

Data Structures and Algorithms Practice

People Tech Group Week 2(L1 Batch 2)

Name: Shrestha Sai Laxmi Yadavilli
Email: shrestha.yadavilli22@gmail.com

Table of Contents

1. Introduction	2
1.1 Topics Covered	2
1.2 Objectives of the Week's Tasks	2
2. Binary Trees	3
2.1 Basic Operations in a Binary Tree.....	3
2.2 Properties of a Binary Tree	9
2.3 Types of Binary Tree	9
2.4 Applications, Benefits and Limitations of a Binary Tree	10
2.5 Key Takeaways from Resources	10
3. Binary Search Trees	11
3.1 Basic Operations in a Binary Search Tree	11
3.2 Applications, Benefits and Limitations of a Binary Search Tree	16
3.3 Key Takeaways from Resources	17
4. Binary Tree and Binary Search Tree Problems on Leetcode	18
4.1 Problem 1: Maximum Depth of a Binary Tree	18
4.2 Problem 2: Count Complete Tree Nodes.....	19
4.3 Problem 3: Second Minimum Node in a Binary Tree	22
4.4 Problem 4: Convert Sorted Array to Binary Search Tree	24
5. Dynamic Programming	26
5.1 Characteristics of Dynamic Programming algorithm	26
5.2 Dynamic Programming vs Recursion	26
5.3 Techniques to solve Dynamic Programming Problems.....	26
5.4 How to Solve a Dynamic Programming Problem?	27
5.5 Greedy Approach Vs Dynamic Programming.....	27
5.6 Applications, Benefits and Limitations.....	27
5.7 Key Takeaways from Resources	28
6. Dynamic Programming Problems on Leetcode	29
6.1 Problem 1: Fibonacci Number	29
6.2 Problem 2: Longest Common Subsequence	30
6.3 Problem 3: Longest Palindromic Subsequence	32
6.4 Problem 4: Coin Change	33
7. References.....	36

1. Introduction

In this document, fundamental concepts and advanced applications of binary trees, binary search trees, and dynamic programming were explored. Emphasis was placed on understanding the operations of these data structures, such as insertion, deletion, traversal, and searching, which are crucial for establishing a strong foundation in computer science. Various coding problems from LeetCode were analyzed, demonstrating how dynamic programming and tree structures can be applied to solve complex algorithmic challenges. The primary objective this week was to ensure that a comprehensive understanding of these topics is achieved and that knowledge was applied to practical coding problems.

1.1 Topics Covered

- **Binary Trees:** The structure, properties, and operations of binary trees are introduced in this topic, including methods for insertion, deletion, and traversal.
- **Binary Search Trees (BSTs):** Binary search trees are examined, highlighting how they differ from binary trees, along with their operations and practical applications.
- **Binary Tree and Binary Search Tree Problems on LeetCode:** Practical problem-solving using tree-based data structures are demonstrated, including problems such as Maximum Depth of a Binary Tree, Count Complete Tree Nodes, and others.
- **Dynamic Programming:** An in-depth look into dynamic programming is provided, covering its characteristics, distinctions from recursion, techniques for solving problems, and comparisons with the greedy approach.
- **Dynamic Programming Problems on LeetCode:** Dynamic programming techniques will be applied to solve various problems, including Fibonacci Number, Longest Common Subsequence, Longest Palindromic Subsequence, and Coin Change.

1.2 Objectives of the Week's Tasks

- To gain a comprehensive understanding of binary trees, binary search trees, and dynamic programming, in-depth exploration of their concepts, properties, and operations.
- To document key takeaways, advantages, limitations, and applications of these data structures and algorithms.
- To implement theoretical knowledge through LeetCode exercises involving tree structures and dynamic programming problems.
- To analyze and document problem-solving approaches by noting the challenges encountered while solving coding problems, along with the strategies and techniques used to arrive at solutions.

2. Binary Trees

A Tree is a structure where there is a root node, and that root node has child nodes and each of these child nodes have child nodes of their own. A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, which are referred to as the **left child** and the **right child**. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.

The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

Representation of Binary Tree

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child

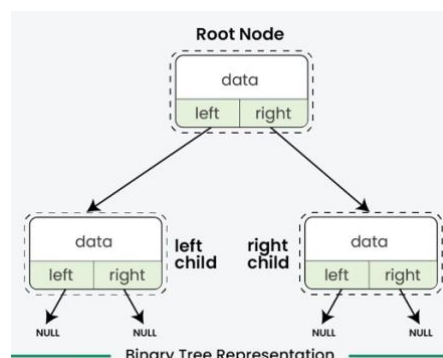


Figure 1 Binary Tree Representation(Source: [geeksforgeeks.com](https://www.geeksforgeeks.com))

Terminologies in Binary Trees

- **Node:** The fundamental unit containing data and links to other nodes.
- **Root:** The topmost node in the tree which is the starting point and has no parent node.
- **Parent Node:** A node that has one or more child nodes.
- **Child Node:** A node that descends from another node.
- **Leaf Node:** A node with no children.
- **Internal Node:** A node with at least one child and not a leaf or the root.
- **Depth of a Node:** Number of edges from the root to the node.
- **Height of a Tree:** Number of edges on the longest path from the node to a leaf.

2.1 Basic Operations in a Binary Tree

2.1.1 Creating a Binary Tree

Here is the python code for creating a binary tree with values 2,3,4 and 5 where 2 is the root.

```
class Node:
    def __init__(self, d):
        self.data = d
        self.left = None
        self.right = None

# Creating nodes
root = Node(2)
node1 = Node(3)
node2 = Node(4)
```

```

node3 = Node(5)

# Building the tree
root.left = node1
root.right = node2
node1.left = node3

```

The tree will have 2 in the root, 3 on the left of 2, 4 on the right of 2, and 5 on the left of 3.

2.1.2 Traversal in a Binary Tree

Traversal in a binary tree can be done using breadth first search or depth first search. Using depth first search there could be three types of traversals: In-Order, Pre-Order, Post-Order.

In-Order: First the program traverses to the left node, then the root node, and then the right node. Time Complexity: $O(n)$, Space Complexity: $O(n)$

```

def inorder(node):
    if node:
        inorder(node.left)
        print(node.data, end=' ')
        inorder(node.right)

```

Pre-Order: First the program traverses to the root node, then the left node, and then the right node. Time Complexity: $O(n)$, Space Complexity: $O(n)$

```

def preorder(node):
    if node:
        print(node.data, end=' ')
        preorder(node.left)
        preorder(node.right)

```

Post-Order: First the program traverses to the left node, then the right node, and then the root node. Time Complexity: $O(n)$, Space Complexity: $O(n)$

```

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.data, end=' ')

```

Using Breadth-first search, level-order traversal could be done. This traversal technique, explores all nodes at the present depth before moving on to nodes at the next level.

```

from collections import deque

def level_order(root):

```

```

if not root:
    return
queue = deque([root])
while queue:
    node = queue.popleft()
    print(node.data, end=' ')
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

```

Example:

```

# Creating the tree
root = Node(2)
root.left = Node(3)
root.right = Node(4)
root.left.left = Node(5)

print("In-Order Traversal:")
inorder(root)
print("\nPre-Order Traversal:")
preorder(root)
print("\nPost-Order Traversal:")
postorder(root)
print("\nLevel Order Traversal:")
level_order(root)

# Output:
# In-Order Traversal:
# 5 3 2 4
# Pre-Order Traversal:
# 2 3 5 4
# Post-Order Traversal:
# 5 3 4 2
# Level Order Traversal:
# 2 3 4 5

```

2.1.3 Insertion in a Binary Tree

First available position in level order needs to be found to insert a new node in a binary tree. If the tree is empty, then first a root node is created, then subsequent insertions involve iteratively searching for an empty place at each level of the tree. When an empty left or right child is found then new node is inserted there. By convention, insertion always starts with the left child node.

Time Complexity: $O(n)$, Space Complexity: $O(n)$

```
from collections import deque

def insert(root, key):
    new_node = Node(key)
    if not root:
        return new_node
    queue = deque([root])
    while queue:
        temp = queue.popleft()
        if not temp.left:
            temp.left = new_node
            break
        else:
            queue.append(temp.left)
        if not temp.right:
            temp.right = new_node
            break
        else:
            queue.append(temp.right)
    return root
```

Example:

```
root = Node(2)
root.left = Node(3)
root.right = Node(4)
root.left.left = Node(5)

print("Before Insertion:")
inorder(root)
root = insert(root, 6)
print("\nAfter Insertion:")
inorder(root)
```

```
# Output:
# Before Insertion:
# 5 3 2 4
# After Insertion:
# 5 3 6 2 4
```

2.1.4 Searching in a Binary Tree

Searching for a value involves traversing the tree until the value is found. Searching could be done using Breadth-first search or depth-first search. In **DFS**, we start from the **root** and explore the depth nodes first. In BFS, we explore all the nodes at the present depth level before moving on to the nodes at the next level. This process is continued until the node with the desired value is found or we reach the end of the tree. If the tree is empty or the value isn't found after exploring all possibilities, we conclude that the value does not exist in the tree.

Time Complexity: $O(n)$, Space Complexity: $O(n)$

```
def search(node, key):
    if not node:
        return False
    if node.data == key:
        return True
    return search(node.left, key) or search(node.right, key)
```

```
found = search(root, 6)
print(f"Value 6 {'found' if found else 'not found'} in the tree.")

# Output:
# Value 6 found in the tree.
```

2.1.5 Deletion in a Binary Tree

Deletion involves removing a node while maintaining the tree structure.

Time Complexity: $O(n)$, Space Complexity: $O(n)$

Steps:

1. Find the node to be deleted.
2. Find the deepest and rightmost node.
3. Replace the target node's value with the deepest node's value.
4. Delete the deepest node.

```
def delete(root, key):
    if not root:
        return None
    if not root.left and not root.right:
        if root.data == key:
```

```

        return None
    else:
        return root
key_node = None
queue = deque([root])
while queue:
    temp = queue.popleft()
    if temp.data == key:
        key_node = temp
    if temp.left:
        queue.append(temp.left)
    if temp.right:
        queue.append(temp.right)
if key_node:
    x = temp.data
    delete_deepest(root, temp)
    key_node.data = x
return root

def delete_deepest(root, d_node):
    queue = deque([root])
    while queue:
        temp = queue.popleft()
        if temp is d_node:
            temp = None
            return
        if temp.left:
            if temp.left is d_node:
                temp.left = None
                return
            else:
                queue.append(temp.left)
        if temp.right:
            if temp.right is d_node:
                temp.right = None
                return
            else:
                queue.append(temp.right)

```

Example:


```

print("Before Deletion:")
inorder(root)

root = delete(root, 3)

print("\nAfter Deletion:")
inorder(root)

# Output:
# Before Deletion:
# 5 3 6 2 4
# After Deletion:
# 5 6 2 4

```

2.2 Properties of a Binary Tree

Following are the properties of binary trees:

- The maximum number of nodes at level 'l' of a binary tree is 2^l .
- The Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$
- In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $\log_2(N+1)$
- A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels
- In a Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children
- In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges, then $e = n - 1$

2.3 Types of Binary Tree

Binary tree can be classified into multiple types based on the number of children, completion of levels and node values.

On the basis of Number of Children:

- **Full Binary Tree:** A full binary tree is a binary tree with either zero or two child nodes for each node.
- **Degenerate Binary Tree:** Every non-leaf node has just one child in a binary tree. The tree effectively transforms into a linked list as a result, with each node linking to its single child.
- **Skewed Binary Tree:** A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.

On the basis of Completion of Levels:

- **Complete Binary Tree:** A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.
- **Perfect Binary Tree:** A perfect binary tree is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.
- **Balanced Binary Tree:** A binary tree is balanced if the height of the tree is $O(\log n)$ where n is the number of nodes.

On the basis of Node Values:

- **Binary Search Tree:** Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node.
- **AVL Tree:** An AVL tree is defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.
- **Red Black Tree:** Red Black Trees are a type of balanced binary search tree that use a set of rules to maintain balance, ensuring logarithmic time complexity for operations like insertion, deletion, and

searching, regardless of the initial shape of the tree. Red Black Trees are self-balancing, using a simple color-coding scheme to adjust the tree after each modification.

- **B Tree:** B-Trees handle large data efficiently by storing multiple keys in each node, resulting in a tree with fewer levels. This makes search and insertion faster, making them ideal for storage systems with slower data access, like hard drives and flash memory.
- **B+ Tree:** A B+ Tree is a variation of the B-Tree where data pointers are stored only in the leaf nodes. Leaf nodes are linked for ordered access, while internal nodes guide the search. This structure makes data retrieval faster and more efficient.
- **Segment Tree:** Segment trees allow quick querying and updating of intervals in an array, ideal for tasks like finding the sum, minimum, or maximum of a range. They efficiently handle range queries with a time complexity of $O(\log n)$.

2.4 Applications, Benefits and Limitations of a Binary Tree

Applications:

- A type of Binary trees is Binary Search Trees (BSTs) which allow for efficient searching, insertion, and deletion operations, with an average time complexity of $O(\log n)$, making them suitable for managing sorted data.
- Algorithms like Binary Search Tree Sort and Heap Sort utilize binary tree structures to perform efficient sorting operations.
- Binary trees are used in database indexing to optimize search queries and data retrieval, enabling efficient management of large datasets.
- Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

Benefits:

- Binary trees require lesser memory as compared to other tree data structures, therefore memory-efficient.
- Binary trees can be traversed in different orders (in-order, pre-order, post-order), which allows for operations such as sorting, searching, or expression evaluation.
- Binary trees are relatively easy to implement and understand as each node has at most two children, left child and right child.
- Binary trees require only two pointers per node, making them memory-efficient for hierarchical data representation compared to more complex structures like multi-way trees.
- Insertions and deletions can be performed efficiently in $O(\log n)$ time, ensuring that binary trees remain a dynamic and flexible data structure.

Limitations:

- The binary structure restricts each node to at most two children, which may not be ideal for scenarios where more child nodes are needed.
- If a binary tree becomes unbalanced (e.g., one subtree is significantly deeper than the other), search operations can degrade to $O(n)$ time, reducing the benefits of binary search.
- Although binary trees are generally space-efficient, each node requires storage for two child pointers, which can result in overhead, especially in large datasets.
- In cases where the binary tree becomes skewed (degenerate), it essentially becomes a linked list, resulting in $O(n)$ search, insertion, and deletion times, negating the advantage of binary trees.

2.5 Key Takeaways from Resources

- Binary trees support fast search, insertion, and sorting algorithms, typically in $O(\log n)$ time when balanced.
- Used in search algorithms, database systems, file systems, compression (Huffman coding), decision trees, and game AI.
- Unbalanced trees can degrade performance to $O(n)$, requiring complex balancing techniques like AVL or Red-Black trees.
- Binary trees require only two pointers per node, but their structure can be limited for applications needing more than two child nodes per parent.

3. Binary Search Trees

A Binary Search Tree (BST) is a hierarchical data structure where each node has at most two children. The left child contains values smaller than the parent node, and the right child contains values larger than the parent node (Left child < Parent < Right child). This structure allows efficient search, insertion, and deletion operations.

The left subtree of a node contains only nodes with keys less than the node's key. The right subtree contains only nodes with keys greater than the node's key. Both subtrees must also be BSTs and duplicates are not allowed in Binary Search Trees but can be handled differently in some cases.

3.1 Basic Operations in a Binary Search Tree

3.1.1 Searching for a Node in Binary Search Tree

The search operation begins at the root and compares the target key with the current node's key:

- If the target key matches the node's key, the search is successful.
- If the target is smaller, move to the left subtree.
- If the target is larger, move to the right subtree.
- Repeat this process until the target key is found or the tree is fully traversed (which would return NULL if the key doesn't exist).

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Function to search for a key in a BST
def search(root, key):
    # Base case: root is None or key is found
    if root is None or root.key == key:
        return root

    # If the key is greater, search in the right subtree
    if root.key < key:
        return search(root.right, key)

    # Otherwise, search in the left subtree
    return search(root.left, key)

# Driver code to test the search function
root = Node(50)
root.left = Node(30)
root.right = Node(70)
root.left.left = Node(20)
root.left.right = Node(40)
root.right.left = Node(60)
```

```

root.right.right = Node(80)

# Searching for different keys
print("Found" if search(root, 80) else "Not Found") # Output: Found
print("Found" if search(root, 25) else "Not Found") # Output: Not Found

```

Time Complexity:

- **Best case:** $O(1)$ (when the target is at the root).
- **Worst case:** $O(h)$, where h is the height of the tree.

3.1.2 Inserting a Node in Binary Search Tree

To insert a new node in a BST:

1. Start from the root and recursively find the correct position for the new node by comparing it with the current node's key.
2. Insert the node at the appropriate leaf position (left if smaller, right if greater).

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Function to insert a new node in BST
def insert(root, key):
    # If the tree is empty, return a new node
    if root is None:
        return Node(key)

    # Otherwise, recur down the tree
    if key < root.key:
        root.left = insert(root.left, key)
    elif key > root.key:
        root.right = insert(root.right, key)

    # Return the unchanged node pointer
    return root

# Inorder traversal to print the BST
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")

```

```

        inorder(root.right)

# Driver code to test the insert function
root = Node(50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print("Inorder traversal of the BST: ")
inorder(root)

# Output:
# Inorder traversal of the BST:
# 20 30 40 50 60 70 80

```

Time Complexity:

- **Best case:** $O(1)$ (if inserted at root).
- **Worst case:** $O(h)$ where h is the height of the tree.

3.1.3 Deleting a Node from Binary Search Tree

Deleting a node in a BST involves three cases:

1. **Node to be deleted has no children** (a leaf node) – simply remove the node.
2. **Node has one child** – replace the node with its child.
3. **Node has two children** – find the in-order successor (the smallest node in the right subtree), copy its value to the node, and delete the in-order successor.

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Function to find the in-order successor (smallest in right subtree)
def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

```

```

# Function to delete a node from BST
def deleteNode(root, key):
    # Base case: If the tree is empty
    if root is None:
        return root

    # Recursively traverse the tree
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif key > root.key:
        root.right = deleteNode(root.right, key)
    else:
        # Node with only one child or no child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        # Node with two children: Get the in-order successor
        temp = minValueNode(root.right)
        root.key = temp.key
        root.right = deleteNode(root.right, temp.key)

    return root

# Inorder traversal to print the BST
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

# Driver code to test the delete function
root = Node(50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

```

```

# Deleting node 20
root = deleteNode(root, 20)
print("Inorder traversal after deleting 20: ")
inorder(root)

# Deleting node 30
root = deleteNode(root, 30)
print("\nInorder traversal after deleting 30: ")
inorder(root)

# Deleting node 50
root = deleteNode(root, 50)
print("\nInorder traversal after deleting 50: ")
inorder(root)

# Output:
# Inorder traversal after deleting 20:
# 30 40 50 60 70 80
# Inorder traversal after deleting 30:
# 40 50 60 70 80
# Inorder traversal after deleting 50:
# 40 60 70 80

```

Time Complexity:

- **Best case:** $O(1)$ (deleting root).
- **Worst case:** $O(h)$ where h is the height of the tree.

3.1.4 Traversing a Binary Search Tree

Inorder traversal of a BST allows visiting the nodes in sorted (non-decreasing) order:

1. Recursively visit the left subtree.
2. Visit the root.
3. Recursively visit the right subtree.

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Function for Inorder Traversal

```

```

def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

# Driver code to test the traversal
root = Node(50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print("Inorder traversal of the BST: ")
inorder(root)

# Output:
# Inorder traversal of the BST:
# 20 30 40 50 60 70 80

```

Time Complexity:

- $O(N)$, where N is the number of nodes in the tree.

3.2 Applications, Benefits and Limitations of a Binary Search Tree

Applications:

- BSTs are ideal when you need to maintain a dynamic set of data sorted for efficient searching, insertion, and deletion like tracking online order prices, allowing quick queries for items below or above a certain cost.
- A balanced BST allows efficient extraction of both minimum and maximum values.
- BSR enables sorting by inserting elements into the BST and using in-order traversal, which returns sorted elements while allowing $O(\log n)$ insertions and deletions.
- Self-Balancing BST is the best suited data structure, for algorithms like count smaller elements on right, Smallest Greater Element on Right Side, etc.
- Variations of BST like B Tree and B+ Tree are used in Database indexing.

Benefits:

- Average case search time is $O(\log n)$, making it more efficient than a linear search.
- Elements are stored in sorted order, making it easy to find the next or previous element
- It allows dynamic insertion and deletion, hence elements can be added or removed efficiently.
- Inorder traversal provides sorted data effortlessly.
- Balanced BSTs maintain a logarithmic height, ensuring efficient operations

Limitations:

- BST performance degrades to $O(n)$ when the tree is skewed (all nodes have one child).
- BSTs require additional memory to store pointers to child nodes
- BSTs are not suitable for very large datasets\

3.3 Key Takeaways from Resources

- BSTs allow for efficient searching, insertion, and deletion, all performed in $O(h)$ time, where h is the height of the tree.
- Every node's left child has a smaller value, and the right child has a larger value, maintaining sorted order for in-order traversal.
- Performance degrades to $O(n)$ in the worst case for skewed trees. Self-balancing variants like AVL or Red-Black trees help maintain efficiency.
- Ideal for dynamic sets of data requiring frequent updates while preserving order, such as databases and priority queues.

4. Binary Tree and Binary Search Tree Problems on Leetcode

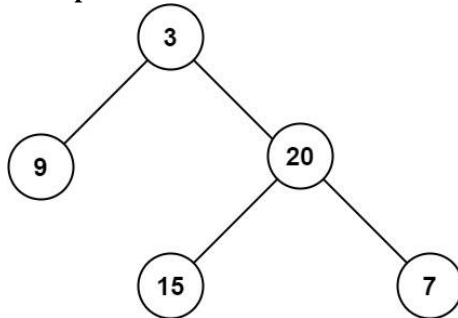
4.1 Problem 1: Maximum Depth of a Binary Tree

Problem Statement([Link](#)):

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:



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

Output: 3

Challenges Faced:

- All nodes need to be traversed to figure out the maximum depth, so the challenge is ensuring that all paths are considered.
- If a node is null, it should be handled appropriately since it means that branch of the tree has ended.
- A recursive check is needed to get the depth of each subtree and decide the larger one.
- The input tree can be visualized as a list but operates as a binary tree structure where null nodes terminate paths.

Approach:

- This problem can be solved using a recursive Depth-First Search (DFS) approach.
- If the node is None (i.e., there is no node), then that is the best case of the algorithm and the depth is 0.
- For each node, the depth of the left subtree and the right subtree is calculated, and then the maximum of those two depths is taken and 1 is added to account for the current node.

Solution:

104. Maximum Depth of Binary Tree Solved

Easy Topics Companies

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:

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

Example 2:

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

```
1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def maxDepth(self, root: Optional[TreeNode]) -> int:
9         # Base case: If the root is None, the depth is 0
10        if root is None:
11            return 0
12
13        # Recursive case: Calculate the depth of the left and right subtrees
14        left_depth = self.maxDepth(root.left)
15        right_depth = self.maxDepth(root.right)
16
17        # The depth of the current node is the max of the left and right depths plus 1
18        return max(left_depth, right_depth) + 1
19
```

Testcase Test Result

Accepted Runtime: 32 ms

Case 1 Case 2

Input

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

Code:

```
def maxDepth(root):  
    # Base case: If the root is None, the depth is 0  
    if root is None:
```

```

    return 0

# Recursive case: Calculate the depth of the left and right subtrees
left_depth = maxDepth(root.left)
right_depth = maxDepth(root.right)

# The depth of the current node is the max of the left and right depths plus 1
return max(left_depth, right_depth) + 1

```

Example:

```

# Leaf nodes
node15 = TreeNode(15)
node7 = TreeNode(7)

# Subtrees
node20 = TreeNode(20, left=node15, right=node7)
node9 = TreeNode(9)

# Root node
root = TreeNode(3, left=node9, right=node20)

# Find the maximum depth
depth = maxDepth(root)
print("Maximum depth of the tree:", depth)

# Output:
# Maximum depth of the tree: 3

```

Time and Space Complexity:

- **Time Complexity:** $O(n)$, where n is the number of nodes in the tree. We must visit each node once to calculate its depth.
- **Space Complexity:** $O(h)$, where h is the height of the tree, due to the recursive call stack. In the worst case (for a skewed tree), this could be $O(n)$, but for a balanced tree, it's $O(\log n)$.

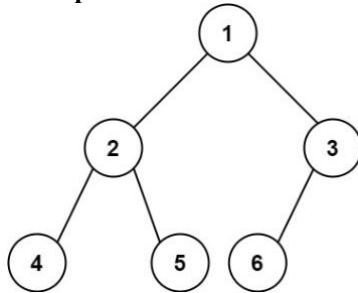
4.2 Problem 2: Count Complete Tree Nodes

Problem Statement([Link](#)):

Given the root of a **complete** binary tree, return the number of the nodes in the tree.

According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .

Design an algorithm that runs in less than $O(n)$ time complexity.

Example 1:

Input: root = [1,2,3,4,5,6]

Output: 6

Challenges Faced:

- Here the requirement is to design an algorithm that has time complexity better than $O(n)$. This means the basic traversal algorithms like depth-first search (DFS) or breadth-first search (BFS) cannot be used as both take $O(n)$ time.
- The challenge lies in using the properties of complete binary trees, where all levels except possibly the last are fully filled, and nodes are as far left as possible.
- Accurately calculating the heights of the left and right subtrees without traversing the entire tree can be tricky. Miscalculating the heights could lead to incorrect results.
- Depending on the tree's height, deep recursion might lead to a stack overflow, especially in trees that are skewed.

Approach:

- If the root is not present then 0 is returned as count of nodes will be 0.
- The height of the left subtree (left_height) is calculated by traversing down the leftmost path.
- The height of the right subtree (right_height) is calculated by traversing down the rightmost path.
- If left_height is equal to right_height, the tree is a perfect binary tree. In this case, the number of nodes can be directly computed using the formula $2^{\text{height}} - 1$.
- If the heights differ, the nodes in the left and right subtrees are recursively counted and one is added to it for the current root.

Time Complexity

- The time complexity of the solution is $O(\log^2 n)$:
 - Calculating the height of the left and right subtrees takes $O(\log n)$ each, resulting in $O(\log^2 n)$ for the entire tree.
- This is efficient for complete binary trees, as it avoids the need to traverse every node.

Solution:

222. Count Complete Tree Nodes

Easy Topics Companies

Solved

Given the `root` of a **complete** binary tree, return the number of the nodes in the tree.

According to [Wikipedia](#), every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .

Design an algorithm that runs in less than $O(n)$ time complexity.

Example 1:

```

Input: root = [1,2,3,4,5,6]
Output: 6

```

Example 2:

```

Input: root = []
Output: 0

```

Example 3:

```

Input: root = [1]
Output: 1

```

```

1 # Definition for a binary tree node.
2 class TreeNode:
3     def __init__(self, val=0, left=None, right=None):
4         self.val = val
5         self.left = left
6         self.right = right
7 class Solution:
8     def countNodes(self, root: Optional[TreeNode]) -> int:
9         if root is None:
10             return 0
11
12         left_height = 0
13         right_height = 0
14
15         current_node = root
16         while current_node:
17             left_height += 1
18             current_node = current_node.left
19
20         current_node = root
21         while current_node:
22             right_height += 1
23             current_node = current_node.right
24
25         if left_height == right_height:
26             return pow(2, left_height) - 1
27
28         return 1 + self.countNodes(root.left) + self.countNodes(root.right)
29

```

Accepted Runtime: 37 ms

Case 1 Case 2 Case 3

Input

```

root = [1,2,3,4,5,6]

```

Code:

```

def countNodes(root):
    if root is None:
        return 0

    left_height = 0
    right_height = 0

    current_node = root
    while current_node:
        left_height += 1
        current_node = current_node.left

    current_node = root
    while current_node:
        right_height += 1
        current_node = current_node.right

    if left_height == right_height:
        return pow(2, left_height) - 1

    return 1 + countNodes(root.left) + countNodes(root.right)

```

Example:

```
root = [1,2,3,4,5,6]
countNodes(root)
```

```
# Output: 6
```

4.3 Problem 3: Second Minimum Node in a Binary Tree

Problem Statement([Link](#)):

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property $\text{root.val} = \min(\text{root.left.val}, \text{root.right.val})$ always holds. Given such a binary tree, you need to output the **second minimum** value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

Challenges Faced:

- The tree is structured such that if a node has two children, its value is the minimum of the values of its children. This unique structure influences how we traverse the tree and compare values.
- The problem requires not just the minimum value, but specifically the second minimum. This means careful tracking of the smallest and second smallest values during the traversal.
- Another challenge is to handle edge cases like: trees where all values are the same (where the second minimum does not exist), trees that only have single nodes or the root with two children having equal values.

Approach:

- Depth-First Search (DFS) can be used to traverse the binary tree and find the second minimum value.
- The smallest is set as the value of the root node.
- The second_smallest is set to infinity ($\text{float}('inf')$) to denote that we have not yet found a second minimum.
- A recursive function `dfs(node)` is defined that returns immediately if the current node is `None`.
- The function `dfs`, checks if the node's value is greater than the smallest but less than second_smallest. If so, it updates the second_smallest and if the node's value equals the smallest, it continues searching its left and right children.
- After the traversal, it is checked if the second_smallest remains $\text{float}('inf')$. If it does, return -1, indicating that no second minimum exists. Otherwise, return the value of second_smallest.

Time Complexity

The DFS traversal visits each node exactly once, resulting in a time complexity of $O(n)$, where n is the number of nodes in the tree.

Space Complexity:

The space complexity is primarily due to the recursion stack used by DFS. In the worst case (a skewed tree), the depth can be n , leading to a space complexity of $O(n)$. In a balanced tree, it would be $O(\log n)$.

Solution:

671. Second Minimum Node In a Binary Tree

Solved

Easy Topics Companies

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-nodes. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property $root.val = \min(root.left.val, root.right.val)$ always holds.

Given such a binary tree, you need to output the second minimum value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

Example 1:

```

graph TD
    2((2)) --> 2L((2))
    2 --> 5R((5))
    5R --> 5L((5))
    5R --> 7R((7))
    
```

Input: root = [2,2,5,null,null,5,7]
Output: 5
Explanation: The smallest value is 2, the second smallest value is 5.

```

Python3
class Solution:
    def findSecondMinimumValue(self, root: Optional[TreeNode]) -> int:
        # Initialize the smallest and second smallest
        smallest = root.val
        second_smallest = float('inf')

        def dfs(node):
            nonlocal second_smallest
            if not node:
                return
            if smallest < node.val < second_smallest:
                second_smallest = node.val
            elif node.val == smallest:
                # Continue the search down the tree
                dfs(node.left)
                dfs(node.right)

        # Start DFS from the root
        dfs(root)

        return second_smallest if second_smallest != float('inf') else -1
    
```

Testcase Test Result

Accepted Runtime: 26 ms

Case 1 Case 2

Input

```

root = [2,2,5,null,null,5,7]
    
```

Code:

```

def findSecondMinimumValue(root):
    # Initialize the smallest and second smallest
    smallest = root.val
    second_smallest = float('inf')

    def dfs(node):
        nonlocal second_smallest
        if not node:
            return
        if smallest < node.val < second_smallest:
            second_smallest = node.val
        elif node.val == smallest:
            # Continue the search down the tree
            dfs(node.left)
            dfs(node.right)

    # Start DFS from the root
    dfs(root)

    return second_smallest if second_smallest != float('inf') else -1
    
```

Example:

```

root = [2,2,5,null,null,5,7]
    
```

```
findSecondMinimumValue(root)
```

```
# Output: 5
```

4.4 Problem 4: Convert Sorted Array to Binary Search Tree

Problem Statement([Link](#)):

Given an integer array `nums` where the elements are sorted in **ascending order**, convert it to a **height-balanced** binary search tree.

Challenges Faced:

- The primary challenge is to ensure that the BST remains height-balanced. This requires choosing the right element as the root at each recursive step.
- At each recursive step, the array needs to be split into left and right halves while ensuring the resulting subtrees remain balanced.
- Edge cases have to be considered like: an empty array where the output should be `None`, arrays with a single element where the result should be a single-node tree, arrays with all elements being the same, ensuring the function handles the recursion correctly without unnecessary complexity.

Approach:

- The solution uses Divide and Conquer method combined with Recursion to build the BST
- Base case is defined as - if the array `nums` is empty, return `None`.
- Find the middle element of the array using `mid = len(nums) // 2`. This element becomes the root of the current subtree, ensuring that the left and right halves are as balanced as possible.
- Recursively the left subtree is built using elements left of `mid`: `self.sortedArrayToBST(nums[:mid])` and the right subtree is built using elements right of `mid`: `self.sortedArrayToBST(nums[mid + 1:])`.
- The root node is returned, which links the left and right subtrees appropriately, forming a height-balanced BST.

Time Complexity

- The solution processes every element exactly once to place it in the tree, resulting in a time complexity of $O(n)$, where n is the number of elements in the array.

Space Complexity:

- The space complexity is primarily due to the recursion stack. The maximum depth of recursion is $O(\log n)$ for a balanced tree. Additionally, slicing the array creates new subarrays, which can require up to $O(n)$ space.

Solution:

108. Convert Sorted Array to Binary Search Tree

Easy Topics Companies

Given an integer array `nums`, where the elements are sorted in **ascending order**, convert it to a **height-balanced** binary search tree.

Example 1:

Input: `nums = [-10,-3,0,5,9]`
Output: `[0,-3,9,-10,null,5]`
Explanation: `[0,-10,5,null,-3,null,9]` is also accepted:

Example 2:

```

1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
9         # Base case: if the array is empty, return None
10        if not nums:
11            return None
12
13        # Find the middle index
14        mid = len(nums) // 2
15
16        # Create the root node with the middle element
17        root = TreeNode(nums[mid])
18
19        # Recursively build the left and right subtrees
20        root.left = self.sortedArrayToBST(nums[:mid]) # Left half excluding the middle element
21        root.right = self.sortedArrayToBST(nums[mid + 1:]) # Right half excluding the middle element
22
23        return root

```

Accepted Runtime: 48 ms

Case 1 Case 2

Input

nums =

Code:

```
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
```

```
        self.left = left
```

```
        self.right = right
```

```
class Solution:
```

```
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
```

```
        # Base case: if the array is empty, return None
```

```
        if not nums:
```

```
            return None
```

```
        # Find the middle index
```

```
        mid = len(nums) // 2
```

```
        # Create the root node with the middle element
```

```
        root = TreeNode(nums[mid])
```

```
        # Recursively build the left and right subtrees
```

```
        root.left = self.sortedArrayToBST(nums[:mid]) # Left half excluding the middle element
```

```
        root.right = self.sortedArrayToBST(nums[mid + 1:]) # Right half excluding the middle element
```

```
        return root
```

Example:

```
nums = [-10,-3,0,5,9]
```

```
sortedArrayToBST(nums)
```

```
# Output: [0,-3,9,-10,null,5]
```

5. Dynamic Programming

Dynamic Programming (DP) is a powerful algorithmic technique used to solve complex problems by breaking them down into simpler overlapping subproblems. Instead of recomputing solutions to these subproblems multiple times, dynamic programming saves and reuses their results, thus significantly reducing computational time. This technique is particularly effective in problems that exhibit optimal substructure and overlapping subproblems, making it widely applicable in algorithm design for tasks such as optimization, counting, and searching.

5.1 Characteristics of Dynamic Programming algorithm

- **Problem-Solving Technique:** Dynamic programming is a powerful technique used for solving problems that can be broken down into smaller, overlapping subproblems. This makes it ideal for optimization problems where subproblem solutions contribute to the overall optimal solution.
- **Optimal Substructure:** A problem has optimal substructure if its optimal solution can be constructed efficiently from the optimal solutions of its subproblems. This means that you can solve smaller pieces of the problem and combine them to solve the larger problem.
- **Overlapping Subproblems:** Dynamic Programming is applicable when subproblems recur multiple times. Instead of solving these subproblems repeatedly, DP saves their solutions to avoid redundant calculations.
- **Storage of Subproblem Results:** DP uses data structures such as arrays, tables, or dictionaries to store the results of solved subproblems, allowing for quick retrieval and reuse.

5.2 Dynamic Programming vs Recursion

- Dynamic Programming (DP) breaks a problem into smaller subproblems to solve the larger problem while recursion is a function that calls itself to execute a specific task.
- Recursion automates a function to solve problems while DP optimizes and accelerates problem-solving, often using recursion.
- Recursive functions recognize when they are needed, execute themselves, then stop working. When the function identifies the moment it is needed, it calls itself and is executed; this is called a recursive case. As a result, the function must stop once the task is completed, known as the base case. Dynamic programming recognizes the problem and divides it into sub-problems in order to solve the whole scene. After solving these sub-problems, or variables, the programmer must establish a mathematical relationship between them. These solutions and results are stored as algorithms, so they can be accessed in the future without having to solve the whole problem again.
- Recursion often solves the same subproblems multiple times, leading to inefficiency, while DP stores the results of these subproblems to avoid redundant calculations.
- Unlike simple recursion, DP employs techniques like memoization (top-down approach) or tabulation (bottom-up approach) to manage stored results.
- DP is much more efficient than recursion in problems with overlapping subproblems since it prevents repeated work by storing intermediate results.

5.3 Techniques to solve Dynamic Programming Problems

5.3.1 Top-Down(Memoization)

- This technique involves solving the problem recursively and storing the results of each subproblem.
- It avoids recalculating subproblems by checking if the result is already stored in a memoization table (e.g., a dictionary or array) before computing.
- This technique solves subproblems on-demand.

- This technique can use more stack space due to recursion.

5.3.2 Bottom-up (Tabulation)

- This approach iteratively solves subproblems starting from the smallest, building up solutions until the final problem is solved.
- You create a table to store the results of subproblems, which are filled in a specific order until you reach the desired solution.
- This technique uses less stack space.

5.4 How to Solve a Dynamic Programming Problem?

- Check if the problem exhibits overlapping subproblems and optimal substructure to see if it can be solved using DP.
- Determine what data you need to store to represent subproblems. For example, for a Fibonacci problem, the state can be defined by the current Fibonacci number you're trying to compute.
- Express the solution to the problem in terms of solutions to smaller subproblems.
- Decide whether to implement a top-down (memoization) or bottom-up (tabulation) approach based on the problem's nature and constraints.
- Code the solution, ensuring it efficiently handles the overlapping subproblems and optimal substructure.

5.5 Greedy Approach Vs Dynamic Programming

- Greedy approach makes locally optimal choices at each step while dynamic programming solves subproblems and combines results for a global solution.
- Greedy approach is suitable when local choices lead to an overall optimal solution while dynamic programming is used when problems have overlapping subproblems and optimal substructure.
- Greedy approach solves problem like shortest path in unweighted graphs, activity selection etc while dynamic programming can be used for problems like fibonacci sequence, knapsack problem, longest common subsequence

5.6 Applications, Benefits and Limitations

Applications:

- Dynamic programming is used for problems that can be broken down into overlapping subproblems, such as finding the shortest path in graphs (e.g., Dijkstra's algorithm) or calculating Fibonacci numbers.
- Dynamic programming is used to solve optimization problems like the Knapsack problem, where resources need to be allocated efficiently.
- It is used in problems like the Longest Common Subsequence (LCS) and edit distance, which involve comparing and processing strings.

Benefits:

- Dynamic programming reduces the computational time by storing results of subproblems, avoiding redundant calculations, and leading to significant speed improvements.
- It guarantees finding the optimal solution for problems by considering all possibilities and building up solutions from smaller subproblems.
- It is suitable for large and complex problems where simple recursion would be inefficient, making it ideal for applications with high computational demands.

Limitations:

- Dynamic programming requires additional space for storing solutions to subproblems, which can lead to high memory consumption, especially for problems with a large state space.
- Designing dynamic programming solutions can be challenging due to the need to identify overlapping subproblems, define states, and establish recursive relationships.

- DP is only effective for problems with overlapping subproblems and optimal substructure; it may not be beneficial for problems without these properties.
- Developing a DP solution can be time-intensive because it requires a thorough understanding of the problem structure and careful planning of state transitions and storage mechanisms.

5.7 Key Takeaways from Resources

- Dynamic Programming is an efficient technique for problems with overlapping subproblems and optimal substructure.
- It can be implemented using **memoization** (top-down) or **tabulation** (bottom-up), each with its pros and cons.
- DP is different from recursion because it avoids redundant calculations by storing intermediate results.
- It is crucial to practice DP to develop intuition for identifying subproblems, defining states, and formulating recurrence relations.

6. Dynamic Programming Problems on Leetcode

6.1 Problem 1: Fibonacci Number

Problem Statement([Link](#)):

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 $F(n)$.

Challenges Faced:

- The standard recursive approach to calculate Fibonacci numbers is highly inefficient. As n increases, the same subproblems are repeatedly solved, leading to an exponential time complexity of $O(2^n)$.
- The recursive approach also consumes a significant amount of space on the call stack due to its deep recursive calls, which can lead to stack overflow for large n .
- The primary challenge was to avoid recomputing Fibonacci numbers that had already been calculated, which significantly increases efficiency.

Approach:

- To tackle these challenges, the memoization technique was used. Memoization stores the results of expensive function calls and reuses them when the same inputs occur again. By doing so, it reduces the time complexity from $O(2^n)$ to $O(n)$ and prevents redundant calculations.
- A dictionary (memo) was used to store the Fibonacci values that had been computed so far.
- The recursive function checks if a value for n already exists in memo. If it does, it returns that value; if not, it calculates it and stores the result in memo for future reference.

Time Complexity: $O(n)$

Space Complexity: $O(n)$

Solution:

509. Fibonacci Number Solved

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:
Input: $n = 4$
Output: 3
Explanation: $F(4) = F(3) + F(2) = 2 + 1 = 3$.

Constraints:
 $0 \leq n \leq 30$

```
class Solution:
    def fib(self, n: int) -> int:
        def fibonacci(n, memo):
            if n == 0 or n == 1:
                return n
            if n not in memo:
                memo[n] = fibonacci(n-2, memo) + fibonacci(n-1, memo)
            return memo[n]
        result = fibonacci(n, {})
        return result
```

Accepted Runtime: 27 ms

Case 1 Case 2 Case 3

Input: $n = 2$

Output: 1

Expected: 1

Code:

```
def fib(n):
    def fibonacci(n, memo):
        if n == 0 or n == 1:
            return n
        if n not in memo:
```

```

memo[n] = fibonacci(n-2, memo) + fibonacci(n-1, memo)

return memo[n]

result = fibonacci(n, {})
return result

```

Example:

```

n = 4
fib(n)

# Output: 3

```

6.2 Problem 2: Longest Common Subsequence

Problem Statement([Link](#)):

Given two strings text1 and text2, return *the length of their longest **common subsequence***. If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

Challenges Faced:

- Finding the Longest Common Subsequence (LCS) is a problem with overlapping subproblems, meaning the same subproblems are solved multiple times when using a naive recursive approach, resulting in inefficiency.
- A brute-force solution that checks all possible subsequences would have an exponential time complexity, making it impractical for larger strings.
- Storing results efficiently was essential to avoid recalculating values and to manage memory usage effectively.

Approach:

- To handle these challenges, the Dynamic Programming (DP) approach was used. This method breaks down the problem into smaller, manageable subproblems, stores their solutions in a table (in this case, a 2D array), and builds the final solution based on these precomputed values.
- The main idea is to create a table dp where dp[i][j] represents the length of the LCS for substrings text1[0.....i-1] and text2[0.....j-1]. By iterating through each character in text1 and text2, the table is filled based on whether characters match or not.
- If text1[i-1] == text2[j-1], then dp[i][j] = dp[i-1][j-1] + 1. This means that the current characters are part of the LCS.
- If text1[i-1] != text2[j-1], then dp[i][j] = max(dp[i-1][j], dp[i][j-1]). This means that the LCS up to this point is determined by ignoring one of the characters.
- The final answer will be found in dp[m][n], where m and n are the lengths of text1 and text2.

Time Complexity: $O(m * n)$, where m and n are the lengths of text1 and text2. This is efficient compared to the exponential time complexity of a naive recursive approach.

Space Complexity: $O(m * n)$ due to the 2D dp table.

Solution:

1143. Longest Common Subsequence

Solved

Medium Topics Companies Hint

Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

Input: `text1 = "abcde", text2 = "ace"`
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: `text1 = "abc", text2 = "abc"`
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: `text1 = "abc", text2 = "def"`
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

Constraints:

- `1 <= text1.length, text2.length <= 1000`
- `text1` and `text2` consist of only lowercase English characters.

```
1 class Solution:
2     def longestCommonSubsequence(self, text1: str, text2: str) -> int:
3         m, n = len(text1), len(text2)
4
5         # Create a (m+1) x (n+1) DP table filled with 0
6         dp = [[0] * (n + 1) for _ in range(m + 1)]
7
8         # Fill the DP table
9         for i in range(1, m + 1):
10             for j in range(1, n + 1):
11                 if text1[i - 1] == text2[j - 1]:
12                     dp[i][j] = dp[i - 1][j - 1] + 1
13                 else:
14                     dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
15
16         # The bottom-right cell contains the length of LCS
17         return dp[m][n]
```

Testcase Test Result

Accepted Runtime: 41 ms

Case 1 Case 2 Case 3

Input

text1 =
"abcde"

text2 =
"ace"

Output

Code:

```
def longestCommonSubsequence(text1, text2):
    m, n = len(text1), len(text2)

    # Create a (m+1) x (n+1) DP table filled with 0
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # The bottom-right cell contains the length of LCS
    return dp[m][n]
```

Example:

```
text1 = "abcde"
text2 = "ace"

longestCommonSubsequence(text1, text2)
# Output: 3
```

6.3 Problem 3: Longest Palindromic Subsequence

Problem Statement([Link](#)):

Given a string s , find the longest palindromic *subsequence's* length in s .

A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Challenges Faced:

- There are overlapping subproblems when trying to identify palindromic subsequences. If we attempt to solve this problem using a naive recursive approach, we would end up recalculating the same subproblems multiple times.
- A brute-force approach would involve checking all possible subsequences, leading to exponential time complexity, which is impractical for long strings.
- Determining whether a subsequence is palindromic involves comparing characters from both ends of the substring, requiring an effective strategy to handle this without redundant calculations.

Approach:

- The most efficient way to solve this problem is using Dynamic Programming (DP). The idea is to build a 2D DP table where $dp[i][j]$ represents the length of the longest palindromic subsequence in the substring $s[i...j]$.
- If $i == j$, it means the substring is a single character, which is always a palindrome of length 1.
- If $s[i] == s[j]$, then $dp[i][j] = dp[i + 1][j - 1] + 2$. This implies that the outer characters are part of the palindromic subsequence, and we add 2 to the length found in the remaining substring.
- If $s[i] != s[j]$, then $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$, meaning we ignore either the left or the right character and take the maximum length found.
- The final result will be stored in $dp[0][n - 1]$, representing the longest palindromic subsequence for the entire string.

Time Complexity: $O(n^2)$, where n is the length of the string s . The nested loop iterates over all substrings of different lengths, ensuring efficiency.

Space Complexity: $O(n^2)$, due to the 2D dp table.

Solution:

The screenshot displays a coding problem titled "516. Longest Palindromic Subsequence" on a platform like LeetCode. The problem description states: "Given a string s , find the longest palindromic subsequence's length in s ." It includes two examples: Example 1 with input "bbbab" and output 4, and Example 2 with input "cbbd" and output 2. Constraints specify that the string length is between 1 and 1000 and consists of lowercase English letters. On the right, a Python solution is shown, implementing a dynamic programming approach with a 2D table dp . The code initializes the table with zeros, sets the diagonal elements to 1, and then fills the table by comparing characters at the ends of substrings. The final result is returned as $dp[0][n - 1]$. The solution is marked as "Accepted" with a runtime of 31 ms.

Code:

```
def longestPalindromeSubseq(s):
```

```
    n = len(s)
```

```
    # Create a 2D DP table filled with 0
```

```
    dp = [[0] * n for _ in range(n)]
```



```

# Each single character is a palindrome of length 1
for i in range(n):
    dp[i][i] = 1

# Fill the DP table
for length in range(2, n + 1): # length of the substring
    for i in range(n - length + 1):
        j = i + length - 1
        if s[i] == s[j]:
            dp[i][j] = dp[i + 1][j - 1] + 2
        else:
            dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

# The top-right corner contains the length of the longest palindromic subsequence
return dp[0][n - 1]

```

Example:

```

s = "bbbab"
longestPalindromeSubseq(s)

# Output: 4

```

6.4 Problem 4: Coin Change

Problem Statement([Link](#)):

You are given an integer array `coins` representing coins of different denominations and an integer amount representing a total amount of money.

Return *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.

You may assume that you have an infinite number of each kind of coin.

Challenges Faced:

- Finding the minimum number of coins requires considering all combinations of coins for the given amount. Determining an optimal substructure (i.e., breaking the problem into smaller subproblems) is crucial to efficiently solving the problem.
- There are multiple ways to reach a particular amount using different coin denominations. Without using dynamic programming, the problem might be solved repeatedly for the same subproblems, leading to redundant calculations and inefficiency.
- Handling situations such as when the amount is 0, when it's not possible to form the amount with the given denominations, or when the list of coins contains only one denomination is a challenge.

Approach:

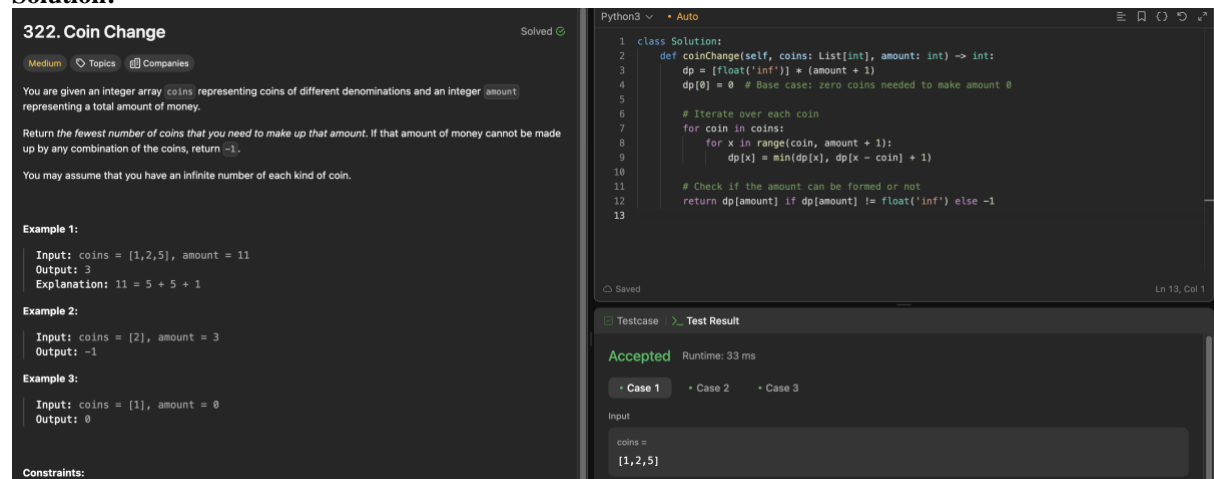
- The most efficient approach to solving this problem is by using Dynamic Programming (DP).
- A dp array is created of size `amount + 1` and all elements are initialised to `float('inf')` (representing that the amount cannot be made initially).
- `dp[0]` is set as 0 because zero coins are needed to make an amount of 0.
- Each coin is iterated in the `coins` array.
- For each coin, all possible amounts are iterated from coin to amount.

- $dp[x]$ is updated to be the minimum of its current value and $dp[x - \text{coin}] + 1$. This represents using one more coin to reach the current amount x from $x - \text{coin}$.
- The value $dp[\text{amount}]$ will give the minimum number of coins needed to make up the amount. If $dp[\text{amount}]$ remains $\text{float}('inf')$, return -1 indicating it's not possible to form the given amount.

Time Complexity: $O(m * n)$, The outer loop iterates over each coin: $O(m)$ where m is the number of coins. The inner loop iterates over all amounts from coin to amount: $O(n)$ where n is the amount. Hence, the overall time complexity is **$O(m * n)$** .

Space Complexity: The solution uses a 1D array dp of size $\text{amount} + 1$ to store results. Thus, the space complexity is **$O(n)$** .

Solution:



The screenshot shows the LeetCode interface for problem 322, 'Coin Change'. The problem description states: 'You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money. Return 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. You may assume that you have an infinite number of each kind of coin.' Examples provided include: Example 1 (coins=[1,2,5], amount=11, output=3), Example 2 (coins=[2], amount=3, output=-1), and Example 3 (coins=[1], amount=0, output=0). The code editor on the right shows a Python solution using dynamic programming.

```

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [float('inf')] * (amount + 1)
        dp[0] = 0 # Base case: zero coins needed to make amount 0

        # Iterate over each coin
        for coin in coins:
            for x in range(coin, amount + 1):
                dp[x] = min(dp[x], dp[x - coin] + 1)

        # Check if the amount can be formed or not
        return dp[amount] if dp[amount] != float('inf') else -1
  
```

The test result shows 'Accepted' with a runtime of 33 ms. The input for Case 1 is coins = [1,2,5].

Code:

```

def coinChange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0 # Base case: zero coins needed to make amount 0

    # Iterate over each coin
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)

    # Check if the amount can be formed or not
    return dp[amount] if dp[amount] != float('inf') else -1
  
```

Example:

```

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

coinChange(coins, amount)
  
```

```
# Output: 3
```

7. References

1. [1] "Learn Data Structures and Algorithms Tutorial," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/>.
2. [2] "Binary Tree Data Structure," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/binary-tree-data-structure/>.
3. [3] "Binary Search Tree Data Structure," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/binary-search-tree-data-structure/>.
4. [4] "Introduction to Dynamic Programming," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/>.
5. [5] "Trees," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=oSWTXtMglKE>.
6. [6] "Binary Search Tree," YouTube, [Online]. Available: https://www.youtube.com/watch?v=pYT9F8_LFTM.
7. [7] "LeetCode Problem Set - Tree," LeetCode, [Online]. Available: <https://leetcode.com/problemset/?page=1&search=tree>.
8. [8] "Dynamic Programming Examples," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=YBSt1jYwVfU>.
9. [9] "Dynamic Programming Explained," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=oBt53YbR9Kk>.
10. [10] "LeetCode Dynamic Programming Problems," LeetCode, [Online]. Available: <https://leetcode.com/problem-list/dynamic-programming/>.