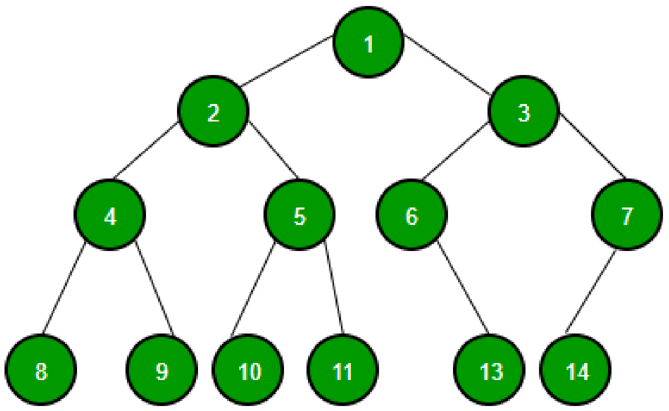
A **tree** is a hierarchical data structure consisting of nodes, where:

- The top node is called the **root**.
- Each node can have **child nodes**.
- A tree can be visualized as a branching structure.

Types of Trees

Binary Trees

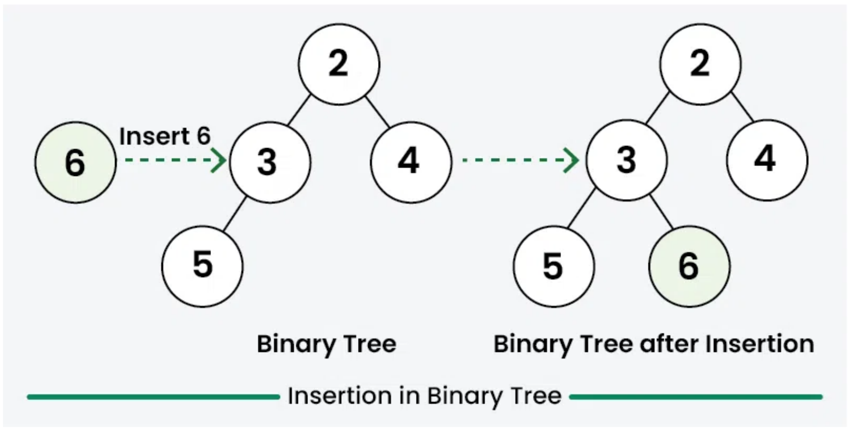
A **binary tree** is a type of tree where each node has at most **two child nodes**. a left child and a right child. The binary tree is a non linear tree data structure, each node will have three parts data, pointer to left node and pointer to right node.



Operations:

Insertion:

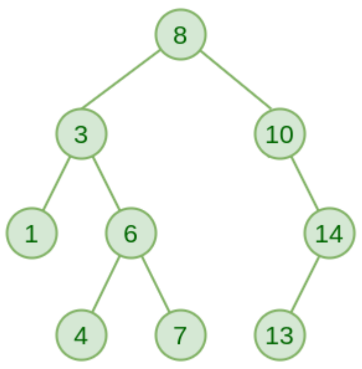
- Start at the root and check the left child; if it's empty, insert the new node there.
- If the left child is occupied, check the right child; if it's empty, insert the new node there.
- If both children are occupied, repeat the process down the tree until you find an empty spot.



Operation	Time Complexity	Auxiliary Space
In-Order Traversal	O(n)	O(n)
Pre-Order Traversal	O(n)	O(n)
Post-Order Traversal	O(n)	O(n)
Insertion (Unbalanced)	O(n)	O(n)
Searching (Unbalanced)	O(n)	O(n)
Deletion (Unbalanced)	O(n)	O(n)

Binary Search Trees (BST)

A **binary search tree** is a binary tree with a specific ordering property. For any given node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater.



Binary Search Tree

Operations on Trees:

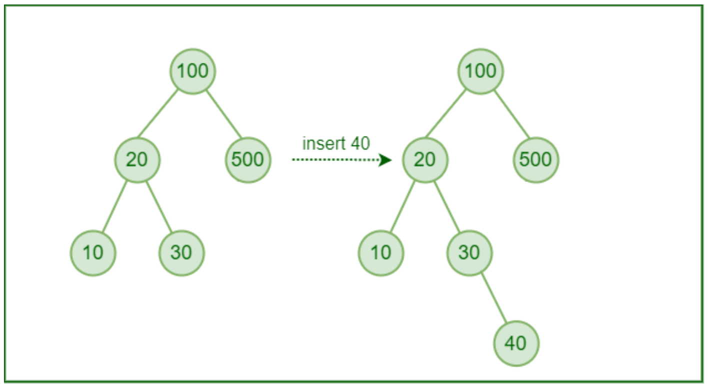
Searching:

- Searching in a BST is efficient due to its ordering.
- Start at the root, compare the target value with the node's value.
- If the target is less, move to the left child; if greater, move to the right child.
- **Example**: Searching for the value 15 in a tree rooted at 10 would involve moving to the right child (if the right child is greater than 10).

Insertion:

- Inserting a value follows the same logic as searching:
- If the value is less than the current node, go left; if greater, go right.
- If a child node is null, insert the new value there.

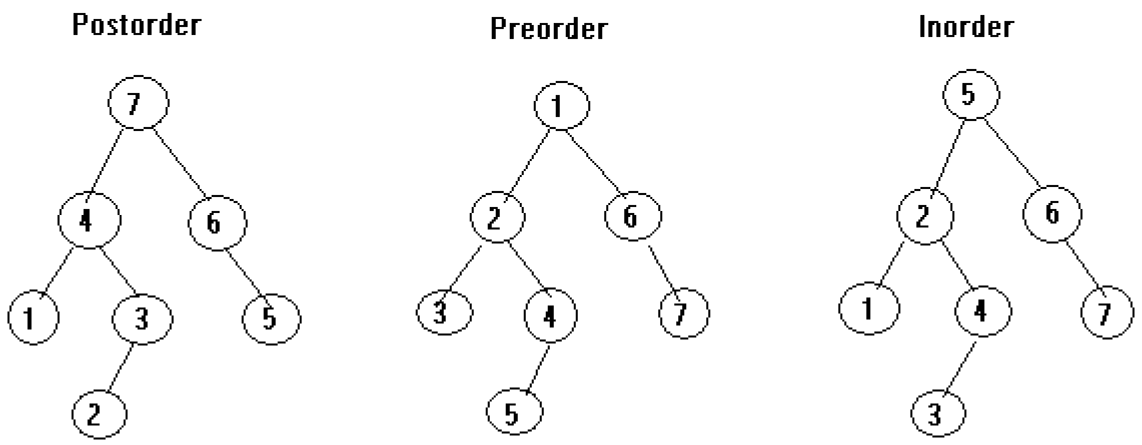
- **Example**: Inserting 12 into a BST that has 10 as the root would lead to navigating to the right child (if it exists) or directly inserting 12 as a child of the appropriate node.



Traversal Methods:

There are three common traversal methods:

- **Preorder Traversal**: Visit the root first, then left, then right.
- **Inorder Traversal**: Visit left nodes first, then the root, then right nodes. This results in sorted order for BSTs.
- **Postorder Traversal**: Visit left nodes, then right nodes, and finally the root.



Deletion:

- Deletion involves adjusting links and is also performed in a similar manner to searching. The average case complexity for deletion is **O(log n)**.

- **Example**: Deleting 15 from a tree requires finding it first and then determining its children for proper pointer adjustments.

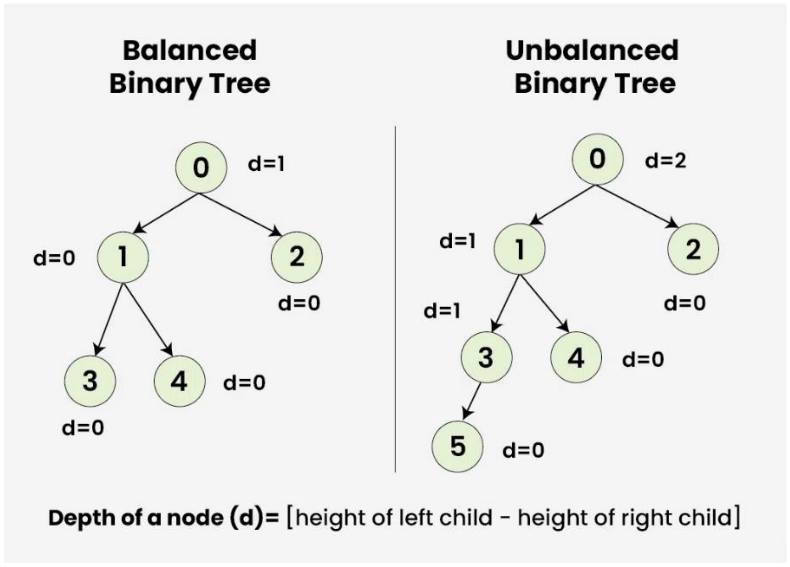
Complexity Analysis:

- **Average Case**: The time complexity for search, insert, and delete operations in a balanced BST is **O(log n)**.

- **Worst Case**: If the tree becomes unbalanced (e.g., nodes are added in sorted order), operations can degrade to **O(n)**, resembling a linked list.

Balancing Trees:

- Trees can become imbalanced if nodes are inserted in a sorted order, resembling a linked list.
- Balanced trees ensure that the number of nodes on the left and right subtrees are roughly equal, maintaining efficient operations.



Real-World Examples:

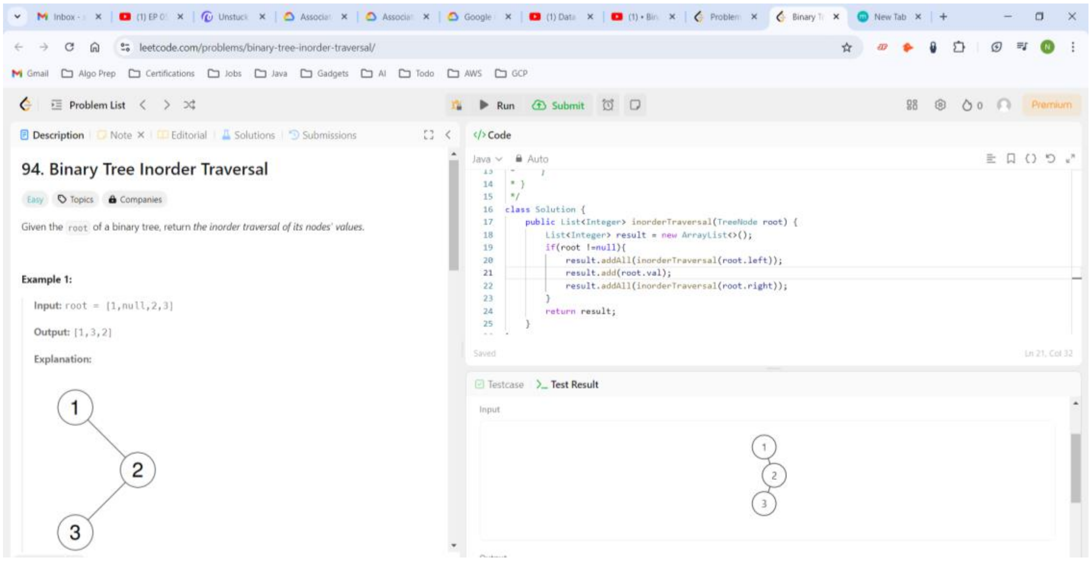
1. **Database Indexing**: Many databases use BSTs or their balanced variants (like AVL trees) to index records efficiently, allowing for rapid queries based on keys.
2. **Autocompletion**: The principles of BSTs can be applied in search engines where input suggestions are generated based on previously indexed data, optimizing retrieval times.
3. **File Systems**: Modern file systems often utilize tree structures to manage directories and files, allowing for quick lookups and organization of data.

Difference between a binary tree and a binary search tree:

- Binary trees are not having any order, but the binary search trees are having specific order.
- Performing the operations in BST's are more efficient than binary trees.
- In BST's performing insertion and deletion are more complex than performing those in binary trees.

Problem 94:

- In reference to the lecture, I attempted to solve the problem but encountered an issue because the lecture used a void method, while the question required returning a list.
- To address this, I developed a modified approach that collects nodes during traversal.
- My approach involves first collecting all the left nodes, followed by the root, and then the right nodes.



94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

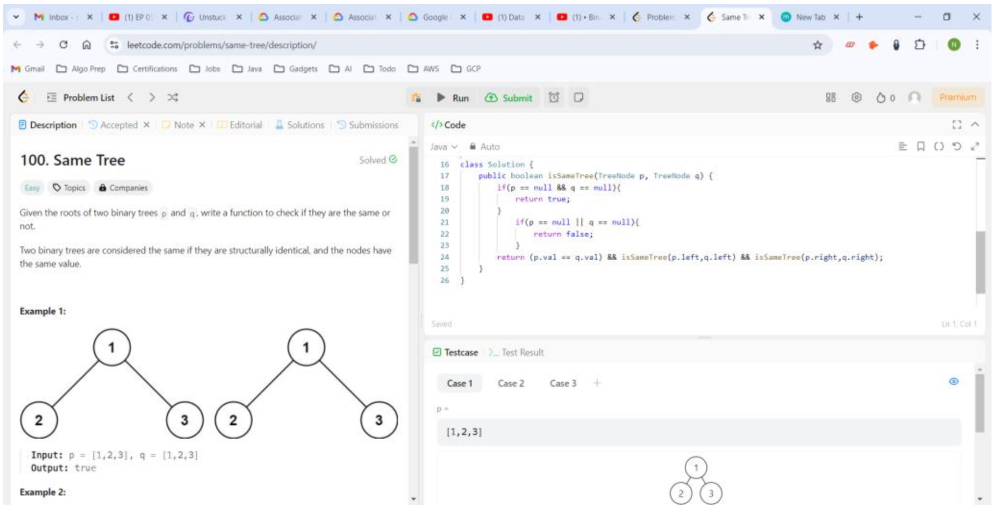
Output: [1,3,2]

Explanation:

```java
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if(root !=null){
            result.addAll(inorderTraversal(root.left));
            result.add(root.val);
            result.addAll(inorderTraversal(root.right));
        }
        return result;
    }
```

Problem 100:

- After solving the inorder traversal problem, I applied similar logic to match each element by traversing the tree in a specific order: exploring the left nodes first, then the root, and finally the right nodes.
- During the traversal, I compared each element to the expected values; if a match is found, the function returns true.
- If no matches are found throughout the traversal, the function returns false, ensuring a comprehensive check of the tree structure while maintaining the correct order.



100. Same Tree

Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:

Input: p = [1,2,3], q = [1,2,3]
Output: true

Example 2:

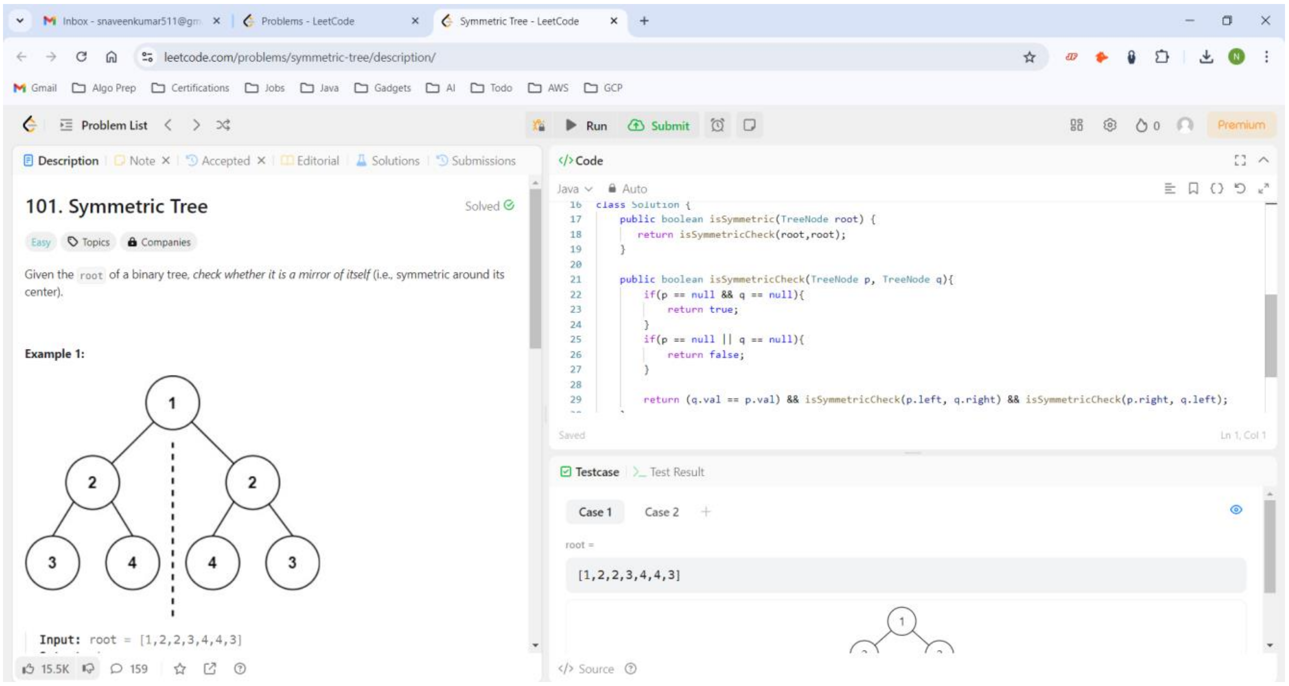
```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == null && q == null){
            return true;
        }
          if(p == null || q == null){
            return false;
          }
        return (p.val == q.val) && isSameTree(p.left,q.left) && isSameTree(p.right,q.right);
    }
}
```

p =

[1,2,3]

Problem 101:

- I adapted the logic used to verify identical trees to check for symmetry by creating a separate method for node comparison.
- Instead of comparing the same nodes, I focused on comparing opposite nodes, such as the left child of one subtree with the right child of the other.
- The method returns true if the opposite nodes match, and false if any comparison fails, effectively checking for symmetry in the binary tree.



101. Symmetric Tree

Solved

Easy

Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Example 1:

Input: root = [1,2,2,3,4,4,3]

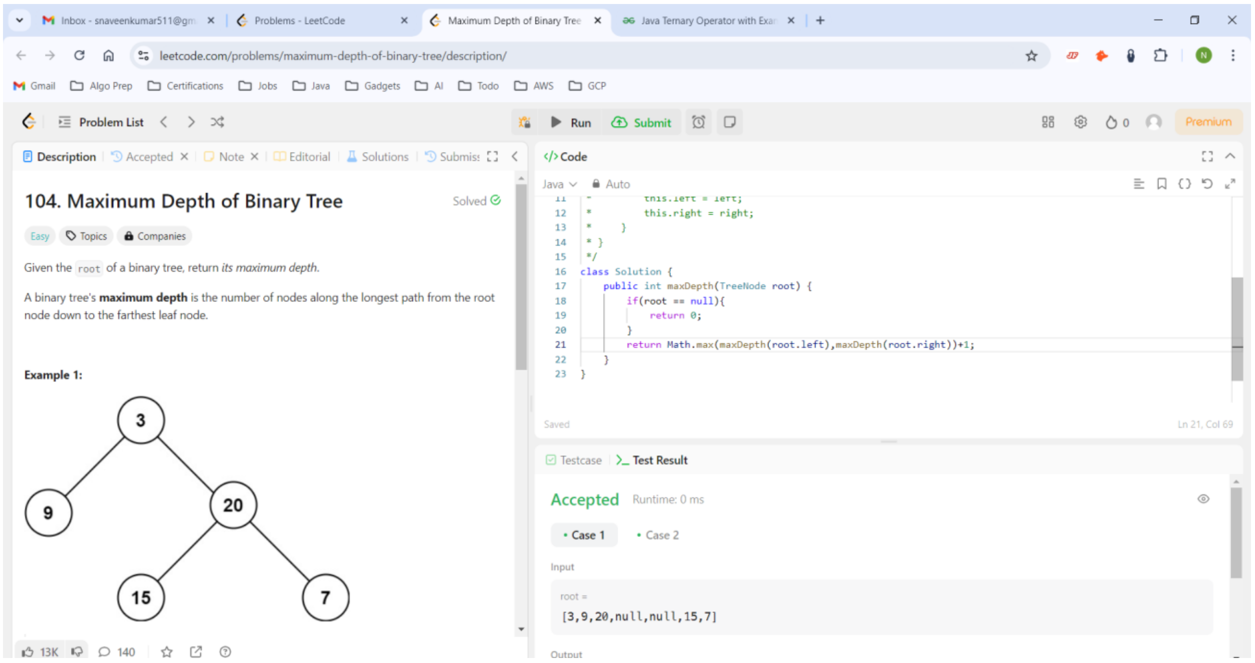
```java
class Solution {
    public boolean isSymmetric(TreeNode root) {
        return isSymmetricCheck(root,root);
    }

    public boolean isSymmetricCheck(TreeNode p, TreeNode q){
        if(p == null && q == null){
            return true;
        }
        if(p == null || q == null){
            return false;
        }

        return (q.val == p.val) && isSymmetricCheck(p.left, q.right) && isSymmetricCheck(p.right, q.left);
```

root =

[1,2,2,3,4,4,3]

Problem 104:

- After solving these problems, I became familiar with recursion and node traversal techniques, which I applied to this problem as well.
- The approach begins by checking if the current node is null; if it is not, I traverse the left subtree to obtain its depth and then the right subtree to obtain its depth.
- After acquiring the depths of both subtrees, I compare them to determine the maximum depth and add 1 for the root node, ultimately calculating the maximum depth of the tree.



104. Maximum Depth of Binary Tree

Solved

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.

Example 1:

```
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null){
            return 0;
        }
        return Math.max(maxDepth(root.left),maxDepth(root.right))+1;
    }
}
```

Accepted Runtime: 0 ms

```
root =
[3,9,20,null,null,15,7]
```

Dynamic Programming:

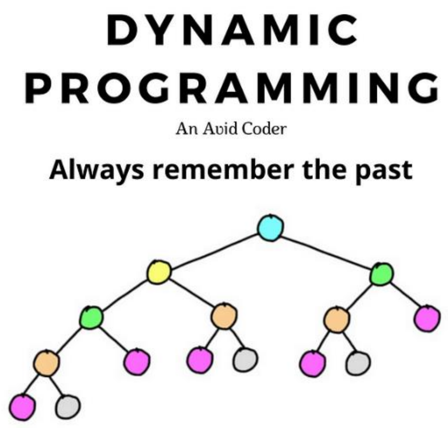
Dynamic Programming (DP) is an efficient algorithmic technique for solving complex problems by dividing them into smaller, overlapping subproblems. It is especially useful in optimization tasks, where naive recursion can lead to redundant calculations. By storing the results of subproblems through memorization or tabulation, DP reduces time complexity from exponential to polynomial. This approach not only enhances performance but also provides a clearer framework for problem-solving.

Example:

- The Fibonacci sequence is defined as using iterative approach:

Pseudo code:

```
F[0] = 0;
F[1] = 1;
for(int i=2;i<n;i++) {
F[n] = F[n-1] + F[n-2];
}
```

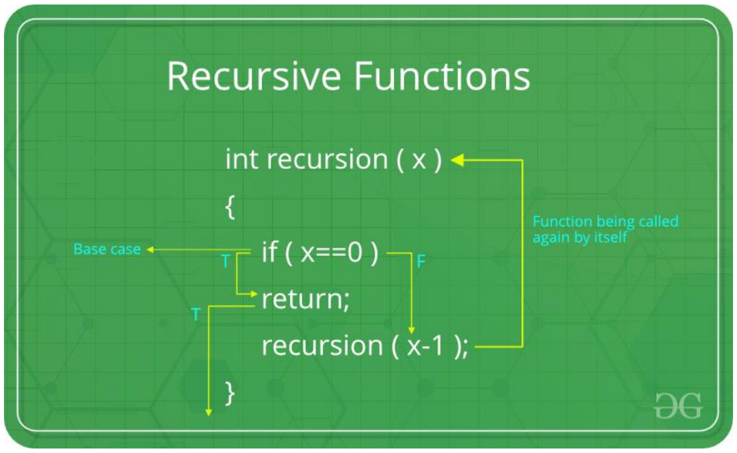


Optimal Substructure:

- Optimal substructure means an optimal solution can be built from optimal solutions of its subproblems.
- This principle is key in algorithms like dynamic programming and greedy methods for solving complex problems efficiently.
- Identifying optimal substructure aids in designing effective algorithms and deepens understanding of problem relationships.

Recursive Approach:

- The recursive approach breaks a problem into smaller subproblems, with a function calling itself until a base case is reached.
- It often faces inefficiencies due to overlapping subproblems, leading to repeated calculations and increased time complexity.
- Despite its limitations, recursion provides clarity and simplicity, making it a fundamental concept in computer science.



```
int recursion ( x )
{
    if ( x==0 )
    return;
    recursion ( x-1 );
}
```

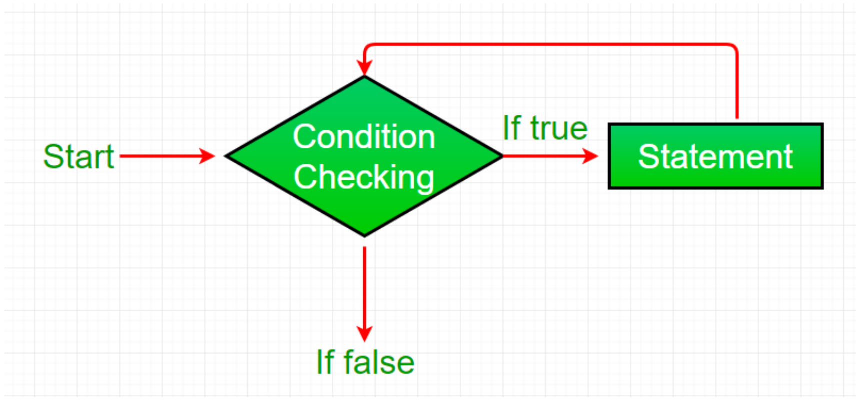
Advantages of Recursive approach:

- **Simplicity:** Recursive solutions are often more straightforward and easier to understand, as they can closely mirror the problem's structure**.**

- **Reduction of Code:** Recursive solutions can lead to more concise code, reducing the need for complex loop structures.
- **Flexibility:** Recursion can be easily adapted for various problem-solving approaches, such as depth-first search in graphs.

Iterative Approach:

- The iterative approach solves problems through repeated execution of instructions using loops, enhancing efficiency by avoiding recursive overhead.
- It is more memory-efficient than recursion, as it maintains a single state without extra stack space.
- This method is often simpler and easier to debug, making it practical for many programming tasks, especially when performance matters.



Advantages of Iteration

- **Speed**: Iterative methods generally have lower overhead and faster execution times.
- **Simplicity**: Code is often shorter and cleaner, making it easier to analyze time and space complexity.

Overlapping Subproblems:

- Both iterative and recursive dynamic programming methods can struggle with larger inputs, leading to significant performance issues or even system crashes due to excessive recursion depth or time complexity.
- By identifying overlapping subproblems, dynamic programming allows us to store and reuse previously computed solutions, effectively reducing the number of calculations needed for similar tasks.
- This approach of reusing previous results not only enhances overall performance but also ensures greater system reliability, making it feasible to tackle larger datasets without sacrificing efficiency.

Summary:

- Dynamic Programming is a powerful technique for optimization problems.
- The Fibonacci sequence can be computed using both recursive and iterative methods, with the iterative approach being more efficient.

- Understanding the trade-offs between recursion and iteration is crucial for effective problem-solving in DP.

Applications/Use Cases:

- Logistics and Supply Chain: Dynamic programming optimizes delivery routes and inventory management to minimize costs and maximize efficiency.
- Finance: It's used for portfolio optimization, helping investors maximize returns while managing risk constraints.
- Telecommunications: Algorithms for network routing utilize dynamic programming to efficiently manage data packets and reduce latency.

Recursive vs Iterative Approaches:

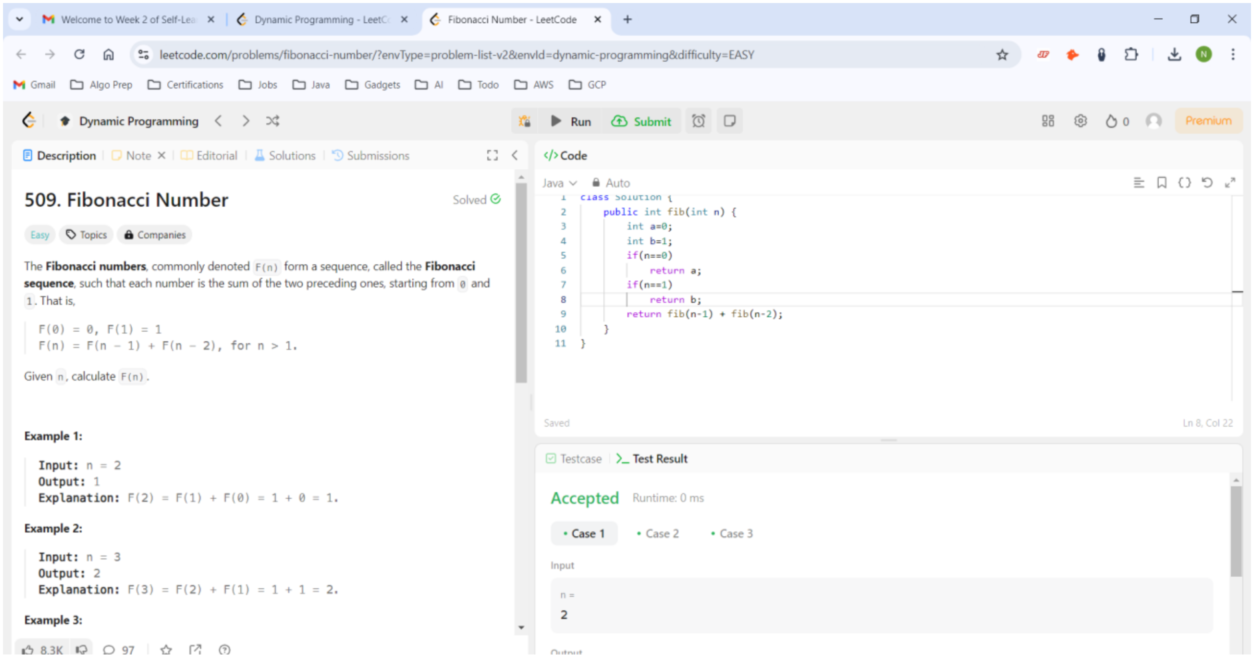
- Recursive approaches break problems into smaller subproblems by calling themselves, while iterative approaches use loops to build solutions from the bottom up.
- Recursive methods can lead to excessive function calls, increasing time complexity and risking stack overflow, whereas iterative methods are generally faster and avoid this overhead.
- Recursive approaches consume more stack space due to each function call, while iterative methods can be optimized to use less memory by storing only necessary values.
- Recursive solutions are often simpler and easier to understand for naturally recursive problems, while iterative solutions may be more complex and require more lines of code.

Advantages of using dynamic programming over naive recursion:

- Dynamic programming reduces time complexity by storing results of overlapping subproblems, eliminating redundant calculations common in naive recursion.
- It enhances efficiency, allowing algorithms to tackle larger problems in a reasonable timeframe, especially in optimization.
- This approach offers a structured way to solve problems, facilitating clearer solutions and better understanding of subproblem relationships.

Problems 509:

By using the videos examples I tried to solve this problem and applied the same logic.



leetcode.com/problems/fibonacci-number/?envType=problem-list-v2&envId=dynamic-programming&difficulty=EASY

509. Fibonacci Number — Solved — Easy

The **Fibonacci numbers**, commonly denoted `F(n)` form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given `n`, calculate `F(n)`.

Example 1:

```
Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Example 2:

```
Input: n = 3
Output: 2
Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.
```

Example 3:

```java
class Solution {
    public int fib(int n) {
        int a=0;
        int b=1;
        if(n==0)
            return a;
        if(n==1)
            return b;
        return fib(n-1) + fib(n-2);
    }
}
```

Accepted — Runtime: 0 ms

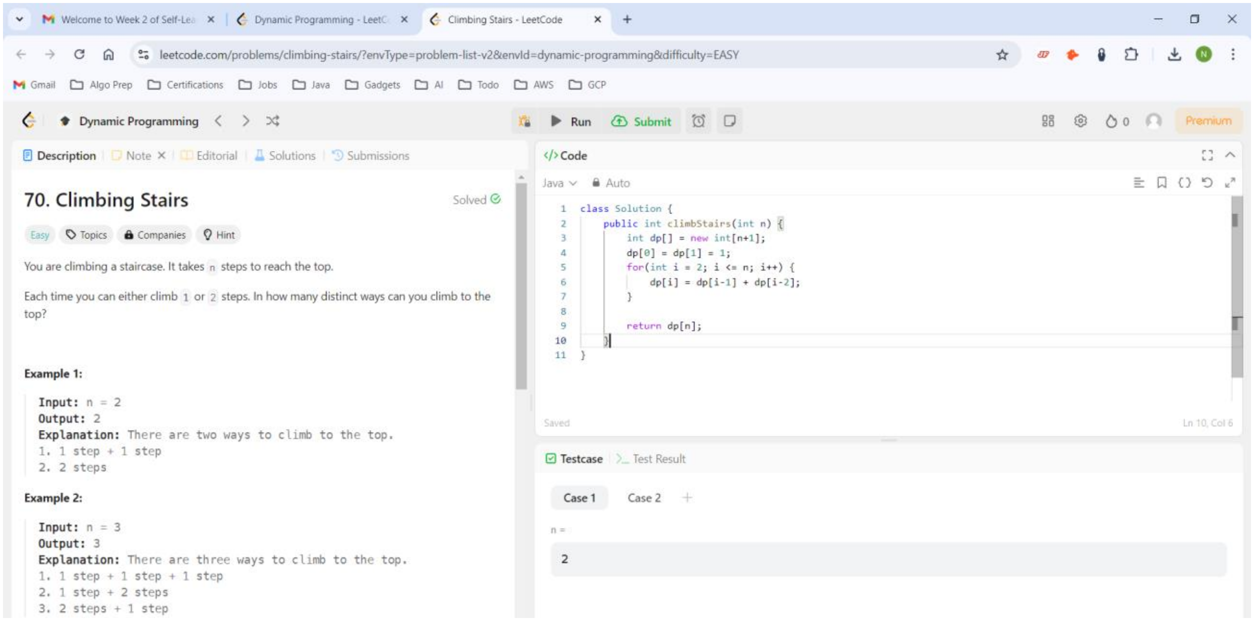
Case 1 · Case 2 · Case 3

Input

n =
2

Problem 70:

After watching the tutorials, I was able to solve the problem using the same method. I have used the iterative approach and we can able to find out the number of ways to climb the given steps.



leetcode.com/problems/climbing-stairs/?envType=problem-list-v2&envId=dynamic-programming&difficulty=EASY

70. Climbing Stairs — Solved — Easy

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

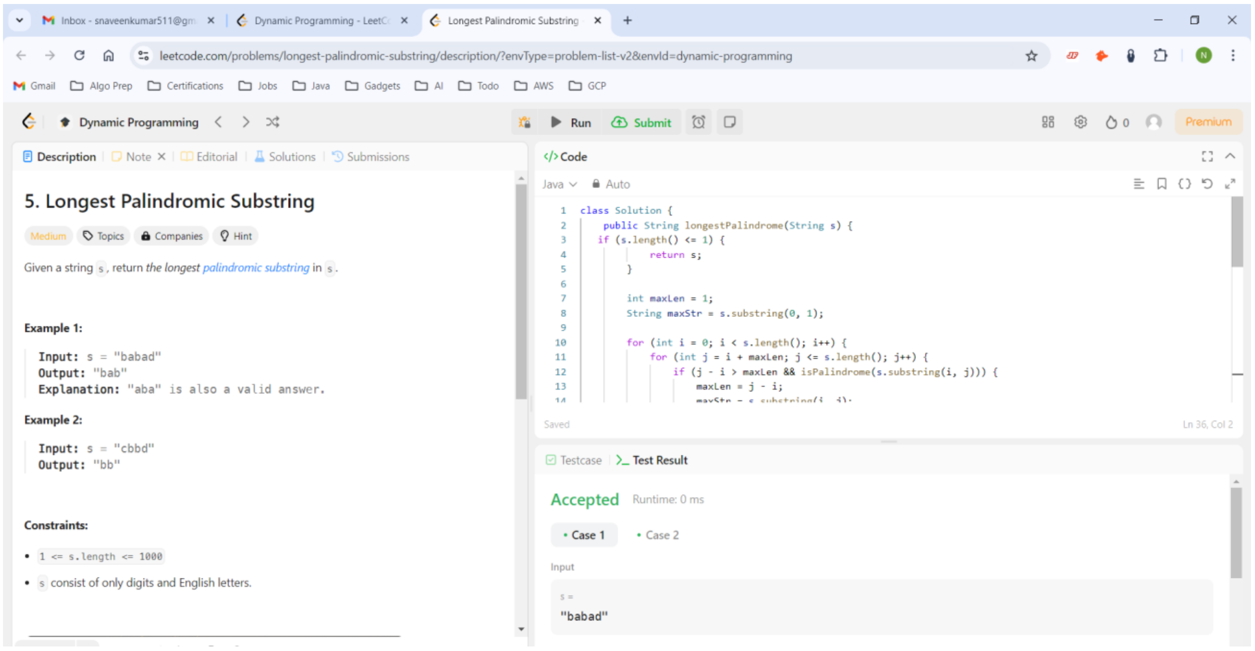
```java
class Solution {
    public int climbStairs(int n) {
        int dp[] = new int[n+1];
        dp[0] = dp[1] = 1;
        for(int i = 2; i <= n; i++) {
            dp[i] = dp[i-1] + dp[i-2];
        }

        return dp[n];
    }
}
```

Testcase — Case 1 · Case 2

n =
2

Problem 5:

To solve this I first tried using the same method but I was facing an issue in counting the palindromes, then I tried to implement using the another method. Later I tried using that also in another method and by using the recursion I'm counting the palindromes and return the count.



5. Longest Palindromic Substring

Medium

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

```java
class Solution {
    public String longestPalindrome(String s) {
   if (s.length() <= 1) {
            return s;
        }

        int maxLen = 1;
        String maxStr = s.substring(0, 1);

        for (int i = 0; i < s.length(); i++) {
            for (int j = i + maxLen; j <= s.length(); j++) {
                if (j - i > maxLen && isPalindrome(s.substring(i, j))) {
                    maxLen = j - i;
```

Accepted Runtime: 0 ms

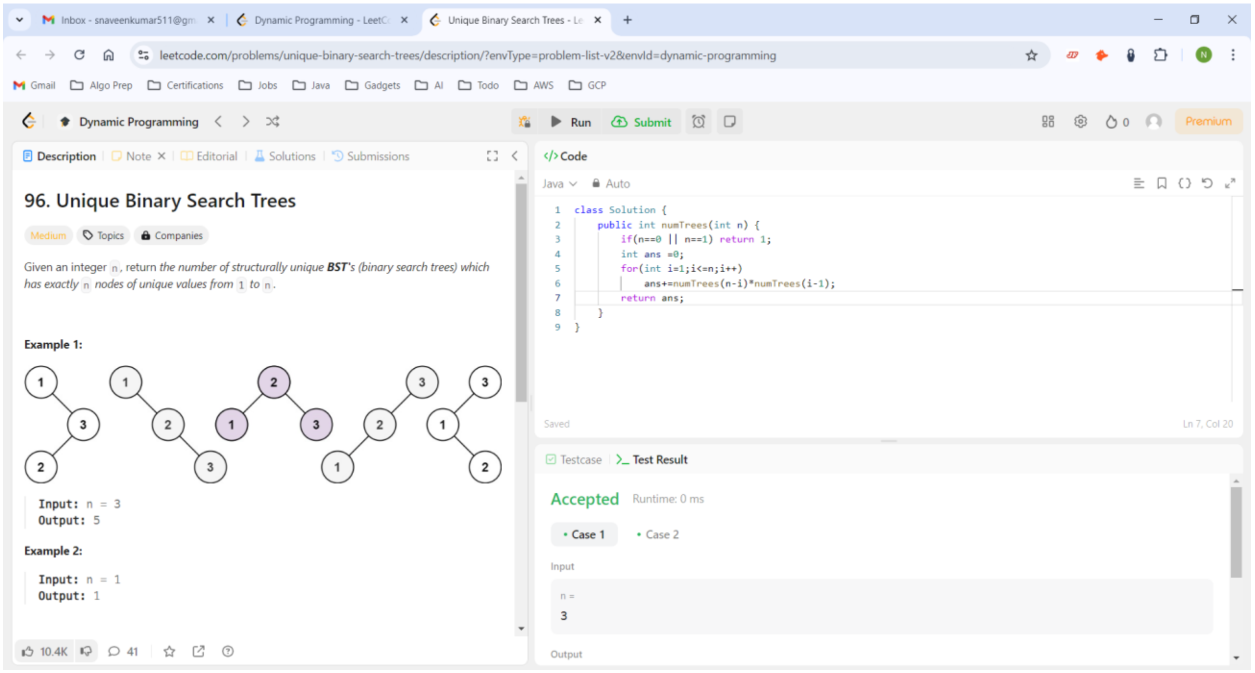
Case 1 Case 2

Input

s =
"babad"

Problem 96:

In this problem I'm using both iterative and recursive approach to solve this one. First we will iterate till n numbers and after that by using iteration how many trees can be form on every iteration and we will follow the same on both left and right sides and will multiply them we will get the total number of trees can be form uniquely.



96. Unique Binary Search Trees

Medium

Given an integer `n`, return *the number of structurally unique* ***BST****'s (binary search trees) which has exactly* `n` *nodes of unique values from* `1` *to* `n`.

Example 1:

```
Input: n = 3
Output: 5
```

Example 2:

```
Input: n = 1
Output: 1
```

```java
class Solution {
    public int numTrees(int n) {
        if(n==0 || n==1) return 1;
        int ans =0;
        for(int i=1;i<=n;i++)
            ans+=numTrees(n-i)*numTrees(i-1);
        return ans;
    }
}
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

n =
3

Output