

ASSIGNMENT-6

1.Height of Binary Tree After Subtree Removal Queries

You are given the root of a binary tree with n nodes. Each node is assigned a unique value

from 1 to n . You are also given an array queries of size m . You have to perform m

independent queries on the tree where in the i th query you do the following:

- Remove the subtree rooted at the node with the value queries[i] from the tree. It is

guaranteed that queries[i] will not be equal to the value of the root. Return an array answer of size m where answer[i] is the height of the tree after performing

the i th query. Note:

- The queries are independent, so the tree returns to its initial state after each query.
- The height of a tree is the number of edges in the longest simple path from the root to

some node in the tree. Example 1:

Input: root = [1,3,4,2,null,6,5,null,null,null,null,7], queries = [4]

Output: [2]

Explanation: The diagram above shows the tree after removing the subtree rooted at node

with value 4. The height of the tree is 2 (The path 1 -> 3 -> 2). Example 2:

Input: root = [5,8,9,2,1,3,7,4,6], queries = [3,2,4,8]

Output: [3,2,3,2]

Explanation: We have the following queries: - Removing the subtree rooted at node with value 3. The height of the tree becomes 3 (The

path 5 -> 8 -> 2 -> 4). - Removing the subtree rooted at node with value 2. The height of the tree becomes 2 (The

path 5 -> 8 -> 1). - Removing the subtree rooted at node with value 4. The height of the tree becomes 3 (The

path 5 -> 8 -> 2 -> 6). - Removing the subtree rooted at node with value 8. The height of the tree becomes 2 (The

path 5 -> 9 -> 3). Constraints:

- The number of nodes in the tree is n. ● $2 \leq n \leq 105$
- $1 \leq \text{Node.val} \leq n$
- All the values in the tree are unique. ● $m == \text{queries.length}$
- $1 \leq m \leq \min(n, 104)$
- $1 \leq \text{queries}[i] \leq n$
- $\text{queries}[i] \neq \text{root.val}$

```
main.py
1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
6     def tree_height(root):
7         if not root:
8             return 0
9         return 1 + max(tree_height(root.left), tree_height(root.right))
10    def find_node(root, val, parent=None):
11        if not root:
12            return None, None
13        if root.val == val:
14            return root, parent
15        left, p = find_node(root.left, val, root)
16        if left:
17            return left, p
18        return find_node(root.right, val, root)
19    def remove_node(root, node, parent):
20        if parent:
21            if parent.left == node:
22                parent.left = None
23            else:
24                parent.right = None
25        else:
26            root = None
27        return root
28    def height_after_removal(root, queries):
29        answer = []
30        for val in queries:
31            node, parent = find_node(root, val)
32            if node:
33                root = remove_node(root, node, parent)
34                answer.append(tree_height(root))
35            if parent:
36                if parent.left is None:
37                    parent.left = node
38                else:
39                    parent.right = node
40            else:
41                root = node
42        return answer
43    root = TreeNode(1)
44    root.left = TreeNode(3)
45    root.right = TreeNode(4)
46    root.left.left = TreeNode(2)
47    root.right.left = TreeNode(6)
48    root.right.right = TreeNode(5)
49    root.right.right.right = TreeNode(7)
50    queries = [4]
51    print(height_after_removal(root, queries))
```

Output

```
[3]
=== Code Execution Successful ===
```

2. Sort Array by Moving Items to Empty Space

You are given an integer array `nums` of size `n` containing each element from 0 to `n - 1`

(inclusive). Each of the elements from 1 to `n - 1` represents an item, and the element 0

represents an empty space. In one operation, you can move any item to the empty space. `nums` is considered to be sorted

if the numbers of all the items are in ascending order and the empty space is either at the

beginning or at the end of the array. For example, if $n = 4$, `nums` is sorted if:

- `nums = [0,1,2,3]` or

- `nums = [1,2,3,0]`

...and considered to be unsorted otherwise. Return the minimum number of operations needed to sort `nums`.

Example 1:

Input:

`nums = [4,2,0,3,1]`

Output: 3

Explanation: - Move item 2 to the empty space. Now, `nums = [4,0,2,3,1]`. - Move item 1 to the empty space. Now, `nums = [4,1,2,3,0]`. - Move item 4 to the empty space. Now, `nums = [0,1,2,3,4]`. It can be proven that 3 is the minimum number of operations needed. Example 2:

Input: `nums = [1,2,3,4,0]`

Output: 0

Explanation: `nums` is already sorted so return 0.

Example 3:

Input: `nums = [1,0,2,4,3]`

Output: 2

Explanation: - Move item 2 to the empty space. Now, `nums = [1,2,0,4,3]`. - Move item 3 to the empty space. Now, `nums = [1,2,3,4,0]`. It can be proven that 2 is the minimum number of operations needed.

Constraints:

- $n == \text{nums.length}$

- $2 \leq n \leq 105$

- $0 \leq \text{nums}[i] < n$

- All the values of `nums` are unique.

```

1 def min_operations_to_sort(nums):
2     n = len(nums)
3     pos = {num: i for i, num in enumerate(nums)}
4     target = list(range(n))
5
6     operations = 0
7
8     for i in range(n):
9         if nums[i] != target[i]:
10             swap_with = pos[target[i]]
11             pos[nums[i]], pos[nums[swap_with]] = pos[target[i]], pos[nums[i]]
12             nums[i], nums[swap_with] = nums[swap_with], nums[i]
13             operations += 1
14
15     return operations
16
17 nums = [4, 2, 0, 3, 1]
18 print(min_operations_to_sort(nums))

```

3

=== Code Execution Successful ===

3. Apply Operations to an Array

You are given a 0-indexed array `nums` of size `n` consisting of non-negative integers. You need to apply `n - 1` operations to this array where, in the `i`th operation (0-indexed), you will apply the following on the `i`th element of `nums`:

- If `nums[i] == nums[i + 1]`, then multiply `nums[i]` by 2 and set `nums[i + 1]` to 0. Otherwise, you skip this operation. After performing all the operations, shift all the 0's to the end of the array. ● For example, the array `[1,0,2,0,0,1]` after shifting all its 0's to the end, is `[1,2,1,0,0,0]`. Return the resulting array. Note that the operations are applied sequentially, not all at once. Example 1:

Input: `nums = [1,2,2,1,1,0]`

Output: `[1,4,2,0,0,0]`

Explanation: We do the following operations: - `i = 0`: `nums[0]` and `nums[1]` are not equal, so we skip this operation. - `i = 1`: `nums[1]` and `nums[2]` are equal, we multiply `nums[1]` by 2 and change `nums[2]` to 0. The array becomes `[1,4,0,1,1,0]`. - `i = 2`: `nums[2]` and `nums[3]` are not equal, so we skip this operation. - `i = 3`: `nums[3]` and `nums[4]` are equal, we multiply `nums[3]` by 2 and change `nums[4]` to 0. The array becomes `[1,4,0,2,0,0]`. - `i = 4`: `nums[4]` and `nums[5]` are equal, we multiply `nums[4]` by 2 and change `nums[5]` to 0. The array becomes `[1,4,0,2,0,0]`. After that, we shift the 0's to the end, which gives the array `[1,4,2,0,0,0]`. Example 2:

Input: `nums = [0,1]`

Output: `[1,0]`

Explanation: No operation can be applied, we just shift the 0 to the end. Constraints:

- $2 \leq \text{nums.length} \leq 2000$

- $0 \leq \text{nums}[i] \leq 1000$

main.py	Save	Run	Output
<pre>1 def apply_operations(nums): 2 n = len(nums) 3 for i in range(n - 1): 4 if nums[i] == nums[i + 1]: 5 nums[i] *= 2 6 nums[i + 1] = 0 7 8 result = [num for num in nums if num != 0] + [0] * nums.count(0) 9 return result 10 nums = [1, 2, 2, 1, 1, 0] 11 print(apply_operations(nums)) 12</pre>			<pre>[1, 4, 2, 0, 0, 0] === Code Execution Successful ===</pre>

4. Maximum Sum of Distinct Subarrays With Length K

You are given an integer array `nums` and an integer `k`. Find the maximum subarray sum of all the subarrays of `nums` that meet the following conditions:

- The length of the subarray is `k`, and
- All the elements of the subarray are distinct. Return the maximum subarray sum of all the subarrays that meet the conditions. If no

subarray meets the conditions, return 0. A subarray is a contiguous non-empty sequence of elements within an array. Example 1:

Input: `nums = [1,5,4,2,9,9,9]`, `k = 3`

Output: 15

Explanation: The subarrays of `nums` with length 3 are: - `[1,5,4]` which meets the requirements and has a sum of 10. - `[5,4,2]` which meets the requirements and has a sum of 11. - `[4,2,9]` which meets the requirements and has a sum of 15. - `[2,9,9]` which does not meet the requirements because the element 9 is repeated. - `[9,9,9]` which does not meet the requirements because the element 9 is repeated. We return 15 because it is the maximum subarray sum of all the subarrays that meet the conditions

Example 2:

Input: `nums = [4,4,4]`, `k = 3`

Output: 0

Explanation: The subarrays of nums with length 3 are: - [4,4,4] which does not meet the requirements because the element 4 is repeated. We return 0 because no subarrays meet the conditions. Constraints:

● $1 \leq k \leq \text{nums.length} \leq 105$

● $1 \leq \text{nums}[i] \leq 105$

```
def max_sum_distinct_subarray(nums, k):
    from collections import defaultdict
    if k > len(nums):
        return 0
    max_sum = 0
    current_sum = 0
    count = defaultdict(int)
    for i in range(len(nums)):
        current_sum += nums[i]
        count[nums[i]] += 1
        if i >= k:
            current_sum -= nums[i - k]
            count[nums[i - k]] -= 1
            if count[nums[i - k]] == 0:
                del count[nums[i - k]]

        if i >= k - 1 and len(count) == k:
            max_sum = max(max_sum, current_sum)
    return max_sum
nums = [1, 5, 4, 2, 9, 9, 9]
k = 3
print(max_sum_distinct_subarray(nums, k))
```

15

=== Code Execution Successful ===

5. Total Cost to Hire K Workers

You are given a 0-indexed integer array costs where costs[i] is the cost of hiring the ith worker. You are also given two integers k and candidates. We want to hire exactly k workers according to the following rules:

● You will run k sessions and hire exactly one worker in each session. ● In each hiring session, choose the worker with the lowest cost from either the first

candidates workers or the last candidates workers. Break the tie by the smallest index. ○ For example, if costs = [3,2,7,7,1,2] and candidates = 2, then in the first hiring

session, we will choose the 4th worker because they have the lowest cost

[3,2,7,7,1,2]. ○ In the second hiring session, we will choose 1st worker because they have the

same lowest cost as 4th worker but they have the smallest index [3,2,7,7,2]. Please note that the indexing may be changed in the process. ● If there are fewer than candidates workers remaining, choose the worker with the

lowest cost among them. Break the tie by the smallest index. ● A worker can only be chosen once. Return the total cost to hire exactly k workers.

Example 1:

Input: costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4

Output: 11

Explanation: We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [17,12,10,2,7,2,11,20,8]. The lowest

cost is 2, and we break the tie by the smallest index, which is 3. The total cost = $0 + 2 = 2$. - In the second hiring round we choose the worker from [17,12,10,7,2,11,20,8]. The lowest

cost is 2 (index 4). The total cost = $2 + 2 = 4$. - In the third hiring round we choose the worker from [17,12,10,7,11,20,8]. The lowest cost is

7 (index 3). The total cost = $4 + 7 = 11$. Notice that the worker with index 3 was common in

the first and last four workers. The total hiring cost is 11. Example 2:

Input: costs = [1,2,4,1], k = 3, candidates = 3

Output: 4

Explanation: We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [1,2,4,1]. The lowest cost is 1, and we

break the tie by the smallest index, which is 0. The total cost = $0 + 1 = 1$. Notice that workers

with index 1 and 2 are common in the first and last 3 workers. - In the second hiring round we choose the worker from [2,4,1]. The lowest cost is 1 (index 2). The total cost = $1 + 1 = 2$. - In the third hiring round there are less than three candidates. We choose the worker from the

remaining workers [2,4]. The lowest cost is 2 (index 0). The total cost = $2 + 2 = 4$. The total hiring cost is 4. Constraints:

● $1 \leq \text{costs.length} \leq 105$

● $1 \leq \text{costs}[i] \leq 105$

● $1 \leq k, \text{candidates} \leq \text{costs.length}$

```

import heapq
def totalCost(costs, k, candidates):
    left_heap = []
    right_heap = []
    for i, cost in enumerate(costs[:candidates]):
        heapq.heappush(left_heap, (cost, i))
    for i, cost in enumerate(costs[-candidates:]):
        heapq.heappush(right_heap, (cost, -(len(costs) - i)))
    total_cost = 0
    for _ in range(k):
        if left_heap and right_heap:
            left_cost, left_index = left_heap[0]
            right_cost, right_index = right_heap[0]
            if left_cost < right_cost or (left_cost == right_cost and left_index <
                -right_index):
                total_cost += left_cost
                heapq.heappop(left_heap)
                if len(left_heap) < candidates and left_index + 1 < len(costs[
                    :candidates]):
                    next_cost = costs[left_index + 1]
                    heapq.heappush(left_heap, (next_cost, left_index + 1))
            else:
                total_cost += right_cost
                heapq.heappop(right_heap)
                if len(right_heap) < candidates and -(right_index + 1) >= 0:
                    next_cost = costs[-(right_index + 3) - 1]
                    heapq.heappush(right_heap, (next_cost, -(right_index + 1)))
        elif left_heap:
            total_cost += heapq.heappop(left_heap)[0]
        else:
            total_cost += heapq.heappop(right_heap)[0]
    return total_cost
print(totalCost([1,2,4,1], 3, 3))

```

4

=== Code Execution Successful ===

6. Minimum Total Distance Traveled

There are some robots and factories on the X-axis. You are given an integer array robot

where robot[i] is the position of the ith robot. You are also given a 2D integer array factory

where factory[j] = [positionj, limitj] indicates that positionj is the position of the jth factory

and that the jth factory can repair at most limitj robots. The positions of each robot are unique. The positions of each factory are also unique. Note

that a robot can be in the same position as a factory initially. All the robots are initially broken; they keep moving in one direction. The direction could be

the negative or the positive direction of the X-axis. When a robot reaches a factory that did

not reach its limit, the factory repairs the robot, and it stops moving. At any moment, you can set the initial direction of moving for some robot. Your target is to

minimize the total distance traveled by all the robots. Return the minimum total distance traveled by all the robots. The test cases are generated

such that all the robots can be repaired. Note that

● All robots move at the same speed. ● If two robots move in the same direction, they will never collide. ● If two robots move in opposite directions and they meet at some point, they do not collide. They cross each other. ● If a robot passes by a factory that reached its limits, it crosses it as if it does not exist. ● If the robot moved from a position x to a position y , the distance it moved is $|y - x|$.

Example 1:

Input: robot = [0,4,6], factory = [[2,2],[6,2]]

Output: 4

Explanation: As shown in the figure: - The first robot at position 0 moves in the positive direction. It will be repaired at the first

factory. - The second robot at position 4 moves in the negative direction. It will be repaired at the first

factory. - The third robot at position 6 will be repaired at the second factory. It does not need to move. The limit of the first factory is 2, and it fixed 2 robots. The limit of the second factory is 2, and it fixed 1 robot. The total distance is $|2 - 0| + |2 - 4| + |6 - 6| = 4$. It can be shown that we cannot achieve a

better total distance than 4. Example 2:

Input: robot = [1,-1], factory = [[-2,1],[2,1]]

Output: 2

Explanation: As shown in the figure: - The first robot at position 1 moves in the positive direction. It will be repaired at the second

factory. - The second robot at position -1 moves in the negative direction. It will be repaired at the

first factory. The limit of the first factory is 1, and it fixed 1 robot. The limit of the second factory is 1, and it fixed 1 robot. The total distance is $|2 - 1| + |(-2) - (-1)| = 2$. It can be shown that we cannot achieve a better

total distance than 2. Constraints:

● $1 \leq \text{robot.length}, \text{factory.length} \leq 100$

● $\text{factory}[j].\text{length} == 2$

● $-109 \leq \text{robot}[i], \text{positionj} \leq 109$

● $0 \leq \text{limitj} \leq \text{robot.length}$

● The input will be generated such that it is always possible to repair every robot.

```
def min_total_distance_traveled(robot, factory):
    robot.sort()
    factory.sort()
    total_distance = 0

    i = 0
    for pos, limit in factory:
        while limit > 0 and i < len(robot):
            total_distance += abs(pos - robot[i])
            limit -= 1
            i += 1

    return total_distance

robot = [0, 4, 6]
factory = [[2, 2], [6, 2]]
print(min_total_distance_traveled(robot, factory))
```

4

=== Code Execution Successful ===

7. Minimum Subarrays in a Valid Split

You are given an integer array `nums`. Splitting of an integer array `nums` into subarrays is valid

if:

- the greatest common divisor of the first and last elements of each subarray is greater than 1, and
- each element of `nums` belongs to exactly one subarray. Return the minimum number of subarrays in a valid subarray splitting of `nums`. If a valid

subarray splitting is not possible, return -1. Note that:

- The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers. ● A subarray is a contiguous non-empty part of an array. Example 1:

Input: `nums = [2,6,3,4,3]`

Output: 2

Explanation: We can create a valid split in the following way: `[2,6] | [3,4,3]`. - The starting element of the 1st subarray is 2 and the ending is 6. Their greatest common

divisor is 2, which is greater than 1. - The starting element of the 2nd subarray is 3 and the ending is 3. Their greatest common

divisor is 3, which is greater than 1. It can be proved that 2 is the minimum number of subarrays that we can obtain in a valid split. Example 2:

Input: nums = [3,5]

Output: 2

Explanation: We can create a valid split in the following way: [3] | [5]. - The starting element of the 1st subarray is 3 and the ending is 3. Their greatest common

divisor is 3, which is greater than 1. - The starting element of the 2nd subarray is 5 and the ending is 5. Their greatest common

divisor is 5, which is greater than 1. It can be proved that 2 is the minimum number of subarrays that we can obtain in a valid split.

Example 3:

Input: nums = [1,2,1]

Output: -1

Explanation: It is impossible to create valid split. Constraints:

- $1 \leq \text{nums.length} \leq 1000$

- $1 \leq \text{nums}[i] \leq 105$

```
import math
def min_valid_subarrays(nums):
    n = len(nums)
    res = n
    for i in range(n):
        gcd = nums[i]
        for j in range(i, n):
            gcd = math.gcd(gcd, nums[j])
            if gcd > 1:
                res = min(res, 1 + min_valid_subarrays(nums[:i] + nums[j+1:]))
    return res if res != n else -1
print(min_valid_subarrays([2,6,3,4,3]))
```

2
=== Code Execution Successful ===

8. Number of Distinct Averages

You are given a 0-indexed integer array nums of even length. As long as nums is not empty, you must repetitively:

- Find the minimum number in nums and remove it.
- Find the maximum number in nums and remove it.
- Calculate the average of the two removed numbers. The average of two numbers a and b is $(a + b) / 2$.
- For example, the average of 2 and 3 is $(2 + 3) / 2 = 2.5$.

Return the number of distinct averages calculated using the above process. Note that when

there is a tie for a minimum or maximum number, any can be removed. Example 1:

Input: nums = [4,1,4,0,3,5]

Output: 2

Explanation:

1. Remove 0 and 5, and the average is $(0 + 5) / 2 = 2.5$. Now, nums = [4,1,4,3]. 2. Remove 1 and 4. The average is $(1 + 4) / 2 = 2.5$, and nums = [4,3]. 3. Remove 3 and 4, and the average is $(3 + 4) / 2 = 3.5$. Since there are 2 distinct numbers among 2.5, 2.5, and 3.5, we return 2. Example 2:

Input: nums = [1,100]

Output: 1

Explanation:

There is only one average to be calculated after removing 1 and 100, so we return 1. Constraints:

- $2 \leq \text{nums.length} \leq 100$
- nums.length is even. ● $0 \leq \text{nums}[i] \leq 100$

main.py	Output
<pre>1 def distinct_averages(nums): 2 nums.sort() 3 averages = set() 4 while nums: 5 min_val = nums.pop(0) 6 max_val = nums.pop(-1) 7 averages.add((min_val + max_val) / 2) 8 return len(averages) 9 nums = [4, 1, 4, 0, 3, 5] 10 print(distinct_averages(nums))</pre>	<pre>2 === Code Execution Successful ===</pre>

9. Count Ways To Build Good Strings

Given the integers zero, one, low, and high, we can construct a string by starting with an empty string, and then at each step perform either of the following:

- Append the character '0' zero times.
- Append the character '1' one times.

This can be performed any number of times. A good string is a string constructed by the

above process having a length between low and high (inclusive). Return the number of different good strings that can be constructed satisfying these

properties. Since the answer can be large, return it modulo $10^9 + 7$. Example 1:

Input: low = 3, high = 3, zero = 1, one = 1

Output: 8

Explanation:

One possible valid good string is "011". It can be constructed as follows: "" -> "0" -> "01" -> "011". All binary strings from "000" to "111" are good strings in this example. Example 2:

Input: low = 2, high = 3, zero = 1, one = 2

Output: 5

Explanation: The good strings are "00", "11", "000", "110", and "011". Constraints:

● $1 \leq \text{low} \leq \text{high} \leq 105$

● $1 \leq \text{zero}, \text{one} \leq \text{low}$

```
1 def count_good_strings(low, high, zero, one):
2     MOD = 10**9 + 7
3     dp = [0] * (high + 1)
4     dp[0] = 1
5
6     for i in range(1, high + 1):
7         if i >= zero:
8             dp[i] = (dp[i] + dp[i - zero]) % MOD
9         if i >= one:
10            dp[i] = (dp[i] + dp[i - one]) % MOD
11
12    return sum(dp[low:high + 1]) % MOD
13
14    low = 3
15    high = 3
16    zero = 1
17    one = 1
18    print(count_good_strings(low, high, zero, one))
```

8

=== Code Execution Successful ===

10. Most Profitable Path in a Tree

There is an undirected tree with n nodes labeled from 0 to $n - 1$, rooted at node 0. You are

given a 2D integer array `edges` of length $n - 1$ where `edges[i] = [ai, bi]` indicates that there is

an edge between nodes `ai` and `bi` in the tree. At every node `i`, there is a gate. You are also given an array of even integers `amount`, where

`amount[i]` represents:

● the price needed to open the gate at node `i`, if `amount[i]` is negative, or, ● the cash reward obtained on opening the gate at node `i`, otherwise. The game goes on as follows:

● Initially, Alice is at node 0 and Bob is at node `bob`. ● At every second, Alice and Bob each move to an adjacent node. Alice moves towards

some leaf node, while Bob moves towards node 0. ● For every node along their path, Alice and Bob either spend money to open the gate at

that node, or accept the reward. Note that:

○ If the gate is already open, no price will be required, nor will there be any cash

reward. ○ If Alice and Bob reach the node simultaneously, they share the price/reward

for opening the gate there. In other words, if the price to open the gate is c , then both Alice and Bob pay $c / 2$ each. Similarly, if the reward at the gate is c , both of them receive $c / 2$ each. ● If Alice reaches a leaf node, she stops moving. Similarly, if Bob reaches node 0, he

stops moving. Note that these events are independent of each other. Return the maximum net income Alice can have if she travels towards the optimal leaf node.

Example 1:

Input: edges = [[0,1],[1,2],[1,3],[3,4]], bob = 3, amount = [-2,4,2,-4,6]

Output: 6

Explanation:

The above diagram represents the given tree. The game goes as follows: - Alice is initially on node 0, Bob on node 3. They open the gates of their respective nodes. Alice's net income is now -2. - Both Alice and Bob move to node 1. Since they reach here simultaneously, they open the gate together and share the reward. Alice's net income becomes $-2 + (4 / 2) = 0$. - Alice moves on to node 3. Since Bob already opened its gate, Alice's income remains

unchanged. Bob moves on to node 0, and stops moving. - Alice moves on to node 4 and opens the gate there. Her net income becomes $0 + 6 = 6$. Now, neither Alice nor Bob can make any further moves, and the game ends. It is not possible for Alice to get a higher net income. Example 2:

Input: edges = [[0,1]], bob = 1, amount = [-7280,2350]

Output: -7280

Explanation:

Alice follows the path 0->1 whereas Bob follows the path 1->0. Thus, Alice opens the gate at node 0 only. Hence, her net income is -7280. Constraints:

● $2 \leq n \leq 105$

● edges.length == n - 1

- `edges[i].length == 2`
- `0 <= ai, bi < n`
- `ai != bi`
- `edges` represents a valid tree. ● `1 <= bob < n`
- `amount.length == n`
- `amount[i]` is an even integer in the range `[-104, 104]`

<pre> def maximize_greatness(nums): nums.sort() n = len(nums) dp = [0] * n for i in range(1, n): for j in range(i): if nums[i] > nums[j]: dp[i] = max(dp[i], dp[j] + 1) return max(dp) nums = [1, 3, 5, 2, 1, 3] print(maximize_greatness(nums)) </pre>	<pre> 3 === Code Execution Successful === </pre>
--	---