DFS

Coding Style

- The name for Class, Method, Variable: Camel case
- Constant Value: All Upper Case
- Space, empty lines, Braces
- How long should be one method?
- Reusable code
- Comments

Google Java Style

Google C++ Style

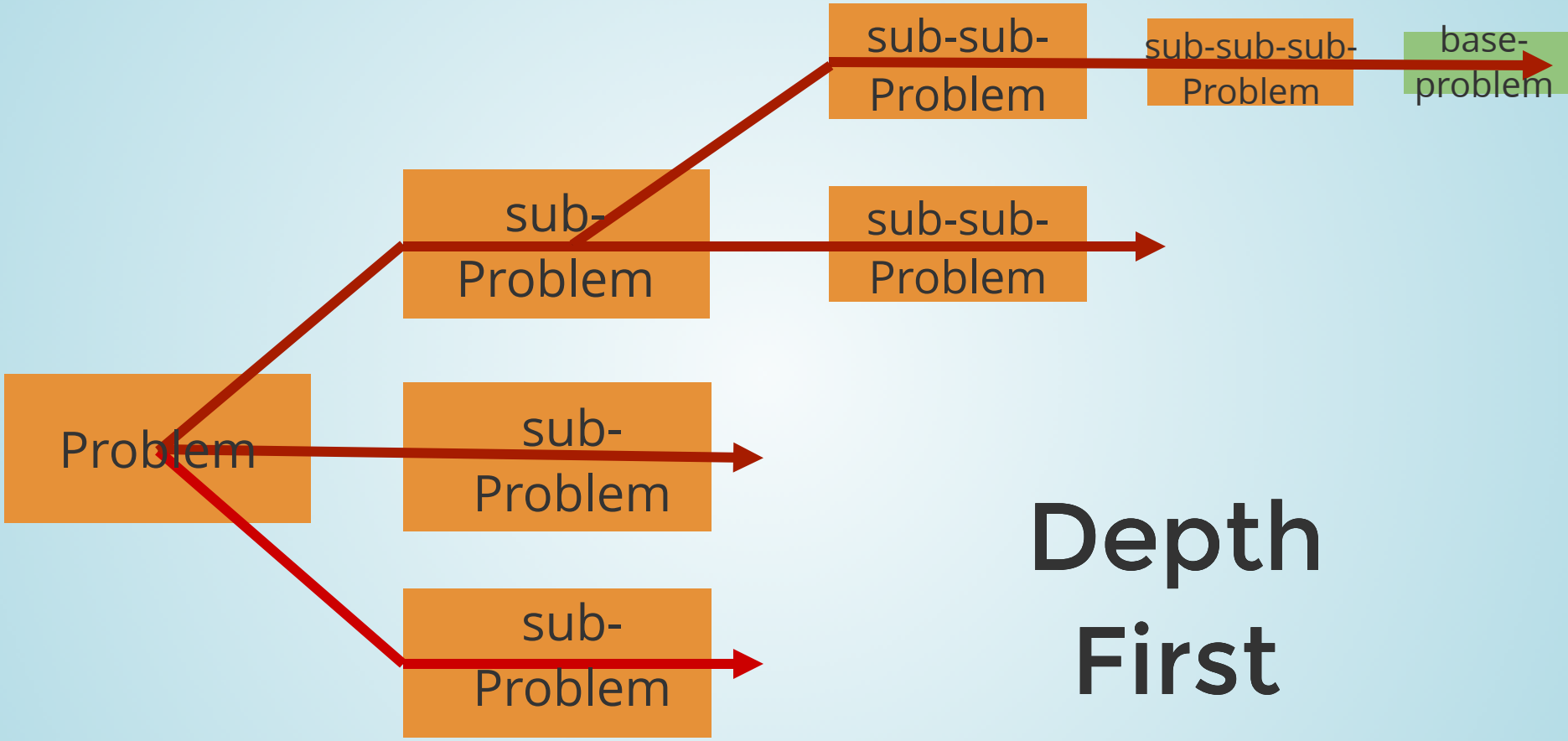
Searching Algorithm

DFS

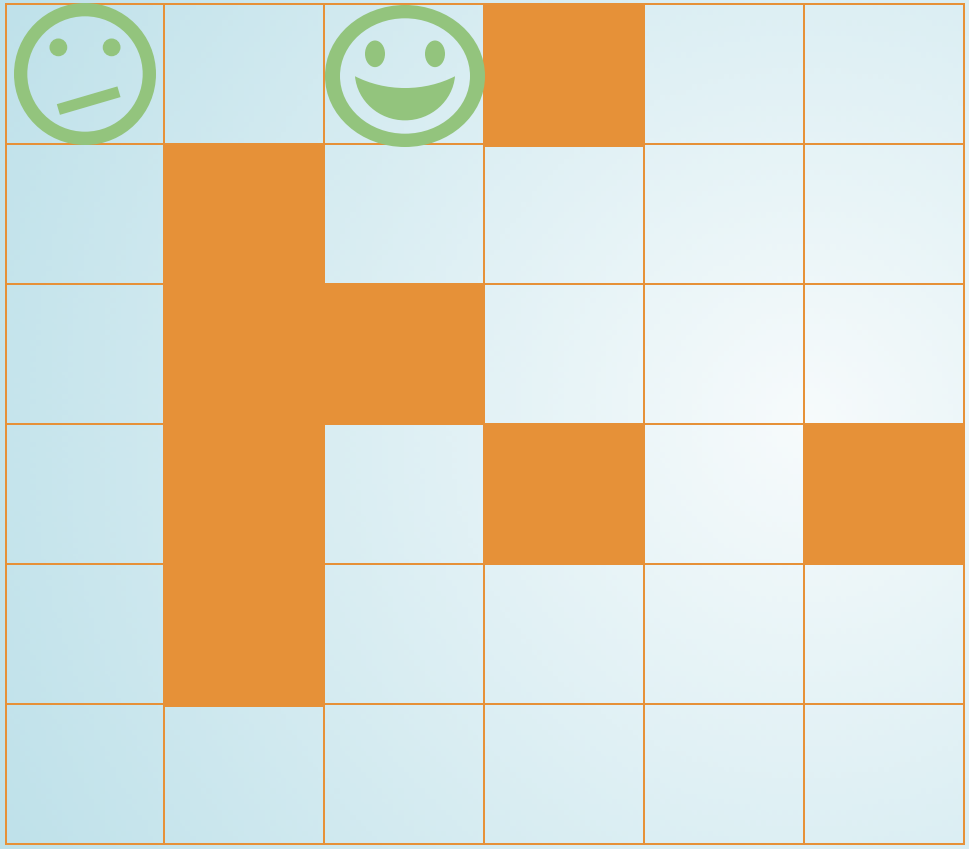
- Depth First Search
- One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

BFS

- Breadth First Search
- One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores the neighbor nodes first, before moving to the next level neighbors.

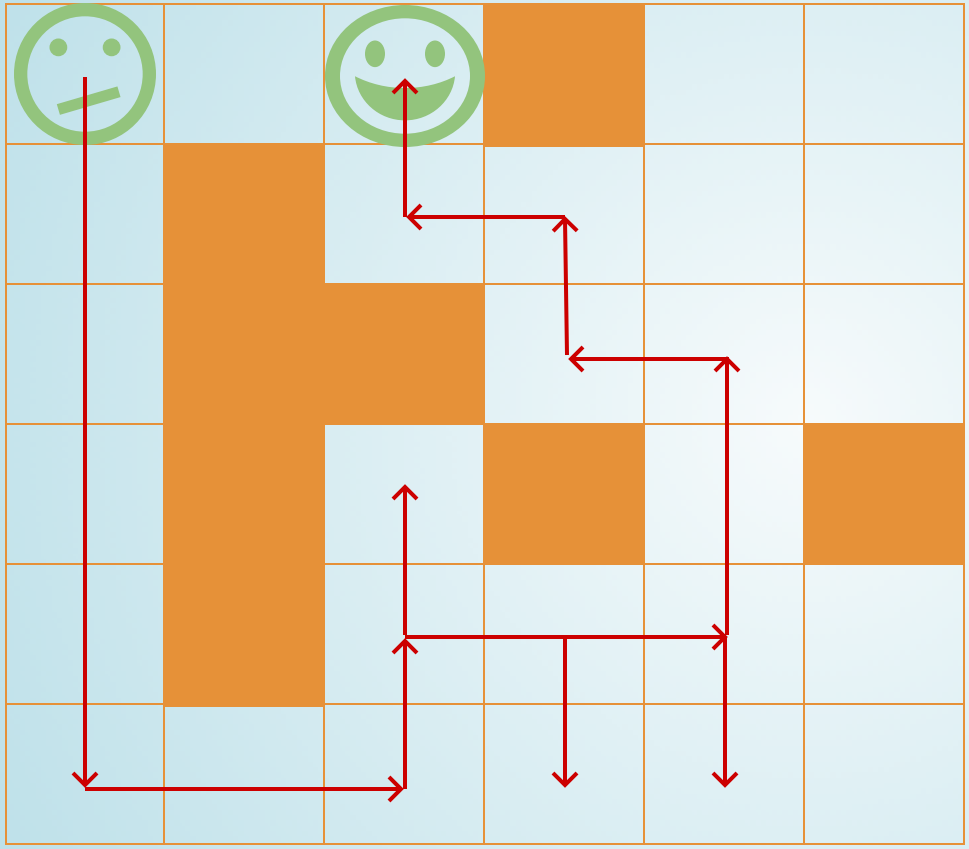


Depth First Search



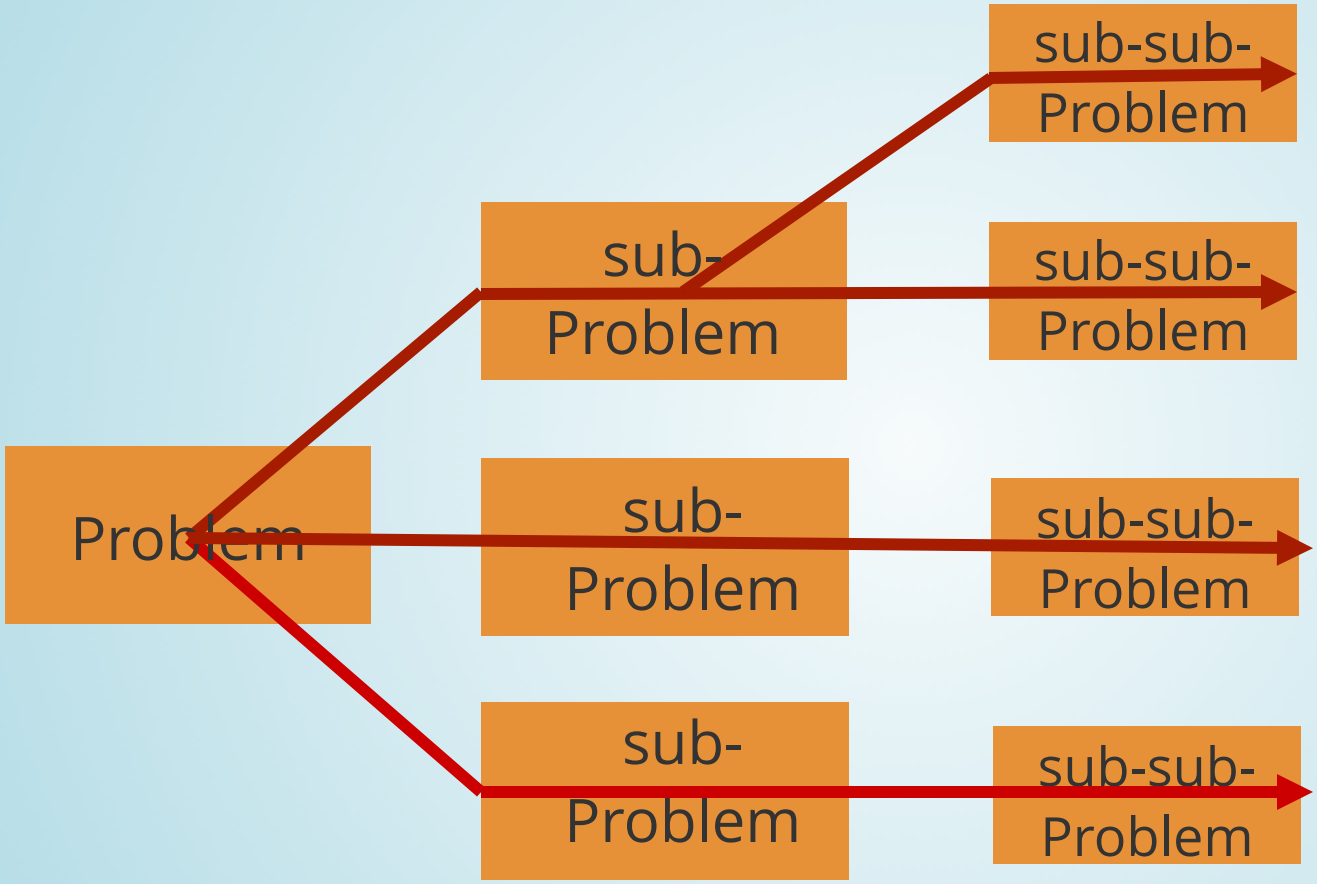
Search Order:

- U, R, D, L
- D, L, U, R

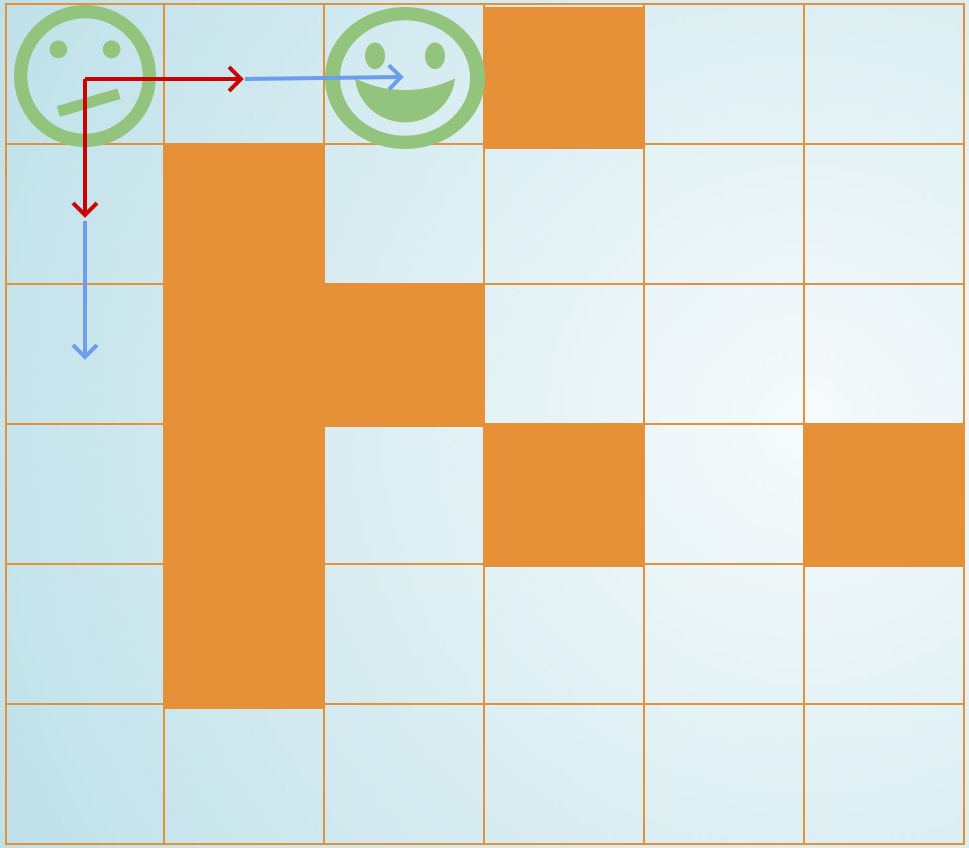


Search Order:

- U, R, D, L
- D, L, U, R

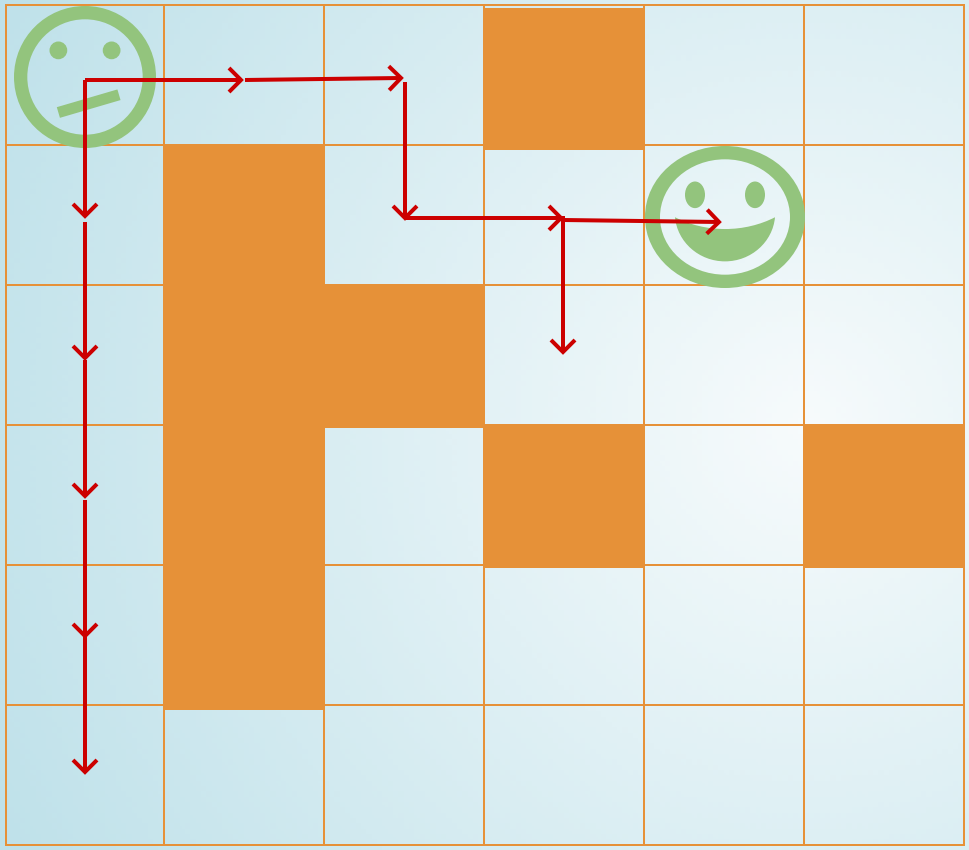


Breadth First Search



Search Order:

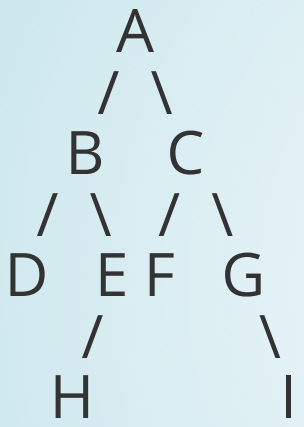
- U, R, D, L
- D, L, U, R



Search Order:

- U, R, D, L
- D, L, U, R

Tree Traversal



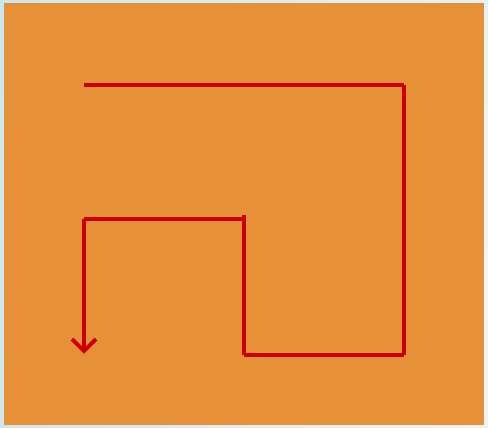
- Preorder Traversal
  - A B D E H C F G I
  - DFS Search Path
- Level Order Traversal
  - A B C D E F G H I
  - BFS Search Path

Filling the Grid

DFS Search Order:

U, R, D, L

Filling the Grid



DFS Search Order:

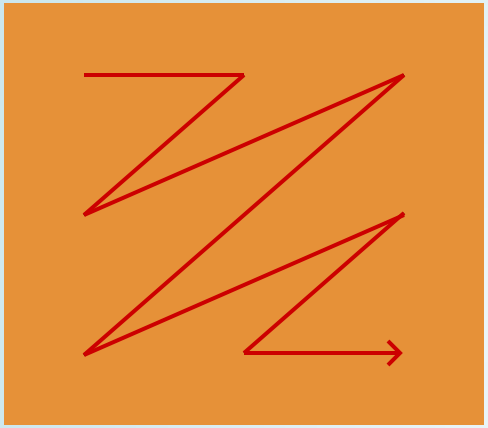
U, R, D, L

Filling the Grid

BFS Search Order:

U, R, D, L

Filling the Grid



BFS Search Order:

U, R, D, L

Use Case

DFS and BFS are both algorithms for searching TREE and GRAPH, but we will ONLY focus on trees, which is more common in interviews.

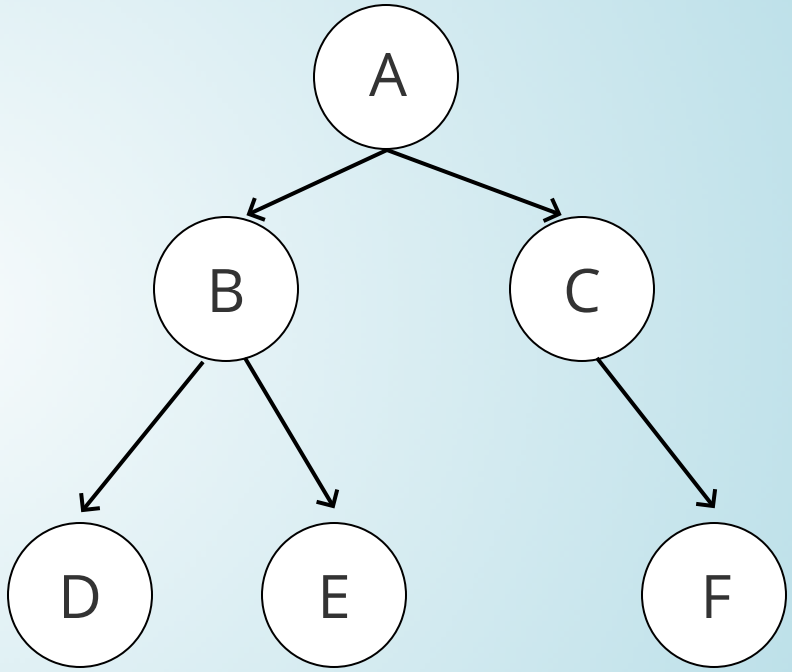
- DFS
  - Recursion and backtrace
  - Stack
- BFS
  - Find minimum path/distance/...
  - Queue

DFS

- Most DFS related problems in interview is for trees.
  - Balanced Binary Tree
  - Symmetric Tree
  - Minimum Depth of Binary Tree
  - etc.
- More general, most recursion problems are also DFS.
  - Permutation, Combination, Combination Sum.
  - Letter Combination of Phone Number, Generate Parenthesis.
  - Surrounded Regions, Word Search,

Basic Terms of Tree

- Root
- Node (internal node)
- Leaf (external node)
- Parent
- Child
- Siblings
- Ancestor
- Descendant
- Edge (N - 1)
- Path
- Height
- Depth



PreOrder Traversal

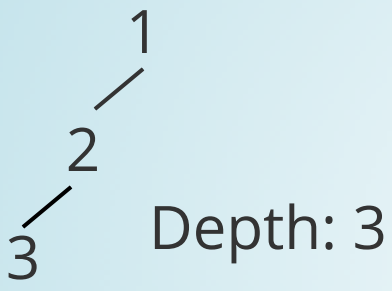
InOrder Traversal

PostOrder Traversal

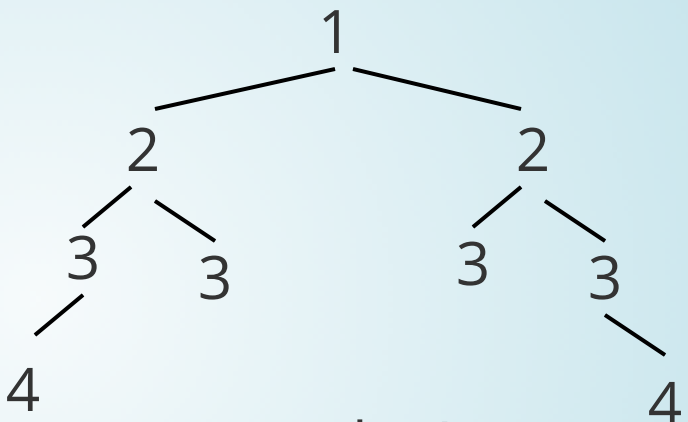
They are all DFS

Depth of Tree

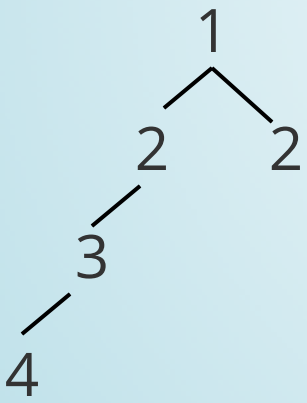
Given a binary tree, calculate its depth.



Depth: 3



Depth: 4



Depth: 4

Depth of Tree

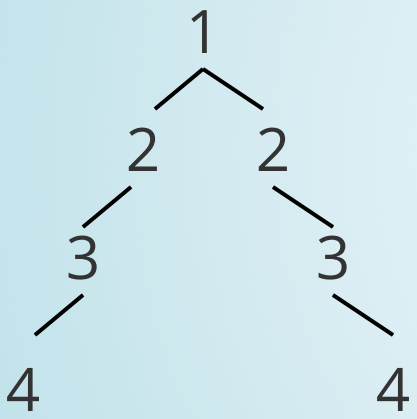
Given a binary tree, calculate its depth.

```
public int depth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    return Math.max(depth(root.left), depth(root.right)) + 1;
}
```

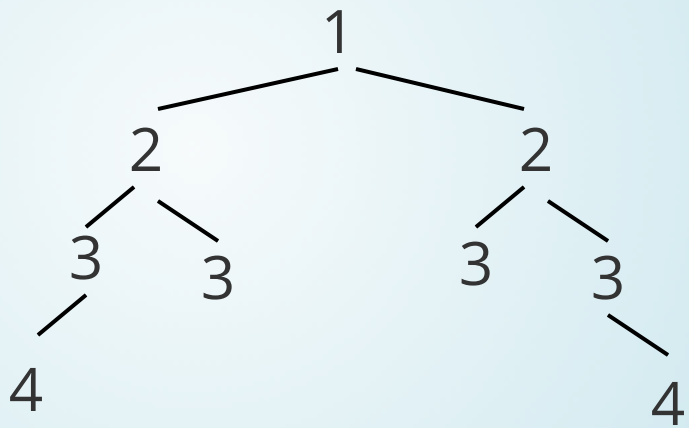
Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.



False



True

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

- root is balanced
  - Diff(depth(root.left), depth(root.right)) <= 1
  - root.left is balanced && root.right is balanced.

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

```
public boolean isBalanced(TreeNode root) {
    if (root == null)
        return true;
    if (Math.abs(getDepth(root.left) - getDepth(root.right)) > 1)
        return false;
    return isBalanced(root.left) && isBalanced(root.right);
}

int getDepth(TreeNode root) {
    if (root == null)
        return 0;
    return Math.max(getDepth(root.left), getDepth(root.right)) + 1;
}
```

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

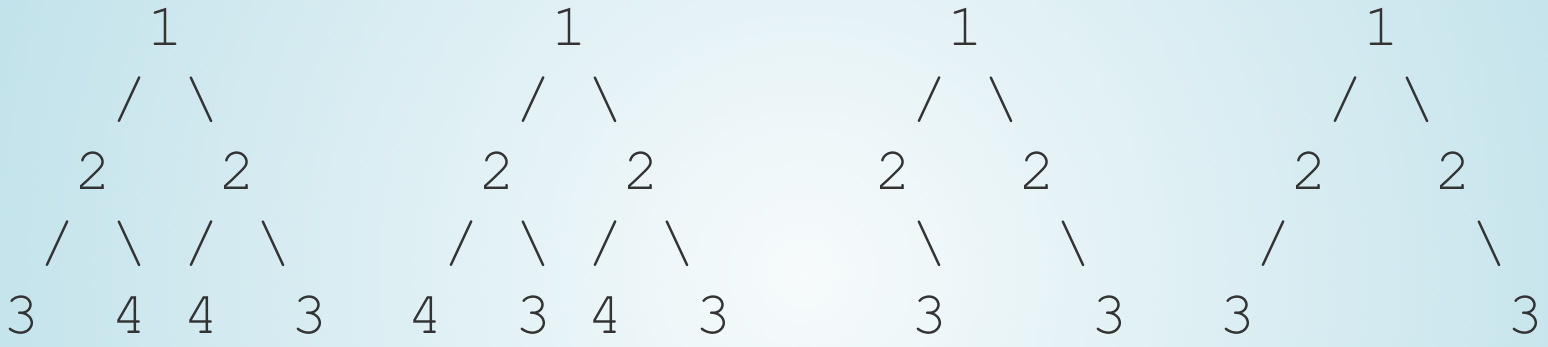
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

```
public boolean isBalanced(TreeNode root) {
    return depth(root) != -1;
}

public int depth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftDepth = depth(root.left);
    int rightDepth = depth(root.right);
    if (leftDepth == -1 || rightDepth == -1 ||
        Math.abs(leftDepth - rightDepth) > 1) {
        return -1;
    }
    return Math.max(leftDepth, rightDepth) + 1;
}
```

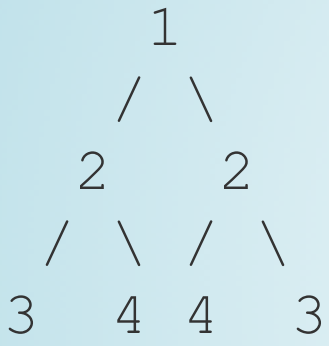
Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

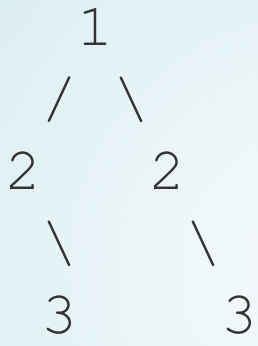


Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).



True



False

- Root is Symmetric
  - root.left is symmetric to root.right
- A is symmetric to B
  - A.val == B.val
  - A.left is symmetric to B.right
  - A.right is symmetric to B.left

Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

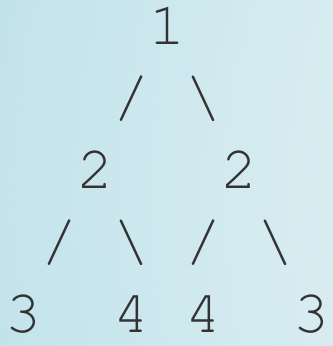
```
public boolean isSymmetric(TreeNode root) {
    if (root == null) {
        return true;
    }
    return isSymmetric(root.left, root.right);

}
public boolean isSymmetric(TreeNode left, TreeNode right) {
    if (left == null && right == null) {
        return true;
    }
    if (left == null && right != null) {
        return false;
    }
    if (left != null && right == null) {
        return false;
    }
    if (left.val != right.val) {
        return false;
    }
    return isSymmetric(left.left, right.right)
        && isSymmetric(left.right, right.left);
}
```

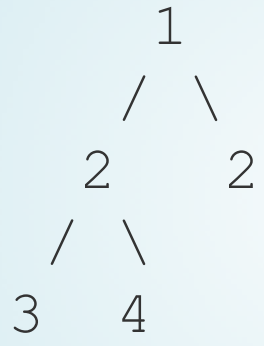
Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

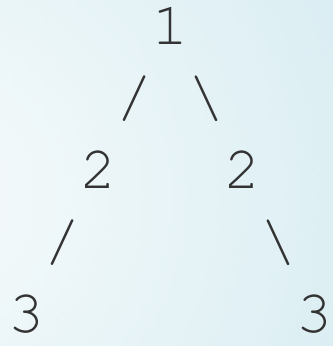
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.



3



2



3

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

- Minimum depth of Root
  - if root is leaf, 1.
  - if root.left is null, return right + 1.
  - if root.right is null, return left + 1.
  - return the minimum of (left, right) + 1.

Minimum Depth of Binary Tree

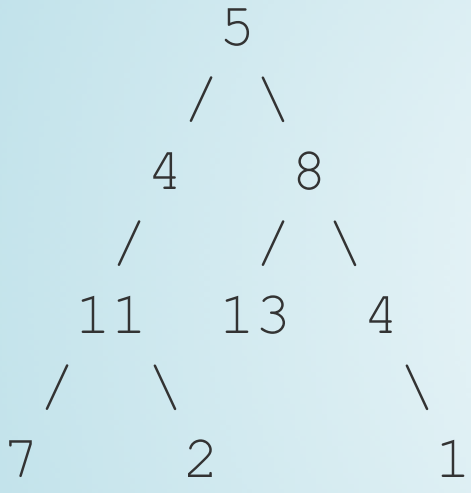
Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

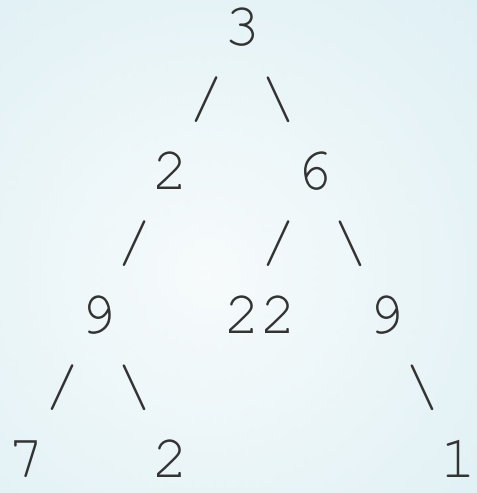
```
public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftDepth = minDepth(root.left);
    int rightDepth = minDepth(root.right);
    if (leftDepth == 0) {
        return rightDepth + 1;
    } else if (rightDepth == 0) {
        return leftDepth + 1;
    }
    return Math.min(leftDepth, rightDepth) + 1;
}
```

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.



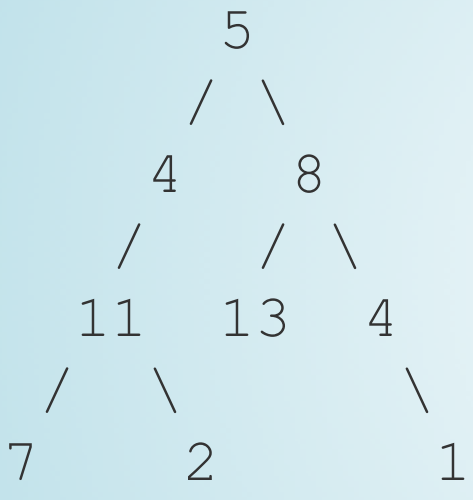
22: [5, 4, 11, 2]



31: [3, 6, 22]

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.



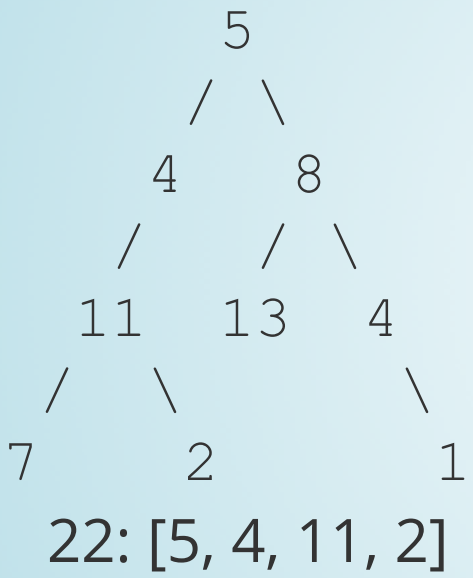
22: [5, 4, 11, 2]

Knapsack:

- Except picking from all of numbers, only path from root to leaf is acceptable.
- hasSum(TreeNode root, int sum)
  - Base case: root is leaf.
  - hasSum(root, sum) = hasSum(root.left, sum-root.val) || hasSum(root.right, sum-root.val)

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.



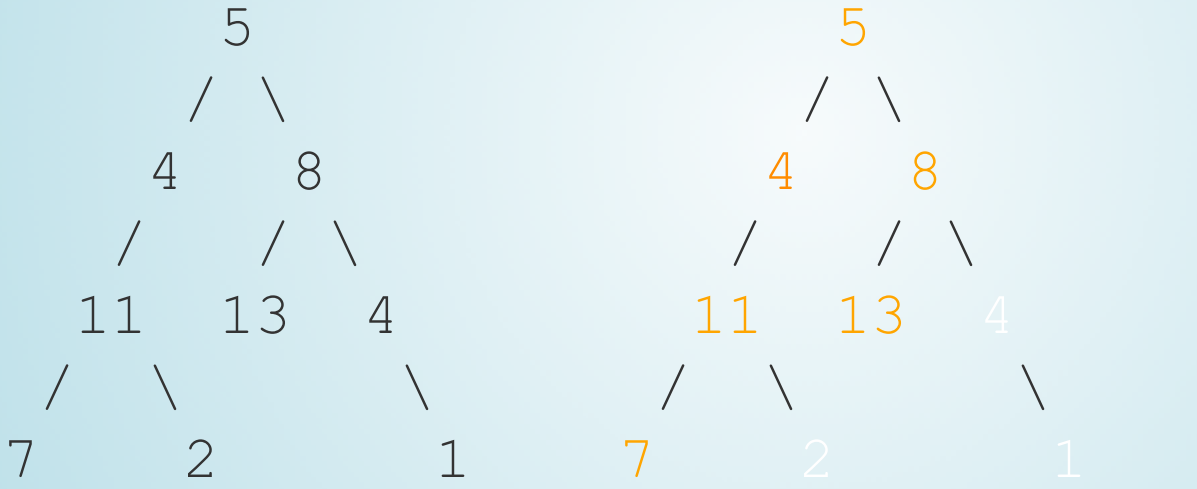
22: [5, 4, 11, 2]

```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        if (sum == root.val) {
            return true;
        }
        return false;
    }
    return hasPathSum(root.left, sum-root.val) ||
           hasPathSum(root.right, sum-root.val);
}
```

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.



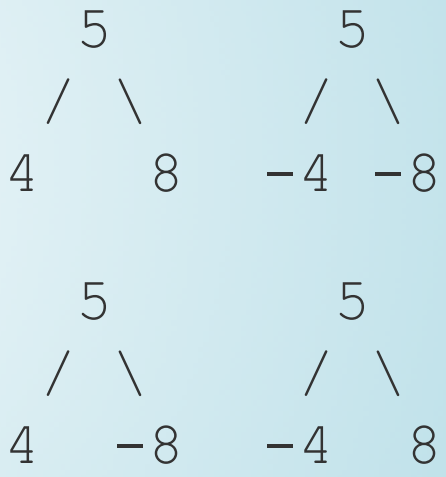
maxSum = 7 + 11 + 4 + 5 + 8 + 13 = 48

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

- Branch Sum (root):
  - Sum of nodes that end at root.
- Maximum Path Sum (root):
  - root.val
  - root.val + maxbranchSum(root.left)
  - root.val + maxbranchSum(root.right)
  - root.val + maxbranchSum(root.left) + maxbranchSum(root.right)



Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

- branchSum (root)
  - root.val
  - root.val + branchSum(root.left)
  - root.val + branchSum(root.right)

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

```
int max_sum = Integer.MIN_VALUE;

public int maxPathSum(TreeNode root) {
    if (root == null)
        return 0;
    max_sum = Integer.MIN_VALUE;
    maxBranchSum(root);
    return max_sum;
}

public int maxBranchSum(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftSum = maxBranchSum(root.left);
    int rightSum = maxBranchSum(root.right);
    int branchMaxSum = root.val + Math.max(0, Math.max(leftSum, rightSum));
    max_sum = Math.max(max_sum,
                       Math.max(branchMaxSum, leftSum + root.val + rightSum));
    return branchMaxSum;
}
```

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

Use int[] to pass the maximum through the recursion

```
public static int maxPathSum(TreeNode root) {
    if (root == null)
        return 0;
    int[] max = {Integer.MIN_VALUE};
    maxBranchSum(root, max);
    return max[0];
}

public static int maxBranchSum(TreeNode root, int[] max) {
    if (root == null) {
        return 0;
    }
    int leftSum = maxBranchSum(root.left, max);
    int rightSum = maxBranchSum(root.right, max);
    int branchMaxSum = root.val + Math.max(0, Math.max(leftSum, rightSum));
    max[0] = Math.max(max[0], Math.max(branchMaxSum, leftSum + root.val + rightSum));

    return branchMaxSum;
}
```

DFS for Trees

- Complete Search
  - If you need to traverse all the nodes in the tree.
  - No need for minimum/shortest/... depth, etc.
- Normal Recursion
  - Tree is recursive definition, so applying recursion on trees are very direct and easy understanding
  - Sub-problem
  - Base case
  - Recursion rule
  - ALWAYS think about two branches (left and right), pseudo code is helpful.

More general DFS

- One Dimension
  - Letter Combination of Phone Number
  - Generate Parenthesis
  - Palindrome Partitioning
  - etc.
- Two Dimensions
  - Maze
  - N Queens
  - Word Search
  - etc.

Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example, given a board,

```
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
```

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

Word Search

```
public boolean exist(char[][] board, String word) {
    if (board.length == 0 || board[0].length == 0)
        return false;
    boolean[][] flag = new boolean[board.length][board[0].length];
    for (int i = 0; i < board.length; i++) {
       Arrays.fill(flag[i], false);
    }
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (exist(board, i, j, word, 0, flag))
                return true;
        }
    }
    return false;
}
public boolean exist(char[][] board, int x, int y, String word, int index,
                     boolean[][] flag) {
    if (index == word.length())
        return true;
    if (x < 0 || y < 0 || x >= board.length || y >= board[0].length
        || flag[x][y] || board[x][y] != word.charAt(index))
        return false;
    int[] dx = {1, 0, -1, 0};
    int[] dy = {0, 1, 0, -1};
    for (int i = 0; i < 4; i++) {
        flag[x][y] = true;
        if (exist(board, x+dx[i], y+dy[i], word, index+1, flag))
            return true;
        flag[x][y] = false;
    }
    return false;
}
```

Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

Example:

Input: "aab"

Output: [["aa", "b"], ["a", "a", "b"]]

Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

- Try substring from the beginning.
  - If it's palindrome, then palindrome partition the left part and merge the results.

Palindrome Partitioning

```
public List<List<String>> partition(String s) {
    List<List<String>> results = new ArrayList<>();
    partition(results, s, 0, new ArrayList<String>());
    return results;
}

public void partition(List<List<String>> results, String s, int start,
                      List<String> path) {
    if (start == s.length()) {
        results.add(new ArrayList<>(path));
        return;
    }
    for (int i = start + 1; i <= s.length(); i++) {
        String sub = s.substring(start, i);
        if (isPalindrome(sub)) {
            path.add(sub);
            partition(results, s, i, path);
            path.remove(path.size() - 1);
        }
    }
}
public boolean isPalindrome(String s) {
    for (int i = 0, j = s.length()-1; i < j; i++, j--) {
        if (s.charAt(i) != s.charAt(j))
            return false;
    }
    return true;
}
```

Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

- Three cases:
  - if you get '(', two conditions: add it or not
  - if you get ')', two conditions: add it or not. But if you want to add it, check if you have available '('
  - if you get others, just add it

Remove Invalid Parentheses

```
private List<String> result = new ArrayList<String>();
private int max = 0;
public List<String> removeInvalidParentheses(String s) {
    dfs(s, "", 0, 0);
    if (result.size() == 0) {
        result.add("");
    }
    return result;
}
public void dfs(String str, String subRes, int left, int tempMax) {
    if (str.length() == 0) {
        if (left == 0 && subRes.length() != 0) {
            if (tempMax > max) {
                max = tempMax;
                result.clear();
            }
            if (max == tempMax && !result.contains(subRes)) {
                result.add(subRes.toString());
            }
        }
        return;
    }
    if (str.charAt(0) == '(') {
        dfs(str.substring(1), subRes.concat("("), left + 1, tempMax + 1);
        dfs(str.substring(1), subRes, left, tempMax);
    } else if (str.charAt(0) == ')') {
        if (left > 0) {
            dfs(str.substring(1), subRes.concat(")"), left - 1, tempMax);
        }
        dfs(str.substring(1), subRes, left, tempMax);
    } else {
        String skipLetter = String.valueOf(str.charAt(0));
        dfs(str.substring(1), subRes.concat(skipLetter), left, tempMax);
    }
}
```

Not very good

Remove Invalid Parentheses

This is a question to get the best answers, so the condition to do the DFS is important.

Using BFS is also a good approach.

Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

coins = [1, 2, 5], amount = 11

return 3 (11 = 5 + 5 + 1)

Example 2:

coins = [2], amount = 3

return -1.

Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Two choices:

Use it or not.

Coin Change

WRONG ANSWER! WHY?

```
static int min;

public int coinChange(int[] coins, int amount) {
    Arrays.sort(coins);
    min = Integer.MAX_VALUE;
    if (helper(coins, amount, coins.length - 1, 0)) {
        return min;
    }
    return -1;
}

public boolean helper(int[] coins, int rest, int current, int count) {
    if (rest < 0) return false;
    if (rest == 0) {
        if (count < min) {
            min = count;
        }
        return true;
    }
    if (count >= min) return false;
    boolean result = false;
    for (int j = current; j >= 0; j --) {
        result = result || helper(coins, rest - coins[j], j, count + 1);
    }
    return result;
}
```

Coin Change

```
static int min;

public int coinChange(int[] coins, int amount) {
    Arrays.sort(coins);
    min = Integer.MAX_VALUE;
    if (helper(coins, amount, coins.length - 1, 0)) {
        return min;
    }
    return -1;
}

public boolean helper(int[] coins, int rest, int current, int count) {
    if (rest == 0) {
        if (count < min) {
            min = count;
        }
        return true;
    }
    if (current < 0 || rest < 0) return false;
    if (count >= min) return false;
    boolean result = false;
    for (int i = rest / coins[current]; i >= 0; i --) {
        if (helper(coins, rest - i * coins[current], current - 1, count + i)) {
                result = true;
        }
    }
    return result;
}
```

Coin Change

Still it won't pass all the test cases because it is not efficient.
We will introduce a better solution in Dynamic Programming

General DFS

- Depth First Search
- Search solutions from the whole possible search space
- Search is a process of trying and backtracking
- Pruning is important

Homework

Same Tree

Validate Binary Search Tree

Flatten Binary Tree to Linked List (you can avoid DFS also)

Path Sum II

Matchsticks to Square

Number Of Islands

Concatenated Words