Problem 3593: Minimum Increments to Equalize Leaf Paths

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given an integer

n

and an undirected tree rooted at node 0 with

n

nodes numbered from 0 to

n - 1

. This is represented by a 2D array

edges

of length

n - 1

, where

edges[i] = [u

i

, v

i

]

indicates an edge from node

u

i

to

v

i

.

Each node

i

has an associated cost given by

cost[i]

, representing the cost to traverse that node.

The

score

of a path is defined as the sum of the costs of all nodes along the path.

Your goal is to make the scores of all

root-to-leaf

paths

equal

by

increasing

the cost of any number of nodes by

any non-negative

amount.

Return the

minimum

number of nodes whose cost must be increased to make all root-to-leaf path scores equal.

Example 1:

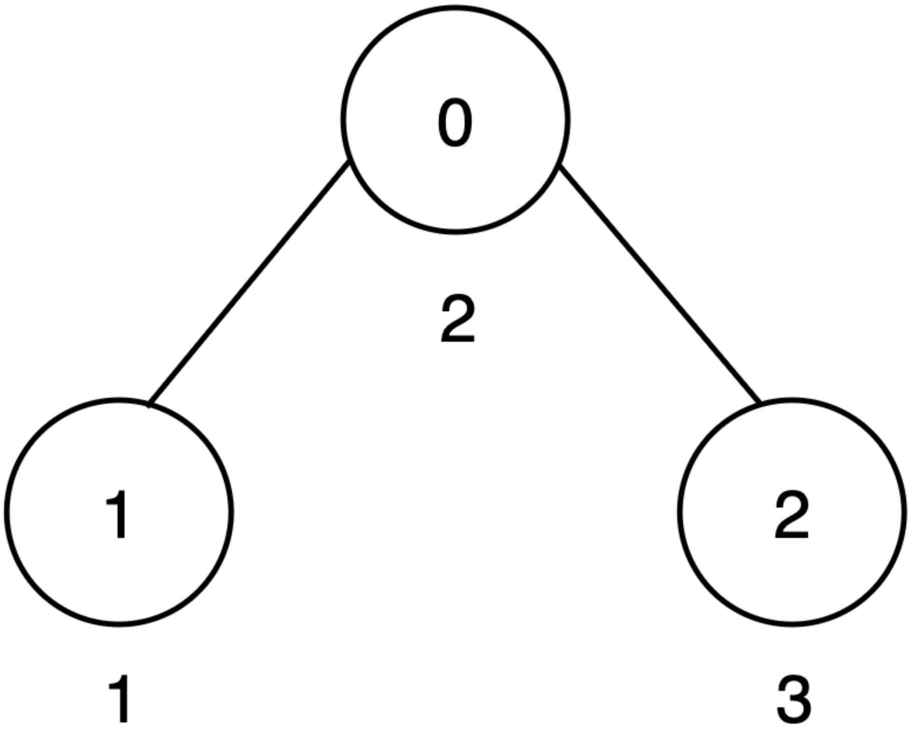
Input:

n = 3, edges = [[0,1],[0,2]], cost = [2,1,3]

Output:

1

Explanation:



There are two root-to-leaf paths:

Path

0 → 1

has a score of

2 + 1 = 3

.

Path

0 → 2

has a score of

2 + 3 = 5

.

To make all root-to-leaf path scores equal to 5, increase the cost of node 1 by 2.

Only one node is increased, so the output is 1.

Example 2:

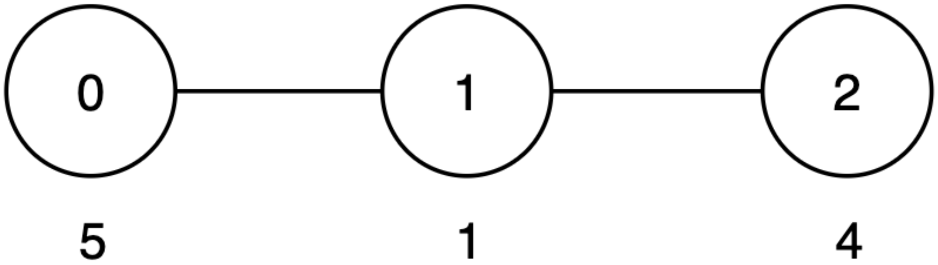
Input:

n = 3, edges = [[0,1],[1,2]], cost = [5,1,4]

Output:

0

Explanation:



There is only

one root-to-leaf path:

Path

0 → 1 → 2

has a score of

5 + 1 + 4 = 10

.

Since only one root-to-leaf path exists, all path costs are trivially equal, and the output is 0.

Example 3:

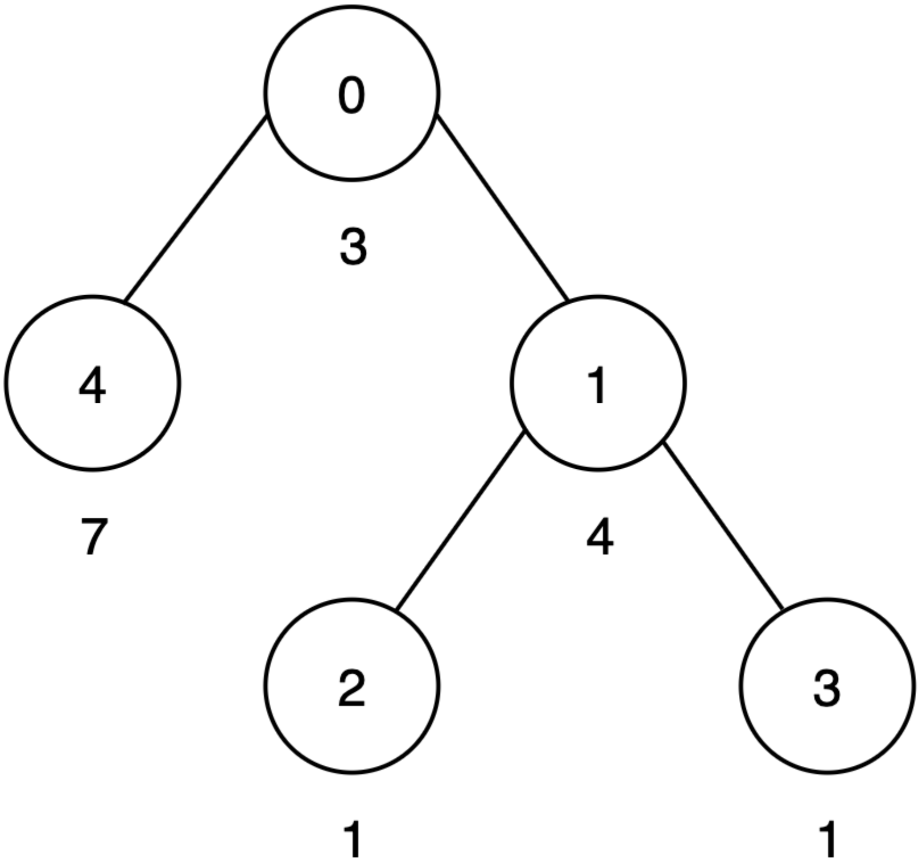
Input:

n = 5, edges = [[0,4],[0,1],[1,2],[1,3]], cost = [3,4,1,1,7]

Output:

1

Explanation:



There are three root-to-leaf paths:

Path

0 → 4

has a score of

3 + 7 = 10

.

Path

0 → 1 → 2

has a score of

3 + 4 + 1 = 8

.

Path

0 → 1 → 3

has a score of

3 + 4 + 1 = 8

.

To make all root-to-leaf path scores equal to 10, increase the cost of node 1 by 2. Thus, the output is 1.

Constraints:

2 <= n <= 10

5

edges.length == n - 1

edges[i] == [u

i

, v

i

]

0 <= u

i

, v

i

< n

cost.length == n

1 <= cost[i] <= 10

9

The input is generated such that

edges

represents a valid tree.

Code Snippets

C++:

```
class Solution {
public:
int minIncrease(int n, vector<vector<int>>& edges, vector<int>& cost) {
}
};
```

Java:

```
class Solution {
public int minIncrease(int n, int[][] edges, int[] cost) {

}
}
```

Python3:

```
class Solution:
def minIncrease(self, n: int, edges: List[List[int]], cost: List[int]) ->
int:
```

Python:

```
class Solution(object):
def minIncrease(self, n, edges, cost):
"""
:type n: int
:type edges: List[List[int]]
:type cost: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} edges
* @param {number[]} cost
* @return {number}
*/
var minIncrease = function(n, edges, cost) {

};
```

TypeScript:

```
function minIncrease(n: number, edges: number[][], cost: number[]): number {

};
```

C#:

```
public class Solution {
public int MinIncrease(int n, int[][] edges, int[] cost) {

}
}
```

C:

```
int minIncrease(int n, int** edges, int edgesSize, int* edgesColSize, int*
cost, int costSize) {

}
```

Go:

```
func minIncrease(n int, edges [][]int, cost []int) int {

}
```

Kotlin:

```
class Solution {
fun minIncrease(n: Int, edges: Array<IntArray>, cost: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func minIncrease(_ n: Int, _ edges: [[Int]], _ cost: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_increase(n: i32, edges: Vec<Vec<i32>>, cost: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @param {Integer[]} cost
# @return {Integer}
def min_increase(n, edges, cost)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @param Integer[] $cost
* @return Integer
*/
function minIncrease($n, $edges, $cost) {

}
}
```

Dart:

```
class Solution {
int minIncrease(int n, List<List<int>> edges, List<int> cost) {

}
}
```

Scala:

```
object Solution {
def minIncrease(n: Int, edges: Array[Array[Int]], cost: Array[Int]): Int = {
}
}
```

Elixir:

```
defmodule Solution do
@spec min_increase(n :: integer, edges :: [[integer]], cost :: [integer]) ::
integer
def min_increase(n, edges, cost) do

end
end
```

Erlang:

```
-spec min_increase(N :: integer(), Edges :: [[integer()]], Cost ::
[integer()]) -> integer().
min_increase(N, Edges, Cost) ->
.
```

Racket:

```
(define/contract (min-increase n edges cost)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

class Solution {
public:
int minIncrease(int n, vector<vector<int>>& edges, vector<int>& cost) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int minIncrease(int n, int[][] edges, int[] cost) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Increments to Equalize Leaf Paths
Difficulty: Medium
Tags: array, tree, dp, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def minIncrease(self, n: int, edges: List[List[int]], cost: List[int]) ->
```

```
int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minIncrease(self, n, edges, cost):
"""
:type n: int
:type edges: List[List[int]]
:type cost: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @param {number[][]} edges
* @param {number[]} cost
* @return {number}
*/
var minIncrease = function(n, edges, cost) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function minIncrease(n: number, edges: number[][], cost: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int MinIncrease(int n, int[][] edges, int[] cost) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Increments to Equalize Leaf Paths
* Difficulty: Medium
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
int minIncrease(int n, int** edges, int edgesSize, int* edgesColSize, int*
cost, int costSize) {

}
```

Go Solution:

```
// Problem: Minimum Increments to Equalize Leaf Paths
// Difficulty: Medium
// Tags: array, tree, dp, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func minIncrease(n int, edges [][]int, cost []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun minIncrease(n: Int, edges: Array<IntArray>, cost: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func minIncrease(_ n: Int, _ edges: [[Int]], _ cost: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Increments to Equalize Leaf Paths
// Difficulty: Medium
// Tags: array, tree, dp, search
//
```

```
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn min_increase(n: i32, edges: Vec<Vec<i32>>, cost: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @param {Integer[]} cost
# @return {Integer}
def min_increase(n, edges, cost)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @param Integer[] $cost
* @return Integer
*/
function minIncrease($n, $edges, $cost) {

}
}
```

Dart Solution:

```
class Solution {
int minIncrease(int n, List<List<int>> edges, List<int> cost) {

}

}
```

Scala Solution:

```
object Solution {
def minIncrease(n: Int, edges: Array[Array[Int]], cost: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_increase(n :: integer, edges :: [[integer]], cost :: [integer]) ::
integer
def min_increase(n, edges, cost) do

end
end
```

Erlang Solution:

```
-spec min_increase(N :: integer(), Edges :: [[integer()]], Cost ::
[integer()]) -> integer().
min_increase(N, Edges, Cost) ->
.
```

Racket Solution:

```
(define/contract (min-increase n edges cost)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```