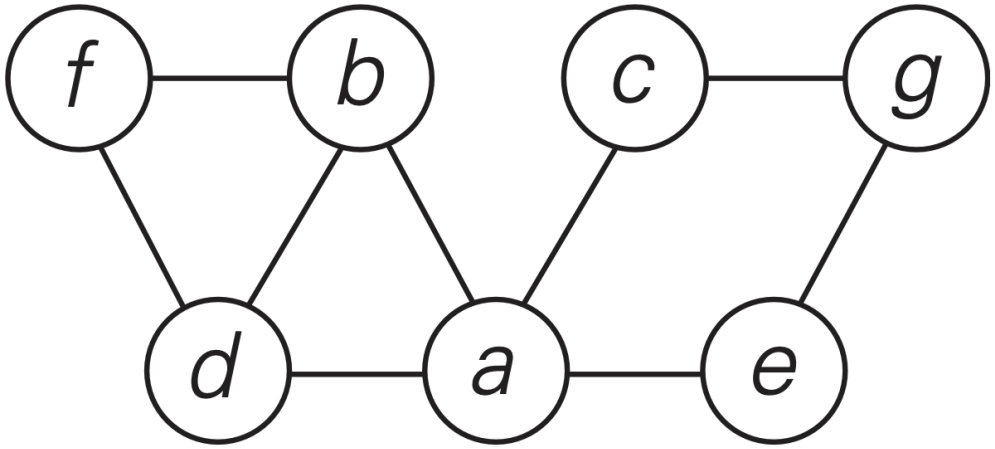
CS427 Fall Assignment #5
Student name: _____

Rules:

- Each pseudocode must specify its input and output.
- Necessary comments should be included.
- When analyzing time/space complexity, specific reasons must be provided.
- For proof questions, a specific derivation process is required. Please refer to the example in the slide.

1. Consider the following graph. (20pts)



a. Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.) (10pts)

->	A	B	C	D	E	F	G
A	-	1	1	1	1	0	0
B	1	-	0	1	0	1	0
C	1	0	-	0	0	0	1
D	1	1	0	-	0	1	0
E	1	0	0	0	-	0	1
F	0	1	0	1	0	-	0
G	0	0	1	0	1	0	-

A	B,C,D,E
B	A,D,F
C	A,G
D	A,B,F
E	A,G
F	B,D

b. Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack). (10pts)

Last Pushed

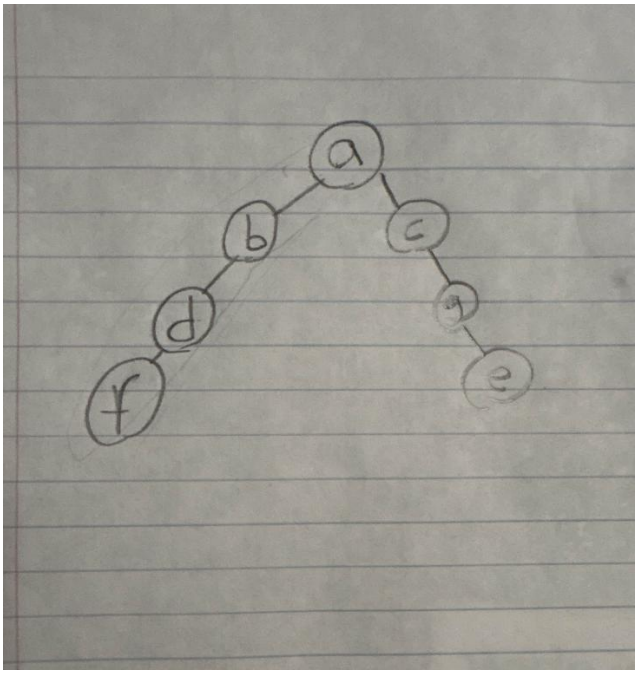
E
G
C
F
D
B
A

First Pushed

First Popped

F
D
B
E
G
C
A

Last Popped



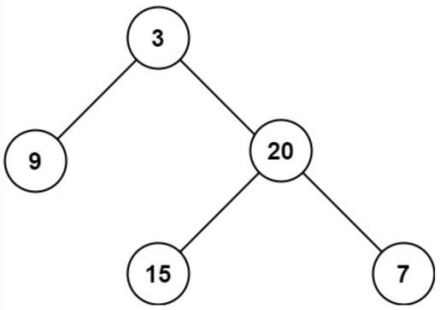
2 (20pts). Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Please provide your algorithm with input/output and time/space analysis.

(Pseudocode will be fine).

Example 1:



Input: root = [3,9,20,null,null,15, 7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

Constraints:

The number of nodes in the tree is in the range [0, 104].

-100 <= Node.val <= 100

```
class Solution {
   // Input: the root of a binary tree
   // Output: The maximum depth of the binary tree
    public int maxDepth(TreeNode root) {
        // Base case: empty tree
       if (root == null) {
           return 0;
       }

       // If only one child is null, take the other child
       if(root.left == null) {
           return 1 + maxDepth(root.right);
       }

        if (root.right == null) {
           return 1 + maxDepth(root.left);
       }

       // If both children exist, take the max of the two depths
       return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
```

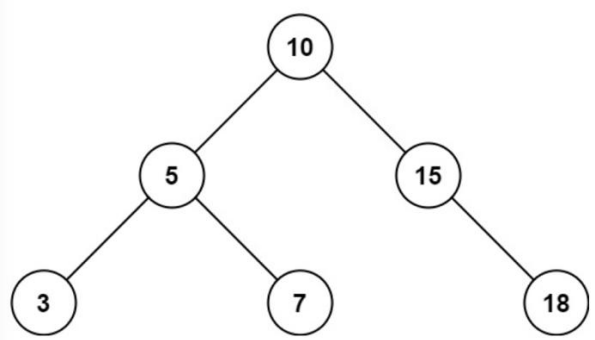
// Time Complexity - O(2^n): Explores both the left and right subtrees

// Space Complexity – O(2^n): At worst creates 2^n stack frames

}

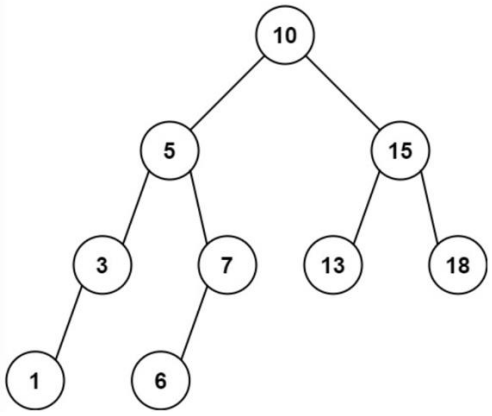
3 (20pts). Given the root node of a binary search tree and two integers low and high, return the sum of values of all nodes with a value in the inclusive range [low, high]. **Please provide your algorithm input & output and time/space analysis. (Pseudocode will be fine).**

Example 1:



Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32
Explanation: Nodes 7, 10, and 15 are in the range [7, 15]. 7 + 10 + 15 = 32.

Example 2:



Input: root = [10,5,15,3,7,13,18,1,null,6], low = 6, high = 10
Output: 23
Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. 6 + 7 + 10 = 23.

Constraints:
The number of nodes in the tree is in the range [1, 2 * 104].
1 <= Node.val <= 105, 1 <= low <= high <= 105
All Node.val are unique.

```
// Input: A binary search tree with 2 integers that define the range (low - high)

 // Output: The sum of all node values within the designated range

class Solution {

    public int rangeSumBST(TreeNode root, int low, int high) {

        // Empty Tree

        if (root == null) {

            return 0;

        }


        // If value is too small, everything left is smaller, skip left

        if (root.val < low) {

            return rangeSumBST(root.right, low, high);

        }


        // If value is too big, everything right is bigger, skip right

        if (root.val > high) {
```

```
            return rangeSumBST(root.left, low, high);
        }

        // Within range: include root and explore both sides
        return root.val + rangeSumBST(root.left, low, high) +
rangeSumBST(root.right, low, high);
    }
    // Time Complexity - O(2^n): At worst, all nodes are in range and therefore need
to be visited
    // Space Complexity - O(2^n): At worst, a stack frame will need to be created for
the entire tree
}
```

4 (20pts). Given an m x n 2D binary grid “grid” which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water. **Please provide your algorithm with input/output, and time/space analysis. (Pseudocode will be fine).**

Example 1:

Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
Example 2:

Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3

Constraints:
m == grid.length, n == grid[i].length
1 <= m, n <= 300, grid[i][j] is '0' or '1'.

```
class Solution {
    // Input: A mxn matrix representing land (1) and water (0)
    // Output: The number of islands
    // Counts number of islands in the grid
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int m = grid.length;
        int n = grid[0].length;
        int islands = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    islands++;
                    dfs(i, j, m, n, grid);
                }
            }
        }
        return islands;
```

```
    }

    // Depth-First Search to mark all connected land cells
    private void dfs(int r, int c, int m, int n, char[][] grid) {
        // Base case: out of bounds OR not land
        if (r < 0 || c < 0 || r >= m || c >= n || grid[r][c] != '1') {
            return;
        }

        // Mark current cell as visited
        grid[r][c] = '0';

        // Visit all four neighbors
        dfs(r + 1, c, m, n, grid);
        dfs(r - 1, c, m, n, grid);
        dfs(r, c + 1, m, n, grid);
        dfs(r, c - 1, m, n, grid);
    }
}

// Time: O(n^2) - At worst we have all water or all land so we have to traverse every spot on the matrix
```

// Space O(n^2) - We are at worst creating a stack frame for every cell in the matrix