654. Maximum Binary Tree

Problem Description

You are given an array named `nums`, which contains a set of integers and has no duplicate elements. Your task is to use this array to build what's called a *maximum binary tree*. This tree is constructed using a specific set of rules:

- The maximum value in the array becomes the root node of the tree.
- The subarray to the left of the maximum value is used to construct the left subtree of the tree by applying the same rule.
- Similarly, the subarray to the right of the maximum value is used to build the right subtree by applying the same rule.

This process is recursive, meaning that you apply the same set of rules to each subarray (left and right of the maximum value found) to build the entire tree. The goal is to construct and return the maximum binary tree.

Intuition

To build the maximum binary tree, we need to find the maximum element in the array and make it the root of the tree. We then split the array into two subarrays: one to the left and one to the right of where the maximum value was found. These subarrays will be used to construct the left and right subtrees, respectively. We follow the same strategy recursively for the subtrees.

This approach leads us to divide-and-conquer strategy, where we divide the problem into smaller instances of the same problem (finding the maximum element and building left and right subtrees) and then combine the solutions (subtrees) to construct the final tree.

The provided solution uses a helper function `dfs` that implements this recursive strategy. The base case for the recursion is when there are no elements in the current subarray (`nums`), in which case the function returns `None` because there's nothing left to construct.

When elements are present, the function:

1. Finds the maximum value in the current subarray.
2. Determines the index of this maximum value.
3. Creates a new tree node with the maximum value.
4. Recursively calls itself to build the left subtree with the elements to the left of the maximum value.
5. Recursively calls itself to build the right subtree with the elements to the right of the maximum value.
6. Links the constructed left and right subtrees to the created node (now the parent node).
7. Returns the current tree node (which forms a part of the larger tree).

The recursion unwinds, building up the entire tree, which is finally returned by the outermost call to `dfs`.

Solution Approach

The solution implementation makes use of the divide-and-conquer strategy, recursion, and the binary tree data structure. Here's how these concepts are applied to construct the maximum binary tree:

1. **Divide and Conquer**: The array `nums` is repeatedly divided into two subarrays around the maximum value found. This approach splits the problem into smaller problems, gradually building the left and right subtrees.

2. **Recursion**: To manage the repetitive task of building subtrees from subarrays, the solution implements a recursive helper function (`dfs`). Recursion continues until the base case is met, which occurs when the passed-in subarray is empty (`not nums`), at which point the function returns `None` since there are no more nodes to create.

3. **Binary Tree Construction**: The nature of the problem requires constructing a binary tree, so the `TreeNode` class is used to create tree nodes. Each node has a value as well as potential left and right children, which correspond to the left and right subtrees.

When employing recursion to build the tree, the approach is as follows:

a. Identify the maximum value `val` in the current subarray `nums` using the `max()` function and find its index `i` with `nums.index(val)`.

b. Create a new `TreeNode` with `val` as its value, which will serve as the root node for the current (sub)tree.

c. For the left child of the current node, recursively call `dfs(nums[:i])`, which constructs the subtree from the elements to the left of `val`.

d. For the right child, a similar recursive call is made with `dfs(nums[i + 1:])` to build the subtree from the elements to the right of `val`.

e. After the recursive calls, the left and right children are connected to the root node of the subtree, thereby completing the subtree construction.

Each recursive call builds a part of the tree and returns it to the calling function, which then attaches it to the appropriate left or right child of the current node. This step-by-step process continues until the original call returns the entire maximum binary tree, which is the final output of the `constructMaximumBinaryTree` function.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Let's say we have the following array `nums`:

```
nums = [3, 2, 1, 6, 0, 5]
```

According to the problem description, we need to construct a maximum binary tree following the given rules. Here's how the approach solves this:

1. Find the maximum value in `nums`. In this case, it is `6`.
2. Since `6` is the maximum value, it will be the root of the maximum binary tree.
3. Divide the array into two subarrays around the maximum value found. Here we have `left_sub = [3, 2, 1]` and `right_sub = [0, 5]`.
4. To construct the left subtree, repeat the same process for `left_sub`. Find the maximum value, which is `3`, and make it the left child of `6`.
5. The subarray to the left of `3` is empty, so the left child of `3` would be `None`. The subarray to the right of `3` is `[2, 1]`.
   - Find the maximum value in `[2, 1]`, which is `2`, and make it the right child of `3`.
   - Now, `2` will have a right child which is `1`, as `1` is the maximum and only element in the subarray to its left, and no elements are to its right.
6. For the right subtree of the root `6`, we examine `right_sub`. The maximum value in `[0, 5]` is `5`, so `5` becomes the right child of `6`.
7. Repeat the process for the subarray to the left of `5`, which is `[0]`. Since `0` is the only element, it becomes the left child of `5`. No elements are to the right of `5`, so its right child is `None`.

The final maximum binary tree constructed from `nums` looks like this:

```
    6
   / \
  3   5
 / \ /
2  1 0
 \
  1
```

This example demonstrates the divide-and-conquer and recursive nature of the solution, where the maximum value in each subarray becomes the root of a subtree, with recursive calls constructing its children until the entire tree is built.

Python Solution

```python
from typing import List, Optional

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def constructMaximumBinaryTree1(self, nums: List[int]) -> Optional[TreeNode]:
        # Helper function to construct the tree using a divide and conquer approach
        def build(left, right):
            # Base case: if there are no numbers, return None
            if not left <= right:
                return None

            # Find the maximum value in the list of numbers
            max_value = max(nums[left:right+1])
            # Find the index of the maximum value
            max_index = nums.index(max_value, left, right+1)

            # Create a tree node with the maximum value
            root = TreeNode(max_value)
            # Recursively build the left subtree using the elements to the left of the max value
            root.left = build(left, max_index-1)
            # Recursively build the right subtree using the elements to the right of the max value
            root.right = build(max_index+1, right)

            return root

        # Call the helper function with the initial list of numbers
        return build(0, len(nums)-1)

# This constructs a maximum binary tree as defined by the problem statement,
# where the root is the maximum number in the array, and the left and right subtrees
# are constructed from the elements before and after the maximum number, respectively.
```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // Declare an array to hold the input values
    private int[] nodeValues;

    /**
     * This method constructs a maximum binary tree from the given array.
     *
     * @param nums The input array containing elements to be used in the tree.
     * @return The constructed maximum binary tree's root node.
     */
    public TreeNode constructMaximumBinaryTree(int[] nums) {
        this.nodeValues = nums;
        // Start the recursive tree construction process from the full range (0 to length-1)
        return constructTreeInRange(0, nums.length - 1);
    }

    /**
     * This private helper method creates the maximum binary tree recursively.
     *
     * @param left The left boundary (inclusive) of the current subarray.
     * @param right The right boundary (inclusive) of the current subarray.
     * @return The root node of the constructed subtree.
     */
    private TreeNode constructTreeInRange(int left, int right) {
        // Base cases: when the left index is greater than the right, we've gone past the leaf node
        if (left > right) {
            return null;
        }

        int maxIndex = left; // Start with the leftmost index

        // Find the index of the maximum element in the current subarray
        for (int i = left; i <= right; ++i) {
            if (nodeValues[maxIndex] < nodeValues[i]) {
                maxIndex = i;
            }
        }

        // Create a new tree node with the maximum element as its value
        TreeNode root = new TreeNode(nodeValues[maxIndex]);

        // Recursively construct the left subtree using elements left to the maximum element
        root.left = constructTreeInRange(left, maxIndex - 1);
        // Recursively construct the right subtree using elements right to the maximum element
        root.right = constructTreeInRange(maxIndex + 1, right);

        return root; // Return the root node of the constructed subtree
    }
}
```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val; // The value of the node
    TreeNode *left; // Pointer to the left child
    TreeNode *right; // Pointer to the right child

    // Constructor for a tree node with a given value, initially with no children
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor for a tree node with given value, left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Constructs a maximum binary tree from the given integer array.
     * @param nums The vector of integers.
     * @return The root TreeNode of the constructed maximum binary tree.
     */
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        return constructTree(nums, 0, nums.size() - 1);
    }

private:
    /**
     * Helper function to construct the maximum binary tree using Depth First Search.
     * @param nums The vector of integers.
     * @param leftBoundary The left boundary of the current segment.
     * @param rightBoundary The right boundary of the current segment.
     * @return The root TreeNode of the constructed maximum binary tree for the segment.
     */
    TreeNode* constructTree(vector<int>& nums, int left, int right) {
        // If the current segment is invalid, return nullptr to indicate no subtree
        if (left > right) return nullptr;

        // Find the index of the maximum element in the current segment
        int maxIndex = left;
        for (int currentIndex = left; currentIndex <= right; ++currentIndex) {
            if (nums[maxIndex] < nums[currentIndex]) {
                maxIndex = currentIndex;
            }
        }

        // Create the root node of the subtree with the maximum element
        TreeNode* root = new TreeNode(nums[maxIndex]);

        // Recursively construct the left subtree with the elements before the maximum element
        root->left = constructTree(nums, left, maxIndex - 1);
        // Recursively construct the right subtree with the elements after the maximum element
        root->right = constructTree(nums, maxIndex + 1, right);

        // Return the root of the subtree
        return root;
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    // Constructor to create a tree node with given values, default to 0 for val and null for left and right.
    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Function to construct a maximum binary tree from an array of numbers.
// A maximum binary tree is a binary tree where every node has a value greater than its children.
function constructMaximumBinaryTree(nums: number[]): TreeNode | null {
    // If the array is empty, return null as no tree can be constructed.
    if (nums.length === 0) {
        return null;
    }

    // Find the maximum value and its index in the array.
    // The reduce method processes each number, keeping track of the largest number and its index.
    let [maxValue, maxIndex] = nums.reduce((number, number[]) => currentMax, value, index) => {
        return value > currentMax[0] ? [value, index] : currentMax;
    }, [-Infinity, -1]);

    // Destructure the result to get max value and index.
    const [maxValue, maxIndex] = maxValueAndIndex;

    // Recursively build the tree.
    // - The maximum value becomes the root node.
    // - The left child is constructed from the subarray left of the maximum value.
    // - The right child is constructed from the subarray right of the maximum value.
    let newNode = new TreeNode(
        maxValue,
        constructMaximumBinaryTree(nums.slice(0, maxIndex)),
        constructMaximumBinaryTree(nums.slice(maxIndex + 1))
    );

    // Return the constructed tree node.
    return newNode;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function `constructMaximumBinaryTree` is determined by multiple factors: the number of elements in the `nums` list, finding the maximum value in the current portion of the list, splitting the list into two parts, and recursively building the left and right subtrees.

1. For each node of the binary tree, finding the maximum element takes $O(n)$ time where n is the number of elements in the current portion of the list.
2. The `nums.index(val)` operation also takes $O(n)$ time for each node to find the index of the maximum element.
3. The `dfs` function is called recursively for each element in the list to create a node, meaning there will be n calls in total (where n is the size of the original list).

Considering the recursive nature and that finding the max and index takes linear time at each level of recursion, the total time complexity is $O(n^2)$ in the worst case, when the input list is sorted in ascending or descending order, causing the divide step to only reduce the problem size by 1 each time.

Space Complexity

The space complexity is determined by the recursive stack depth and the space needed to store the constructed binary tree.

1. In the best case (when the input list is already balanced), the depth of the recursive stack is $O(\log n)$ since the tree would be roughly balanced. This would happen when the maximum element always ends up being in the middle of the current subarray.
2. In the worst case (input list is sorted), the depth of the recursive stack is $O(n)$ because the constructed tree would be skewed (like a linked list), and hence we would have a chain of recursive calls equal to the size of the input list.

Because the space needed to store the binary tree is $O(n)$ and the recursive stack space in the worst case is also $O(n)$, the overall space complexity is $O(n)$.