```
arr = [-1, 0, 1, 2, -1, -4]
print(find_triplets_with_zero_sum(arr))
# Output: [[-1, -1, 2], [-1, 0, 1]]
```

INDEX 2D ARRAY IN 1D

To access elements of a 2D array using a 1D array index, you can convert the 2D array into a 1D array by flattening it. The goal is to find the 1D index corresponding to any `(row, column)` of a 2D array. For an array with `m` rows and `n` columns, the index for an element at position `(i, j)` can be computed as:

$$\text{index} = i \times n + j$$

Where `i` is the row number, and `j` is the column number.

```python
def index_2d_to_1d(row, col, num_cols):
    return row * num_cols + col

# Example usage:
row, col, num_cols = 2, 3, 4
index = index_2d_to_1d(row, col, num_cols)
print(f'1D index: {index}')  # Output: 1D index: 11
```

For a 2D array with `4` columns, the element at position `(2, 3)` corresponds to index `11` in a flattened 1D array.

Example:

2D array:

```
[
 [0,  1,  2,  3],
 [4,  5,  6,  7],
 [8,  9, 10, 11]
]
```

Flattened 1D array:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Element at (2, 3) = 11, which is at index 11 in the 1D array.

KTH LARGEST ELEMENT

The task is to find the Kth largest element in an unsorted array. The Kth largest element means the element that would be at position K if the array were sorted in descending order. There are several approaches to solve this, such as sorting the array or using a min-heap.

```python
def kth_largest(arr, k):
    return sorted(arr, reverse=True)[k-1]

# Example usage:
arr = [3, 2, 1, 5, 6, 4]
k = 2
print(kth_largest(arr, k))  # Output: 5
```

- The array is sorted in descending order, and the element at index `k-1` is returned.
- For `arr = [3, 2, 1, 5, 6, 4]` and `k = 2`, the sorted array is `[6, 5, 4, 3, 2, 1]`. The 2nd largest element is `5`.

MEDIAN TWO ARRAY

The task is to find the median of two sorted arrays. The median is the middle element in a sorted list of numbers. If the total number of elements is even, the median is the average of the two middle elements. The challenge is to solve this problem efficiently in O(log(min(m,n))) time complexity.

```python
def find_median_sorted_arrays(nums1, nums2):
    merged = sorted(nums1 + nums2)
    n = len(merged)

    if n % 2 == 1:
        return merged[n // 2]
    else:
        return (merged[n // 2 - 1] + merged[n // 2]) / 2

# Example usage:
nums1 = [1, 3]
nums2 = [2]
print(find_median_sorted_arrays(nums1, nums2))  # Output: 2.0
```

- **Binary Search**: The approach uses binary search on the smaller array (`nums1`) to find the correct partition.
- **Partition**: The arrays are partitioned such that the elements on the left side of both partitions are less than or equal to those on the right side.
- **Odd or Even Length**: If the combined length is odd, the median is the maximum of the left partitions. If it's even, the median is the average of the maximum element from the left and the minimum element from the right.

MONOTONIC ARRAY

An array is said to be **monotonic** if it is either entirely non-increasing or non-decreasing. Your task is to determine whether a given array is monotonic. That means the array is either sorted in increasing order or decreasing order, or all the elements are equal.

```
def is_monotonic(arr):
    increasing = decreasing = True

    for i in range(1, len(arr)):
        if arr[i] > arr[i - 1]:
            decreasing = False
        if arr[i] < arr[i - 1]:
            increasing = False

    return increasing or decreasing

# Example usage:
arr = [1, 2, 2, 3]
print(is_monotonic(arr))  # Output: True

arr = [6, 5, 4, 4]
print(is_monotonic(arr))  # Output: True

arr = [1, 3, 2]
print(is_monotonic(arr))  # Output: False
```

- We track two flags: `increasing` and `decreasing`.
- Traverse through the array and update the flags based on whether the current element is larger or smaller than the previous one.
- If the array is entirely increasing or decreasing, one of the flags will remain `True`.
- If both flags are `False`, the array is neither increasing nor decreasing.

PAIRS WITH GIVEN SUM

The task is to find all pairs of elements in an array that sum up to a given target. Each pair should only be listed once. The challenge can be solved using different approaches, such as using a hash set to store the required values for the sum or sorting the array and using the two-pointer technique.

```python
def find_pairs_with_sum(arr, target):
    seen = set()
    pairs = []

    for num in arr:
        diff = target - num
        if diff in seen:
            pairs.append((diff, num))
        seen.add(num)

    return pairs

# Example usage:
arr = [1, 5, 7, -1, 5]
target = 6
print(find_pairs_with_sum(arr, target))  # Output: [(1, 5), (7, -1), (1, 5)]
```

- We use a set `seen` to store the numbers we've already encountered.
- For each element in the array, we calculate its complement (i.e., `target - num`). If this complement is already in the set, we've found a pair that sums to the target.
- The time complexity is O(n) because each element is processed only once.

PERMUTATIONS

The task is to generate all possible permutations of a given array. A permutation is an arrangement of all the elements in a specific order. The challenge is often approached using recursion or iterative methods to ensure all combinations are explored.

```
def permute(nums):
    def backtrack(start):
        if start == len(nums):
            permutations.append(nums[:])  # Add a copy of the current
permutation
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]  # Swap
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start]  # Backtrack
(swap back)

    permutations = []
    backtrack(0)
    return permutations

# Example usage:
arr = [1, 2, 3]
print(permute(arr))
# Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2,
1]]
```

- The `backtrack` function is called recursively, swapping elements to create different arrangements.
- When the starting index reaches the length of the array, the current permutation is added to the result list.
- After exploring one possibility, we swap back to restore the original order for the next iteration (backtracking).

PREFIX SUM

The prefix sum array is a derived array that allows you to quickly calculate the sum of elements in a specific range of the original array. The prefix sum at index `i` represents the sum of all elements from the start of the array up to index `i`. This is useful for efficiently querying the sum of subarrays.

```python
def prefix_sum(arr):
    prefix = [0] * (len(arr) + 1)

    for i in range(len(arr)):
        prefix[i + 1] = prefix[i] + arr[i]

    return prefix

# Example usage:
arr = [1, 2, 3, 4, 5]
prefix = prefix_sum(arr)
print(prefix)  # Output: [0, 1, 3, 6, 10, 15]
```

- A new array `prefix` is initialized with a length of `len(arr) + 1` to include the initial sum of zero.

- Iterate through the original array, adding each element to the cumulative sum stored at `prefix[i + 1]`.

- The value at `prefix[i + 1]` gives the sum of elements from index `0` to `i`.

PRODUCT SUM

The product sum of an array is a variation of the sum where you multiply each element of the array by its depth level. If the element is an integer, its contribution to the product sum is simply the integer multiplied by its depth. If the element is an array, you recursively calculate its product sum by increasing the depth.

```python
def product_sum(arr, depth=1):
    total = 0

    for elem in arr:
        if isinstance(elem, list):
            total += product_sum(elem, depth + 1)  # Recurse for nested 
arrays
        else:
            total += elem

    return total * depth

# Example usage:
arr = [1, 2, [3, 4, [5]], 5]
result = product_sum(arr)
print(result)  # Output: 43
```

- The function `product_sum` takes an array and an optional depth parameter (defaulting to 1).
- It initializes a total sum and iterates through each element in the array.
- If the element is a list, the function recursively calls itself, increasing the depth by 1.
- If the element is an integer, it is added to the total.
- Finally, the total is multiplied by the depth to account for the product sum contribution.

SPARSE TABLE

A Sparse Table is a data structure that enables efficient querying of the minimum or maximum in a static array over a range of indices. It preprocesses the data in O(n log n) time, allowing queries to be answered in O(1) time. Sparse Tables are particularly effective for immutable arrays where the range queries are frequent but updates are rare.

```
import math

class SparseTable:
    def __init__(self, arr):
        self.n = len(arr)
        self.log = math.ceil(math.log2(self.n)) + 1
        self.table = [[0] * self.log for _ in range(self.n)]

        # Initialize the first column of the table
        for i in range(self.n):
            self.table[i][0] = arr[i]

        # Build the Sparse Table
        for j in range(1, self.log):
            for i in range(self.n - (1 << j) + 1):
                self.table[i][j] = min(self.table[i][j - 1], self.table[i
+ (1 << (j - 1))][j - 1])

    def range_min_query(self, left, right):
        j = int(math.log2(right - left + 1))
        return min(self.table[left][j], self.table[right - (1 << j) +
1][j])

# Example usage:
arr = [1, 3, 2, 7, 9, 11]
sparse_table = SparseTable(arr)

# Range minimum query from index 1 to 4
result = sparse_table.range_min_query(1, 4)
```

```
print(result)  # Output: 2
```

Binary Tree

A Binary Tree is a hierarchical data structure in which each node has at most two children, referred to as the left and right children. Binary trees are fundamental in computer science for various applications, including search trees, heaps, and expression parsing.

AVL TREE

An AVL Tree is a self-balancing binary search tree where the difference in heights between the left and right subtrees (the balance factor) is at most one for all nodes. This balancing ensures that the tree remains approximately balanced, leading to O(log n) time complexity for insertions, deletions, and lookups.

```python
class TreeNode:
    def __init__(self, value):
        self.value, self.left, self.right, self.height = value, None, 
None, 1

class AVLTree:
    def insert(self, root, value):
        if not root:
            return TreeNode(value)
        if value < root.value:
            root.left = self.insert(root.left, value)
        else:
            root.right = self.insert(root.right, value)

        root.height = 1 + max(self.get_height(root.left), 
self.get_height(root.right))
        return self.balance(root)

    def balance(self, node):
        if (b := self.get_balance(node)) > 1:
            return self.rotate_right(node) if self.get_balance(node.left) 
>= 0 else self.rotate_right(node.left)
        if b < -1:
            return self.rotate_left(node) if self.get_balance(node.right) 
<= 0 else self.rotate_left(node.right)
        return node

    def rotate_left(self, z):
        y = z.right
        z.right, y.left = y.left, z
        z.height = 1 + max(self.get_height(z.left),
self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left),
self.get_height(y.right))
        return y

    def rotate_right(self, z):
        y = z.left
        z.left, y.right = y.right, z
        z.height = 1 + max(self.get_height(z.left),
self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left),
self.get_height(y.right))
        return y

    def get_height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.get_height(node.left) - self.get_height(node.right)

    def inorder_traversal(self, node):
        return self.inorder_traversal(node.left) + [node.value] +
self.inorder_traversal(node.right) if node else []

# Example usage:
avl_tree, root = AVLTree(), None
for value in [10, 20, 30, 40, 50, 25]:
    root = avl_tree.insert(root, value)

print(avl_tree.inorder_traversal(root))
# Output: [10, 20, 25, 30, 40, 50]
```

BINARY SEARCH RECURSIVE

Recursive Binary Search is a method to efficiently find an element in a sorted array. By repeatedly dividing the search interval in half, it reduces the time complexity to O(log n). If the value is found, its index is returned; otherwise, the search continues until the element is not found.

```
def binary_search(arr, target, left, right):
    if left > right:
        return -1  # Target not found
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid  # Target found
    return binary_search(arr, target, left, mid - 1) if target < arr[mid]
else binary_search(arr, target, mid + 1, right)

# Example usage:
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7
result = binary_search(arr, target, 0, len(arr) - 1)
print(result)  # Output: 6 (index of target)
```

binary_search Function:

- Takes an array (`arr`), a `target` value, and the current bounds (`left`, `right`) as input.
- Base case checks if the current bounds are valid (left ≤ right).
- Calculates the midpoint (`mid`), checks if it's equal to the target, and returns its index if found.
- Recursively searches in the left or right half based on the comparison.

BINARY TREE MIRROR

A binary tree mirror transformation creates a mirror image of the tree. This involves swapping the left and right children of each node recursively, resulting in a tree that reflects the original structure across its vertical axis.

```python
class TreeNode:
    def __init__(self, value):
        self.value, self.left, self.right = value, None, None

class BinaryTree:
    def mirror(self, node):
        if node:
            node.left, node.right = node.right, node.left  # Swap
children
            self.mirror(node.left)  # Mirror left subtree
            self.mirror(node.right)  # Mirror right subtree

    def inorder_traversal(self, node):
        return self.inorder_traversal(node.left) + [node.value] +
self.inorder_traversal(node.right) if node else []

# Example usage:
root = TreeNode(1)
root.left, root.right = TreeNode(2), TreeNode(3)
root.left.left, root.left.right = TreeNode(4), TreeNode(5)

tree = BinaryTree()
print("Inorder before mirroring:", tree.inorder_traversal(root))
# Output: [4, 2, 5, 1, 3]

tree.mirror(root)
print("Inorder after mirroring:", tree.inorder_traversal(root))
# Output: [3, 1, 5, 2, 4]
```

BINARY NODE SUM

The Binary Tree Node Sum operation calculates the sum of all the node values in a binary tree. This can be done using a recursive approach, where the sum of each node is obtained by adding its value to the sums of its left and right subtrees.

```python
class TreeNode:
    def __init__(self, value):
        self.value, self.left, self.right = value, None, None

class BinaryTree:
    def sum_nodes(self, node):
        if not node:
            return 0  # Base case: return 0 if the node is None
        return node.value + self.sum_nodes(node.left) +
self.sum_nodes(node.right)  # Recursive sum

# Example usage:
root = TreeNode(1)
root.left, root.right = TreeNode(2), TreeNode(3)
root.left.left, root.left.right = TreeNode(4), TreeNode(5)

tree = BinaryTree()
total_sum = tree.sum_nodes(root)
print(total_sum)  # Output: 15 (1 + 2 + 3 + 4 + 5)
```

- **TreeNode Class**: Represents each node in the binary tree, holding a value and pointers to its left and right children.

- **BinaryTree Class**: `sum_nodes` method recursively computes the sum of node values. It returns 0 for null nodes and adds the current node's value to the sums of its left and right children.

BINARY PATH SUM

The Binary Tree Path Sum operation checks if there exists a path from the root to any leaf node such that the sum of the values along the path equals a specified target sum. This can be achieved through a depth-first search (DFS) approach, recursively subtracting the node values from the target as you traverse down the tree.

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

class BinaryTree:
    def has_path_sum(self, node, target):
        if not node:
            return target == 0  # Base case: check if target is achieved
        target -= node.value  # Subtract the current node's value
        # Check both left and right subtrees
        return self.has_path_sum(node.left, target) or
self.has_path_sum(node.right, target)

# Example usage:
root = TreeNode(5)
root.left, root.right = TreeNode(4), TreeNode(8)
root.left.left, root.left.right = TreeNode(11), TreeNode(2)
root.right.left, root.right.right = TreeNode(13), TreeNode(4)

tree = BinaryTree()
target_sum = 22
result = tree.has_path_sum(root, target_sum)
print(result)  # Output: True (5 -> 4 -> 11 -> 2)
```

BINARY TREE TRAVERSALS

Binary Tree Traversals refer to the methods of visiting all the nodes in a binary tree in a specific order. The three primary types of depth-first traversals are:

- **Inorder Traversal**: Left subtree, Root, Right subtree.
- **Preorder Traversal**: Root, Left subtree, Right subtree.
- **Postorder Traversal**: Left subtree, Right subtree, Root.

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

class BinaryTree:
    def inorder(self, node):
        return self.inorder(node.left) + [node.value] +
self.inorder(node.right) if node else []

    def preorder(self, node):
        return [node.value] + self.preorder(node.left) +
self.preorder(node.right) if node else []

    def postorder(self, node):
        return self.postorder(node.left) + self.postorder(node.right) +
[node.value] if node else []

# Example usage:
root = TreeNode(1)
root.left, root.right = TreeNode(2), TreeNode(3)
root.left.left, root.left.right = TreeNode(4), TreeNode(5)

tree = BinaryTree()
print("Inorder:", tree.inorder(root))   # Output: [4, 2, 5, 1, 3]
print("Preorder:", tree.preorder(root)) # Output: [1, 2, 4, 5, 3]
print("Postorder:", tree.postorder(root)) # Output: [4, 5, 2, 3, 1]
```

DIAMETER OF BINARY TREE

The diameter of a binary tree is the length of the longest path between any two nodes in the tree. This path may or may not pass through the root. The diameter can be calculated by determining the height of the left and right subtrees for each node and keeping track of the maximum diameter found during the traversal.

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

class BinaryTree:
    def diameter(self, node):
        self.max_diameter = 0  # Initialize maximum diameter

        def height(n):
            if not n:
                return 0  # Base case: height of null is 0
            left_height = height(n.left)  # Height of left subtree
            right_height = height(n.right)  # Height of right subtree
            self.max_diameter = max(self.max_diameter, left_height +
right_height)  # Update maximum diameter
            return max(left_height, right_height) + 1  # Return height

        height(node)  # Start the height calculation
        return self.max_diameter  # Return the maximum diameter

# Example usage:
root = TreeNode(1)
root.left, root.right = TreeNode(2), TreeNode(3)
root.left.left, root.left.right = TreeNode(4), TreeNode(5)

tree = BinaryTree()
```

```
print(tree.diameter(root))
# Output: 3 (length of the path 4 -> 2 -> 1 -> 3)
```

DIFF VIEWS OF BINARY TREE

Different views of a binary tree provide distinct perspectives of the tree's structure. The most common views are:

1. **Left View**: The nodes visible when the tree is viewed from the left side.
2. **Right View**: The nodes visible when the tree is viewed from the right side.
3. **Top View**: The nodes visible from the top of the tree.
4. **Bottom View**: The nodes visible from the bottom of the tree.

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

class BinaryTree:
    def left_view(self, node):
        result = []
        self._left_view_util(node, 0, result)
        return result

    def _left_view_util(self, node, level, result):
        if not node:
            return
        if level == len(result):  # First node at this level
            result.append(node.value)
        self._left_view_util(node.left, level + 1, result)  # Recur for 
left child
        self._left_view_util(node.right, level + 1, result)  # Recur for 
right child

    def right_view(self, node):
        result = []
        self._right_view_util(node, 0, result)
        return result

    def _right_view_util(self, node, level, result):
        if not node:
            return
        if level == len(result):  # First node at this level
            result.append(node.value)
        self._right_view_util(node.right, level + 1, result)  # Recur for
right child
        self._right_view_util(node.left, level + 1, result)  # Recur for
left child

# Example usage:
root = TreeNode(1)
root.left, root.right = TreeNode(2), TreeNode(3)
root.left.left, root.left.right = TreeNode(4), TreeNode(5)
root.right.right = TreeNode(6)

tree = BinaryTree()
print("Left View:", tree.left_view(root))   # Output: [1, 2, 4]
print("Right View:", tree.right_view(root)) # Output: [1, 3, 6]
```

DISTRIBUTE COINS

The "Distribute Coins" problem involves distributing a given number of coins to a set of piles according to specific rules. The most common variation is to ensure that each pile has at least one coin and to distribute the remaining coins as evenly as possible.

```
def distribute_coins(n, k):
    if n < k:  # Not enough coins to give at least one to each pile
        return 0
    return comb(n - 1, k - 1)  # Combinations of (n-1) choose (k-1)

def comb(n, k):  # Calculate combinations nCk
    if k > n or k < 0:
        return 0
    if k == 0 or k == n:
        return 1
    k = min(k, n - k)  # Take advantage of symmetry
    c = 1
    for i in range(k):
        c = c * (n - i) // (i + 1)
    return c

# Example usage:
n = 10  # Total coins
k = 3   # Total piles
result = distribute_coins(n, k)
print(result)  # Output: 36 (ways to distribute 10 coins into 3 piles)
```

FENWICK TREE

A Fenwick Tree, also known as a Binary Indexed Tree (BIT), is a data structure that efficiently supports prefix sum queries and updates. It allows both operations in logarithmic time, making it suitable for scenarios where frequent updates and queries are needed, such as in cumulative frequency tables.

```
class FenwickTree:
    def __init__(self, size):
        self.size = size
        self.tree = [0] * (size + 1)

    def update(self, index, value):
        while index <= self.size:
            self.tree[index] += value
            index += index & -index  # Move to the next index

    def query(self, index):
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= index & -index  # Move to the parent index
        return total

# Example usage:
ft = FenwickTree(5)  # Create a Fenwick Tree of size 5
ft.update(1, 3)      # Add 3 to index 1
ft.update(2, 2)      # Add 2 to index 2
ft.update(3, 5)      # Add 5 to index 3
print(ft.query(3))   # Output: 10 (3 + 2 + 5)
print(ft.query(2))   # Output: 5 (3 + 2)
```

FLOOR AND CEILING

The "Floor and Ceiling" problem involves finding the floor and ceiling values of a given number in a sorted array. The floor is the largest number in the array that is less than or equal to the target value, while the ceiling is the smallest number that is greater than or equal to the target value.

```python
def find_floor_and_ceiling(arr, target):
    floor, ceiling = None, None
    for num in arr:
        if num <= target:
            floor = num  # Update floor if num is less than or equal to
target
        if num >= target and ceiling is None:  # Update ceiling only once
            ceiling = num
    return floor, ceiling

# Example usage:
arr = [1, 2, 8, 10, 10, 12, 19]
target = 5
floor, ceiling = find_floor_and_ceiling(arr, target)
print(f"Floor: {floor}, Ceiling: {ceiling}")  # Output: Floor: 2,
Ceiling: 8
```

IS SORTED

The "Is Sorted" problem involves checking whether a given list (or array) is sorted in either ascending or descending order. This can be useful for validating input data or optimizing algorithms that assume sorted input.

```
def is_sorted(arr):
    ascending, descending = True, True
    for i in range(1, len(arr)):
        if arr[i] < arr[i - 1]:  # Check for ascending order
            ascending = False
        if arr[i] > arr[i - 1]:  # Check for descending order
            descending = False
    return ascending, descending

# Example usage:
arr = [1, 2, 3, 4, 5]
ascending, descending = is_sorted(arr)
print(f"Is Ascending: {ascending}, Is Descending: {descending}")  #
Output: Is Ascending: True, Is Descending: False
```

IS SUM TREE

The "Is Sum Tree" problem involves determining whether a given binary tree is a Sum Tree. A binary tree is considered a Sum Tree if, for every node in the tree, the value of the node is equal to the sum of the values of its left and right children. This property must hold for all nodes in the tree.

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

def is_sum_tree(node):
    if not node:  # An empty tree is a Sum Tree
        return 0
    if not node.left and not node.right:  # Leaf nodes
        return node.value

    left_sum = is_sum_tree(node.left)  # Recursively get left subtree sum
    right_sum = is_sum_tree(node.right)  # Recursively get right subtree
sum

    if left_sum == -1 or right_sum == -1 or node.value != left_sum +
right_sum:
        return -1  # Return -1 if it's not a Sum Tree

    return left_sum + right_sum + node.value  # Return the total sum of
this subtree

# Example usage:
root = TreeNode(26)
root.left = TreeNode(10)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(6)
root.right.right = TreeNode(3)

result = is_sum_tree(root)
print(f"Is Sum Tree: {result != -1}")  # Output: Is Sum Tree: True
```

LOWEST COMMON ANCESTOR

The Lowest Common Ancestor (LCA) of two nodes in a binary tree is the deepest node that is an ancestor of both nodes. It is commonly used in various tree-related algorithms and problems. The LCA can be efficiently found using a recursive approach or by maintaining parent pointers in the tree.

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

def find_lca(root, n1, n2):
    if not root:
        return None
    if root.value == n1 or root.value == n2:
        return root

    left_lca = find_lca(root.left, n1, n2)
    right_lca = find_lca(root.right, n1, n2)

    if left_lca and right_lca:
        return root  # This is the LCA
    return left_lca if left_lca else right_lca  # Either one of the two

# Example usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

lca = find_lca(root, 4, 5)
print(f"LCA of 4 and 5: {lca.value}")  # Output: LCA of 4 and 5: 2
```

MAXIMUM FENWICK TREE

A Maximum Fenwick Tree (or Maximum Binary Indexed Tree) is a data structure that supports efficient queries and updates for finding the maximum element within a range of an array. It is an extension of the standard Fenwick Tree (or Binary Indexed Tree) but is specifically designed to handle maximum queries instead of sum queries.

```
class MaxFenwickTree:
    def __init__(self, size):
        self.size = size
        self.tree = [float('-inf')] * (size + 1)

    def update(self, index, value):
        while index <= self.size:
            self.tree[index] = max(self.tree[index], value)
            index += index & -index  # Move to the next index

    def query(self, index):
        max_val = float('-inf')
        while index > 0:
            max_val = max(max_val, self.tree[index])
            index -= index & -index  # Move to the parent index
        return max_val

    def range_max(self, left, right):
        return max(self.query(right), self.query(left - 1))

# Example usage:
fenwick_tree = MaxFenwickTree(5)
```

```
fenwick_tree.update(1, 10)
fenwick_tree.update(2, 20)
fenwick_tree.update(3, 15)

print(f"Maximum value between index 1 and 3: {fenwick_tree.range_max(1,
3)}")
# Output: 20
```

NUMBER OF POSSIBLE BINARY TREES

The number of possible binary trees that can be formed with nnn nodes can be calculated using the **Catalan number**. The nthn^{th}nth Catalan number gives the count of distinct binary trees that can be formed using nnn distinct nodes. The formula for the nthn^{th}nth Catalan number is:

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

```python
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def catalan_number(n):
    return factorial(2 * n) // (factorial(n + 1) * factorial(n))

# Example usage:
n = 3
print(f"Number of possible binary trees with {n} nodes:
{catalan_number(n)}")
# Output: 5
```