Problem 3203: Find Minimum Diameter After Merging Two Trees

Problem Information

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

Problem Description

There exist two

undirected

trees with

n

and

m

nodes, numbered from

0

to

n - 1

and from

0

to

m - 1

, respectively. You are given two 2D integer arrays

edges1

and

edges2

of lengths

n - 1

and

m - 1

, respectively, where

edges1[i] = [a

i

, b

i

]

indicates that there is an edge between nodes

a

i

and

b

i

in the first tree and

edges2[i] = [u

i

, v

i

]

indicates that there is an edge between nodes

u

i

and

v

i

in the second tree.

You must connect one node from the first tree with another node from the second tree with an edge.

Return the

minimum

possible

diameter

of the resulting tree.

The

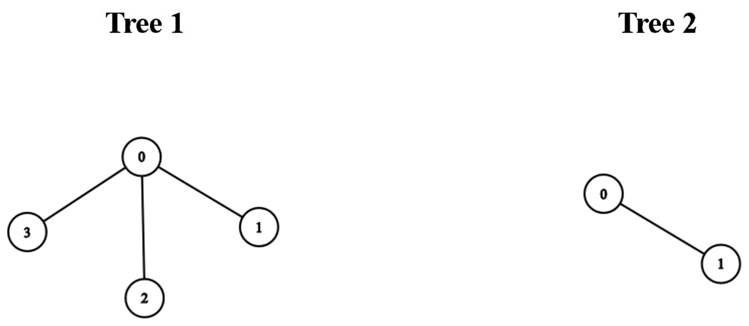
diameter

of a tree is the length of the

longest

path between any two nodes in the tree.

Example 1:



Input:

edges1 = [[0,1],[0,2],[0,3]], edges2 = [[0,1]]

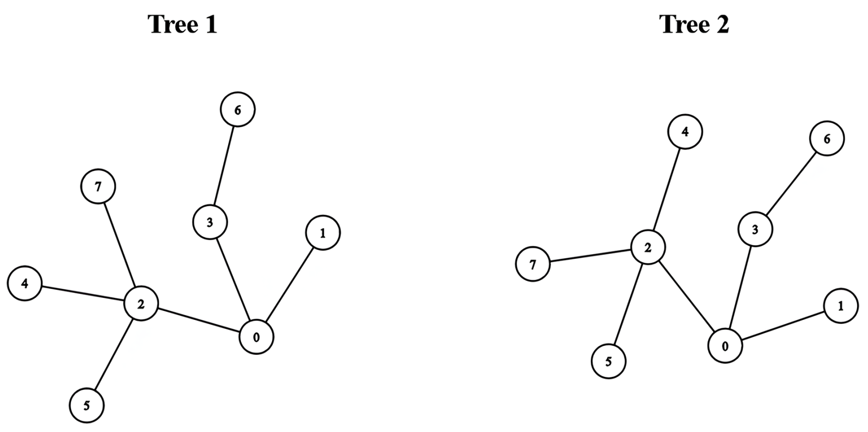
Output:

3

Explanation:

We can obtain a tree of diameter 3 by connecting node 0 from the first tree with any node from the second tree.

Example 2:



Input:

edges1 = [[0,1],[0,2],[0,3],[2,4],[2,5],[3,6],[2,7]], edges2 = [[0,1],[0,2],[0,3],[2,4],[2,5],[3,6],[2,7]]

Output:

5

Explanation:

We can obtain a tree of diameter 5 by connecting node 0 from the first tree with node 0 from the second tree.

Constraints:

1 <= n, m <= 10

5

edges1.length == n - 1

edges2.length == m - 1

edges1[i].length == edges2[i].length == 2

edges1[i] = [a

i

, b

i

]

0 <= a

i

, b

i

< n

edges2[i] = [u

i

, v

i

]

0 <= u

i

, v

i

< m

The input is generated such that

edges1

and

edges2

represent valid trees.

Code Snippets

C++:

```
class Solution {
public:
int minimumDiameterAfterMerge(vector<vector<int>>& edges1,
vector<vector<int>>& edges2) {

}
};
```

Java:

```
class Solution {
public int minimumDiameterAfterMerge(int[][] edges1, int[][] edges2) {

}
}
```

Python3:

```
class Solution:
def minimumDiameterAfterMerge(self, edges1: List[List[int]], edges2:
List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def minimumDiameterAfterMerge(self, edges1, edges2):
```

```
"""
:type edges1: List[List[int]]
:type edges2: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} edges1
* @param {number[][]} edges2
* @return {number}
*/
var minimumDiameterAfterMerge = function(edges1, edges2) {

};
```

TypeScript:

```
function minimumDiameterAfterMerge(edges1: number[][], edges2: number[][]):
number {

};
```

C#:

```
public class Solution {
public int MinimumDiameterAfterMerge(int[][] edges1, int[][] edges2) {

}
}
```

C:

```
int minimumDiameterAfterMerge(int** edges1, int edges1Size, int*
edges1ColSize, int** edges2, int edges2Size, int* edges2ColSize) {

}
```

Go:

```
func minimumDiameterAfterMerge(edges1 [][]int, edges2 [][]int) int {

}
```

Kotlin:

```
class Solution {
fun minimumDiameterAfterMerge(edges1: Array<IntArray>, edges2:
Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func minimumDiameterAfterMerge(_ edges1: [[Int]], _ edges2: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_diameter_after_merge(edges1: Vec<Vec<i32>>, edges2:
Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} edges1
# @param {Integer[][]} edges2
# @return {Integer}
def minimum_diameter_after_merge(edges1, edges2)

end
```

PHP:

```
class Solution {
```

```
/**
* @param Integer[][] $edges1
* @param Integer[][] $edges2
* @return Integer
*/
function minimumDiameterAfterMerge($edges1, $edges2) {

}
}
```

Dart:

```
class Solution {
int minimumDiameterAfterMerge(List<List<int>> edges1, List<List<int>> edges2)
{

}
}
```

Scala:

```
object Solution {
def minimumDiameterAfterMerge(edges1: Array[Array[Int]], edges2:
Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_diameter_after_merge(edges1 :: [[integer]], edges2 ::
[[integer]]) :: integer
def minimum_diameter_after_merge(edges1, edges2) do

end
end
```

Erlang:

```
-spec minimum_diameter_after_merge(Edges1 :: [[integer()]], Edges2 ::
[[integer()]]) -> integer().
```

```
minimum_diameter_after_merge(Edges1, Edges2) ->
.
```

Racket:

```
(define/contract (minimum-diameter-after-merge edges1 edges2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?))
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* 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 minimumDiameterAfterMerge(vector<vector<int>>& edges1,
vector<vector<int>>& edges2) {

}
};
```

Java Solution:

```
/**
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* 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 minimumDiameterAfterMerge(int[][] edges1, int[][] edges2) {

}
}
```

Python3 Solution:

```
"""
Problem: Find Minimum Diameter After Merging Two Trees
Difficulty: Hard
Tags: array, tree, graph, search

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 minimumDiameterAfterMerge(self, edges1: List[List[int]], edges2:
List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumDiameterAfterMerge(self, edges1, edges2):
"""
:type edges1: List[List[int]]
:type edges2: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* 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[][]} edges1
* @param {number[][]} edges2
* @return {number}
*/
var minimumDiameterAfterMerge = function(edges1, edges2) {

};
```

TypeScript Solution:

```
/**
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* 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 minimumDiameterAfterMerge(edges1: number[][], edges2: number[][]):
number {

};
```

C# Solution:

```
/*
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
```

```
*
* 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 MinimumDiameterAfterMerge(int[][] edges1, int[][] edges2) {

}
}
```

C Solution:

```
/*
* Problem: Find Minimum Diameter After Merging Two Trees
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* 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
*/

int minimumDiameterAfterMerge(int** edges1, int edges1Size, int*
edges1ColSize, int** edges2, int edges2Size, int* edges2ColSize) {

}
```

Go Solution:

```
// Problem: Find Minimum Diameter After Merging Two Trees
// Difficulty: Hard
// Tags: array, tree, graph, search
//
// 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 minimumDiameterAfterMerge(edges1 [][]int, edges2 [][]int) int {
}
```

Kotlin Solution:

```
class Solution {
fun minimumDiameterAfterMerge(edges1: Array<IntArray>, edges2:
Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func minimumDiameterAfterMerge(_ edges1: [[Int]], _ edges2: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Find Minimum Diameter After Merging Two Trees
// Difficulty: Hard
// Tags: array, tree, graph, search
//
// 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 minimum_diameter_after_merge(edges1: Vec<Vec<i32>>, edges2:
Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} edges1
# @param {Integer[][]} edges2
# @return {Integer}
```

```
def minimum_diameter_after_merge(edges1, edges2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $edges1
* @param Integer[][] $edges2
* @return Integer
*/
function minimumDiameterAfterMerge($edges1, $edges2) {

}
}
```

Dart Solution:

```
class Solution {
int minimumDiameterAfterMerge(List<List<int>> edges1, List<List<int>> edges2)
{

}
}
```

Scala Solution:

```
object Solution {
def minimumDiameterAfterMerge(edges1: Array[Array[Int]], edges2:
Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_diameter_after_merge(edges1 :: [[integer]], edges2 ::
[[integer]]) :: integer
```

```
def minimum_diameter_after_merge(edges1, edges2) do

end
end
```

Erlang Solution:

```
-spec minimum_diameter_after_merge(Edges1 :: [[integer()]], Edges2 ::
[[integer()]]) -> integer().
minimum_diameter_after_merge(Edges1, Edges2) ->
.
```

Racket Solution:

```
(define/contract (minimum-diameter-after-merge edges1 edges2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?))
exact-integer?)
)
```