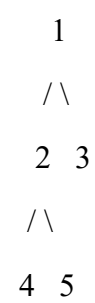
Binary Trees:

A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child.

Key Concepts:

1. **Node**: The basic unit of a binary tree, containing a value and pointers to left and right children.
2. **Root**: The top node of the tree, from which all other nodes branch.
3. **Child**: A node directly connected to another node when moving away from the root.
4. **Parent**: A node that has one or more children.
5. **Leaf**: A node that has no children (both left and right pointers are null).
6. **Height**: The number of edges on the longest path from the root to a leaf.
7. **Depth**: The number of edges from the root to a particular node.
8. **Subtree**: A tree consisting of a node and its descendants.

Example:



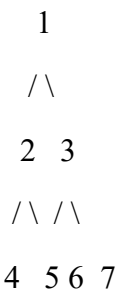
- Root: 1
- Left child of 1: 2
- Right child of 1: 3
- Left child of 2: 4
- Right child of 2: 5

Types

Full Binary Tree:

A full binary tree is where each node has either 0 or 2 children.

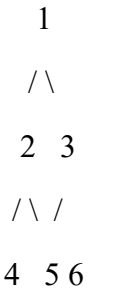
Every node has either 2 children or none.



Complete Binary Tree:

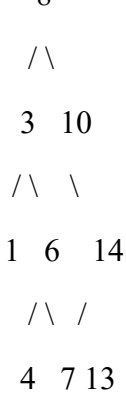
A complete binary tree has all levels fully filled except possibly the last level, and the last level is filled from left to right.

All levels are filled except the last one, where nodes are filled from the left.



Binary Search Tree (BST):

A binary search tree is a tree in which the left child of a node contains only nodes with values less than the node's value, and the right child only contains nodes with values greater than the node's value.



Tree Traversal

Tree traversal is the process of visiting each node in a tree data structure exactly once in a systematic way. There are several types of tree traversal methods, primarily categorized into Depth-First Traversal and Breadth-First Traversal.

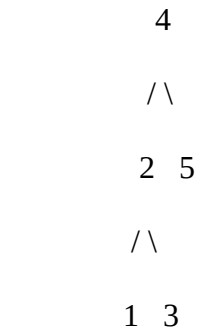
1. Depth-First Traversal (DFT):

In depth-first traversal, we go as deep as possible along a branch before backtracking.

A. In-order Traversal (Left → Root → Right):

In this traversal, the left subtree is visited first, then the root node, and finally the right subtree. This traversal is commonly used in binary search trees (BSTs) to retrieve nodes in non-decreasing order.

For example, consider the following binary tree:

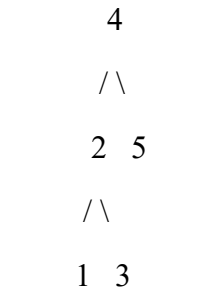


In-order Traversal: 1 → 2 → 3 → 4 → 5

B. Pre-order Traversal (Root → Left → Right):

In this traversal, the root node is visited first, then the left subtree, and finally the right subtree. This is useful when you want to copy the tree or evaluate expressions.

Using the same tree:



Pre-order Traversal: 4 → 2 → 1 → 3 → 5

C. Post-order Traversal (Left → Right → Root):

Here, we visit the left subtree first, then the right subtree, and finally the root node. This is often used to delete the tree or to evaluate postfix expressions.

Using the same tree:

```
    4
   / \
  2   5
 / \
1   3
```

Post-order Traversal: 1 → 3 → 2 → 5 → 4

2. Breadth-First Traversal (BFT) or Level-order Traversal:

In level-order traversal, nodes are visited level by level starting from the root, then moving left to right across the tree. This traversal is particularly useful for tree structure exploration or shortest path problems.

For the same tree:

```
    4
   / \
  2   5
 / \
1   3
```

- In-order: Left → Root → Right (useful for binary search trees)
- Pre-order: Root → Left → Right (useful for copying the tree)
- Post-order: Left → Right → Root (useful for deleting the tree)

Binary Search Tree (BST)

A **Binary Search Tree (BST)** is a type of binary tree that maintains a specific order in which:

- The left subtree of a node contains only nodes with values **less than** the node's key.

- The right subtree of a node contains only nodes with values **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

This structure allows for efficient searching, insertion, and deletion operations, all of which have an average time complexity of **O(log n)**.

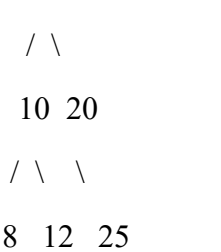
Operations on Binary Search Trees

1. **Search Operation**: The search operation is used to check whether a given key is present in the BST.

Algorithm:

- Start at the root node.
- If the key is equal to the root's value, the search is successful.
- If the key is smaller than the root's value, search the left subtree.
- If the key is larger than the root's value, search the right subtree.
- Repeat the process until the key is found or a leaf node is reached.

Time Complexity: O(log n) in a balanced tree, O(n) in the worst case (if the tree is skewed).



 Searching for 12:

- Start at 15 → 12 is smaller, go left.
- Check 10 → 12 is larger, go right.
- Check 12 → Match found.

 Insertion Operation: Insertion adds a new node to the tree while maintaining the binary search property.

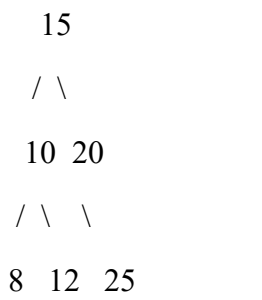
Algorithm:

- Start at the root.

- If the key is smaller than the root's value, move to the left subtree; if larger, move to the right subtree.
- Continue this process until finding an empty spot to insert the new node.

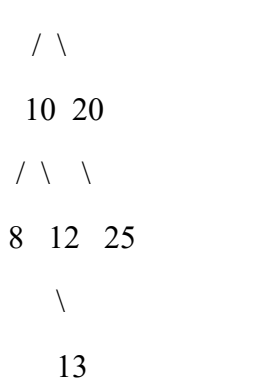
Time Complexity: O(log n) for a balanced tree, O(n) in the worst case (if the tree is skewed).

Example:
Insert 13 into the tree:



- Start at 15 → 13 is smaller, go left.
- Check 10 → 13 is larger, go right.
- Check 12 → 13 is larger, go right → Insert 13 as the right child of 12.

The resulting tree:



Deletion Operation: The deletion operation removes a node from the tree while maintaining the binary search tree property. There are three cases for deletion:

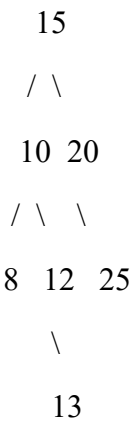
Algorithm:

- **Case 1: Node has no children (leaf node)**: Simply remove the node.
- **Case 2: Node has one child**: Remove the node and replace it with its child.

- **Case 3: Node has two children**: Find the node's **in-order predecessor** (largest value in the left subtree) or **in-order successor** (smallest value in the right subtree), replace the node with that value, and delete the predecessor/successor node.

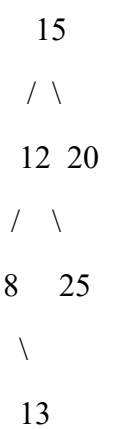
Time Complexity: O(log n) for a balanced tree, O(n) in the worst case.

Example:
Deleting node 10 from this tree:



- Node 10 has two children.
- Find the in-order successor, which is 12.
- Replace 10 with 12, then delete the original node 12.

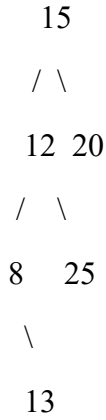
Resulting tree:



Traversal Operations: Traversing a BST involves visiting its nodes in a specific order. Common traversal methods:

- **In-order Traversal**: Visits the left subtree, root, then right subtree. In a BST, this gives nodes in ascending order.
- **Pre-order Traversal**: Visits the root, left subtree, then right subtree.
- **Post-order Traversal**: Visits the left subtree, right subtree, then the root.

Example (for the tree after deletion):



1.
    - **In-order**: 8 → 12 → 13 → 15 → 20 → 25
    - **Pre-order**: 15 → 12 → 8 → 13 → 20 → 25
    - **Post-order**: 8 → 13 → 12 → 25 → 20 → 15

Advantages of Binary Search Tree:

- Efficient searching, insertion, and deletion when the tree is balanced.
- Searching takes O(log n) in the average case.

Disadvantages:

- Performance degrades to O(n) if the tree becomes unbalanced (i.e., skewed).

Balanced BSTs:

To avoid performance degradation, balanced trees like **AVL trees** or **Red-Black trees** are used. They maintain balance by performing rotations during insertions and deletions, ensuring that the height remains logarithmic in relation to the number of nodes.

In **Binary Search Trees (BSTs)**, several challenges can arise, especially in cases where the tree becomes unbalanced. These challenges can degrade the performance of operations such as

search, insertion, and deletion. Below are some common challenges with BSTs and their solutions:

Challenges in Binary Search Trees (BSTs)

1. **Skewed Trees**:
    - **Problem**: If nodes are inserted in ascending or descending order (or in a sorted manner), the tree can become skewed to the left or right. In such cases, the BST degenerates into a **linked list** where each node has only one child, causing operations like search, insert, and delete to take **O(n)** time instead of the expected **O(log n)** for balanced trees.

```
1
 \
  2
   \
    3
     \
      4
```

In this tree, searching for node 4 would require traversing all nodes, taking O(n) time.

Solution: Use **self-balancing binary trees** like **AVL trees** or **Red-Black trees**, which automatically perform rotations to maintain balance during insertion and deletion.

2. **Duplicate Keys**:

- **Problem**: BSTs are often required to maintain unique values, as the tree's ordering property might break with duplicate values. In some use cases, duplicates may arise and the handling of such cases becomes problematic.

Solution:

- **Approach 1**: Modify the tree structure to allow duplicates by storing duplicates in a separate list or counter at each node.
- **Approach 2**: Define a rule for duplicates: always insert duplicates in the left subtree or always insert them in the right subtree.

3. **Maintaining Balance**:

- **Problem**: Even if the tree starts balanced, insertion and deletion of nodes may eventually cause the tree to become unbalanced over time, degrading performance.

Solution:

- **AVL Trees**: These are **height-balanced** trees where the heights of the left and right subtrees of any node differ by at most 1. After every insertion or deletion, rotations (single or double) are used to maintain balance. AVL trees guarantee **O(log n)** time for search, insertion, and deletion.
- **Red-Black Trees**: A type of balanced tree that relaxes the strict balancing condition of AVL trees. Red-Black trees use color coding and rotations to maintain a semi-balanced structure, providing O(log n) operations in the worst case while reducing the overhead of frequent rotations seen in AVL trees.

4. **Deletion Complexity**:
   - **Problem**: The deletion operation in a BST can be tricky, especially when the node to be deleted has two children. In this case, the tree needs to be restructured to maintain the BST properties. Incorrect deletion handling can break the tree structure.

Solution:

- The most common approach is to replace the node with its **in-order predecessor** (largest value in the left subtree) or **in-order successor** (smallest value in the right subtree), ensuring that the tree remains properly ordered after deletion.
- Use **self-balancing trees** to automatically adjust and ensure the tree remains balanced after deletions.

---

5. **Large Height in Sparse Trees**:
   - **Problem**: Even when not fully skewed, a sparse BST can develop a large height due to unbalanced insertions. This increases the time complexity of operations that depend on the height of the tree.

Solution: Implement a **balanced BST** like an **AVL tree**, **Red-Black tree**, or **B-trees** (which are useful in databases) to maintain a logarithmic height and ensure efficient operation times.

---

6. **Traversal Efficiency**:

- **Problem**: Traversing large BSTs for operations like **in-order traversal** can become inefficient if the tree is large and operations like printing all nodes or finding k-th smallest elements are required frequently.

Solution:

- Use iterative traversal methods or maintain additional metadata (e.g., size of subtrees) at each node to enhance traversal operations like finding k-th smallest elements in O(log n) time.
- **Threaded Binary Trees** can be used to optimize in-order traversals, making the process faster by threading the nodes to point to in-order successors, which can reduce the space and overhead of recursive calls.

---

7. **Handling Range Queries**:
   - **Problem**: Given a range of values [low, high], retrieving all values from a BST that fall within this range can be inefficient if the tree is not optimized for range queries.

Solution:

- Modify the tree structure or use **augmented BSTs** with additional information (e.g., subtree sizes, ranges) stored at each node to facilitate efficient range queries.
- An alternative approach could be using **segment trees** or **interval trees** for range-based operations.

---

8. **Memory Overhead in Balanced Trees**:
   - **Problem**: Self-balancing trees like AVL and Red-Black trees introduce overhead by storing additional information at each node (e.g., height in AVL trees, color in Red-Black trees) and require frequent rebalancing operations.

Solution:

- Although this overhead is unavoidable for keeping the tree balanced, Red-Black trees provide a compromise between balance and the number of rotations compared to AVL trees. In cases where memory and speed are critical, Red-Black trees are often preferred for practical implementations.

Other Advanced Solutions

1. **Splay Trees**:

- **Solution to Skewing**: A type of self-adjusting binary search tree where recently accessed elements are moved to the root using a series of tree rotations. This ensures that frequently accessed elements are quickly accessible, providing **amortized O(log n)** performance for operations like search, insertion, and deletion.

2. **Treaps (Tree + Heap)**:
   - A randomized binary search tree that combines properties of a **BST** and a **heap**. Each node contains a key (BST property) and a priority (heap property), ensuring a balanced structure with high probability, reducing the likelihood of skewing.
3. **B-trees**:
   - **Solution for Disk-based Storage**: B-trees are used in databases to handle large datasets. They minimize disk access by keeping nodes with a large number of children, ensuring that height remains low and that the number of I/O operations is minimized.

Summary of Solutions:

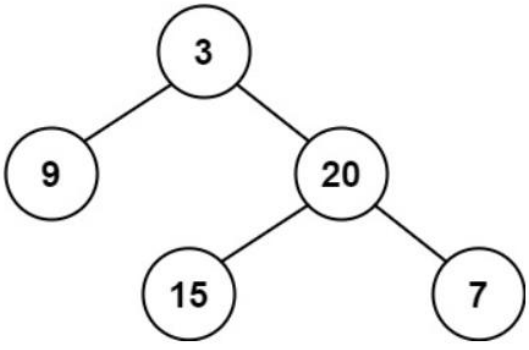
- **Self-balancing trees** (like **AVL** and **Red-Black** trees) maintain tree balance to ensure efficient operations.
- **Threaded binary trees** and **splay trees** provide alternatives for traversal and access optimization.
- **Treaps** and **B-trees** offer advanced solutions for specialized use cases, such as randomized balancing and disk-based storage optimization.

110. Balanced Binary Tree

Easy | Topics | Companies

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

Example 1:




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

```
public class BalancedBinaryTree {

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

    private int checkHeight(TreeNode node) {
        if (node == null) {
            return 0;
        }

        int leftHeight = checkHeight(node.left);
        if (leftHeight == -1) return -1;

        int rightHeight = checkHeight(node.right);
        if (rightHeight == -1) return -1;

        if (Math.abs(leftHeight - rightHeight) > 1) {
            return -1;
        }

        return Math.max(leftHeight, rightHeight) + 1;
    }
```

Key Points:

- The TreeNode class defines the structure of a node in the binary tree.
- The BalancedBinaryTree class contains the logic to check if the binary tree is balanced using the isBalanced method, which calls the recursive checkHeight method.
- The checkHeight method returns -1 if the subtree is not balanced, and the height of the subtree otherwise.
- The main method provides an example of how to create a binary tree and check if it's balanced.

144. Binary Tree Preorder Traversal

Easy | Topics | Companies

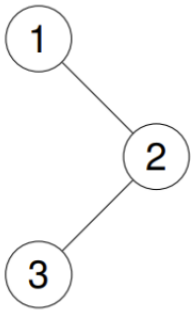
Given the `root` of a binary tree, return *the preorder traversal of its nodes' values.*

Example 1:

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

Output: `[1,2,3]`

Explanation:




```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {

        List<Integer> result = new ArrayList<>();
        preorder(root, result);
        return result;
    }

    private void preorder(TreeNode node, List<Integer> result) {
        if (node == null) return;

        result.add(node.val);
        preorder(node.left, result);
        preorder(node.right, result);
    }
}
```

Key Points:

- The preorderTraversal method initializes a result list and starts the recursive preorder method.
- The preorder method processes the current node, then recursively visits the left and right children.

145. Binary Tree Postorder Traversal

Easy | Topics | Companies

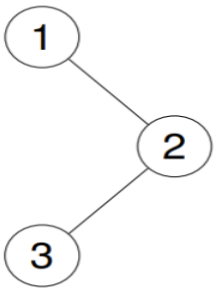
Given the `root` of a binary tree, return *the postorder traversal of its nodes' values.*

Example 1:

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

Output: `[3,2,1]`

Explanation:




```
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        postorder(root, result);
        return result;
    }

    private void postorder(TreeNode node, List<Integer> result) {
        if (node == null) return;

        postorder(node.left, result);
        postorder(node.right, result);
        result.add(node.val);

    }
}
```

Key Points:

- The postorderTraversal method initializes the result list and calls the recursive postorder method.
- The postorder method first visits the left and right children, then processes the current node, which is the characteristic of postorder traversal.

94. Binary Tree Inorder Traversal

Easy | Topics | Companies

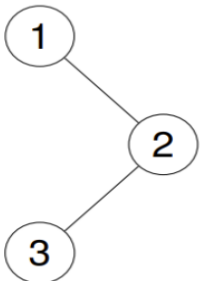
Given the `root` of a binary tree, return *the inorder traversal of its nodes' values.*

Example 1:

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

Output: `[1,3,2]`

Explanation:




```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {

        List<Integer> result = new ArrayList<>();
        inorder(root, result);
        return result;
    }

    private void inorder(TreeNode node, List<Integer> result) {
        if (node == null) return;

        inorder(node.left, result);
        result.add(node.val);
        inorder(node.right, result);
    }
}
```

Key Points:

- The inorderTraversal method initializes a result list and calls the recursive inorder method.
- The inorder method first visits the left child, processes the current node, and then visits the right child.

Dynamic Programming:

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into overlapping subproblems and solving each subproblem only once. The key idea is to avoid redundant calculations by storing intermediate results and reusing them whenever needed. It's particularly useful in problems where the same subproblems are solved multiple times in naive recursive approaches.

There are two main strategies in DP:

1. Memoization (Top-Down Approach): Solve the problem recursively but store the results of each subproblem in a table or array so that if the same subproblem is encountered again, its value can be retrieved without recomputation.
2. Tabulation (Bottom-Up Approach): Build the solution iteratively by solving smaller subproblems first and using their results to solve larger subproblems. This avoids the need for recursion and minimizes stack overhead.

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into overlapping subproblems and solving each subproblem only once. The key idea is to avoid redundant calculations by storing intermediate results and reusing them whenever needed. It's particularly useful in problems where the same subproblems are solved multiple times in naive recursive approaches.

There are two main strategies in DP:

1. **Memoization (Top-Down Approach)**: Solve the problem recursively but store the results of each subproblem in a table or array so that if the same subproblem is encountered again, its value can be retrieved without recomputation.
2. **Tabulation (Bottom-Up Approach)**: Build the solution iteratively by solving smaller subproblems first and using their results to solve larger subproblems. This avoids the need for recursion and minimizes stack overhead.

1. Recursive Approach (Naive Recursion)

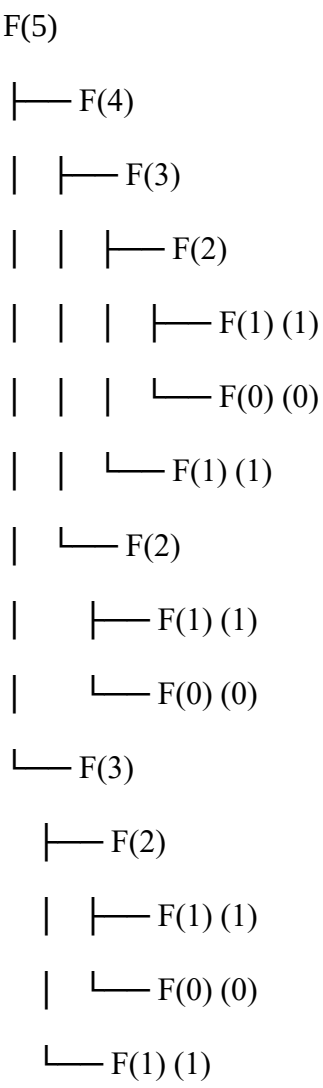
The simplest way to calculate Fibonacci numbers is using a **recursive approach** that directly mirrors the mathematical definition:

```
def fibonacci_recursive(n):

    if n <= 1:

        return n

    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Problems with Naive Recursion:

- **Exponential Time Complexity**: The time complexity is **O(2^n)**, because for every nnn, the function makes two recursive calls. This leads to repeated recalculation of the same Fibonacci numbers.
- **Redundant Calculations**: For example, calculating F(5)F(5)F(5) requires recalculating F(4)F(4)F(4) and F(3)F(3)F(3) multiple times.

Recursive Fibonacci Tree for F(5)F(5)F(5):



1. Recursive Approach (Naive Recursion)

The simplest way to calculate Fibonacci numbers is using a **recursive approach** that directly mirrors the mathematical definition:

python

Copy code

```
def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Problems with Naive Recursion:

- **Exponential Time Complexity**: The time complexity is **O(2^n)**, because for every nnn, the function makes two recursive calls. This leads to repeated recalculation of the same Fibonacci numbers.
- **Redundant Calculations**: For example, calculating F(5)F(5)F(5) requires recalculating F(4)F(4)F(4) and F(3)F(3)F(3) multiple times.

Recursive Fibonacci Tree for F(5)F(5)F(5):

scss

Copy code

```
F(5)
├── F(4)
│   ├── F(3)
│   │   ├── F(2)
│   │   │   ├── F(1) (1)
│   │   │   └── F(0) (0)
│   │   └── F(1) (1)
│   └── F(2)
│       ├── F(1) (1)
│       └── F(0) (0)
```

└── F(3)

    ├── F(2)

    │   ├── F(1) (1)

    │   └── F(0) (0)

    └── F(1) (1)

In this tree, the same subproblems, like F(3) and F(2), are calculated multiple times, leading to inefficiency.

Time Complexity:

- **O(2^n)** because of the excessive recursive calls.

Space Complexity:

- **O(n)** due to the function call stack in recursion.

---

2. Iterative Approach

To avoid the inefficiencies of recursion, we can compute Fibonacci numbers iteratively. This approach builds the Fibonacci sequence in a **bottom-up** manner.

python

Copy code

```
def fibonacci_iterative(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for i in range(2, n+1):
        a, b = b, a + b
    return b
```

Iterative Approach Explanation:

- **Initialize**: Start with the first two Fibonacci numbers, F(0)=0F(0) = 0F(0)=0 and F(1)=1F(1) = 1F(1)=1.
- **Loop**: For each number from 222 to nnn, update two variables a and b to store the last two Fibonacci values.
- **Return**: After the loop, b will hold the value of F(n)F(n)F(n).

Time Complexity:

- **O(n)** because each Fibonacci number is computed once.

Space Complexity:

- **O(1)** because we only need two variables to store the last two Fibonacci values.

---

3. Dynamic Programming Approach (Memoization)

The **memoization** technique is a form of dynamic programming where we store results of subproblems to avoid redundant calculations. This combines the benefits of recursion with efficient time complexity.

python

Copy code

```
def fibonacci_memoization(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memoization(n-1, memo) + fibonacci_memoization(n-2, memo)
    return memo[n]
```

Memoization Breakdown:

- **Top-Down**: Solve the problem recursively but store intermediate results in a memoization table (dictionary) so that repeated subproblems are not recalculated.

- **Recursive but Efficient**: Memoization ensures that each subproblem is solved only once, reducing the time complexity to **O(n)**.

Time Complexity:

- **O(n)** because each Fibonacci number is calculated only once and stored in memory.

Space Complexity:

- **O(n)** due to the memoization table and the recursive call stack.

---

4. Dynamic Programming Approach (Tabulation)

Tabulation is a bottom-up dynamic programming approach. Instead of starting from the top and solving recursively, you start from the base cases and build up to the final solution.

python

Copy code

```
def fibonacci_tabulation(n):
    if n <= 1:
        return n
    table = [0] * (n+1)
    table[1] = 1
    for i in range(2, n+1):
        table[i] = table[i-1] + table[i-2]
    return table[n]
```

Tabulation Breakdown:

- **Bottom-Up**: Start with base cases F(0)=0F(0) = 0F(0)=0 and F(1)=1F(1) = 1F(1)=1, then iteratively compute F(2)F(2)F(2) to F(n)F(n)F(n).
- **Table**: Use an array to store the values of Fibonacci numbers from 0 to nnn.

Time Complexity:

- **O(n)** because each Fibonacci number is computed once.

Space Complexity:

- **O(n)** for the table.

Space Optimization:

Since each Fibonacci number only depends on the previous two, we can reduce space complexity to **O(1)** by using two variables instead of a table.

python

Copy code

```
def fibonacci_space_optimized(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for i in range(2, n+1):
        a, b = b, a + b
    return b
```

Comparison of Approaches

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursive	O(2^n)	O(n)	Simple, intuitive, mirrors math definition	Extremely slow due to redundant calls
Iterative	O(n)	O(1)	Fast, memory efficient	Lacks the elegance of recursion
Memoization	O(n)	O(n)	Combines recursion with efficiency	Requires additional memory for memo table
Tabulation	O(n)	O(n) (or O(1))	Bottom-up, no recursion depth issues	Requires space for the table

---

Conclusion

Dynamic programming provides a powerful way to solve problems that exhibit overlapping subproblems and optimal substructure, like the Fibonacci sequence. Both **memoization** and **tabulation** improve upon naive recursion, turning an exponential-time solution into a linear-time solution. Depending on the problem requirements, space optimization techniques can further reduce the memory overhead while maintaining efficiency.

70. Climbing Stairs

Easy | Topics | Companies | Hint

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

```
class Solution {
    public int climbStairs(int n) {
        if (n == 1) return 1;

        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;

        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];

    }
}
```

Key Points:

- This problem is solved using dynamic programming.
- dp[i] represents the number of distinct ways to reach step i.
- The recurrence relation is: dp[i] = dp[i - 1] + dp[i - 2] because from step i, you can either come from step i-1 or step i-2.

70. Climbing Stairs

Easy | Topics | Companies | Hint

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
```

```
class Solution {
    public int fib(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;

        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;

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

        return dp[n];
    }
}
```

Key Points:

- The Fibonacci sequence is solved using dynamic programming.
- dp[i] stores the Fibonacci value for index i, with the recurrence relation dp[i] = dp[i-1] + dp[i-2].

121. Best Time to Buy and Sell Stock

Easy | Topics | Companies

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1
= 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must
buy before you sell.
```

```
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;

        for (int price : prices) {
            if (price < minPrice) {
                minPrice = price;
            } else if (price - minPrice > maxProfit) {
                maxProfit = price - minPrice;
            }
        }

        return maxProfit;

    }
}
```

Key Points:

- The algorithm iterates through the prices array, tracking the minimum price seen so far and calculating the maximum profit possible at each step.
- The maximum profit is the difference between the current price and the minimum price before that.