654. Maximum Binary Tree Medium **Divide and Conquer** Stack Array **Binary Tree Monotonic Stack Leetcode Link**

Problem Description

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.

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

Similarly, the subarray to the right of the maximum value is used to build the right subtree by applying the same rule.

The subarray to the left of the maximum value is used to construct the left subtree of the tree 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.

tree.

Intuition

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

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 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.

1. Divide and Conquer: The array nums is repeatedly divided into two subarrays around the maximum value found. This approach

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.

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**

5. Recursively calls itself to build the right subtree with the elements to the right of the maximum value.

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:

splits the problem into smaller problems, specifically building the left and right subtrees.

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

the final output of the constructMaximumBinaryTree function.

1. Find the maximum value in nums. In this case, it is 6.

elements are to the right of 5, so its right child is None.

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

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.

val.

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

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

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

child of the current node. This step-by-step process continues until the original call returns the entire maximum binary tree, which is

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

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

Let's walk through a small example to illustrate the solution approach. Let's say we have the following array nums: 1 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

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

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].

to its right.

Python Solution

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

29

30

31

32

33

34

32

33

34

35

36

37

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

65

64 }

C++ Solution

4 struct TreeNode {

*/

6

8

9

10

11

12

13

15

14 };

*/

if (left > right) {

return null;

from typing import List, Optional

self.val = val

self.left = left

self.right = right

def build_max_tree(sub_nums):

return None

max_value = max(sub_nums)

root = TreeNode(max_value)

if not sub_nums:

return root

return build_max_tree(nums)

Definition for a binary tree node.

child of 6.

Example Walkthrough

approach solves this:

• Find the maximum value in [2, 1], which is 2, and make it the right child of 3. o Now, 2 will have a right child which is 1, as 2 is the maximum and only element in the subarray to its left, and no elements are

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

- becomes the root of a subtree, with recursive calls constructing its children until the entire tree is built.

This example demonstrates the divide-and-conquer and recursive nature of the solution, where the maximum value in each subarray

9 class Solution: def constructMaximumBinaryTree(self, nums: List[int]) -> Optional[TreeNode]: 12 # Helper function to construct the tree using a divide and conquer approach

Find the index of the maximum value

max_index = sub_nums.index(max_value)

Create a tree node with the maximum value

root.left = build_max_tree(sub_nums[:max_index])

root.right = build_max_tree(sub_nums[max_index + 1:])

Call the helper function with the initial list of numbers

This constructs a maximum binary tree as defined by the problem statement,

36 # where the root is the maximum number in the array, and the left and right subtrees

* This private helper method creates the maximum binary tree recursively.

// Base case: when the left index is greater than the right, we've gone past the leaf node

// Recursively construct the left subtree using elements left to the maximum element

// Recursively construct the right subtree using elements right to the maximum element

* @param left The left boundary (inclusive) of the current subarray.

* @return The root node of the constructed subtree.

for (int j = left; j <= right; ++j) {</pre>

maxIndex = j;

* Definition for a binary tree node.

int val; // The value of the node

TreeNode *left; // Pointer to the left child

TreeNode *right; // Pointer to the right child

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

private TreeNode constructTreeInRange(int left, int right) {

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

if (nodeValues[maxIndex] < nodeValues[j]) {</pre>

TreeNode root = new TreeNode(nodeValues[maxIndex]);

root.left = constructTreeInRange(left, maxIndex - 1);

root.right = constructTreeInRange(maxIndex + 1, right);

* @param right The right boundary (inclusive) of the current subarray.

// Find the index of the maximum element in the current subarray

// Create a new tree node with the maximum element as its value

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

// Constructor for a tree node with a given value, initially with no children

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

// Constructor for a tree node with given value, left and right children

Recursively build the left subtree using the elements to the left of the max value

Recursively build the right subtree using the elements to the right of the max value

Base case: if there are no numbers, return None

Find the maximum value in the list of numbers

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

```
# are constructed from the elements before and after the maximum number, respectively.
38
Java Solution
1 // 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;
9
           this.left = left;
10
           this.right = right;
13 }
14
   class Solution {
       // Declare an array to hold the input values
16
       private int[] nodeValues;
17
18
19
       /**
20
        * This method constructs a maximum binary tree from the given array.
21
22
        * @param nums The input array containing elements to be used in the tree.
23
        * @return The constructed maximum binary tree's root node.
24
        */
25
       public TreeNode constructMaximumBinaryTree(int[] nums) {
26
           this.nodeValues = nums;
27
           // Start the recursive tree construction process from the full range (0 to length-1)
           return constructTreeInRange(0, nums.length - 1);
28
29
30
31
       /**
```

46 47 48 49

21

22

23

24

26

28

29

32

33

39

40

41

43

45

44 }

);

```
16 class Solution {
 17 public:
 18
         /**
          * Constructs a maximum binary tree from the given integer array.
 19
          * @param nums The vector of integers.
 20
 21
          * @return The root TreeNode of the constructed maximum binary tree.
 22
          */
 23
         TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
             return constructTree(nums, 0, nums.size() - 1);
 24
 25
 26
    private:
 28
         /**
 29
          * Helper function to construct the maximum binary tree using Depth First Search.
          * @param nums The vector of integers.
 30
 31
          * @param left The left boundary of the current segment.
 32
          * @param right The right boundary of the current segment.
 33
          * @return The root TreeNode of the constructed maximum binary tree for the segment.
 34
 35
         TreeNode* constructTree(vector<int>& nums, int left, int right) {
 36
             // If the current segment is invalid, return nullptr to indicate no subtree
             if (left > right) return nullptr;
 37
 38
 39
             // Find the index of the maximum element in the current segment
 40
             int maxIndex = left;
 41
             for (int currentIndex = left; currentIndex <= right; ++currentIndex) {</pre>
                 if (nums[maxIndex] < nums[currentIndex]) {</pre>
 42
 43
                     maxIndex = currentIndex;
 44
 45
             // Create the root node of the subtree with the maximum element
             TreeNode* root = new TreeNode(nums[maxIndex]);
 50
             // Recursively construct the left subtree with the elements before the maximum element
 51
             root->left = constructTree(nums, left, maxIndex - 1);
 52
 53
             // Recursively construct the right subtree with the elements after the maximum element
 54
             root->right = constructTree(nums, maxIndex + 1, right);
 55
             // Return the root of the subtree
 56
 57
             return root;
 58
 59 };
 60
Typescript Solution
   // 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) {
8
           this.val = val;
9
           this.left = left;
10
           this.right = right;
11
12
13 }
14
  // 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.
18
       if (nums.length === 0) {
           return null;
20
```

Time Complexity The time complexity of the function constructMaximumBinaryTree is determined by multiple factors: the number of elements in the

reduce the problem size by 1 each time.

Time and Space Complexity

}, [-Infinity, -1]);

maxValue,

return rootNode;

// Recursively build the tree:

let rootNode = new TreeNode(

// Find the maximum value and its index in the array.

// Destructure the result to get max value and index.

constructMaximumBinaryTree(nums.slice(0, maxIndex)),

constructMaximumBinaryTree(nums.slice(maxIndex + 1))

const [maxValue, maxIndex] = maxValueIndex;

// - The maximum value becomes the root.

// Return the constructed tree node.

// The reduce method processes each number, keeping track of the largest number and its index.

let maxValueIndex = nums.reduce<[number, number]>((currentMax, value, index) => {

// - 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.

return (currentMax[0] < value) ? [value, index] : currentMax;</pre>

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.

and right subtrees.

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 0(n^2) in the worst case, when the input list is sorted in ascending or descending order, causing the divide step to only

nums list, finding the maximum value in the current portion of the list, splitting the list into two parts, and recursively building the left

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

- **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 0(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 0(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).