2792. Count Nodes That Are Great Enough

Medium

In this problem, we are given a binary tree where each node has an integer value. Alongside, we are given an integer k. The task is to

Problem Description

 The subtree rooted with this node has at least k nodes. The value of this node is greater than the value of at least k nodes in its subtree. We must explore the tree and count how many

find and count the nodes in the binary tree that are considered "great enough". A node is "great enough" if it meets the following

Leetcode Link

- nodes satisfy these conditions.
 - Intuition The intuition behind the solution is to perform a depth-first search (DFS) starting from the root node, to explore the entire tree. While

exploring, we need to keep track of the nodes and their values in a way that enables us to check if a node is "great enough"

To check if a node is "great enough", we accumulate the greatest values we encounter in its subtree, including the node itself. We

according to the given conditions. use a min-heap data structure, which allows us to efficiently keep track of the largest elements (since in Python's heapq the smallest

criteria:

Here's the key to our approach:

Finally, push the negative of the current node's value onto our heap (to include the current node in the subtree for future parent

elements come first, we push negatives of the values so that when we pop, we pop the smallest of the largest values).

· Perform DFS from the root.

 Once merged, we compare the negative of the top of our min-heap (which is the kth largest value) with current node's value to determine if our node is "great enough". If it is, increment our counter ans.

Ensure our heap does not exceed k elements.

nodes) before returning the heap to its parent call.

The main advantages of this approach are: Space efficiency, as we carry a single heap up the tree while we are gathering subsets of values.

For each node, merge the heaps of its left and right children to congregate their greatest values.

• Time efficiency, since heaps provide O(log k) pushing and popping, making the overall complexity better compared to sorting the values or other brute force methods that might seem tempting at first glance.

node is greater than at least k nodes in its subtree, and we increment our answer ans.

Solution Approach The provided solution uses a Depth-First Search (DFS) algorithm in combination with a min-heap to solve the problem. Here is a

 A DFS dfs function is defined, which will be called recursively to explore the tree nodes. At each node, this function will return a min-heap containing the largest values (negated) from its subtree, with size at most k.

DFS call.

1 def dfs(node):

10

11

18

if node is None:

Merge heaps

ans += 1

Imagine a binary tree like this:

greater than at least k nodes in their subtree.

2. Perform DFS on the left child (3).

1. Begin DFS at the root node (5). Since it's not None, we proceed.

Left child (3) is not None, we perform DFS on its children.

return new empty heap

left_heap = dfs(node.left)

for value in right_heap:

right_heap = dfs(node.right)

push(left_heap, value)

push(left_heap, -node's value)

Check if the current node is "great enough"

if heap size is k and (-top of heap < node's value):

step-by-step explanation of the implementation:

- indicating that a leaf has been reached. When at a non-leaf node, DFS is called on both the left and right children, and their returned heaps are merged. It's important

Within this dfs function, when the base case is encountered (i.e., when a None node is encountered), an empty heap is returned,

- to note that we're using the push helper function to maintain the heap's size at no more than k. • The push helper function works as follows: It pushes an element onto the heap and if the heap size is greater than k after the
- push, it pops the smallest element (which is represented as the largest negative value). This ensures that only the k largest (negated) elements remain.

Back in the dfs function, after merging the heaps from the left and right subtrees, we check if the current node's value is "great"

k largest negated values) is less than the negated value of the current node. If these conditions are met, it means the current

Finally, the current node's value (negated) is pushed onto the heap, and the heap is returned to be merged in the parent node's

enough". This check is done by ensuring that the heap size is exactly k and the top element of the heap (i.e., the smallest of the

- After the DFS traversal completes, the counter ans holds the number of "great enough" nodes, and this value is returned as the solution to the problem. Here is the pseudocode for the DFS and push function:
- 13 return left_heap def push(heap, value): add value to heap 16 if size of heap > k: 17 remove the smallest element from heap
- creating a new heap at each node, which saves memory and potentially decreases runtime due to fewer allocations. Example Walkthrough

In this tree, we want to count the number of "great enough" nodes, i.e., nodes with at least k nodes in their subtree and a value

Now, left_heap has 2 elements (size is k) and top of heap is -4, which is smaller than -3. So, node 3 is "great enough".

○ After merging, the combined heap will have [-10, -8, -4, -3]. We need to maintain only the largest k values, so we pop off

Notice that the dfs function's merge step iterates over the right heap and pushes its elements into the left heap. This merging avoids

 Increment ans to 1 and push -3 into the left_heap before returning it. 3. Perform DFS on the right child (8).

 Right child (8) is not None, perform DFS on its left and right children. It has no left child, so returns an empty heap.

Left child (1) has no children (both None), so it returns an empty heap.

Right child (4) has no children (both None), so it returns an empty heap.

Right child (10) has no children (both None), so it returns an empty heap.

Merge returned heaps from 1 and 4 and push negated values: left_heap contains [-4, -3].

Let's use a small binary tree example and k = 2 to illustrate the solution approach.

 Merge returned heaps and push negated values: right_heap contains [-10]. Heap size is less than k, so we cannot say node 8 is "great enough" yet.

def countGreatEnoughNodes(self, root: Optional[TreeNode], k: int) -> int:

 Push –8 into the right_heap before returning it. 4. Back to the root now, left_heap contains [-4, -3] and right_heap contains [-10, -8]. Merge them.

Python Solution

class Solution:

10

11

12

18

19

22

23

24

25

26

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

10

11

12

13

14

16

17

18

19

20

21

22

23

24

16

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

61

62

63

64

65

66

67

68

70

8

9

10

11

13

16

17

18

19

20

27

28

29

30

31

32

33

34

36

41

42

43

44

45

46

47

48

50

55

49 };

51 // Example usage:

35 };

12 }

69 };

self.left = left

def dfs(node):

self.right = right

from heapq import heappush, heappop

heappop(heap)

if node is None:

return []

left_heap = dfs(node.left)

for val in right_heap:

nonlocal count

Return the updated heap

* @param root Root of the binary tree.

this.threshold = k;

this.greatEnoughCount = 0;

depthFirstSearch(root);

return greatEnoughCount;

* @param k The k-th largest value to be considered.

public int countGreatEnoughNodes(TreeNode root, int k) {

count += 1

return left_heap

Start DFS from the root

Return the final count

count = 0

dfs(root)

return count

right_heap = dfs(node.right)

push_to_heap(left_heap, val)

push_to_heap(left_heap, -node.val)

Initialize count of great enough nodes as 0

Helper method to maintain a heap of size 'k'

DFS traversal to count the great enough nodes

If the node is None, return an empty heap

Combine heaps from children and maintain size 'k'

if len(left_heap) == k and -left_heap[0] < node.val:</pre>

If condition satisfies, increment the result

Recursively traverse left and right subtrees

-3 to get left_heap as [-10, -8, -4]. The top of heap is now -4, which is smaller than -5. Thus, the root node (5) is "great enough". Increment ans to 2. Push -5 into left_heap and return left_heap.

5. After DFS traversal, the ans is 2, representing the number of "great enough" nodes (nodes 3 and 5).

1 # Definition for a binary tree node. 2 class TreeNode: def __init__(self, val=0, left=None, right=None): self.val = val

Thus, the output for our binary tree example with k = 2 is 2, meaning there are 2 nodes that are "great enough" by those criteria.

13 def push_to_heap(heap, val): # Add 'val' to 'heap' and ensure the heap size does not exceed 'k' 14 heappush(heap, val) 16 # If the heap size is > k, pop the smallest element if len(heap) > k: 17

Check if current node's value is greater than k-th value (if heap size is k)

Add current node's value to the heap (negated to convert to max-heap)

private int greatEnoughCount; // Counter to keep track of nodes meeting the condition

private int threshold; // The value k used as a threshold to compare node values

* is greater than the k-th largest value of all nodes from the root to that node.

* Counts the number of nodes in a binary tree where the node's value

* Performs a depth-first search on the binary tree, examining each node

* for finding the k-th largest value from the root to the current node.

// This function returns the number of nodes in a binary tree 'root'

// If the node is null, return an empty priority queue.

// Recursively apply DFS to the left and right subtrees.

// Combine the max values from left and right subtrees.

// Maintain the size of the priority queue as 'k'.

if (leftMaxes.size() == k && leftMaxes.top() < node->val) {

// all 'k' values because all others are larger).

// Return the final count of nodes that meet the condition.

// Check if the current node's value is greater than the smallest

// Add the current node's value to the collection of max values.

constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {

// If the current node's value is greater than all 'k' values seen so far, increment the count.

// This function performs Depth-First Search on the binary tree to find 'k' largest values.

const dfs = (node: TreeNode | null, k: number, answer: { count: number }): number[] => {

// Ensure the size of the priority queue does not exceed 'k'.

// value in the max 'k' values seen so far (means it's greater than

// Define a lambda function that uses Depth-First Search (DFS) to iterate

// through the tree and keep track of the largest 'k' values found so far.

std::function<std::priority_queue<int>(TreeNode*)> dfs = [&](TreeNode* node) -> std::priority_queue<int> {

// whose values are greater than all of the last 'k' visited nodes

int countGreatEnoughNodes(TreeNode* root, int k) {

auto leftMaxes = dfs(node->left);

while (!rightMaxes.empty()) {

rightMaxes.pop();

answer++;

return leftMaxes;

};

Typescript Solution

val: number;

class TreeNode {

dfs(root);

return answer;

1 // Definition for a binary tree node.

left: TreeNode | null;

right: TreeNode | null;

this.val = val;

if (node === null) {

answer.count++;

maxes.push(node.val);

39 // in a depth-first search order.

dfs(root, k, answer);

return answer.count;

let answer = { count: 0 };

return [];

this.left = left;

this.right = right;

// Recursively traverse left and right children.

return maxes.sort((a, b) => b - a).slice(0, k);

// Start the DFS from the root of the tree.

53 const result = countGreatEnoughNodes(rootNode, 2);

37 // This function returns the number of nodes in a binary tree 'root'

// Return the final count of nodes that meet the condition.

52 const rootNode = new TreeNode(2, new TreeNode(1), new TreeNode(3));

38 // whose values are greater than all of the last 'k' visited nodes

if (maxes.length === k && maxes[maxes.length - 1] < node.val) {</pre>

// Add the current node's value to the list of maxes and return.

const countGreatEnoughNodes = (root: TreeNode | null, k: number): number => {

// Initialize count as a reference object to keep track during recursive calls.

console.log(result); // Output should be based on the tree structure and 'k' value.

leftMaxes.push(node->val);

if (leftMaxes.size() > k) {

// Start the DFS from the root of the tree.

leftMaxes.pop();

auto rightMaxes = dfs(node->right);

if (leftMaxes.size() > k) {

leftMaxes.pop();

return std::priority_queue<int>();

leftMaxes.push(rightMaxes.top());

// in a depth-first search order.

int answer = 0;

if (!node) {

* and updating the greatEnoughCount when conditions are met using priority queue

* @return The count of nodes meeting the mentioned criterion.

Java Solution

/**

*/

/**

class Solution {

```
25
        * @param node The current node being examined.
        * @return A priority queue containing the largest values encountered up to the current node,
26
27
                  with size maintained at most k.
28
29
       private PriorityQueue<Integer> depthFirstSearch(TreeNode node) {
           // If we reach a null node, return a priority queue that orders elements in reverse order
30
           if (node == null) {
31
32
               return new PriorityQueue<>(Comparator.reverseOrder());
33
34
35
           // Recursively traverse left and right subtrees
36
           PriorityQueue<Integer> leftQueue = depthFirstSearch(node.left);
37
           PriorityQueue<Integer> rightQueue = depthFirstSearch(node.right);
38
           // Merge the right queue into the left queue
           for (int value : rightQueue) {
40
               leftQueue.offer(value);
41
               // If the size of the queue exceeds k, remove the smallest element
               if (leftQueue.size() > threshold) {
43
                   leftQueue.poll();
44
45
46
47
           // If the k-th largest value is defined and is less than the current node's value
           if (leftQueue.size() == threshold && leftQueue.peek() < node.val) {</pre>
50
               greatEnoughCount++; // Increment our count
51
52
53
           // Add the current node's value to the queue
54
           leftQueue.offer(node.val);
55
           // Ensure the size of the queue doesn't exceed k
           if (leftQueue.size() > threshold) {
56
               leftQueue.poll();
57
58
59
60
           return leftQueue; // Return the priority queue holding the largest k values up to and including this node
61
62 }
63
C++ Solution
    #include <functional>
    #include <queue>
     // Definition for a binary tree node.
    struct TreeNode {
         int val;
         TreeNode *left;
         TreeNode *right;
  8
         TreeNode(): val(0), left(nullptr), right(nullptr) {}
  9
         TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 10
 11
         TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 12 };
 13
 14 class Solution {
 15 public:
```

21 const leftMaxes = dfs(node.left, k, answer); 22 const rightMaxes = dfs(node.right, k, answer); 23 24 // Flatten left and right max values and sort them in descending order. const maxes = leftMaxes.concat(rightMaxes).sort((a, b) => b - a).slice(0, k); 25 26

Time and Space Complexity The time complexity of the given countGreatEnoughNodes function is O(n * log(n)), where n is the number of nodes in the binary tree. The function performs a depth-first search (DFS) on the tree, visiting each node once (O(n)). At each node, the push function potentially performs a heap push and pop operation in the priority queue. The length of the priority queue is capped at k, so each heap operation is O(log(k)). Since k can, at maximum, be as large as n (the total number of nodes), in the worst-case scenario, this becomes O(log(n)). When multiplied by the total number of nodes visited, we obtain O(n * log(n)). The space complexity of the function is O(n). This is because the recursive DFS function call stack could potentially store every node

in a path from the root to a leaf in the case of a skewed binary tree. In addition, the priority queue 1 also stores up to k elements, but since k is at most n, it does not exceed O(n) space. Thus, the overall space complexity is determined by the maximum between the height of the tree and the number length of the priority queue, which is O(n).