 return false;
104             }
105         }
106     }
107     return true;
108 }

```

Codeium: Refactor | Explain | Generate Javadoc | ✕

```

109 public static boolean isBipartite(int n, ArrayList<ArrayList<Integer>>adj) {
110     int[] color = new int[n];
111     Arrays.fill(color, -1);
112     for(int i = 0; i < n; i++){
113         if(color[i] == -1){
114             if(check(i, n, adj, color) == false){
115                 return false;
116             }
117         }
118     }
119     return true;
120 }

```



```

121 public static void main(String[] args) {
122     ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
123     for (int i = 0; i < 4; i++) {
124         adj.add(new ArrayList<>());
125     }
126     adj.get(0).add(1);
127     adj.get(0).add(3);
128     adj.get(1).add(0);
129     adj.get(1).add(2);
130     adj.get(2).add(1);
131     adj.get(2).add(3);
132     adj.get(3).add(0);
133     adj.get(3).add(2);
134
135     System.out.println(isBipartite(adj.size(), adj));
136 }
137 }
138

```

Kindly explain the code along with example and output



This code determines whether a given graph is **bipartite**. A graph is bipartite if its vertices can be divided into two disjoint sets such that no two adjacent vertices belong to the same set.

Problem Description

Given an adjacency list `adj` of V vertices:

- Check whether the graph is bipartite.
- A bipartite graph can be colored using two colors such that no two adjacent vertices share the same color.

Examples

Example 1:

Input:

```
lua
0 -- 1 -- 2
|      |
4      3
```

Output: `true`

Explanation: The graph can be divided into two sets: $\{0, 2, 4\}$ and $\{1, 3\}$, and it is possible to color the vertices using two colors.

Example 2:

Input:

```
lua
1 -- 2 -- 3
|      |
4 -- 5
```

Output: `false`

Explanation: This graph forms an odd cycle (1-2-3-5-4-1), which cannot be divided into two sets, making it non-bipartite.

Example 3:

Input:

```
lua
1 -- 2
|    |
```

Output: ``true``

Explanation: This graph is bipartite, as the vertices can be divided into two sets: $\{1, 4\}$ and $\{2, 3\}$.

Code Explanation

The code provides two methods to check if the graph is bipartite: one using **DFS** and the other using **BFS**.

DFS Implementation

The method ``dfs`` checks bipartiteness using **Depth-First Search** (DFS).

1. Input:

- ``start``: The current node.
- ``adj``: The adjacency list of the graph.
- ``color``: An array storing the colors of the nodes.
- ``currCol``: The current color.

2. Steps:

- Color the current node (``start``) with ``currCol``.
- Traverse all its neighbors.
- If a neighbor is uncolored, color it with the opposite color (``1 - currCol``) and perform DFS.
- If a neighbor has the same color as the current node, the graph is not bipartite, and the method returns ``false``.

3. Main Method:

- Initializes a ``color`` array with ``-1`` (uncolored).
 - Calls ``dfs`` for every unvisited node to ensure all components are checked.
 - If all components are bipartite, returns ``true``.
-

BFS Implementation

The method ``check`` uses **Breadth-First Search** (BFS).

1. Input:

- Same as the DFS method.

2. Steps:

- Start BFS from the given node.
- Assign the current node a color (e.g., `0`).
- Process all neighbors:
 - If a neighbor is uncolored, color it with the opposite color and add it to the queue.
 - If a neighbor has the same color as the current node, return `false`.

3. Main Method:

- Similar to the DFS main method but calls `check` instead.

Complexity

1. Time Complexity: $O(V + E)$

- Each node and edge is processed once.

2. Space Complexity: $O(V)$

- Space for the `color` array and recursion stack/queue.

Example Execution

Input:

Graph (adjacency list):

CSS

```
0 -> [1, 3]
1 -> [0, 2]
2 -> [1, 3]
3 -> [0, 2]
```

Execution:

1. Using DFS:

- Start at node `0` and color it `0`.
- Color `1` as `1`, `3` as `1`, and `2` as `0`.
- All neighbors have opposite colors. Result: `true`.

2. Using BFS:

- Start at node `0` and enqueue it.
- Process each node, assigning colors. Result: `true`.

Output:

```
plaintext
```

```
true
```

Test Case Implementation in `main`

The `main` method sets up the adjacency list and calls `isBipartite`:

```
java

ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
for (int i = 0; i < 4; i++) {
    adj.add(new ArrayList<>());
}
adj.get(0).add(1);
adj.get(0).add(3);
adj.get(1).add(0);
adj.get(1).add(2);
adj.get(2).add(1);
adj.get(2).add(3);
adj.get(3).add(0);
adj.get(3).add(2);

System.out.println(isBipartite(adj.size(), adj)); // Output: true
```



```

1  /*
2  Detect cycle in a directed graph
3  Stand out from the crowd. Prepare with Complete Interview Preparation
4
5  Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, check whether it contains any cycle or not.
6
7
8  Example 1:
9
10 Input:
11 Example 3:
12
13 1----->2
14 |         |
15 |         |
16 |         |
17 3<-----4
18
19 Output: true
20
21 Example 2:
22
23 Input:
24 1----->2
25 |         | \
26 |         |  5
27 |         | /
28 3----->4
29
30 Output: false
31 Explanation: no cycle in the graph
32 */

```

```

36  public class Cycle_Detection_in_Directed_Graph {
37      /*
38       * Time Complexity: O(N + E)
39       * Space Complexity: O(N)
40       */
41
42      // DFS - Depth First Search
43
44      Codeium: Refactor | Explain | Generate Javadoc | ✕
45      public static boolean dfs(int node, ArrayList<ArrayList<Integer>> adj, int[] vis, int[] pathVis){
46          vis[node] = 1;
47          pathVis[node] = 1;
48          for(int it : adj.get(node)){
49              if(vis[it] == 0){
50                  if(dfs(it, adj, vis, pathVis) == true) return true;
51              }
52              else if(pathVis[it] == 1) return true;
53          }
54          pathVis[node] = 0;
55          return false;
56      }
57
58      Codeium: Refactor | Explain | Generate Javadoc | ✕
59      public static boolean isCyclic(int n, ArrayList<ArrayList<Integer>> adj) {
60          int[] vis = new int[n];
61          int[] pathVis = new int[n];
62          for(int i = 0; i < n; i++){
63              if(vis[i] == 0){
64                  if(dfs(i, adj, vis, pathVis) == true) return true;
65              }
66          }
67          return false;
68      }
69  }

```

Codeium: Refactor | Explain | Generate Javadoc | X

```

56 public static boolean isCyclic(int n, ArrayList<ArrayList<Integer>> adj) {
57     int[] vis = new int[n];
58     int[] pathVis = new int[n];
59     for(int i = 0; i < n; i++){
60         if(vis[i] == 0){
61             if(dfs(i, adj, vis, pathVis) == true) return true;
62         }
63     }
64     return false;
65 }

```

Codeium: Refactor | Explain | Generate Javadoc | X

```

66 public static void main(String[] args) {
67     int n = 6;
68     ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
69     for (int i = 0; i < n; i++) {
70         adj.add(new ArrayList<>());
71     }
72     adj.get(1).add(2);
73     adj.get(2).add(3);
74     adj.get(3).add(4);
75     adj.get(3).add(5);
76     adj.get(4).add(2);
77
78     System.out.println(isCyclic(n, adj));
79 }
80 }
81

```

Kindly explain the code along with example and output



The given code determines whether a **directed graph** contains a cycle using **Depth First Search (DFS)**. Here's a detailed explanation:

Problem Description

Input:

- A directed graph with V vertices and E edges, represented as an adjacency list.

Output:

- ``true`` if the graph contains a cycle, ``false`` otherwise.

Examples

Example 1:

Graph:

```

1 → 2
↑   ↓
4 ← 3

```

Adjacency List:

```

0 → 1
1 → 2
2 → 3
3 → 0

```

Output: ``true``

Explanation: The graph contains a cycle: $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

Example 2:

Graph:

```
1 → 2
↓   ↓
3 → 4
```

Adjacency List:

```
0 → 1
1 → 2
2 → 3
3 → 4
```

Output: `false`

Explanation: The graph does not contain any cycles.

Code Explanation

Key Idea

To detect a cycle in a **directed graph**, we use DFS and maintain two arrays:

1. `vis`: Tracks whether a node has been visited.
 2. `pathVis`: Tracks whether a node is part of the current DFS path. If a node is revisited in the current path, a cycle exists.
-

Functions

`dfs` Function

- Performs DFS to check for cycles.
- **Input:**

- ``node``: The current vertex.
- ``adj``: Adjacency list of the graph.
- ``vis``: Array indicating whether a node is visited.
- ``pathVis``: Array indicating whether a node is part of the current path.

- **Steps:**

1. Mark the current node as visited (``vis[node] = 1``) and part of the current path (``pathVis[node] = 1``).
2. Traverse all neighbors:
 - If a neighbor is unvisited, recursively perform DFS.
 - If a neighbor is already in the current path (``pathVis[neighbor] = 1``), a cycle exists.
3. After exploring all neighbors, backtrack by unmarking the node from the current path (``pathVis[node] = 0``).

``isCyclic`` Function

- Initiates the cycle detection process.
- **Steps:**
 1. Initialize ``vis`` and ``pathVis`` arrays with ``0``.
 2. For each unvisited node, perform DFS.
 3. If DFS returns ``true`` for any node, the graph contains a cycle.

``main`` Function

- Constructs the graph using adjacency lists.
- Calls the ``isCyclic`` function to check for cycles.

Complexity

1. **Time Complexity:** $O(V + E)$
 - Each vertex and edge is processed once.
2. **Space Complexity:** $O(V)$
 - Space for the ``vis``, ``pathVis`` arrays, and recursion stack.

Example Walkthrough

Input:

```
java
```

```
int n = 4;
ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
adj.get(0).add(1);
adj.get(1).add(2);
adj.get(2).add(3);
adj.get(3).add(0);
```

Adjacency List:

```
0 → 1
1 → 2
2 → 3
3 → 0
```

Execution:

1. Start DFS from node `0`:
 - Mark `0` as visited and part of the current path.
 - Move to `1`.
2. DFS for `1`:
 - Mark `1` as visited and part of the current path.
 - Move to `2`.
3. DFS for `2`:
 - Mark `2` as visited and part of the current path.
 - Move to `3`.
4. DFS for `3`:
 - Mark `3` as visited and part of the current path.
 - Move back to `0`, which is already in the current path (`pathVis[0] = 1`).
 - **Cycle detected.**

Output:

```
plaintext
```

```
true
```

Input:

```
java
```

```
int n = 4;
ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
adj.get(0).add(1);
adj.get(1).add(2);
adj.get(2).add(3);
```

Adjacency List:

```
0 → 1
1 → 2
2 → 3
```

Execution:

1. Start DFS from node `0`:
 - Traverse `1 → 2 → 3`.
2. No back edge exists.

Output:

plaintext

false



```

1  /*
2   Detect cycle in an undirected graph Using DFS
3   Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.
4   Graph is in the form of adjacency list where adj[i] contains all the nodes ith node is having edge with.
5
6   Example 1:
7
8   Input:
9   V = 5, E = 5
10  adj = {{1}, {0, 2, 4}, {1, 3}, {2, 4}, {1, 3}}
11  Output: 1
12  Explanation:
13
14  1->2->3->4->1 is a cycle.
15  Example 2:
16
17  Input:
18  V = 4, E = 2
19  adj = {{}, {2}, {1, 3}, {2}}
20  Output: 0
21  Explanation:
22
23  No cycle in the graph.
24
25
26  Ex: Input:
27  ①-----②
28  |       | \
29  |       |  ⑤
30  |       | /
31  ③-----④
32
33  Output: True (Yes, there is a cycle in this graph)
34
35  */

```

```

39  public class Cycle_Detection_in_undirected_Graph_using_dfs {
40      /*
41       * Time Complexity: O(N + 2E) + O(N) ----> For every node four directions
42       * Space Complexity: O(N) + O(N)  $\cong$  O(N)
43       */
44      public static boolean dfs(int src, int parent, int n, ArrayList<ArrayList<Integer>> adj, boolean[] vis){
45          vis[src] = true;
46          for(int it : adj.get(src)){
47              if(vis[it] == false){
48                  if(dfs(it, src, n, adj, vis)) return true;
49              }
50              else if(it != parent) return true;
51          }
52          return false;
53      }
54      public static boolean isCycle(int n, ArrayList<ArrayList<Integer>> adj) {
55          boolean[] vis = new boolean[n];
56          int parent = -1;
57          for(int i = 0; i < n; i++){
58              if(vis[i] == false){
59                  if(dfs(i, parent, n, adj, vis)) return true;
60              }
61          }
62          return false;
63      }
64      public static void main(String[] args) {
65          ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
66          for (int i = 0; i < 5; i++) {
67              adj.add(new ArrayList<>());
68          }
69          adj.get(0).add(1);
70          adj.get(1).add(0);
71          adj.get(1).add(2);
72          adj.get(1).add(4);
73          adj.get(2).add(1);
74          adj.get(2).add(3);
75          adj.get(3).add(2);
76          adj.get(3).add(4);
77          adj.get(4).add(1);
78          adj.get(4).add(3);
79          adj.get(4).add(2);
80      }
81  }

```



```

75     adj.get(3).add(2);
76     adj.get(3).add(4);
77     adj.get(4).add(1);
78     adj.get(4).add(3);
79     System.out.println(isCycle(5, adj));
80 }

```

Kindly explain the code along with example and output



The provided code detects a **cycle in an undirected graph** using **Depth First Search (DFS)**. Here's an explanation of the code and how it works:

Problem Description

You are given:

1. An undirected graph with V vertices and E edges.
2. The graph is represented as an adjacency list.

Task: Check whether the graph contains any cycle.

Examples

Example 1:

Input:

- $V = 5, E = 5$
- Adjacency list:

```
makefile
```

```

0: [1]
1: [0, 2, 4]
2: [1, 3]

```

```
3: [2, 4]
4: [1, 3]
```

Output: `true` (1)

Explanation: The graph contains a cycle: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$.

Example 2:

Input:

- $V = 4, E = 2$
- Adjacency list:

```
makefile
```

```
0: []
1: [2]
2: [1, 3]
3: [2]
```

Output: `false` (0)

Explanation: No cycle exists in this graph.

Code Explanation

Key Idea

In an **undirected graph**, a cycle exists if:

1. A node is visited more than once during DFS, and
 2. The second visit is **not from the immediate parent** node.
-

Functions

``dfs`` Function

This function performs a DFS to check for cycles.

- **Input:**
 - ``src``: Current node.
 - ``parent``: The parent of the current node.
 - ``adj``: The adjacency list of the graph.
 - ``vis``: Array indicating whether a node is visited.
- **Steps:**
 1. Mark the current node (``src``) as visited.
 2. Traverse all its neighbors:
 - If a neighbor is unvisited, recursively perform DFS.
 - If a neighbor is already visited but is **not the parent** of the current node, a cycle exists.
 3. Return ``true`` if a cycle is found; otherwise, return ``false``.

``isCycle`` Function

This function initializes the cycle detection process.

- **Steps:**
 1. Create a ``vis`` array to track visited nodes.
 2. For each unvisited node, perform DFS:
 - If DFS detects a cycle, return ``true``.
 3. Return ``false`` if no cycle is found.

``main`` Function

The main function constructs the graph and calls the ``isCycle`` function.

Complexity

1. **Time Complexity:** $O(V + E)$
 - Each vertex and edge is processed once.
 2. **Space Complexity:** $O(V)$
 - Space for the `vis` array and recursion stack.
-

Example Execution

Input:

Graph:

```
makefile
0: [1]
1: [0, 2, 4]
2: [1, 3]
3: [2, 4]
4: [1, 3]
```

Execution:

1. Start DFS from node `0`:
 - Traverse $0 \rightarrow 1$.
2. DFS for node `1`:
 - Traverse $1 \rightarrow 2$ and $1 \rightarrow 4$.
3. DFS for node `2`:
 - Traverse $2 \rightarrow 3$.
4. DFS for node `3`:
 - Traverse $3 \rightarrow 4$, which is already visited but not the parent of `3`.
 - **Cycle detected.**

Output:

```
plaintext
```

```
true
```

Input:

Graph:

```
makefile
```

```
0: []  
1: [2]  
2: [1, 3]  
3: [2]
```

Execution:

1. Start DFS from node `1`:
 - Traverse $1 \rightarrow 2$.
2. DFS for node `2`:
 - Traverse $2 \rightarrow 3$.
3. No back edges exist.

Output:

```
plaintext
```

```
false
```



```

1  /*
2   Detect cycle in an undirected graph
3   Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.
4   Graph is in the form of adjacency list where adj[i] contains all the nodes ith node is having edge with.
5
6   Example 1:
7
8   Input:
9   V = 5, E = 5
10  adj = {{1}, {0, 2, 4}, {1, 3}, {2, 4}, {1, 3}}
11  Output: 1
12  Explanation:
13
14  1->2->3->4->1 is a cycle.
15  Example 2:
16
17  Input:
18  V = 4, E = 2
19  adj = {{}, {2}, {1, 3}, {2}}
20  Output: 0
21  Explanation:
22
23  No cycle in the graph.
24
25
26  Ex: Input:
27  ①-----②
28  |       | \
29  |       |  ⑤
30  |       | /
31  ③-----④
32
33  Output: True (Yes, there is a cycle in this graph)
34
35  */

```

```

41 class Pair{
42     int node;
43     int parent;
44     Pair(int node, int parent){
45         this.node = node;
46         this.parent = parent;
47     }
48 }
Codeium: Refactor | Explain
49 public class Cycle_Detection_in_unirected_Graph_using_bfs {
Codeium: Refactor | Explain | Generate Javadoc | X
50 public static boolean detectCycle(int src, int n, ArrayList<ArrayList<Integer>> adj, boolean[] vis){
51     vis[src] = true;
52     Queue<Pair> q = new LinkedList<>();
53     q.add(new Pair(src, -1));
54     while(!q.isEmpty()){
55         int node = q.peek().node;
56         int parent = q.peek().parent;
57         q.remove();
58         for(int adjacentNode : adj.get(node)){
59             if(vis[adjacentNode] == false){
60                 vis[adjacentNode] = true;
61                 q.add(new Pair(adjacentNode, node));
62             }
63             else if(adjacentNode != parent) return true;
64         }
65     }
66     return false;
67 }
Codeium: Refactor | Explain | Generate Javadoc | X
68 public static boolean isCycle(int n, ArrayList<ArrayList<Integer>> adj) {
69     boolean[] vis = new boolean[n];
70     for(int i = 0; i < n; i++){
71         if(vis[i] == false){
72             if(detectCycle(i, n, adj, vis)) return true;
73         }
74     }
75     return false;
76 }

```

```

77 public static void main(String[] args) {
78     ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
79     for (int i = 0; i < 5; i++) {
80         adj.add(new ArrayList<>());
81     }
82     adj.get(0).add(1);
83     adj.get(1).add(0);
84     adj.get(1).add(2);
85     adj.get(1).add(4);
86     adj.get(2).add(1);
87     adj.get(2).add(3);
88     adj.get(3).add(2);
89     adj.get(3).add(4);
90     adj.get(4).add(1);
91     adj.get(4).add(3);
92     System.out.println(isCycle(5, adj));
93 }
94 }
95

```

Kindly explain the code along with example and output



The provided code detects a **cycle in an undirected graph** using **Breadth First Search (BFS)**. Here's a detailed explanation:

Problem Description

You are given:

1. An undirected graph with V vertices and E edges.
2. The graph is represented as an adjacency list.

Task: Check whether the graph contains any cycle.

Examples

Example 1:

Input:

- $V = 5, E = 5$
- Adjacency list:

```
makefile
```

```
0: [1]
1: [0, 2, 4]
2: [1, 3]
3: [2, 4]
4: [1, 3]
```

Output: `true` (1)

Explanation: The graph contains a cycle: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$.

Example 2:

Input:

- $V = 4, E = 2$
- Adjacency list:

```
makefile
```

```
0: []
1: [2]
2: [1, 3]
3: [2]
```

Output: ``false`` (0)

Explanation: No cycle exists in this graph.

Code Explanation

Key Idea

In an **undirected graph**, a cycle exists if:

1. A node is visited more than once during BFS, and
 2. The second visit is **not from the immediate parent** node.
-

Functions

``detectCycle`` Function

This function performs BFS to check for cycles.

- **Input:**

- ``src``: Starting node for BFS.
- ``adj``: Adjacency list of the graph.
- ``vis``: Array to track visited nodes.

- **Steps:**

1. Mark the source node (``src``) as visited.
2. Initialize a queue and enqueue the source node along with its parent (set to ``-1`` for the root).
3. Process the queue:
 - Dequeue a node and its parent.
 - Traverse all its adjacent nodes:
 - If an adjacent node is unvisited, mark it as visited and enqueue it.
 - If an adjacent node is already visited but is not the parent of the current node, a cycle exists.

``isCycle`` Function

This function initializes the cycle detection process.

- **Steps:**
 1. Create a `vis` array to track visited nodes.
 2. For each unvisited node, perform BFS using `detectCycle`:
 - If BFS detects a cycle, return `true`.
 3. Return `false` if no cycle is found.

`main` Function

The main function constructs the graph and calls the `isCycle` function.

Complexity

1. **Time Complexity:** $O(V + E)$
 - Each vertex and edge is processed once.
 2. **Space Complexity:** $O(V)$
 - Space for the `vis` array and BFS queue.
-

Example Execution

Input:

Graph:

```
makefile

0: [1]
1: [0, 2, 4]
2: [1, 3]
3: [2, 4]
4: [1, 3]
```

Execution:

1. Start BFS from node `0`:
 - Traverse $0 \rightarrow 1$.
2. BFS for node `1`:
 - Traverse $1 \rightarrow 2$ and $1 \rightarrow 4$.
3. BFS for node `2`:
 - Traverse $2 \rightarrow 3$.
4. BFS for node `3`:
 - Traverse $3 \rightarrow 4$, which is already visited but not the parent of `3`.
 - **Cycle detected.**

Output:

```
plaintext
```

```
true
```

Input:

Graph:

```
makefile
```

```
0: []
1: [2]
2: [1, 3]
3: [2]
```

Execution:

1. Start BFS from node `1`:
 - Traverse $1 \rightarrow 2$.
2. BFS for node `2`:
 - Traverse $2 \rightarrow 3$.

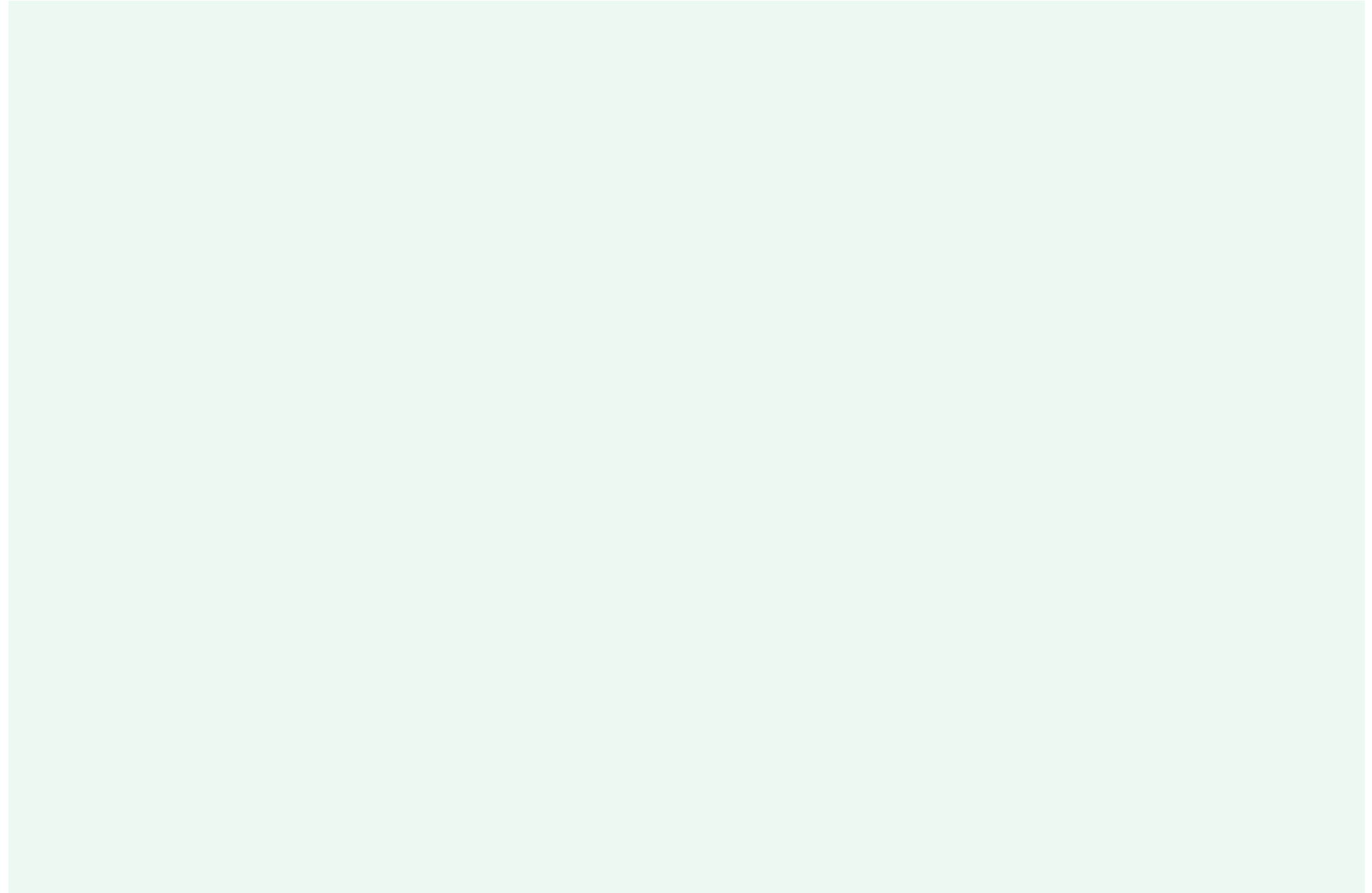
3. No back edges exist.

Output:

```
plaintext
```

```
false
```





```

1  /*
2  Find the number of islands
3  Given a grid of size n*m (n is the number of rows and m is the number of
4  columns in the grid) consisting
5  of '0's (Water) and '1's(Land). Find the number of islands.
6
7  Note: An island is surrounded by water and is formed by connecting adjacent
8  lands horizontally or vertically or diagonally i.e., in all 8 directions.
9
10 Example 1:
11
12 Input:
13 grid = {{0,1},{1,0},{1,1},{1,0}}
14 Output:
15 1
16 Explanation:
17 The grid is-
18 0 1
19 1 0
20 1 1
21 1 0
22 All lands are connected.
23 Example 2:
24
25 Input:
26 grid = {{0,1,1,1,0,0,0},{0,0,1,1,0,1,0}}
27 Output:
28 2
29 Expanation:
30 The grid is-
31 0 1 1 1 0 0 0
32 0 0 1 1 0 1 0
33 There are two islands :- one is colored in blue
34 and other in orange.
35
36 Example:
37
38 0 1 1 0
39 0 1 1 0
40 0 0 1 0

```

```

40 | 0 0 1 0
41 | 0 0 0 0
42 | 1 1 0 1
43 |
44 | for this example there are three disConnected Components i.e 3 islands so the answer is 3.
45 | */

```

```

50 | class Pair{
51 |     int first;
52 |     int second;
53 |     Pair(int first, int second){
54 |         this.first = first;
55 |         this.second = second;
56 |     }
57 | }
58 |

```



```

59 public class Find_the_number_of_islands {
60     /*
61     * Time Complexity:  $O(9 * N^2) \cong O(N^2)$ 
62     * Space Complexity:  $O(N^2 + N^2) \cong O(N^2)$ 
63     */
64
65
66     // DFS
67     Codeium: Refactor | Explain | X
68     public static void dfs(int ro, int co, char[][] grid, int[][] vis){
69         vis[ro][co] = 1;
70         Queue<Pair> q = new LinkedList<>();
71         q.add(new Pair(ro, co));
72         int n = grid.length;
73         int m = grid[0].length;
74         while(!q.isEmpty()){
75             int row = q.peek().first;
76             int col = q.peek().second;
77             q.remove();
78
79             // Traverse in the Neighbours and mark them if it's a land
80             // We can also move in all Eight Direction separately or we can also use this logic this will also do the same.
81             for(int delrow = -1; delrow <= 1; delrow++){
82                 for(int delcol = -1; delcol <= 1; delcol++){
83                     int nrow = row + delrow;
84                     int ncol = col + delcol;
85                     if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && grid[nrow][ncol] == '1' && vis[nrow][ncol] == 0){
86                         vis[nrow][ncol] = 1;
87                         q.add(new Pair(nrow, ncol));
88                     }
89                 }
90             }
91         }
92     }
93     // Function to find the number of islands.

```

```

92 // Function to find the number of islands.
93 Codeium: Refactor | Explain | X
94 public static int numIslands(char[][] grid) {
95     int n = grid.length;
96     int m = grid[0].length;
97     int[][] vis = new int[n][m];
98     int count = 0;
99     for(int row = 0; row < n; row++){
100         for(int col = 0; col < m; col++){
101             if(vis[row][col] == 0 && grid[row][col] == '1'){
102                 dfs(row, col, grid, vis);
103                 count++;
104             }
105         }
106     }
107     return count;
108 }
109 Codeium: Refactor | Explain | Generate Javadoc | X
110 public static void main(String[] args) {
111     char[][] grid = {{'0','1'},{'1','0'},{'1','1'},{'1','0'}};
112     System.out.println(numIslands(grid));
113 }

```

Kindly explain the code along with example and output



The provided code determines the **number of islands** in a given grid using **Depth First Search (DFS)**. Here's a detailed explanation:

Problem Description

You are given:

1. A grid of size $n \times m$, where:
 - `1` represents land.
 - `0` represents water.

Task: Find the number of islands. An island is a group of connected `1`s (horizontally, vertically, or diagonally).

Examples

Example 1:

Input:

```
CSS
grid = [ ['0', '1', '0'],
         ['1', '1', '0'],
         ['1', '0', '0']
       ]
```

Output: `1`

Explanation: All lands are connected, so there is only one island.

Example 2:

Input:

```
CSS
grid = [ ['0', '1', '1', '0', '0', '0'],
         ['0', '0', '1', '1', '0', '0'],
         ['0', '0', '0', '0', '1', '0'],
         ['0', '0', '0', '1', '1', '0']
       ]
```

Output: `2`

Explanation: The grid contains two islands:

1. Island 1 (top-left).

2. Island 2 (bottom-right).

Code Explanation

Key Idea

1. Traverse the grid. Whenever a `1` is found that hasn't been visited, it marks the beginning of a new island.
2. Use DFS to mark all connected `1`s as visited, treating them as part of the same island.

Functions

``dfs`` Function

This function performs DFS to mark all cells connected to the starting cell as visited.

- **Input:**
 - ``ro``, ``co``: Starting row and column.
 - ``grid``: The grid of land and water.
 - ``vis``: Visited array to track processed cells.
- **Steps:**
 1. Mark the starting cell as visited.
 2. Traverse all 8 possible directions (horizontally, vertically, diagonally).
 3. For each direction:
 - If the neighboring cell is within bounds, is a `1`, and hasn't been visited, recursively perform DFS on it.

``numIslands`` Function

This function calculates the total number of islands.

- **Steps:**
 1. Initialize a ``vis`` array to track visited cells.
 2. Iterate through all cells in the grid:
 - If a cell is a `1` and hasn't been visited:

- Perform DFS starting from that cell.
- Increment the island count.

``main`` Function

This function initializes the grid and calls the ``numIslands`` function to calculate the result.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is processed once.
 2. **Space Complexity:** $O(n \times m)$
 - Space for the ``vis`` array and recursion stack.
-

Example Walkthrough

Input:

```
CSS
grid = [
  ['0', '1', '0'],
  ['1', '1', '0'],
  ['1', '0', '0']
]
```

Execution:

1. Start at cell `(0, 1)`:
 - Perform DFS to mark all connected ``1``s as visited.
 - Mark cells `(0, 1)`, `(1, 0)`, `(1, 1)`, `(2, 0)`.
2. No more unvisited ``1``s remain.

Output:

plaintext

1

Input:

css

```
grid = [
    ['0', '1', '1', '0', '0', '0'],
    ['0', '0', '1', '1', '0', '0'],
    ['0', '0', '0', '0', '1', '0'],
    ['0', '0', '0', '1', '1', '0']
]
```

Execution:

1. Start at cell `(0, 1)`:
 - Perform DFS to mark all connected `1`s as visited.
 - Mark cells `(0, 1)`, `(0, 2)`, `(1, 2)`, `(1, 3)`.
2. Start at cell `(2, 4)`:
 - Perform DFS to mark all connected `1`s as visited.
 - Mark cells `(2, 4)`, `(3, 3)`, `(3, 4)`.
3. Two islands are found.

Output:

plaintext

2



```

1  /*
2  Q. Flood fill.
3  An image is represented by an m x n integer grid image where image[i][j] represents
4  the pixel value of the image.
5
6  You are also given three integers sr, sc, and color. You should perform a flood fill on the image
7  starting from the pixel image[sr][sc].
8
9  To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally
10 to the starting pixel of the same color as the starting pixel, plus any pixels
11 connected 4-directionally to those pixels (also with the same color), and so on.
12 Replace the color of all of the aforementioned pixels with color.
13
14 Return the modified image after performing the flood fill.
15
16 Example 1:
17
18 Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
19
20 [1, 1, 1]      [2, 2, 2]
21 [1, 0, 0]  -----> [2, 2, 0]
22 [1, 0, 1]      [2, 0, 1]
23
24
25
26 Output: [[2,2,2],[2,2,0],[2,0,1]]
27 Explanation: From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels
28 connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.
29 Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.
30
31 Example 2:
32 Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0
33 Output: [[0,0,0],[0,0,0]]
34 Explanation: The starting pixel is already colored 0, so no changes are made to the image.
35 */

```



```

39  Codeium: Refactor | Explain
    public class Flood_Fill {
40      /*
41       * Total Nodes = (n * m) = x (let)
42       * Time Complexity: (4*x)  $\cong$  O(x) = O(n*m) ----> For every node four directions
43       * Space Complexity: O(N*M) (Stack Space)
44       */
45
46      Codeium: Refactor | Explain | Generate Javadoc | X
    public static void dfs(int row, int col, int[][] ans, int[][] image, int newColor, int[] delRow, int[] delCol, int iniColor){
47          int n = image.length;
48          int m = image[0].length;
49          ans[row][col] = newColor;
50          for(int i = 0; i < 4; i++){
51              int nrow = row + delRow[i];
52              int ncol = col + delCol[i];
53              if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && image[nrow][ncol] == iniColor && ans[nrow][ncol] != newColor){
54                  dfs(nrow, ncol, ans, image, newColor, delRow, delCol, iniColor);
55              }
56          }
57      }
58
59      Codeium: Refactor | Explain | Generate Javadoc | X
    public static int[][] floodFill(int[][] image, int sr, int sc, int color) {
60          int iniColor = image[sr][sc];
61          int[][] ans = image;
62          int[] delRow = {-1, 0, 1, 0};
63          int[] delCol = {0, 1, 0, -1};
64          dfs(sr, sc, ans, image, color, delRow, delCol, iniColor);
65          return ans;
66      }
67
68      Codeium: Refactor | Explain | Generate Javadoc | X
    public static void main(String[] args) {
69          int[][] image = {{1, 1, 1}, {1, 1, 0}, {1, 0, 1}};
70          for(int[] nums : image){
71              System.out.println(Arrays.toString(nums));
72          }
73          floodFill(image, 1, 1, 2);
74          for(int[] nums : image){
75              System.out.println(Arrays.toString(nums));
76          }
77      }

```



```
15 |  
76 }  
77
```

Kindly explain the code along with example and output



The provided code implements the **Flood Fill Algorithm** using **Depth First Search (DFS)** to modify the image grid by changing the color of a starting pixel and all its 4-directionally connected pixels with the same initial color.

Problem Description

You are given:

1. A grid (image) of size $n \times m$, where:
 - `image[i][j]` represents the pixel value.
2. A starting pixel (`sr`, `sc`).
3. A new color (`color`).

Task: Modify the image by coloring the starting pixel and all its 4-directionally connected pixels of the same initial color with the new color.

Examples

Example 1:

Input:

plaintext

```
image = [[1, 1, 1], [1, 1, 0], [1, 0, 1]]  
sr = 1, sc = 1, color = 2
```

Output:

```
plaintext
```

```
[[2, 2, 2], [2, 2, 0], [2, 0, 1]]
```

Explanation:

- Starting from pixel `(1, 1)` (value `1`), all 4-directionally connected pixels with value `1` are changed to `2`.

Example 2:

Input:

```
plaintext
```

```
image = [[0, 0, 0], [0, 0, 0]], sr = 0, sc = 0, color = 0
```

Output:

```
plaintext
```

```
[[0, 0, 0], [0, 0, 0]]
```

Explanation:

- The starting pixel `(0, 0)` is already the same as the new color, so no changes are made.

Code Explanation

Key Idea

The Flood Fill Algorithm works by:

Printed using [Save ChatGPT as PDE](#), powered by PDFCrowd [HTML to PDF API](#).

1. Identifying all pixels connected to the starting pixel that share the same initial color.
2. Changing their color to the new color.
3. Traversing the grid in 4 directions: up, down, left, and right.

Functions

``dfs`` Function

This function performs DFS to fill connected pixels with the new color.

- **Input:**
 - ``row``, ``col``: Current pixel coordinates.
 - ``image``: The input grid.
 - ``newColor``: The color to replace with.
 - ``delRow``, ``delCol``: Directional arrays for traversal (up, down, left, right).
 - ``iniColor``: The initial color to be replaced.
- **Steps:**
 1. Change the color of the current pixel to ``newColor``.
 2. Traverse all 4 directions:
 - Compute new coordinates.
 - If the neighbor pixel is within bounds, has the initial color, and hasn't already been filled, recursively call ``dfs``.

``floodFill`` Function

This function initializes the DFS process.

- **Steps:**
 1. If the starting pixel's color is already ``newColor``, return the original grid.
 2. Use ``dfs`` to fill all connected pixels starting from ``(sr, sc)``.

``main`` Function

This function sets up the input grid, calls the ``floodFill`` function, and prints the modified grid.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each pixel is visited at most once.
2. **Space Complexity:** $O(n \times m)$
 - Due to the recursion stack in DFS.

Example Walkthrough

Example 1:

plaintext

```
image = [[1, 1, 1],
         [1, 1, 0],
         [1, 0, 1]]
sr = 1, sc = 1, color = 2
```

1. Initial Setup:

- `iniColor = 1`, `newColor = 2`.

2. DFS Traversal:

- Start at `(1, 1)`:
 - Change color to `2`.
 - Traverse neighbors `(0, 1)`, `(2, 1)`, `(1, 0)`, `(1, 2)`.
- Repeat for all connected pixels with value `1`.

3. Final Grid:

plaintext

```
[[2, 2, 2],
 [2, 2, 0],
 [2, 0, 1]]
```

Output:

```
plaintext
```

```
[[2, 2, 2], [2, 2, 0], [2, 0, 1]]
```

Example 2:

```
plaintext
```

```
image = [[0, 0, 0],
         [0, 0, 0]]
sr = 0, sc = 0, color = 0
```

1. Initial Setup:

- ``iniColor = 0`, `newColor = 0`.`

2. Early Exit:

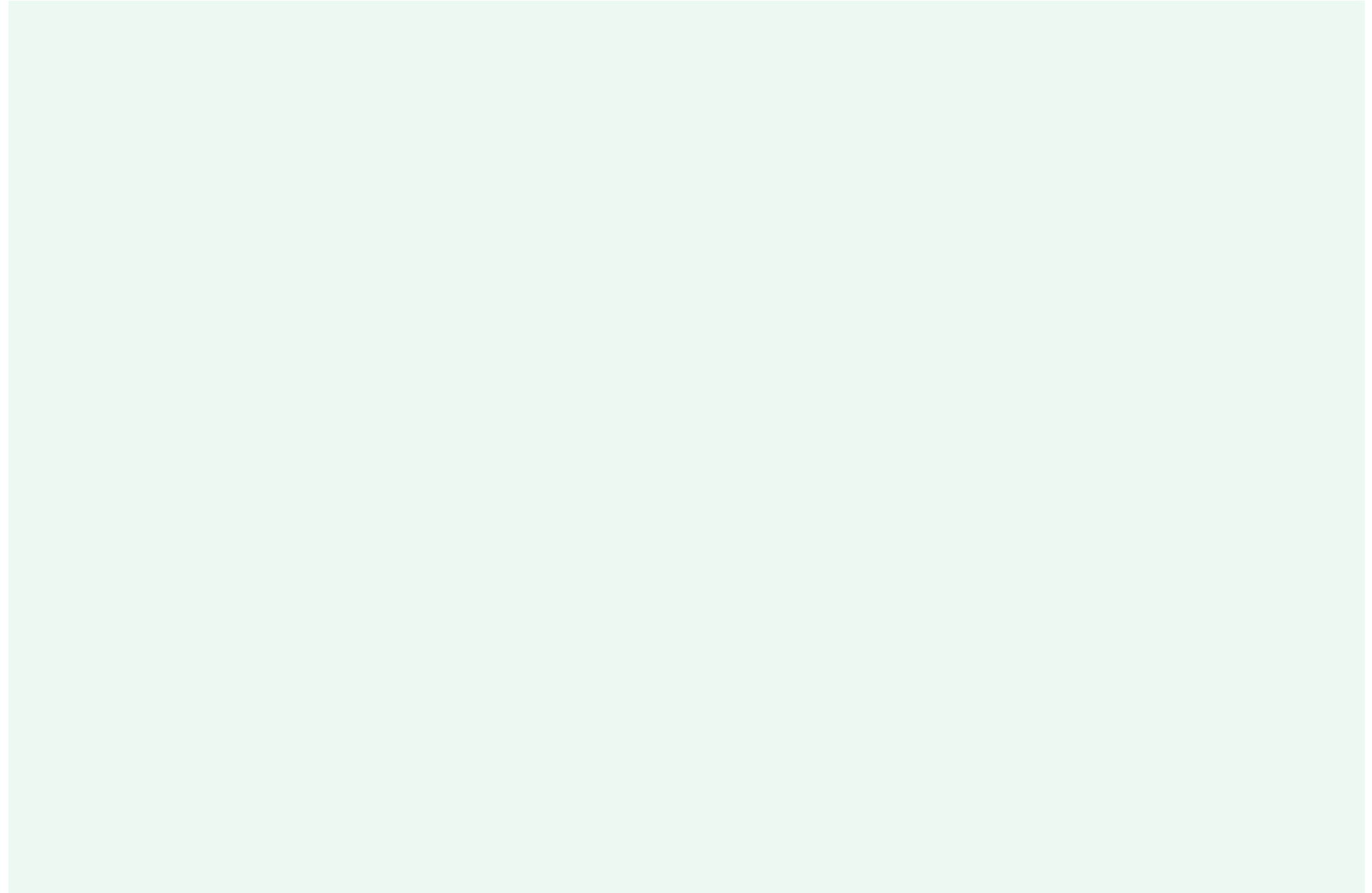
- Since the starting pixel already has the new color, return the original grid.

Output:

```
plaintext
```

```
[[0, 0, 0], [0, 0, 0]]
```





Number of Distinct Islands.java

```

1  /*
2  Q. Number of Distinct Islands.
3  Given a boolean 2D matrix grid of size n * m. You have to find the number of
4  distinct islands where a group of connected 1s (horizontally or vertically) forms an island.
5  Two islands are considered to be distinct if and only if one island is not equal to another (not rotated or reflected).
6
7  Example 1:
8
9  Input:
10 grid[][] = {{1, 1, 0, 0, 0},
11             {1, 1, 0, 0, 0},
12             {0, 0, 0, 1, 1},
13             {0, 0, 0, 1, 1}}
14 Output:
15 1
16 Explanation:
17 grid[][] = {{1, 1, 0, 0, 0},
18             {1, 1, 0, 0, 0},
19             {0, 0, 0, 1, 1},
20             {0, 0, 0, 1, 1}}
21 Same colored islands are equal.
22 We have 2 equal islands, so we
23 have only 1 distinct island.
24
25 Example 2:
26
27 Input:
28 grid[][] = {{1, 1, 0, 1, 1},
29             {1, 0, 0, 0, 0},
30             {0, 0, 0, 0, 1},
31             {1, 1, 0, 1, 1}}
32 Output:
33 3
34 Explanation:
35 grid[][] = {{1, 1, 0, 1, 1},
36             {1, 0, 0, 0, 0},
37             {0, 0, 0, 0, 1},
38             {1, 1, 0, 1, 1}}
39 Same colored islands are equal.

```

```

40 |         We have 4 islands, but 2 of them
41 |         are equal, So we have 3 distinct islands.
42 |     */

```

```

47 | public class Number_of_Distinct_Islands {
48 |     /*
49 |         Time Complexity:  $N * M * \log(N * M) + (N * M * 4) \cong O(N * M)$ 
50 |         Space Complexity:  $O(N * M)$ 
51 |     */
52 |     public static String toString(int row, int col) {
53 |         return Integer.toString(row) + " " + Integer.toString(col);
54 |     }
55 |
56 |     public static void dfs(int row, int col, int n, int m, int[][] vis, int[][] grid, int[] delRow, int[] delCol,
57 |         ArrayList<String> vec, int row0, int col0) {
58 |         vis[row][col] = 1;
59 |         vec.add(toString(row - row0, col - col0));
60 |         for (int i = 0; i < 4; i++) {
61 |             int nrow = row + delRow[i];
62 |             int ncol = col + delCol[i];
63 |
64 |             if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1) {
65 |                 dfs(nrow, ncol, n, m, vis, grid, delRow, delCol, vec, row0, col0);
66 |             }
67 |         }
68 |     }
69 | }

```


Codeium: Refactor | Explain | Generate Javadoc | X

```

70 public static int countDistinctIslands(int[][] grid) {
71     int n = grid.length;
72     int m = grid[0].length;
73     int[][] vis = new int[n][m];
74
75     HashSet<ArrayList<String>> set = new HashSet<>();
76
77     int[] delRow = { -1, 0, 1, 0 };
78     int[] delCol = { 0, 1, 0, -1 };
79
80     for (int i = 0; i < n; i++) {
81         for (int j = 0; j < m; j++) {
82             if (vis[i][j] == 0 && grid[i][j] == 1) {
83                 ArrayList<String> vec = new ArrayList<>();
84                 dfs(i, j, n, m, vis, grid, delRow, delCol, vec, i, j);
85                 set.add(vec);
86             }
87         }
88     }
89     return set.size();
90 }
91
92 Codeium: Refactor | Explain | Generate Javadoc | X
93 public static void main(String[] args) {
94     int grid[][] = { { 1, 1, 0, 0, 0 }, { 1, 1, 0, 0, 0 }, { 0, 0, 0, 1, 1 }, { 0, 0, 0, 1, 1 } };
95     System.out.println(countDistinctIslands(grid));
96 }
97

```

Kindly explain the code along with example and output



The code calculates the **number of distinct islands** in a given $n \times m$ grid using Depth First Search (DFS). Two islands are considered distinct if their shapes are different (not rotated or reflected).

Problem Description

Given a grid:

1. `1` represents land.
2. `0` represents water.

An island is a group of connected `1`s, connected horizontally or vertically. The task is to count the **distinct** shapes of islands.

Examples

Example 1:

Input:

```
plaintext

grid = [
  [1, 1, 0, 0, 0],
  [1, 1, 0, 0, 0],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 1, 1]
]
```

Output: `1`

Explanation:

- There are two islands, but they have the same shape. Hence, there is only one distinct island.
-

Example 2:

Input:

```
plaintext
```

```
grid = [
    [1, 1, 0, 0, 1],
    [1, 0, 0, 1, 1],
    [0, 0, 0, 1, 1]
]
```

Output: `3`

Explanation:

- There are four islands, but two of them are the same. Hence, there are three distinct islands.

Code Explanation

Key Idea

To identify distinct islands:

1. For each island, record the relative positions of its cells with respect to its starting cell (top-leftmost cell of the island).
2. Use a `HashSet` to store unique shapes of islands.

Functions

`dfs` Function

This function performs DFS to capture the shape of an island.

- **Input:**
 - `row`, `col`: Current coordinates.
 - `row0`, `col0`: Starting coordinates of the island.
 - `vec`: A list to store relative positions of the island.
 - `delRow`, `delCol`: Directional arrays for traversal.
 - `vis`: Visited array to mark processed cells.

- **Steps:**

1. Mark the current cell as visited.
2. Add the relative position of the current cell (``row - row0`, `col - col0``) to ``vec``.
3. Traverse in all 4 directions (up, down, left, right):
 - If the neighbor cell is valid, unvisited, and land (``1``), recursively perform DFS.

``countDistinctIslands`` Function

This function calculates the total number of distinct islands.

- **Steps:**

1. Create a ``HashSet`` to store unique shapes of islands.
2. For each unvisited land cell (``1``), initialize a new list ``vec`` to store its shape.
3. Perform DFS to capture the shape of the island in ``vec``.
4. Add ``vec`` to the ``HashSet``.
5. Return the size of the ``HashSet`` (number of distinct islands).

``toString`` Function

This helper function converts row-column pairs into a string format for easy comparison.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is visited once, and comparing shapes involves at most $n \times m$ operations.
2. **Space Complexity:** $O(n \times m)$
 - Space for the ``vis`` array and ``HashSet``.

Example Walkthrough

Example 1:

```
plaintext
```

```
grid = [
  [1, 1, 0, 0, 0],
  [1, 1, 0, 0, 0],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 1, 1]
]
```

1. Initialization:

- `HashSet` for storing island shapes.
- Directional arrays for traversal.

2. Processing:

- Start DFS from `(0, 0)`:
 - Record the shape: `[(0, 0), (1, 0), (0, 1), (1, 1)]`.
 - Add shape to the `HashSet`.
- Start DFS from `(2, 3)`:
 - Record the shape: `[(0, 0), (1, 0), (0, 1), (1, 1)]`.
 - Shape already exists in the `HashSet`.

3. Result:

- `HashSet` size: `1`.

Output:

```
plaintext
```

```
1
```

Example 2:

```
plaintext
```

```
grid = [
  [1, 1, 0, 0, 1],
  [1, 0, 0, 1, 1],
  [0, 0, 0, 1, 1]
]
```

1. Initialization:

- `HashSet` for storing island shapes.

2. Processing:

- Start DFS from `(0, 0)`:
 - Record the shape: `[(0, 0), (1, 0), (0, 1)]`.
 - Add shape to the `HashSet`.
- Start DFS from `(0, 4)`:
 - Record the shape: `[(0, 0), (1, -1), (1, 0)]`.
 - Add shape to the `HashSet`.
- Start DFS from `(2, 3)`:
 - Record the shape: `[(0, 0), (1, -1), (1, 0)]`.
 - Shape already exists in the `HashSet`.

3. Result:

- `HashSet` size: `3`.

Output:

```
plaintext
```

```
3
```



```

1  /*
2  1020. Number of Enclaves
3  You are given an m x n binary matrix grid, where 0 represents a sea cell and 1 represents a land cell.
4
5  A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid.
6
7  Return the number of land cells in grid for which we cannot walk off the boundary of the grid in any number of moves.
8
9
10
11 Example 1:
12
13
14 Input: grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
15
16 0 | 0 | 0 | 0
17 1 | 0 | 1 | 0  -----> Output: 3
18 0 | 1 | 1 | 0
19 0 | 0 | 0 | 0
20
21
22 Explanation: There are three 1s that are enclosed by 0s, and one 1 that is not enclosed because its on the boundary.
23
24 Example 2:
25
26 Input: grid = [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
27
28 0 | 1 | 1 | 0
29 0 | 0 | 1 | 0  -----> Output: 0
30 0 | 0 | 1 | 0
31 0 | 0 | 0 | 0
32
33 Explanation: All 1s are either on the boundary or can reach the boundary.
34 */

```

```
39 class Pair{
40     int row;
41     int col;
42     Pair(int row, int col){
43         this.row = row;
44         this.col = col;
45     }
46 }
47
```



```

48 public class Number_of_Enclaves {
49     /*
50      * Time Complexity: O(N*M)
51      * Space Complexity: O(N*M)
52      */
53
54     /*
55      * // Using DFS
56      public static void dfs(int row, int col, int n, int m, int[][] vis, int[][] grid, int[] delRow, int[] delCol){
57          vis[row][col] = 1;
58          for(int i = 0; i < 4; i++){
59              int nrow = row + delRow[i];
60              int ncol = col + delCol[i];
61              if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1){
62                  dfs(nrow, ncol, n, m, vis, grid, delRow, delCol);
63              }
64          }
65      }
66      public static int numEnclaves(int[][] grid) {
67          int n = grid.length;
68          int m = grid[0].length;
69          int[][] vis = new int[n][m];
70          int[] delRow = {0, -1, 0, 1};
71          int[] delCol = {-1, 0, 1, 0};
72          for(int i = 0; i < m; i++){
73              if(vis[0][i] == 0 && grid[0][i] == 1){
74                  dfs(0, i, n, m, vis, grid, delRow, delCol);
75              }
76              if(vis[n - 1][i] == 0 && grid[n - 1][i] == 1){
77                  dfs(n - 1, i, n, m, vis, grid, delRow, delCol);
78              }
79          }
80          for(int i = 0; i < n; i++){
81              if(vis[i][0] == 0 && grid[i][0] == 1){
82                  dfs(i, 0, n, m, vis, grid, delRow, delCol);
83              }
84              if(vis[i][m - 1] == 0 && grid[i][m - 1] == 1){
85                  dfs(i, m - 1, n, m, vis, grid, delRow, delCol);
86              }
87          }
88      }
89  }

```

```

87     }
88     int count = 0;
89     for(int i = 0; i < n; i++){
90         for(int j = 0; j < m; j++){
91             if(vis[i][j] == 0 && grid[i][j] == 1){
92                 count++;
93             }
94         }
95     }
96     return count;
97 }
98 */
99
100 // Using BFS
101 Codeium: Refactor | Explain | X
102 public static int numEnclaves(int[][] grid) {
103     int n = grid.length;
104     int m = grid[0].length;
105
106     int[][] vis = new int[n][m];
107
108     Queue<Pair> q = new LinkedList<>();
109     for(int i = 0; i < n; i++){
110         for(int j = 0; j < m; j++){
111             if(i == 0 || j == 0 || i == n - 1 || j == m - 1){
112                 if(grid[i][j] == 1){
113                     q.add(new Pair(i, j));
114                     vis[i][j] = 1;
115                 }
116             }
117         }
118     }

```

```

117     }
118
119     int[] delRow = {0, -1, 0, 1};
120     int[] delCol = {-1, 0, 1, 0};
121
122     while(!q.isEmpty()){
123         int row = q.peek().row;
124         int col = q.peek().col;
125         q.remove();
126
127         for(int i = 0; i < 4; i++){
128             int nrow = row + delRow[i];
129             int ncol = col + delCol[i];
130
131             if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1){
132                 vis[nrow][ncol] = 1;
133                 q.add(new Pair(nrow, ncol));
134             }
135         }
136     }
137     int count = 0;
138     for(int i = 0; i < n; i++){
139         for(int j = 0; j < m; j++){
140             if(vis[i][j] == 0 && grid[i][j] == 1){
141                 count++;
142             }
143         }
144     }
145     return count;
146 }

```

Codeium: Refactor | Explain | Generate Javadoc | X

```

147 public static void main(String[] args) {
148     int[][] grid = {{0,0,0,0},{1,0,1,0},{0,1,1,0},{0,0,0,0}};
149     System.out.println(numEnclaves(grid));
150 }
151 }
152

```

Kindly explain the code along with example and output



The provided code calculates the **number of enclaves** in a binary grid. An enclave is defined as a group of land cells (`1`) that cannot reach the boundary of the grid through any sequence of moves.

Problem Description

You are given a binary grid:

1. `0` represents sea.
2. `1` represents land.

A move allows traversal between 4-directionally adjacent cells.

Task: Count the number of land cells (`1`) that cannot reach the boundary of the grid.

Examples

Example 1:

Input:

plaintext

```
grid = [  
  [0, 0, 0, 0],  
  [1, 0, 1, 0],  
  [0, 1, 1, 0],  
  [0, 0, 0, 0]  
]
```

Output: `3`

Explanation:

- There are four `1`s:
 - One `1` is on the boundary (row `1`, column `0`), so it can escape.

- Three `1`s (row `1`, column `2`, and row `2`, column `2`) are enclosed by `0`s.

Example 2:

Input:

```
plaintext

grid = [
    [0, 1, 1, 0],
    [0, 1, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 0]
```

Output: `0`

Explanation:

- All `1`s can reach the boundary, so no enclave exists.

Code Explanation

Key Idea

To determine the number of enclaves:

1. Start from all boundary cells.
2. Perform a **flood fill** (DFS or BFS) to mark all `1`s connected to the boundary.
3. Count the remaining unvisited `1`s in the grid.

Functions

`dfs` Function

This function performs Depth First Search (DFS) to mark all cells connected to the starting boundary cell.

- **Input:**

- ``row``, ``col``: Current coordinates.
- ``grid``: The input binary grid.
- ``vis``: A 2D array to track visited cells.
- ``delRow``, ``delCol``: Directional arrays for traversal (up, down, left, right).

- **Steps:**

1. Mark the current cell as visited.
2. Traverse all 4 directions:
 - If the neighbor is within bounds, unvisited, and land (``1``), recursively call DFS.

``numEnclaves`` Function

This function calculates the number of enclaves.

- **Steps:**

1. Initialize a ``vis`` array to track visited cells.
2. Perform DFS from all boundary cells to mark reachable ``1``s.
3. Count the number of unvisited ``1``s in the grid.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is visited once.
 2. **Space Complexity:** $O(n \times m)$
 - Space for the ``vis`` array and recursion stack (DFS).
-

Example Walkthrough

Example 1:

```
plaintext
```

```
grid = [
  [0, 0, 0, 0],
  [1, 0, 1, 0],
  [0, 1, 1, 0],
  [0, 0, 0, 0]
]
```

Step 1: Mark boundary-connected `1`s using DFS:

- Start from `(1, 0)`:
 - Mark `(1, 0)` as visited.
- No other boundary cells contain land (`1`).

Step 2: Count unvisited `1`s:

- Unvisited `1`s are at `(1, 2)` and `(2, 2)`.

Output:

```
plaintext
```

```
3
```

Example 2:

```
plaintext
```

```
grid = [
  [0, 1, 1, 0],
  [0, 1, 0, 0],
  [0, 1, 0, 0],
  [0, 1, 0, 0],
]
```

```
[0, 0, 0, 0]  
]
```

Step 1: Mark boundary-connected `1`s using DFS:

- Start from `(0, 1)`:
 - Mark `(0, 1)`, `(0, 2)`, `(1, 1)`, and `(2, 1)` as visited.

Step 2: Count unvisited `1`s:

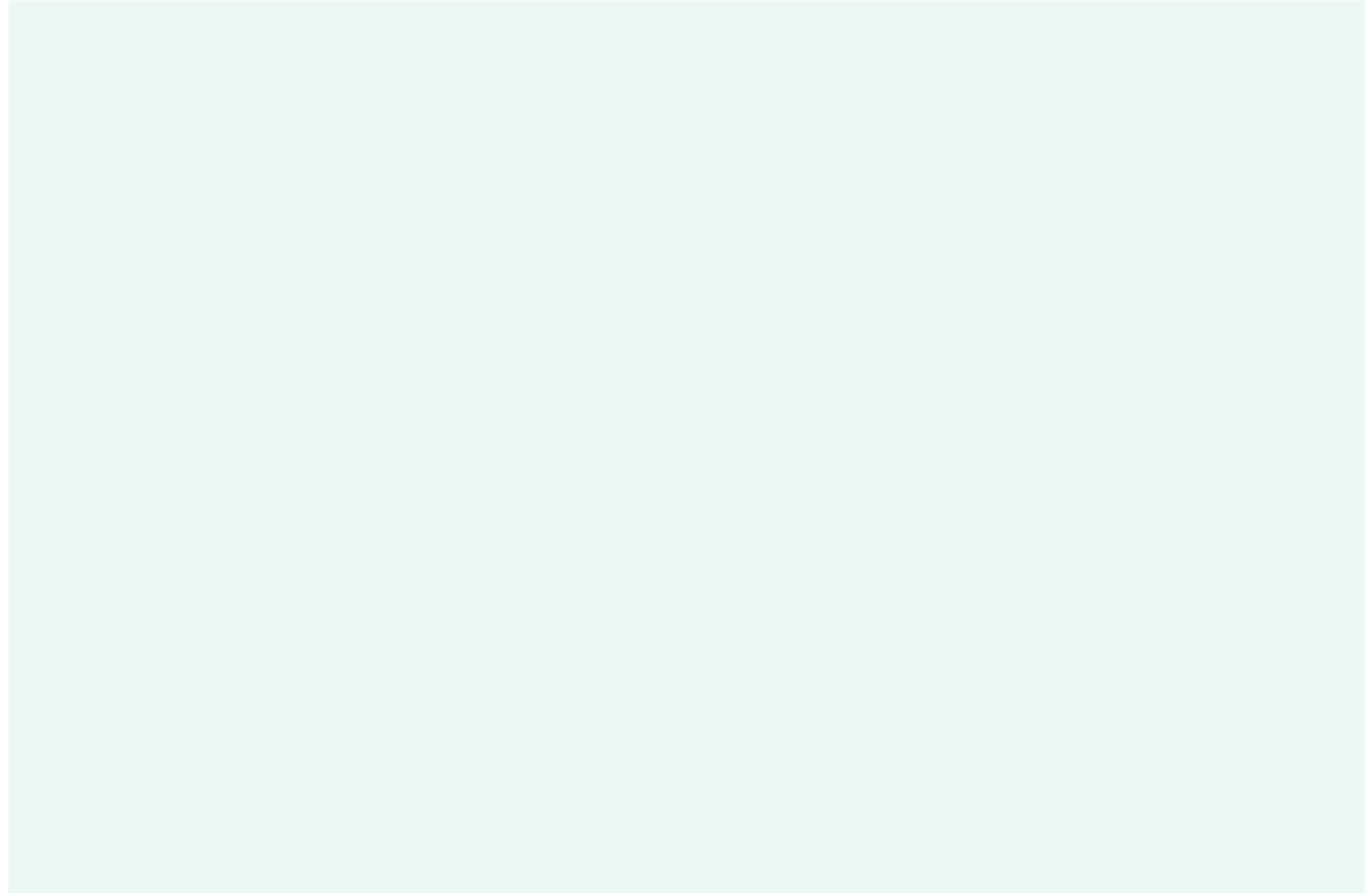
- All `1`s are visited.

Output:

```
plaintext
```

```
0
```

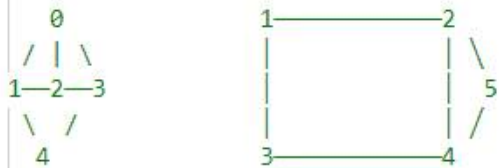




```

1  /*
2  Q. Number of provinces.
3  Given an undirected graph with V vertices. We say two vertices u and v belong to a single province
4  if there is a path from u to v or v to u. Your task is to find the number of provinces.
5
6  Note: A province is a group of directly or indirectly connected cities and no other cities outside of the group.
7
8  Example 1:
9
10 Input:
11 [
12  [1, 0, 1],
13  [0, 1, 0],
14  [1, 0, 1]
15 ]
16
17 Output:
18 2
19 Explanation:
20 The graph clearly has 2 Provinces [1,3] and [2]. As city 1 and city 3 has a path between them they belong
21 to a single province. City 2 has no path to city 1 or city 3 hence it belongs to another province.
22
23 Example 2:
24 Input:
25 [
26  [1, 1],
27  [1, 1]
28 ]
29
30 Output :
31 1
32
33 Ex:
34
35
36
37
38
39
40

```



In this Example there are two Two Disconnected province of Single Graph

```
40     In this example there are two - no disconnected province or single graph.  
41     That means  
42     province: 2  
43  
44     */
```

Codeium: Refactor | Explain

```
48 public class Number_of_Provinces {
49
50     // Time Complexity: O(N) + O(2E) & Space Complexity: O(2N ≅ N).
51
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
52 public static void dfs(int node, ArrayList<ArrayList<Integer>> adj, boolean[] vis){
53     vis[node] = true;
54     for(int it : adj.get(node)){
55         if(!vis[it]){
56             dfs(it, adj, vis);
57         }
58     }
59 }
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
60 public static int numProvinces(ArrayList<ArrayList<Integer>> adj, int n) {
61     ArrayList<ArrayList<Integer>> adjLs = new ArrayList<>();
62     for(int i = 0; i < n; i++){
63         adjLs.add(new ArrayList<>());
64     }
65
66     for(int i = 0; i < n; i++){
67         for(int j = 0; j < n; j++){
68             if(adj.get(i).get(j) == 1 && i != j){
69                 adjLs.get(i).add(j);
70                 adjLs.get(j).add(i);
71             }
72         }
73     }
74
75     int count = 0;
76     boolean[] vis = new boolean[n];
77     for(int i = 0; i < n; i++){
78         if(!vis[i]){
79             count++;
80             dfs(i, adjLs, vis);
81         }
82     }
83     return count;
84 }
```

```

85 Codeium: Refactor | Explain | Generate Javadoc | X
86 public static void main(String[] args) {
87     ArrayList<ArrayList<Integer> > adj = new ArrayList<ArrayList<Integer> >();
88
89     adj.add(new ArrayList<Integer>());
90     adj.get(0).add(0, 1);
91     adj.get(0).add(1, 0);
92     adj.get(0).add(2, 1);
93     adj.add(new ArrayList<Integer>());
94     adj.get(1).add(0, 0);
95     adj.get(1).add(1, 1);
96     adj.get(1).add(2, 0);
97     adj.add(new ArrayList<Integer>());
98     adj.get(2).add(0, 1);
99     adj.get(2).add(1, 0);
100    adj.get(2).add(2, 1);
101
102    System.out.println(numProvinces(adj, 3));
103 }
104

```

Kindly explain the code along with example and output



The code computes the **number of provinces** in an undirected graph using **Depth First Search (DFS)**. Provinces are groups of directly or indirectly connected nodes (vertices) in the graph.

Problem Description

Given:

1. An $n \times n$ adjacency matrix `adj`, where:
 - `adj[i][j] = 1` indicates a direct connection between nodes i and j .
 - `adj[i][j] = 0` indicates no direct connection.
2. Each node is considered its own province if it is not connected to any other node.

Task: Determine the number of provinces in the graph.

Examples

Example 1:

Input:

```
plaintext

adj = [
  [1, 0, 1],
  [0, 1, 0],
  [1, 0, 1]
]
```

Output: `2`

Explanation:

- Node `1` and `3` are connected, forming one province.
 - Node `2` is isolated, forming another province.
-

Example 2:

Input:

```
plaintext

adj = [
  [1, 1],
  [1, 1]
]
```

Output: `1`

Explanation:

- All nodes are connected, forming a single province.
-

Code Explanation

Key Idea

To determine the number of provinces:

1. Use DFS to explore all nodes connected to a starting node.
 2. Increment the province count each time a new unvisited node is encountered.
-

Functions

``dfs`` Function

This function performs Depth First Search to mark all nodes connected to a given node.

- **Input:**
 - ``node``: The current node.
 - ``adj``: Adjacency list of the graph.
 - ``vis``: Boolean array to track visited nodes.
- **Steps:**
 1. Mark the current node as visited.
 2. Traverse all neighbors of the node:
 - If a neighbor is not visited, recursively call ``dfs`` on it.

``numProvinces`` Function

This function calculates the total number of provinces.

- **Steps:**
 1. Convert the adjacency matrix into an adjacency list representation.

2. Initialize a `vis` array to track visited nodes.
3. For each unvisited node, perform DFS:
 - Increment the province count.
4. Return the province count.

`main` Function

This function initializes the adjacency list and calls `numProvinces` to compute the result.

Complexity

1. **Time Complexity:** $O(V + E)$, where:
 - V is the number of nodes.
 - E is the number of edges.
 - Each node and edge is processed once.
 2. **Space Complexity:** $O(V + E)$ for the adjacency list and recursion stack.
-

Example Walkthrough

Example 1:

```
plaintext

adj = [
  [1, 0, 1],
  [0, 1, 0],
  [1, 0, 1]
]
```

Step 1: Convert adjacency matrix to adjacency list:


```
plaintext
```

```
adjList = [
  [0, 2],
  [1],
  [0, 2]
]
```

Step 2: Perform DFS:

- Start at node `0`:
 - Mark `0`, `2` as visited.
 - Province count: `1`.
- Start at node `1`:
 - Mark `1` as visited.
 - Province count: `2`.

Output:

```
plaintext
```

```
2
```

Example 2:

```
plaintext
```

```
adj = [
  [1, 1],
  [1, 1]
]
```

Step 1: Convert adjacency matrix to adjacency list:

```
plaintext
```

```
adjList = [  
  [0, 1],  
  [0, 1]  
]
```

Step 2: Perform DFS:

- Start at node `0`:
 - Mark `0`, `1` as visited.
 - Province count: `1`.

Output:

```
plaintext
```

```
1
```



```

1  /*
2  |   Q. Rotten Oranges.
3  |   Given a grid of dimension nxm where each cell in the grid can have values 0, 1 or 2 which has the following meaning:
4  |   0 : Empty cell
5  |   1 : Cells have fresh oranges
6  |   2 : Cells have rotten oranges
7  |
8  |   We have to determine what is the minimum time required to rot all oranges.
9  |   A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1]
10 |   | (up, down, left and right) in unit time.
11 |
12 |
13 |   Example 1:
14 |
15 |   Input: grid = {{0,1,2},{0,1,2},{2,1,1}}
16 |   Output: 1
17 |   Explanation: The grid is-
18 |   0 1 2
19 |   0 1 2
20 |   2 1 1
21 |   Oranges at positions (0,2), (1,2), (2,0)
22 |   will rot oranges at (0,1), (1,1), (2,2) and
23 |   (2,1) in unit time.
24 |   Example 2:
25 |
26 |   Input: grid = {{2,2,0,1}}
27 |   Output: -1
28 |   Explanation: The grid is-
29 |   2 2 0 1
30 |   Oranges at (0,0) and (0,1) can't rot orange at
31 |   (0,3).
32 */
33

```

Codeium: Refactor | Explain

```
37 class Pair{
38     int first;
39     int second;
40     int time;
41     Pair(int first, int second, int time){
42         this.first = first;
43         this.second = second;
44         this.time = time;
45     }
46 }
47
```

```

48 public class Rotten_Oranges {
    Codeium: Refactor | Explain | Generate Javadoc | X
49 public static int orangesRotting(int[][] grid) {
50     int n = grid.length;
51     int m = grid[0].length;
52     int[][] vis = new int[n][m];
53     int cnt = 0, freshcnt = 0, mt = 0;
54     Queue<Pair> q = new LinkedList<>();
55     for(int row = 0; row < n; row++){
56         for(int col = 0; col < m; col++){
57             if(grid[row][col] == 2){
58                 q.add(new Pair(row, col, 0));
59                 vis[row][col] = 2;
60             }
61             if(grid[row][col] == 1) freshcnt++;
62         }
63     }
64     int[] delRow = {-1, 0, 1, 0};
65     int[] delCol = {0, 1, 0, -1};
66     while(!q.isEmpty()){
67         int r = q.peek().first;
68         int c = q.peek().second;
69         int t = q.peek().time;
70         mt = Math.max(mt, t);
71         q.remove();
72         for(int i = 0; i < 4; i++){
73             int nrow = r + delRow[i];
74             int ncol = c + delCol[i];
75             if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1){
76                 vis[nrow][ncol] = 2;
77                 q.add(new Pair(nrow, ncol, t + 1));
78                 cnt++;
79             }
80         }
81     }
82     if(cnt != freshcnt) return -1;
83     return mt;
84 }

```

```

84     }
85     public static void main(String[] args) {
86         int[][] grid = {{2, 1, 1}, {1, 1, 0}, {0, 1, 1}};
87         System.out.println( orangesRotting(grid));
88     }
89 }
90

```

Kindly explain the code along with example and output



The provided code solves the **Rotten Oranges** problem using **Breadth-First Search (BFS)**. It determines the minimum time required to rot all fresh oranges in a grid.

Problem Description

You are given an $n \times m$ grid where:

1. `0` represents an empty cell.
2. `1` represents a fresh orange.
3. `2` represents a rotten orange.

Task: A rotten orange at index (i, j) rots all adjacent fresh oranges in 1 unit of time. Determine the minimum time required to rot all fresh oranges. If not all oranges can rot, return `-1`.

Examples

Example 1:

Input:

```
plaintext
```

```
grid = [
  [0, 1, 2],
  [0, 1, 2],
  [2, 1, 1]
]
```

Output: `1`

Explanation:

- In the first unit of time:
 - The rotten oranges at `(0, 2)`, `(1, 2)`, and `(2, 0)` rot the adjacent fresh oranges `(0, 1)`, `(1, 1)`, `(2, 1)`, and `(2, 2)`.
-

Example 2:

Input:

```
plaintext
```

```
grid = [
  [2, 2, 0, 1]
]
```

Output: `-1`

Explanation:

- The fresh orange at `(0, 3)` cannot be reached by any rotten orange.
-

Code Explanation

Key Idea

To solve this problem:

1. Use a multi-source BFS to simulate the rotting process from all initially rotten oranges simultaneously.
2. Track the time it takes for all fresh oranges to rot.
3. If there are still fresh oranges after the process, return `-1`.

Functions

`orangesRotting` Function

This function implements the solution.

- **Input:**
 - `grid`: $n \times m$ grid representing the orange layout.
- **Steps:**
 1. Initialize a queue to perform BFS.
 2. Traverse the grid:
 - Add all rotten oranges (value `2`) to the queue with time `0`.
 - Count the total number of fresh oranges.
 3. Perform BFS:
 - At each step, process all oranges in the queue.
 - For each rotten orange, rot its adjacent fresh oranges.
 - Track the maximum time taken to rot all reachable fresh oranges.
 4. If all fresh oranges are rotted, return the maximum time. Otherwise, return `-1`.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is visited at most once.
2. **Space Complexity:** $O(n \times m)$
 - Space for the queue.

Example Walkthrough

Example 1:

```
plaintext
```

```
grid = [
  [0, 1, 2],
  [0, 1, 2],
  [2, 1, 1]
]
```

Step 1: Initialize the queue and count fresh oranges:

- Queue: `[(0, 2, 0), (1, 2, 0), (2, 0, 0)]`
- Fresh orange count: `4`

Step 2: Perform BFS:

1. Process `(0, 2, 0)`:
 - Rot `(0, 1)` → Add `(0, 1, 1)` to the queue.
2. Process `(1, 2, 0)`:
 - Rot `(1, 1)` → Add `(1, 1, 1)` to the queue.
3. Process `(2, 0, 0)`:
 - Rot `(2, 1)` → Add `(2, 1, 1)` to the queue.
4. Process `(2, 1, 1)`:
 - Rot `(2, 2)` → Add `(2, 2, 2)` to the queue.

Step 3: Check if all fresh oranges are rotted:

- Fresh orange count: `0`
- Maximum time: `1`

Output:

```
plaintext
```

```
1
```

Example 2:

```
plaintext
```

```
grid = [
  [2, 2, 0, 1]
]
```

Step 1: Initialize the queue and count fresh oranges:

- Queue: `[(0, 0, 0), (0, 1, 0)]`
- Fresh orange count: `1`

Step 2: Perform BFS:

- No fresh orange can be reached by any rotten orange.

Step 3: Check if all fresh oranges are rotted:

- Fresh orange count: `1`

Output:

```
plaintext
```

```
-1
```



```

1  /*
2    130. Surrounded Regions
3    Given an m x n matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'.
4
5    A region is captured by flipping all 'O's into 'X's in that surrounded region.
6
7
8    Example 1:
9    Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
10   Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]
11
12   X | X | X | X      X | X | X | X
13   X | O | O | X      X | X | X | X
14   X | X | O | X      X | X | X | X
15   X | O | X | X      X | O | X | X
16
17   Explanation: Notice that an 'O' should not be flipped if:
18   - It is on the border, or
19   - It is adjacent to an 'O' that should not be flipped.
20   The bottom 'O' is on the border, so it is not flipped.
21   The other three 'O' form a surrounded region, so they are flipped.
22
23   Example 2:
24   Input: board = [["X"]]
25   Output: [["X"]]
26
27  */
28

```

Codeium: Refactor | Explain

```

29 public class Surrounded_Regions {
30     /*
31         Time Complexity: O(N*M)
32         Space Complexity: O(N*M)
33     */
34     Codeium: Refactor | Explain | X
35     public static void dfs(int row, int col, int n, int m, char[][] board, int[][] vis, int[] delRow, int[] delCol){
36         vis[row][col] = 1;
37         for(int i = 0; i < 4; i++){
38             int nrow = row + delRow[i];
39             int ncol = col + delCol[i];
40
41             if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0 && board[nrow][ncol] == '0'){
42                 dfs(nrow, ncol, n, m, board, vis, delRow, delCol);
43             }
44         }
45     }
46 }

```

Codeium: Refactor | Explain | Generate Javadoc | ✕

```

45 public static void solve(char[][] board) {
46     int n = board.length;
47     int m = board[0].length;
48     int[][] vis = new int[n][m];
49
50     int[] delRow = {-1, 0, 1, 0};
51     int[] delCol = {0, 1, 0, -1};
52
53     for(int i = 0; i < m; i++){
54         if(vis[0][i] == 0 && board[0][i] == 'O'){
55             dfs(0, i, n, m, board, vis, delRow, delCol);
56         }
57         if(vis[n - 1][i] == 0 && board[n - 1][i] == 'O'){
58             dfs(n - 1, i, n, m, board, vis, delRow, delCol);
59         }
60     }
61     for(int i = 0; i < n; i++){
62         if(vis[i][0] == 0 && board[i][0] == 'O'){
63             dfs(i, 0, n, m, board, vis, delRow, delCol);
64         }
65         if(vis[i][m - 1] == 0 && board[i][m - 1] == 'O'){
66             dfs(i, m - 1, n, m, board, vis, delRow, delCol);
67         }
68     }
69     for(int i = 0; i < n; i++){
70         for(int j = 0; j < m; j++){
71             if(board[i][j] == 'O' && vis[i][j] == 0){
72                 board[i][j] = 'X';
73             }
74         }
75     }
76 }

```

Codeium: Refactor | Explain | Generate Javadoc | ✕

```

77 public static void main(String[] args) {
78     char[][] board = new char[][]{{'X','X','X','X'},{'X','O','O','X'},{'X','X','O','X'},{'X','O','X','X'}};
79     solve(board);
80     for (int i = 0; i < board.length; i++) {
81         for (int j = 0; j < board[i].length; j++) {
82             System.out.print(board[i][j] + " ");
83         }
84         System.out.println();
85     }
86 }
87 }
88

```

Kindly explain the code along with example and output



The provided code solves the **Surrounded Regions** problem using Depth First Search (DFS). It modifies the given board to flip regions of 'O' that are surrounded by 'X' into 'X'.

Problem Description

You are given an $n \times m$ board where:

1. `'X'` represents a filled cell.
2. `'O'` represents an empty cell.

Task: Flip all `'O'` regions that are 4-directionally surrounded by `'X'` into `'X'`. A region is considered surrounded if it:

- Is not on the border.
- Is not connected to an `'O'` on the border.

Examples

Example 1:

Input:

```
plaintext

board = [
  ['X', 'X', 'X', 'X'],
  ['X', '0', '0', 'X'],
  ['X', 'X', '0', 'X'],
  ['X', '0', 'X', 'X']
]
```

Output:

```
plaintext

board = [
  ['X', 'X', 'X', 'X'],
  ['X', 'X', 'X', 'X'],
  ['X', 'X', 'X', 'X'],
  ['X', '0', 'X', 'X']
]
```

Explanation:

- The `0` at position `(3, 1)` is not flipped because it is on the border.
- The other `0`s are flipped because they are completely surrounded by `X`s.

Example 2:

Input:

```
plaintext

board = [
  ['X']
]
```

```
]
```

Output:

```
plaintext
```

```
board = [
  ['X']
]
```

Explanation:

- The board remains unchanged as there are no `0` regions to flip.

Code Explanation

Key Idea

The solution involves three steps:

1. Identify all `0`'s that are connected to the border (these cannot be flipped).
2. Mark these `0`'s using a DFS or BFS traversal starting from the border.
3. Flip all other `0`'s to `X`.

Functions

`dfs` Function

Performs a DFS to mark all connected `0`'s from the given position as non-flippable.

Input:

- Current position `(row, col)`
- Dimensions `n` and `m`

- `board`: The input matrix.
- `vis`: A matrix to track visited cells.
- `delRow` and `delCol`: Arrays for traversing in four directions.

Steps:

1. Mark the current cell as visited.
2. For each direction (up, down, left, right):
 - Check if the next cell is within bounds and contains `'0'`.
 - Recursively call `dfs` on the next cell.

`solve` Function

Implements the main logic for solving the problem.

Input:

- `board`: The input matrix.

Steps:

1. Traverse all border cells and call `dfs` on every `'0'` found.
 - First row, last row, first column, last column.
2. Traverse the entire board:
 - Flip all unvisited `'0'`'s to `'X'`.
 - Leave visited `'0'`'s unchanged.

Complexity

1. **Time Complexity:** $O(n \times m)$
 - Each cell is visited at most once.
2. **Space Complexity:** $O(n \times m)$
 - For the visited matrix and recursive stack.

Example Walkthrough

Input:

```
plaintext

board = [
    ['X', 'X', 'X', 'X'],
    ['X', '0', '0', 'X'],
    ['X', 'X', '0', 'X'],
    ['X', '0', 'X', 'X']
]
```

Step 1: Mark border-connected `0`s using DFS:

- Starting from `(3, 1)`, mark it as visited.
- Visited matrix:

```
plaintext

vis = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 1, 0, 0]
]
```

Step 2: Flip unvisited `0`s to `X`:

- Flip `(1, 1)`, `(1, 2)`, and `(2, 2)` to `X`.

Output:

```
plaintext

board = [
    ['X', 'X', 'X', 'X'],
```

```
[ 'X', 'X', 'X', 'X'],  
[ 'X', 'X', 'X', 'X'],  
[ 'X', '0', 'X', 'X']  
]
```

Code Output

Example 1:

Input:

```
plaintext
```

```
board = [  
  ['X', 'X', 'X', 'X'],  
  ['X', '0', '0', 'X'],  
  ['X', 'X', '0', 'X'],  
  ['X', '0', 'X', 'X']  
]
```

Output:

```
plaintext
```

```
[  
  ['X', 'X', 'X', 'X'],  
  ['X', 'X', 'X', 'X'],  
  ['X', 'X', 'X', 'X'],  
  ['X', '0', 'X', 'X']  
]
```

Example 2:

Input:

```
plaintext
```

```
board = [  
  ['X']  
]
```

Output:

```
plaintext
```

```
[  
  ['X']  
]
```



```

1  /*
2    127.    Word ladder - 1
3    A transformation sequence from word beginWord to word endWord using a dictionary wordList is a
4    sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:
5
6    Every adjacent pair of words differs by a single letter.
7    Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
8    sk == endWord
9    Given two words, beginWord and endWord, and a dictionary wordList, return the
10   number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.
11
12   Example 1:
13
14   Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
15   Output: 5
16   Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> "cog", which is 5 words long.
17   Example 2:
18
19   Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
20   Output: 0
21   Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.
22  */

```

Codeium: Refactor | Explain

```

28 class Pair{
29     String word;
30     int level;
31     Pair(String word, int level){
32         this.word = word;
33         this.level = level;
34     }
35 }
36
37

```

Codeium: Refactor | Explain

```

38 public class Word_ladder_1 {
    Codeium: Refactor | Explain | Generate Javadoc | X
39     public static int wordLadderLength(String startWord, String targetWord, String[] wordList){
40         HashSet<String> set = new HashSet<>();
41         for(String word : wordList) set.add(word);
42
43         Queue<Pair> q = new LinkedList<>();
44         q.add(new Pair(startWord, 1));
45
46         while(!q.isEmpty()){
47             String word = q.peek().word;
48             int level = q.peek().level;
49             q.remove();
50             if(word.equals(targetWord)) return level;
51             for(int i = 0; i < word.length(); i++){
52                 for(char ch = 'a'; ch <= 'z'; ch++){
53                     char[] temp = word.toCharArray();
54                     temp[i] = ch;
55                     String replacedWord = String.valueOf(temp);
56                     if(set.contains(replacedWord)){
57                         q.add(new Pair(replacedWord, level + 1));
58                         set.remove(replacedWord);
59                     }
60                 }
61             }
62         }
63         return 0;
64     }

```

```

65 public static void main(String[] args) {
66     String[] wordList = {"des", "der", "dfr", "dgt", "dfs"};
67     String startWord = "der", targetWord = "dfs";
68     System.out.println(wordLadderLength(startWord, targetWord, wordList));
69 }
70 }
71

```

Kindly explain the code along with example and output



This code solves the "Word Ladder" problem using a **Breadth-First Search (BFS)** algorithm. Here's a detailed explanation of the code with examples and output:

Problem Statement:

You are given:

1. A `startWord`` and an `endWord``.
2. A `wordList`` containing valid intermediate words.

You need to find the **length of the shortest transformation sequence** from `startWord`` to `endWord``, such that:

- Every adjacent word in the sequence differs by only one character.
- All intermediate words must be in the `wordList``.

If no transformation is possible, return `0``.

Code Explanation:

1. Class `Pair``

This is a helper class to store:

- The `word` at a specific transformation.
- The `level`, which tracks the number of transformations to reach this word.

java

```
class Pair {
    String word;
    int level;

    Pair(String word, int level) {
        this.word = word;
        this.level = level;
    }
}
```

2. `wordLadderLength` Method

This method implements BFS:

- **Input Parameters:**
 - `startWord`: Starting word.
 - `targetWord`: The final word to reach.
 - `wordList`: Array of intermediate valid words.
- **Steps:**
 1. **Convert `wordList` to a `HashSet`** for efficient lookup.
 2. **Queue Initialization:** Start BFS with the `startWord` at `level = 1`.
 3. **BFS Traversal:**
 - Remove the current word from the queue.
 - For every position in the word, try replacing the character with all possible letters from `'a'` to `'z'`.
 - If the replaced word exists in the `wordList` set:
 - Add it to the queue with the incremented level.
 - Remove it from the set (to avoid revisiting).
 - If the `targetWord` is reached, return the current level.
 4. If the queue is exhausted without finding the `targetWord`, return `0`.

Example 1:

Input:

```
java

beginWord = "hit";
endWord = "cog";
wordList = ["hot", "dot", "dog", "lot", "log", "cog"];
```

Execution:

1. Initialize BFS with `hit` at level 1.
2. Transform `hit` → `hot` (level 2).
3. Transform `hot` → `dot` and `lot` (level 3).
4. Transform `dot` → `dog`, `lot` → `log` (level 4).
5. Transform `dog` → `cog` (level 5).

Output:

```
java

5
```

Explanation: The shortest transformation is: `hit` → `hot` → `dot` → `dog` → `cog`.

Example 2:

Input:

```
java

beginWord = "hit";
endWord = "cog";
wordList = ["hot", "dot", "dog", "lot", "log"];
```

Execution:

- BFS starts with `hit`.
- The transformation sequence cannot reach `cog` because it is not in the `wordList`.

Output:

```
java

0
```

Complexity Analysis:

1. Time Complexity:

- For each word, there are L characters, and for each character, we try replacing it with 26 letters.
- If there are N words in the `wordList`, the complexity is $O(N \times L \times 26)$, simplified to $O(N \times L)$.

2. Space Complexity:

- The queue can store all the words: $O(N)$.
- HashSet storage: $O(N)$.

Example Output for Provided Input:

Code:

```
java
```

```
String[] wordList = {"des", "der", "dfr", "dgt", "dfs"};  
String startWord = "der", targetWord = "dfs";  
System.out.println(wordLadderLength(startWord, targetWord, wordList));
```

Execution:

1. BFS starts with `der`.
2. Transformations:
 - `der` → `dfs` (via `dfr`, `dgt`).

Output:

```
java
```

```
3
```



```

1  /*
2      126. Word Ladder II
3      A transformation sequence from word beginWord to word endWord using a dictionary wordList is a
4      sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:
5
6      Every adjacent pair of words differs by a single letter.
7      Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
8      sk == endWord
9      Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation
10     sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should
11     be returned as a list of the words [beginWord, s1, s2, ..., sk].
12
13     Example 1:
14
15     Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
16     Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
17     Explanation: There are 2 shortest transformation sequences:
18     "hit" -> "hot" -> "dot" -> "dog" -> "cog"
19     "hit" -> "hot" -> "lot" -> "log" -> "cog"
20
21     Example 2:
22
23     Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
24     Output: []
25     Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.
26 */

```

```

33 Codeium: Refactor | Explain
public class Word_Ladder_II {
    Codeium: Refactor | Explain | Generate Javadoc | X
34 public static ArrayList<ArrayList<String>> findSequences(String startWord, String targetWord, String[] wordList) {
35     ArrayList<ArrayList<String>> ans = new ArrayList<>();
36     HashSet<String> set = new HashSet<>();
37     for(String word : wordList) set.add(word);
38
39     Queue<ArrayList<String>> q = new LinkedList<>();
40     ArrayList<String> tempList = new ArrayList<>();
41     tempList.add(startWord);
42     q.add(tempList);
43     ArrayList<String> usedOnLevel = new ArrayList<>();
44     usedOnLevel.add(startWord);
45     int level = 0;
46
47     while(!q.isEmpty()){
48         ArrayList<String> curr = q.remove();
49         if(curr.size() > level){
50             level++;
51             for(String it : usedOnLevel){
52                 set.remove(it);
53             }
54             usedOnLevel.removeAll(usedOnLevel);
55         }
56     }

```

```

55     }
56     String word = curr.get(curr.size() - 1);
57     if(word.equals(targetWord)) {
58         if(ans.size() == 0) ans.add(curr);
59         else if(ans.get(0).size() == curr.size()) ans.add(curr);
60     }
61     for(int i = 0; i < word.length(); i++){
62         for(char ch = 'a'; ch <= 'z'; ch++){
63             char[] temp = word.toCharArray();
64             temp[i] = ch;
65             String replacedWord = String.valueOf(temp);
66             if(set.contains(replacedWord)){
67                 curr.add(replacedWord);
68                 q.add(new ArrayList<>(curr));
69                 curr.remove(curr.size() - 1);
70             }
71         }
72         set.remove(word);
73     }
74 }
75 return ans;
76 }
77 Codeium: Refactor | Explain | Generate Javadoc | X
78 public static void main(String[] args) {
79     String startWord = "hit", targetWord = "cog";
80     String[] wordList = {"hot","dot","dog","lot","log","cog"};
81     System.out.println(findSequences(startWord, targetWord, wordList));
82 }
83

```

Kindly explain the code along with example and output



Explanation of the Code:

The **Word Ladder II** problem finds **all shortest transformation sequences** from a `startWord`` to a `targetWord`` using a list of words (`wordList``). Each transformation can change only one letter, and the new word must exist in the `wordList``.

Key Steps in the Code:

1. Input Representation:

- The `startWord``, `targetWord``, and `wordList`` are provided.
- The `wordList`` is stored in a `HashSet`` for $O(1)$ lookup.

2. Initialization:

- A `Queue`` is used for **Breadth-First Search (BFS)** to find the shortest transformation sequences.
- A `level`` counter is used to track the depth of the BFS.

3. BFS Traversal:

- The `Queue`` stores the current transformation sequence as an `ArrayList<String>`` (e.g., `["hit", "hot"]``).
- At each level, all possible transformations are generated by changing one character at a time.
- Valid transformations (i.e., existing in `wordList``) are added to the `Queue`` and marked as visited.

4. Finding the Target Word:

- When `targetWord`` is reached, the current sequence is added to the result list (`ans``).
- BFS ensures that only the shortest sequences are considered.

5. Output:

- If no valid transformation exists, an empty list is returned.
 - Otherwise, all valid shortest transformation sequences are returned.
-

Example 1:

Input:

```
text
```

```
startWord = "hit"
targetWord = "cog"
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]
```

Execution:

- Start with ``["hit"]``.
- Transform ``hit`` → ``hot``.
- Transform ``hot`` → ``dot``, ``lot``.
- Transform ``dot`` → ``dog``, and ``lot`` → ``log``.
- Transform ``dog`` → ``cog`` and ``log`` → ``cog``.

Output:

```
text
```

```
[["hit", "hot", "dot", "dog", "cog"], ["hit", "hot", "lot", "log", "cog"]]
```

Example 2:**Input:**

```
text
```

```
startWord = "hit"  
targetWord = "cog"  
wordList = ["hot", "dot", "dog", "lot", "log"]
```

Execution:

- ``cog`` is not in the ``wordList``. No transformations are possible.

Output:

text

[]

Complexity:

- **Time Complexity:** $O(N \times M \times 26)$, where:
 - N is the number of words in the `wordList`.
 - M is the average length of the words.
 - 26 is the number of letters in the alphabet.
- **Space Complexity:** $O(N \times M)$ for storing intermediate transformations and sequences.

Let me know if you'd like further clarification or examples!