Experiment – 9

1.Number of Islands

```
class Solution {
public:
    void dfs(vector<vector<char>>& grid, int i, int j) {
        int m = grid.size(), n = grid[0].size();
        if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == '0') return;
        grid[i][j] = '0';
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty()) return 0;
        int count = 0, m = grid.size(), n = grid[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '1') {
                    dfs(grid, i, j);
                    ++count;
                }
            }
        }
        return count;
    }
};
```

2. Word Ladder

```
class Solution {
public:
   int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
   if (!wordSet.count(endWord)) return 0;
   queue<pair<string, int>> q;
   q.push({beginWord, 1});
   while (!q.empty()) {
      auto [word, length] = q.front(); q.pop();
      if (word == endWord) return length;
      for (int i = 0; i < word.size(); ++i) {
         string temp = word;
         for (char c = 'a'; c <= 'z'; ++c) {
            temp[i] = c;
            if (wordSet.count(temp)) {
               q.push({temp, length + 1});
               wordSet.erase(temp); // mark as visited
            }
         }
      }
   }
   return 0;
}
};
```

3 . **Surrounded Regions**

```
class Solution {
  public:
 void solve(vector<vector<char>>& board) {
    int m = board.size();
    if (m == 0) return;
    int n = board[0].size();
    function<void(int, int)> dfs = [&](int i, int j) {
       if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] != 'O') return;
       board[i][j] = '#';
       dfs(i + 1, j);
      dfs(i - 1, j);
       dfs(i, j + 1);
       dfs(i, j - 1);   };
    for (int i = 0; i < m; ++i) {
       dfs(i, 0);
       dfs(i, n - 1);
    }
    for (int j = 0; j < n; ++j) {
       dfs(0, j);
       dfs(m - 1, j);
    }
    for (int i = 0; i < m; ++i) {
       for (int j = 0; j < n; ++j) {
          if (board[i][j] == 'O') board[i][j] = 'X';
          if (board[i][j] == '#') board[i][j] = 'O';
       }
    } }
};
```

4. Binary Tree Maximum Path sum

```
class Solution {
   int maxSum = INT_MIN;
   int maxGain(TreeNode* node) {
      if (!node) return 0;
      int leftGain = max(maxGain(node->left), 0);
      int rightGain = max(maxGain(node->right), 0);
      int currentPath = node->val + leftGain + rightGain;
      maxSum = max(maxSum, currentPath);
      return node->val + max(leftGain, rightGain);
   }
public:
   int maxPathSum(TreeNode* root) {
      maxGain(root);
      return maxSum;
   }
};
```

5. Friend Circles

```
class Solution {
public:
   void dfs(int city, vector<vector<int>>& isConnected, vector<bool>& visited) {
      visited[city] = true;
      for (int i = 0; i < isConnected.size(); ++i) {
         if (isConnected[city][i] == 1 && !visited[i]) {
            dfs(i, isConnected, visited);
         }
      }
   }
   int findCircleNum(vector<vector<int>>& isConnected) {
```

```
        int n = isConnected.size();
        vector<bool> visited(n, false);
        int provinces = 0;
        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                dfs(i, isConnected, visited);
                provinces++;
            }
        }
        return provinces;
    }
};
```

6. Lowest Common Ancestor of a Binary Tree

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || root == p || root == q) return root;

        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        if (left && right) return root;
        return left ? left : right;
    }
};
```

7. Course Schedule

```
class Solution {
public:
```

```
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> graph(numCourses);
        vector<int> visited(numCourses, 0);  // 0: unvisited, 1: visiting, 2: visited

        // Build graph
        for (auto& p : prerequisites) {
            graph[p[1]].push_back(p[0]);
        }

        // DFS to detect cycle
        function<bool(int)> dfs = [&](int course) {
            if (visited[course] == 1) return false;
            if (visited[course] == 2) return true;

            visited[course] = 1;
            for (int next : graph[course]) {
                if (!dfs(next)) return false;
            }
            visited[course] = 2;
            return true;
        };
        for (int i = 0; i < numCourses; ++i) {
            if (visited[i] == 0 && !dfs(i)) {
                return false;
            }
        }
        return true;
    }
};
```

8. Longest Increasing Path in a Matrix

```
class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        if (matrix.empty()) return 0;

        int m = matrix.size(), n = matrix[0].size();
        vector<vector<int>> memo(m, vector<int>(n, 0));
        int longestPath = 0;

        // Directions for up, down, left, right
        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        // Helper function for DFS + memoization
        function<int(int, int)> dfs = [&](int i, int j) {
            if (memo[i][j] != 0) return memo[i][j]; // Return cached result

            int maxPath = 1; // Minimum path length is 1 (the current cell)

            // Explore four directions
            for (auto& dir : directions) {
                int x = i + dir.first, y = j + dir.second;
                if (x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j]) {
                    maxPath = max(maxPath, 1 + dfs(x, y));
                }
            }

            memo[i][j] = maxPath; // Memoize the result
            return maxPath;
```

```
        };

        // Try starting DFS from each cell
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                longestPath = max(longestPath, dfs(i, j));
            }
        }

        return longestPath;
    }
};
```

9. Course Schedule 2

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        vector<int> result;
        vector<int> inDegree(numCourses, 0);
        vector<vector<int>> graph(numCourses);

        // Build the graph and in-degree array
        for (const auto& prereq : prerequisites) {
            int course = prereq[0];
            int prereqCourse = prereq[1];
            graph[prereqCourse].push_back(course);
            inDegree[course]++;
        }

        // Queue to store courses with no prerequisites (in-degree = 0)
```

```
        queue<int> q;
        for (int i = 0; i < numCourses; ++i) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }

        // Process the courses
        while (!q.empty()) {
            int course = q.front();
            q.pop();
            result.push_back(course);

            // Reduce in-degree of dependent courses
            for (int nextCourse : graph[course]) {
                inDegree[nextCourse]--;
                if (inDegree[nextCourse] == 0) {
                    q.push(nextCourse);
                }
            }
        }
        if (result.size() == numCourses) {
            return result;
        } else {
            return {}; // Cycle detected, return empty array
        }
    }
};
```