1. **What is time complexity and space complexity? Explain them in the pages/word “Assignment 2”. (2 scores)**

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

The time complexity is commonly expressed using big O notation, typically O(n), O(nlogn), O(n2), etc., where n is the input size measured by the number of bits needed for representing it.

Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

1. **What is the time complexity and space complexity of the code you wrote (question 3). Paste your code and explain them for each line in the pages/word “Assignment 2”. (2 scores)**
2. Two Sum

package Assignment2.TwoSum;  
  
import java.util.Hashtable;  
  
public class Solution {  
  
 // Time complexity: O(n^2).  
 // Space complexity: O(1).  
 public int[] twoSum1(int[] nums, int target) {  
  
 int[] index = new int[2];  
  
 // An n loop is used. The time complexity here is O(n).  
 for (int i = 0; i < nums.length - 1; i++) {  
  
 // Another n loop is used. The time complexity here is O(n^2).  
 for (int j = i + 1; j < nums.length; j++) {  
  
 if (target == nums[i] + nums[j]) {  
  
 index[0] = i;  
 index[1] = j;  
 }  
 }  
 }  
 // None extra space used. The space complexity is O(1).  
 return index;  
 }  
  
 // Time complexity: O(n).  
 // Space complexity: O(n).  
 public int[] twoSum2(int[] nums, int target) {  
  
 // The extra space required depends on the number of items stored in the hash table,  
 // which stores n elements, so the space complexity here is O(n).  
 Hashtable hash = new Hashtable();  
 int[] index = new int[2];  
  
 // An n loop is used. The time complexity here is O(n).  
 for (int i = 0; i < nums.length; i++) {  
  
 if (hash.containsKey(nums[i])) {  
  
 index[0] = (int)hash.get(nums[i]);  
 index[1] = i;  
 break;  
 } else {  
  
 hash.put(target - nums[i], i) ;  
 }  
  
 }  
 return index;  
 }  
  
 public static void main(String[] args) {  
  
 int[] nums = {1, 4, 45, 6, 10, -8};  
 int target = 37;  
 Solution n = new Solution();  
// int[] result = n.twoSum1(nums,target);  
 int[] result = n.twoSum2(nums,target);  
 System.*out*.println(**"["** + result[0] + **","** + result[1] + **"]"**);  
 }  
}

1. Number of Islands

package Assignment2.NumberOfIsland;  
  
import java.util.Queue;  
import java.util.LinkedList;  
  
public class Solution {  
  
 public int numberOfIsland(int[][] grid){  
  
 if (grid == null || grid.length == 0 || grid[0].length == 0)  
 return 0;  
  
 boolean[][] visited = new boolean[grid.length][grid[0].length];  
 int count = 0;  
 // An m loop is used. The time complexity here is O(m).  
 for(int i = 0; i < grid.length ; i++) {  
  
 // An n loop is used. The time complexity here is O(m \* n).  
 for(int j =0 ; j < grid[i].length ; j++) {  
  
 if (!visited[i][j] && grid[i][j] == 1) {  
  
// dfs(grid,visited,i,j);  
 bfs(grid,visited,i,j);  
 count++;  
 }  
 }  
 }  
 return count;  
 }  
  
 // The worst space complexity is O(m \* n),  
 // in case that the grid map is filled with lands where dfs goes by m \* n deep.  
 public void dfs(int[][] grid, boolean[][] visited, int i, int j){  
  
 if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] == 0 || visited[i][j])  
 return;  
  
 visited[i][j] = true;  
  
 dfs(grid, visited, i + 1, j);  
 dfs(grid, visited, i - 1, j);  
 dfs(grid, visited, i, j + 1);  
 dfs(grid, visited, i, j - 1);  
  
 }  
  
 class Point {  
  
 int i, j;  
 public Point(int i, int j) {  
  
 this.i = i;  
 this.j = j;  
 }  
 }  
  
 // The worst space complexity is O(m \* n),  
 // because in worst case where the grid is filled with lands, the size of queue can grow up to m \* n.  
 public void bfs(int[][] grid, boolean[][] visited, int i, int j) {  
 Queue<Point> queue = new LinkedList();  
 queue.add(new Point(i, j));  
 while (!queue.isEmpty()) {  
  
 Point p = queue.remove();  
 visited[p.i][p.j] = true;  
  
 if(p.i > 0 && grid[p.i - 1][p.j] == 1 && !visited[p.i - 1][p.j])  
 queue.add(new Point(p.i - 1, p.j));  
  
 if(p.j < visited[0].length - 1 && grid[p.i][p.j + 1] == 1 && !visited[p.i][p.j + 1])  
 queue.add(new Point(p.i, p.j + 1));  
  
 if(p.i < visited.length - 1 && grid[p.i + 1][p.j] == 1 && !visited[p.i + 1][p.j])  
 queue.add(new Point(p.i + 1, p.j));  
  
 if(p.j > 0 && grid[p.i][p.j - 1] == 1 && !visited[p.i][p.j - 1])  
 queue.add(new Point(p.i, p.j - 1));  
 }  
 }  
  
  
  
 public static void main(String args[]){  
  
 int matrix[][] = {  
 {1,0,0,1,0},  
 {1,0,1,1,1},  
 {0,1,0,1,0},  
 {1,0,0,0,1},  
 {1,0,0,1,0}  
 };  
 Solution island = new Solution();  
 int count = island.numberOfIsland(matrix);  
 System.*out*.println(count);  
 }  
}