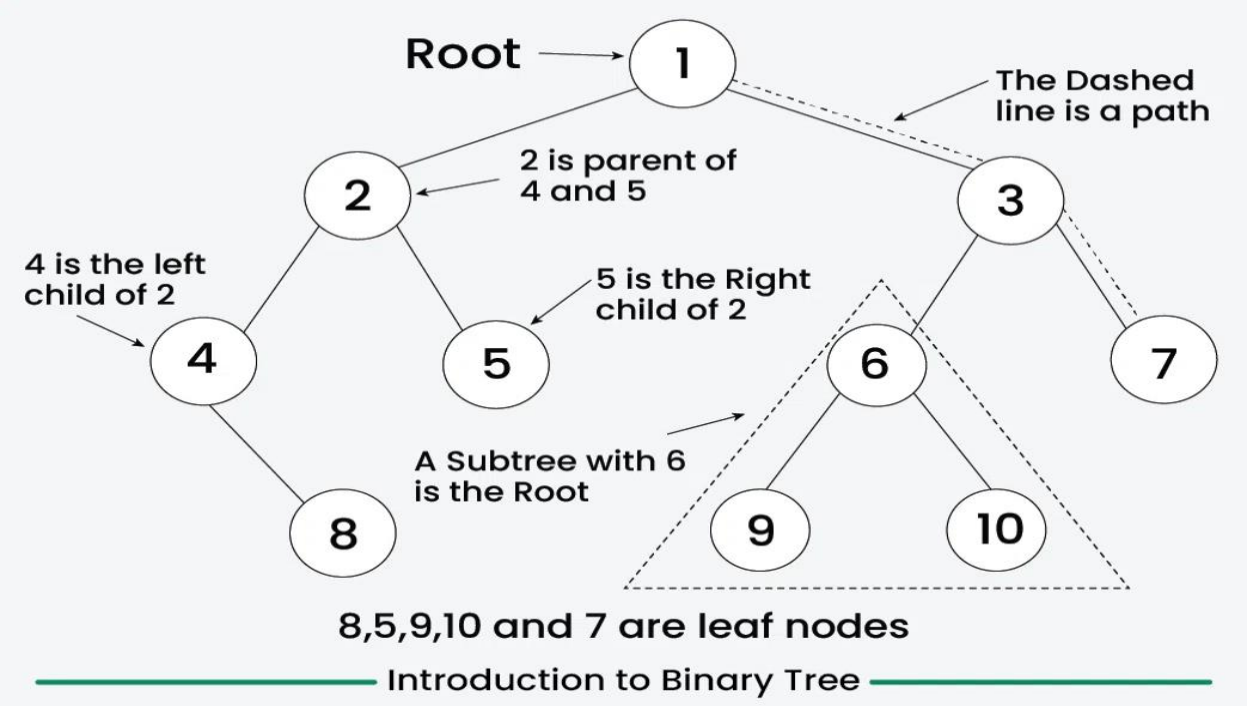
WEEK -2

Binary Tree :

Binary Tree is a non-linear data structure where each node has at most two children .



1. Node: Every element in a binary tree is called a node.
2. Root: The topmost node in a tree.
3. Child Node: Nodes which are directly connected to another node.
4. Parent Node: A node that has children.
5. Leaf Node: A node with no children.
6. Depth/Height: The depth of a node is the number of edges from the node to the root. The height of a tree is the depth of its deepest node.
7. Subtree: Any node in a binary tree, along with its descendants, forms a subtree.

Operations of Binary Tree:

Operation	Time Complexity	Auxiliary Space
In-Order Traversal	O(n)	O(n)
Pre-Order Traversal	O(n)	O(n)
Post-Order Traversal	O(n)	O(n)

Operation	Time Complexity	Auxiliary Space
Insertion (Unbalanced)	O(n)	O(n)
Searching (Unbalanced)	O(n)	O(n)
Deletion (Unbalanced)	O(n)	O(n)

1. Insertion
2. Searching
3. Deletion
4. Traversal

Example:
[10, 5, 15, 3, 7, 13, 18]:

1. Node Initialization and Binary Tree Creation
We will first define the Tree Node class and use the same method bstFromPreorder to build a Binary Search Tree (BST) from a preorder traversal list.

Definition for a binary tree node.

```
class Tree Node:
    def __init__(self, Val=0, left=None, right=None):
        self. Val = Val
        self. Left = left
        self. Right = right

class Solution:
    def bstFromPreorder(self, preorder: list[int]) -> Tree Node:
        if not preorder:
            return None

        root = Tree Node(preorder[0])
        root. Left = self. BstFromPreorder([i for i in preorder if i
< preorder[0]])
        root. Right = self. BstFromPreorder([i for i in preorder if
i > preorder[0]])

        return root

# Preorder list to construct the tree
preorder = [10, 5, 3, 7, 15, 13, 18]

# Create the binary tree
solution = Solution()
root = solution. BstFromPreorder(preorder)
```

Tree Structure:

```
      10
    /    \
   5      15
  / \     / \
 3   7 13 18
```

2. Tree Traversals:
Inorder (Left, Root, Right):

```
def inorder(root: TreeNode):
    if root:
        inorder(root.left)
        print(root.val, end=' ')
        inorder(root.right)

print("Inorder Traversal: ", end='')
inorder(root)  # Output: 3 5 7 10 13
15 18
print()
```

Preorder (Root, Left, Right):

```
def preorder(root: TreeNode):
    if root:
        print(root.val, end=' ')
        preorder(root.left)
        preorder(root.right)

print("Preorder      Traversal:      ",
end='')
preorder(root)  # Output: 10 5 3 7
15 13 18
print()
```

Post order (Left, Right, Root):

```
def postorder(root: TreeNode):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val, end=' ')

print("Postorder Traversal: ", end='')
postorder(root)  # Output: 3 7 5 13 18
15 10
print()
```

Level order:

```
from collections import deque

def level_order(root: TreeNode):
    if not root:
        return

    queue = deque([root])

    while queue:
        node = queue.popleft()
        print(node.val, end=' ')
        if node.left:

queue.append(node.left)
        if node.right:

queue.append(node.right)

print("Level-order Traversal: ", end='')
level_order(root)  # Output: 10 5 15 3 7 13 18
print()
```

3. Insertion in Binary Tree:

```
def insert(root: TreeNode, key: int):
    if not root:
        return TreeNode(key)

    queue = deque([root])

    while queue:
        node = queue.popleft()

        if not node.left:
            node.left = TreeNode(key)
            break
        else:

queue.append(node.left)

        if not node.right:
```

```
            node.right =
  TreeNode(key)
            break
        else:

  queue.append(node.right)
```

```
# Insert a node with value 8
  insert(root, 8)

  print("Inorder Traversal after inserting 8:
  ", end='')
  inorder(root)  # Output: 3 5 7 8 10 13 15
  18
print()
```

Tree Structure:

```
      10
     /  \
    5    15
   / \   / \
  3   7 13 18
      \
       8
```

4.Deletion in Binary Tree:

```
def delete_node(root: TreeNode, key: int) -> TreeNode:
```

```
    if not root:
        return None

    if root.left is None and root.right is None:
        if root.val == key:
            return None
        else:
            return root

    key_node = None
    queue = deque([root])
    temp = None

    # Find the key node and the deepest node
    while queue:
        temp = queue.popleft()

        if temp.val == key:
            key_node = temp

        if temp.left:
            queue.append(temp.left)
        if temp.right:
            queue.append(temp.right)

    if key_node:
        key_node.val = temp.val  # Replace key node's value with the deepest
node's value
        delete_deepest(root, temp)

    return root

# Helper function to delete the deepest node
def delete_deepest(root: TreeNode, d_node: TreeNode):
    queue = deque([root])

    while queue:
        temp = queue.popleft()

        if temp is d_node:
            temp = None
            return
       if temp.right:
            if temp.right is d_node:
                temp.right = None
                return
```

```
            else:
                queue.append(temp.right)

        if temp.left:
            if temp.left is d_node:
                temp.left = None
                return
            else:
                queue.append(temp.left)

# Delete node with value 15
delete_node(root, 15)

print("Inorder Traversal after deleting 15: ", end='')
inorder(root)  # Output: 3 5 7 8 10 13 18
print()
```

- Node 15 is deleted and replaced by node 18, which was the deepest rightmost node in the tree.

Tree Structure:

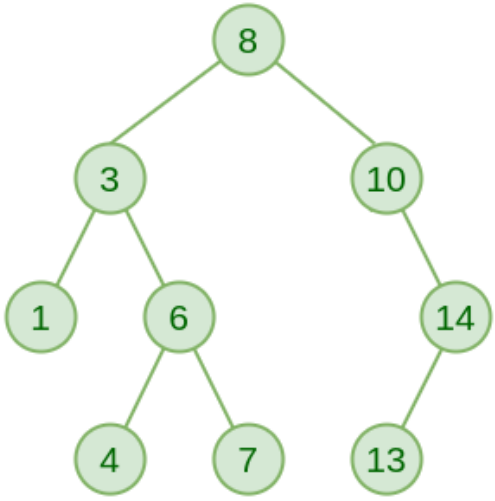
```
    10
   /  \
  5    18
 / \   /
3   7 13
     \
      8
```

5. Traversal Output After Deletion:
The traversals after deleting node 15 would be:

- In-order DFS: 3 5 7 8 10 13 18
- Pre-order DFS: 10 5 3 7 8 18 13
- Post-order DFS: 3 8 7 5 13 18 10
- Level-order BFS: 10 5 18 3 7 13 8

Binary Search Tree

Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property, and it makes it possible to efficiently search, insert, and delete elements in the tree.



Binary Search Tree

Binary Structure:

- Each node in a BST has at most two children: a left and a right child. This ensures the tree maintains a simple structure while organizing data hierarchically.

BST Ordering Principle:

- For every node N:
  - The values of all nodes in the left subtree are strictly less than the value of N.
  - The values of all nodes in the right subtree are strictly greater than the value of N.
  - This rule applies recursively, allowing for efficient data organization and lookup.

Uniqueness of Elements:

- By default, a BST does not store duplicate elements. Each node contains a distinct value. In cases where duplicates are allowed, their placement (usually to the left or right) follows a predetermined rule.

Efficient Lookup Structure:

- The tree's structure facilitates fast searches by narrowing down the search space with each step. In an optimally balanced BST, searching for a value has a time complexity of O(log n) due to its branching nature.

Sorted Output via In-order Traversal:

- An in-order traversal (visiting the left subtree, then the root, then the right subtree) will return the nodes in ascending order. This inherent sorting feature makes BSTs particularly useful for ordered data retrieval.

Tree Height and Balance:

- The height of a BST determines its efficiency. For a well-balanced tree, the height is proportional to O(log n). However, in a degenerate or unbalanced tree (where each node has only one child), the height can degrade to O(n), reducing performance.

Dynamic Adaptation:

- Unlike static data structures, a BST dynamically adjusts as new elements are inserted or existing ones are deleted. However, it is susceptible to becoming unbalanced, especially when data is inserted in sorted order, affecting its overall performance.

Time Complexities:

- **Searching**: Best-case O(log n), but O(n) in the worst case (if the tree is unbalanced).
- **Insertion**: Typically O(log n) in balanced cases, but O(n) if the tree becomes skewed.
- **Deletion**: Like insertion, it depends on the tree's balance, ranging from O(log n) to O(n).

Basic Operation of Binary Search Tree:

1.Searching
2.Insertion
3.Deletion
4. Traversal

Insertion:

The **insert** operation adds a new node to the BST while maintaining the BST property (left subtree contains values less than the parent node, and the right subtree contains values greater than the parent node).

Steps:

- Start at the root.
- Compare the new key with the current node's key:
  - If it's smaller, move to the left subtree.
  - If it's larger, move to the right subtree.
- Insert the new node when an appropriate None position is found.

Code:

```
def insert(self, key):
    if self.root is None:
        self.root = Node(key)
    else:
        self._insert(self.root, key)

def _insert(self, node, key):
    if key < node.key:
        if node.left is None:
            node.left = Node(key)
        else:
            self._insert(node.left, key)
    else:
        if node.right is None:
            node.right = Node(key)
        else:
```

```
                self._insert(node.right,
    key)
```

2. Searching

Steps:

- Start at the root.
- Compare the target key with the current node's key:
  - If the key matches, return the node.
  - If the key is smaller, move to the left subtree.
  - If the key is larger, move to the right subtree.
- If the node is not found, return None.

Code:

```
    def search(self, key):
     return self._search(self.root, key)

    def _search(self, node, key):
        if node is None or node.key == key:
            return node  # Node found or end of tree
    reached
        if key < node.key:
            return self._search(node.left, key)
        else:
return self._search(node.right, key)
```

3. Deletion

The **delete** operation removes a node from the BST while maintaining its structure.

Steps:

- If the node to be deleted is a **leaf**, simply remove it.
- If the node has **one child**, replace the node with its child.
- If the node has **two children**, find its in-order successor (smallest value in the right subtree), copy the successor's value to the node, and delete the successor.

Code:

```
def delete(self, key):
  self.root = self._delete(self.root, key)

 def _delete(self, node, key):
    if node is None:
       return node

    # Traverse the tree to find the node to
 delete
    if key < node.key:
       node.left = self._delete(node.left,
 key)
    elif key > node.key:
       node.right = self._delete(node.right,
 key)
    else:
       # Node to be deleted is found
       if node.left is None:
          return node.right
       elif node.right is None:
          return node.left

       # Node with two children
       temp =
 self._min_value_node(node.right)
       node.key = temp.key
       node.right = self._delete(node.right,
 temp.key)

    return node

 # Helper function to find the minimum
 value node
 def _min_value_node(self, node):
    current = node
    while current.left is not None:
       current = current.left
    return current
```

5. In-order Traversal

In-order traversal visits the nodes in ascending order (left-root-right). This is useful when you want to retrieve the elements of a BST in sorted order.

Steps:

- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.

Code :

```
def inorder(self):
    self._inorder(self.root)

def _inorder(self, node):
    if node:

self._inorder(node.left)
        print(node.key, end="
")

self._inorder(node.right)
```

1.Find Mode in Binary Search Tree

CODE:

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

class Solution:
    def findMode(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        if not root:
            return []


        self.current_val = None
        self.current_count = 0
        self.max_count = 0
        self.modes = []


        def inorder(node):
            if not node:
                return

            inorder(node.left)

            # Process the current node
            self.handle_value(node.val)

            inorder(node.right)
```

```
        def handle_value(val):
            if val != self.current_val:
                self.current_val = val
                self.current_count = 0
            self.current_count += 1

            if self.current_count > self.max_count:
                self.max_count = self.current_count
                self.modes = [val]
            elif self.current_count == self.max_count:
                self.modes.append(val)


        self.handle_value = handle_value

        inorder(root)

        return self.modes


if __name__ == "__main__":

    root = TreeNode(1)
    root.right = TreeNode(2)
    root.right.left = TreeNode(2)


    solution = Solution()
    print(solution.findMode(root))


    root2 = TreeNode(0)
    print(solution.findMode(root2))
```

- Step-by-Step:
    1. Start at root node 1.
    2. Move to the left child (None).
    3. Add 1 to modes and move back to root 1.
    4. Add 2 to modes and move to the right child (2).
    5. Since this 2 has the same value, increment the count.

Time Complexity:

- O(n): Each node is visited once.

Space Complexity:

- O(n): Space for the modes list and the recursion stack.

2. Minimum Absolute Difference in BST

Code:

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


class Solution(object):
    def getMinimumDifference(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        self.values = []


        def inorder(node):
            if not node:
                return
            inorder(node.left)
            self.values.append(node.val)
            inorder(node.right)


        inorder(root)


        min_diff = float('inf')
        for i in range(1, len(self.values)):
            min_diff = min(min_diff, self.values[i] - self.values[i - 1])

        return min_diff
```

3. Two Sum IV - Input is a BST

Code:

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

class Solution(object):
    def findTarget(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: bool
        """
        seen = set()

        def in_order_traverse(node):
            if not node:
                return False
```

```
            if in_order_traverse(node.left):
                return True

            if (k - node.val) in seen:
                return True

            seen.add(node.val)

            return in_order_traverse(node.right)

        return in_order_traverse(root)
```

4. Delete Node in a BST
Code:

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


class Solution(object):
    def deleteNode(self, root, key):
        """
        :type root: TreeNode
        :type key: int
        :rtype: TreeNode
        """
        if not root:
            return root


        if key < root.val:
            root.left = self.deleteNode(root.left, key)
        elif key > root.val:
            root.right = self.deleteNode(root.right, key)
        else:

            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            else:

                successor = self.getMinValueNode(root.right)
                root.val = successor.val
                root.right = self.deleteNode(root.right, successor.val)

        return root

    def getMinValueNode(self, node):
        current = node
        while current.left:
```

```
            current = current.left
        return current
```

5.Range Sum of BST

Code:

```
class Solution(object):
    def rangeSumBST(self, root, low, high):
        """
        :type root: TreeNode
        :type low: int
        :type high: int
        :rtype: int
        """
        if not root:
            return 0

        if root.val < low:
            # All nodes in left subtree are out of range
            return self.rangeSumBST(root.right, low, high)
        elif root.val > high:
            # All nodes in right subtree are out of range
            return self.rangeSumBST(root.left, low, high)
        else:
            # Node is within range, include it in sum
            return (root.val +
                    self.rangeSumBST(root.left, low, high) +
                    self.rangeSumBST(root.right, low, high))
```

1. Finding Mode in a Binary Search Tree (BST)

Challenges:

- **Handling Duplicate Values**: Keeping track of repeated values during in-order traversal is crucial to correctly determine the mode(s) of the BST. Since a mode is the most frequently occurring value, this requires careful attention to increment counts accurately when consecutive values are the same.
- **Stack Overflow in Recursion**: In cases where the BST is deep or unbalanced, the recursion depth can become too large, leading to potential stack overflow issues.
- **Single Pass Optimization**: A common challenge is optimizing the solution to find the mode(s) in a single traversal rather than multiple passes. This requires updating the mode dynamically while traversing the tree.

2. Minimum Absolute Difference in a BST

Challenges:

- **Excessive Memory Usage**: This approach stores all the values of the BST in an array, which can be inefficient in terms of space, especially for large trees. It may be possible to compute the minimum difference during the in-order traversal to avoid storing all values.
- **Sorted Nature of In-order Traversal**: Leveraging the fact that in-order traversal of a BST gives a sorted sequence, the algorithm minimizes the difference between adjacent values. However, ensuring this sorting is applied correctly without missing any values can be tricky in more complex tree structures.

3. Two Sum IV – Input is a BST

Challenges:

- **Efficiently Finding Complements**: The main challenge here is determining if the difference between the target sum and the current node's value has already been encountered. Using a set allows for quick lookups, but ensuring the set doesn't grow unnecessarily large is important for memory efficiency.
- **Traversal Strategy**: It's important to ensure that you don't overlook any potential pairs of values by improperly terminating the traversal early. The order of traversal can influence whether all valid pairs are discovered.

4. Deleting a Node in a BST

Challenges:

- **Handling Different Deletion Scenarios**: When deleting a node, there are three distinct scenarios: removing a leaf node, removing a node with one child, and removing a node with two children. The case involving two children is particularly complex, as it requires finding the node's in-order successor (the smallest value in its right subtree) and carefully restructuring the tree.
- **Tree Imbalance**: Deleting a node can cause the tree to become unbalanced, especially if the BST is already skewed. If the tree becomes imbalanced, it may degrade performance for future operations, making the tree behave more like a linked list.

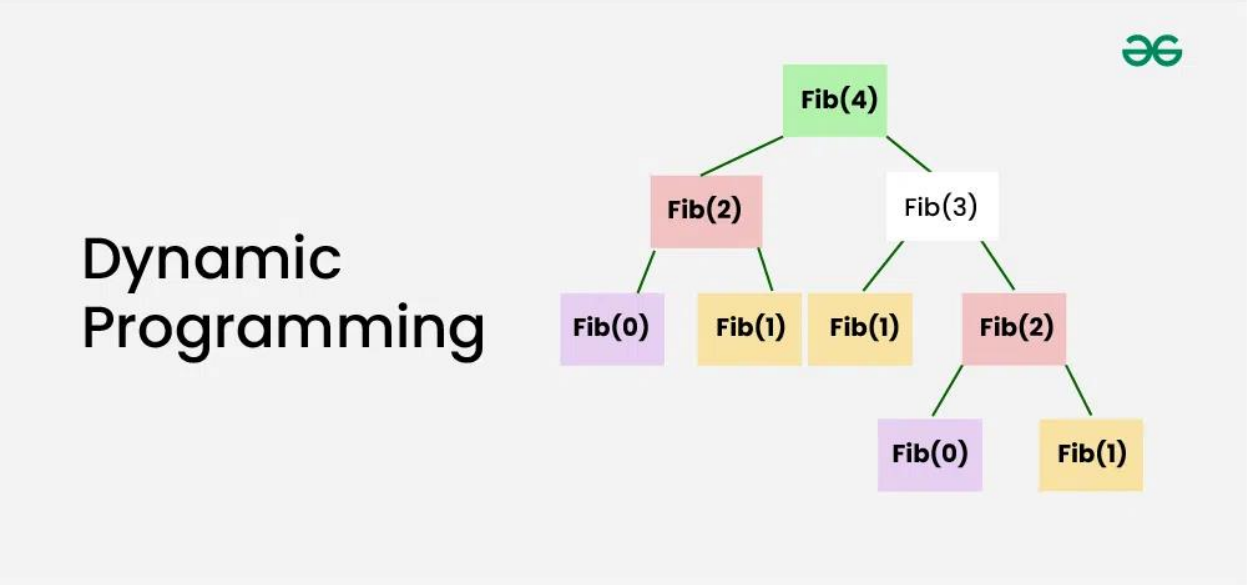
5. Range Sum of a BST

Challenges:

- **Efficiently Narrowing the Range**: The key challenge is efficiently filtering out nodes that lie outside the given range. If not optimized, the algorithm might end up traversing parts of the tree that don't contribute to the sum, which would increase the time complexity.
- **Managing Recursion Depth**: As with other tree problems, an unbalanced tree can lead to deep recursion, which could cause stack overflow errors in environments with limited stack space.

Dynamic Programming:

Dynamic Programming (DP) is a powerful technique used to solve complex problems by breaking them down into simpler subproblems. It is particularly useful for optimization problems, where you're trying to find the best possible solution among many possible options.



Techniques:

1.Memoization (Top-Down Approach):

- In memoization, you start with the original problem and recursively solve the subproblems. If a subproblem has already been solved, its result is simply reused.
- Every time you solve a subproblem, store the result in a lookup table (or a memo array).
- Before solving a subproblem, check if it has been solved earlier. If so, return the stored result instead of solving it again.
- Example: Fibonacci sequence using memorization

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

2.Tabulation (Bottom-Up Approach):

- In tabulation, you solve the subproblems first, storing their solutions in a table, and then use these solutions to build up to the solution for the main problem.
- You solve smaller subproblems first and store their results in a table (like an array).
- You gradually combine these solutions to solve larger and larger subproblems.
- Example: Fibonacci sequence using tabulation

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

How to solve a Dynamic Programming Problem?

To dynamically solve a problem, we need to check two necessary conditions:

- **Overlapping Subproblem** When the solutions to the same subproblems are needed repetitively for solving the actual problem. The problem is said to have overlapping subproblems property.
- **Optimal Substructure Property:** If the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems then the problem is said to have Optimal Substructure Property.

Steps to solve a Dynamic programming problem:

1. Identify if it is a Dynamic programming problem.
2. Decide a state expression with the Least parameters.3. Formulate state and transition relationships.
4. Do tabulation (or memorization).

Longest Palindromic Substring:

Code:

```
class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        if not s or len(s) == 1:
            return s

        def expand_around_center(left, right):

            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1

            return left + 1, right - 1

        start, end = 0, 0

        for i in range(len(s)):

            left1, right1 = expand_around_center(i, i)

            left2, right2 = expand_around_center(i, i + 1)
```

```
            if right1 - left1 > end - start:
                start, end = left1, right1
            if right2 - left2 > end - start:
                start, end = left2, right2


        return s[start:end + 1]



if __name__ == "__main__":
    solution = Solution()

    s1 = "babad"
    print(solution.longestPalindrome(s1))

    s2 = "cbbd"
```

Generate Parentheses:
Code:

```
class Solution(object):
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        result = []

        def backtrack(current_string, open_count, close_count):

            if len(current_string) == 2 * n:
                result.append(current_string)
                return

            if open_count < n:
                backtrack(current_string + '(', open_count + 1, close_count)

            if close_count < open_count:
                backtrack(current_string + ')', open_count, close_count + 1)

        backtrack('', 0, 0)

        return result


solution = Solution()

n = 3
print(solution.generateParenthesis(n))
```

Maximum Subarray
Code:

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        current_sum = nums[0]
        max_sum = nums[0]

        for i in range(1, len(nums)):
            current_sum = max(nums[i], current_sum + nums[i])

            max_sum = max(max_sum, current_sum)

        return max_sum


solution = Solution()

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(solution.maxSubArray(nums))
```

Climbing Stairs

Code:

```
class Solution(object):
    def climbStairs(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n == 0:
            return 1
        elif n == 1:
            return 1

        prev1, prev2 = 1, 1


        for i in range(2, n + 1):
            current = prev1 + prev2
            prev2 = prev1
            prev1 = current

        return prev1


solution = Solution()
print(solution.climbStairs(2))
```

Fibonacci Number

Code:

```
class Solution(object):
    def fib(self, n):
```

```
        """
        :type n: int
        :rtype: int
        """
        if n == 0:
            return 0
        elif n == 1:
            return 1

        dp = [0] * (n + 1)
        dp[0], dp[1] = 0, 1

        for i in range(2, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]

        return dp[n]


solution = Solution()
print(solution.fib(2))
```