Problem 2846: Minimum Edge Weight Equilibrium Queries in a Tree

Problem Information

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

Problem Description

There is an undirected tree with

n

nodes labeled from

0

to

n - 1

. You are given the integer

n

and a 2D integer array

edges

of length

n - 1

, where

edges[i] = [u

i

, v

i

, w

i

]

indicates that there is an edge between nodes

u

i

and

v

i

with weight

w

i

in the tree.

You are also given a 2D integer array

queries

of length

m

, where

queries[i] = [a

i

, b

i

]

. For each query, find the

minimum number of operations

required to make the weight of every edge on the path from

a

i

to

b

i

equal. In one operation, you can choose any edge of the tree and change its weight to any value.

Note

that:

Queries are independent of each other, meaning that the tree returns to its initial state on each new query.

The path from a_i to b_i is a sequence of distinct nodes starting with node a_i and ending with node b_i such that every two adjacent nodes in the sequence share an edge in the tree.

Return

an array

answer

of length

m

where

answer[i]

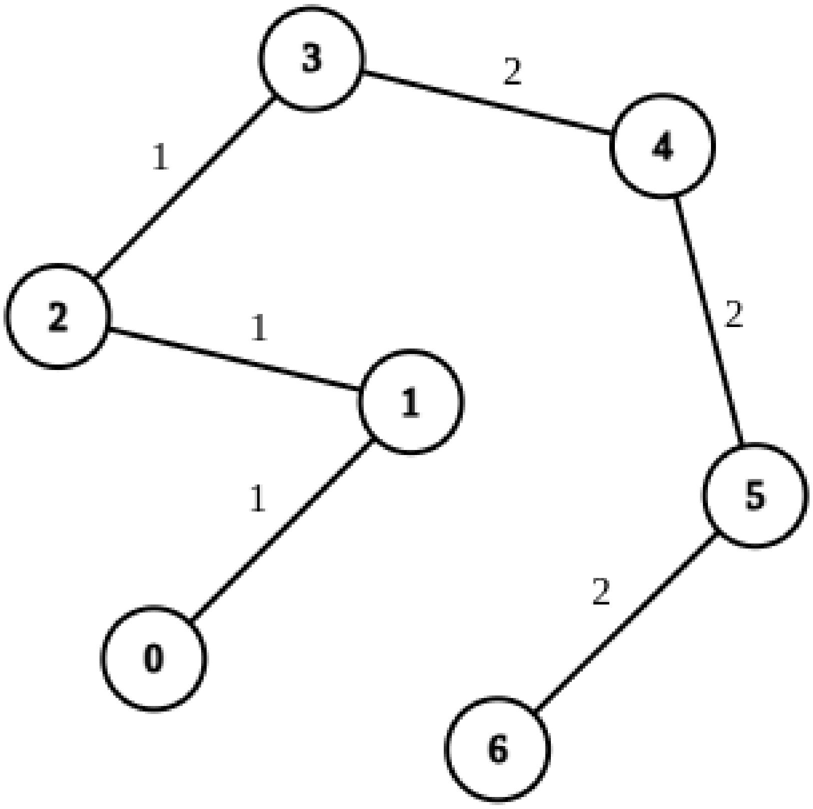
is the answer to the

i

th

query.

Example 1:



Input:

n = 7, edges = [[0,1,1],[1,2,1],[2,3,1],[3,4,2],[4,5,2],[5,6,2]], queries = [[0,3],[3,6],[2,6],[0,6]]

Output:

[0,0,1,3]

Explanation:

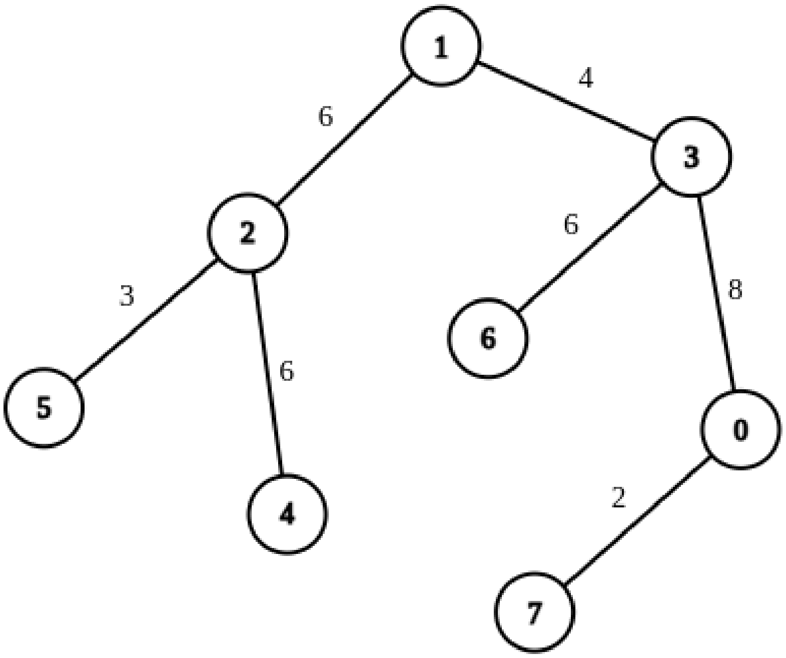
In the first query, all the edges in the path from 0 to 3 have a weight of 1. Hence, the answer is 0. In the second query, all the edges in the path from 3 to 6 have a weight of 2. Hence, the answer is 0. In the third query, we change the weight of edge [2,3] to 2. After this operation, all the edges in the path from 2 to 6 have a weight of 2. Hence, the answer is 1. In the fourth query, we change the weights of edges [0,1], [1,2] and [2,3] to 2. After these operations, all the edges in the path from 0 to 6 have a weight of 2. Hence, the answer is 3. For each queries[i], it can be shown that answer[i] is the minimum number of operations needed to equalize all the edge weights in the path from a i

to b

i

.

Example 2:



Input:

n = 8, edges = [[1,2,6],[1,3,4],[2,4,6],[2,5,3],[3,6,6],[3,0,8],[7,0,2]], queries = [[4,6],[0,4],[6,5],[7,4]]

Output:

[1,2,2,3]

Explanation:

In the first query, we change the weight of edge [1,3] to 6. After this operation, all the edges in the path from 4 to 6 have a weight of 6. Hence, the answer is 1. In the second query, we change the weight of edges [0,3] and [3,1] to 6. After these operations, all the edges in the path from 0 to 4 have a weight of 6. Hence, the answer is 2. In the third query, we change the weight of edges [1,3] and [5,2] to 6. After these operations, all the edges in the path from 6 to 5 have a weight of 6. Hence, the answer is 2. In the fourth query, we change the weights of edges [0,7], [0,3] and [1,3] to 6. After these operations, all the edges in the path from 7 to 4 have a weight of 6. Hence, the answer is 3. For each queries[i], it can be shown that answer[i] is the minimum number of operations needed to equalize all the edge weights in the path from a

i

to b

i

.

Constraints:

1 <= n <= 10

4

edges.length == n - 1

edges[i].length == 3

0 <= u

i

, v

i

< n

1 <= w

i

<= 26

The input is generated such that

edges

represents a valid tree.

1 <= queries.length == m <= 2 * 10

4

queries[i].length == 2

0 <= a

i

, b

i

< n

Code Snippets

C++:

```
class Solution {
public:
vector<int> minOperationsQueries(int n, vector<vector<int>>& edges,
vector<vector<int>>& queries) {

}
};
```

Java:

```
class Solution {
public int[] minOperationsQueries(int n, int[][] edges, int[][] queries) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function minOperationsQueries(n: number, edges: number[][], queries:
number[][]): number[] {

};
```

C#:

```
public class Solution {
public int[] MinOperationsQueries(int n, int[][] edges, int[][] queries) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minOperationsQueries(int n, int** edges, int edgesSize, int*
edgesColSize, int** queries, int queriesSize, int* queriesColSize, int*
returnSize) {

}
```

Go:

```
func minOperationsQueries(n int, edges [][]int, queries [][]int) []int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
List<int> minOperationsQueries(int n, List<List<int>> edges, List<List<int>>
queries) {

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

```
-spec min_operations_queries(N :: integer(), Edges :: [[integer()]], Queries
:: [[integer()]]) -> [integer()].
min_operations_queries(N, Edges, Queries) ->
.
```

Racket:

```
(define/contract (min-operations-queries n edges queries)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<int> minOperationsQueries(int n, vector<vector<int>>& edges,
vector<vector<int>>& queries) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public int[] minOperationsQueries(int n, int[][] edges, int[][] queries) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Edge Weight Equilibrium Queries in a Tree
Difficulty: Hard
Tags: array, tree, graph

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
```

```
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def minOperationsQueries(self, n: int, edges: List[List[int]], queries:
List[List[int]]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function minOperationsQueries(n: number, edges: number[][], queries:
number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public int[] MinOperationsQueries(int n, int[][] edges, int[][] queries) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Edge Weight Equilibrium Queries in a Tree
* Difficulty: Hard
* Tags: array, tree, graph
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minOperationsQueries(int n, int** edges, int edgesSize, int*
edgesColSize, int** queries, int queriesSize, int* queriesColSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Minimum Edge Weight Equilibrium Queries in a Tree
// Difficulty: Hard
// Tags: array, tree, graph
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func minOperationsQueries(n int, edges [][]int, queries [][]int) []int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Minimum Edge Weight Equilibrium Queries in a Tree
// Difficulty: Hard
// Tags: array, tree, graph
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

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

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
List<int> minOperationsQueries(int n, List<List<int>> edges, List<List<int>>
queries) {

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

```
-spec min_operations_queries(N :: integer(), Edges :: [[integer()]], Queries
:: [[integer()]]) -> [integer()].
min_operations_queries(N, Edges, Queries) ->
.
```

Racket Solution:

```
(define/contract (min-operations-queries n edges queries)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```