

# Problem 1719: Number Of Ways To Reconstruct A Tree

## Problem Information

Difficulty: Hard

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

You are given an array

pairs

, where

$\text{pairs}[i] = [x$

$i$

$, y]$

$i$

$]$

, and:

There are no duplicates.

$x$

$i$

$< y$

i

Let

ways

be the number of rooted trees that satisfy the following conditions:

The tree consists of nodes whose values appeared in

pairs

.

A pair

[x

i

, y

i

]

exists in

pairs

if and only if

x

i

is an ancestor of

y

i

or

y

i

is an ancestor of

x

i

Note:

the tree does not have to be a binary tree.

Two ways are considered to be different if there is at least one node that has different parents in both ways.

Return:

0

if

ways == 0

1

if

ways == 1

2

if

ways > 1

A

rooted tree

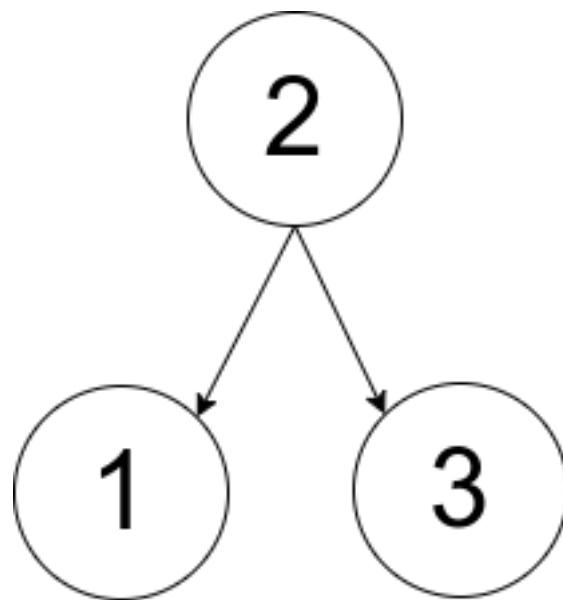
is a tree that has a single root node, and all edges are oriented to be outgoing from the root.

An

ancestor

of a node is any node on the path from the root to that node (excluding the node itself). The root has no ancestors.

Example 1:



Input:

pairs = [[1,2],[2,3]]

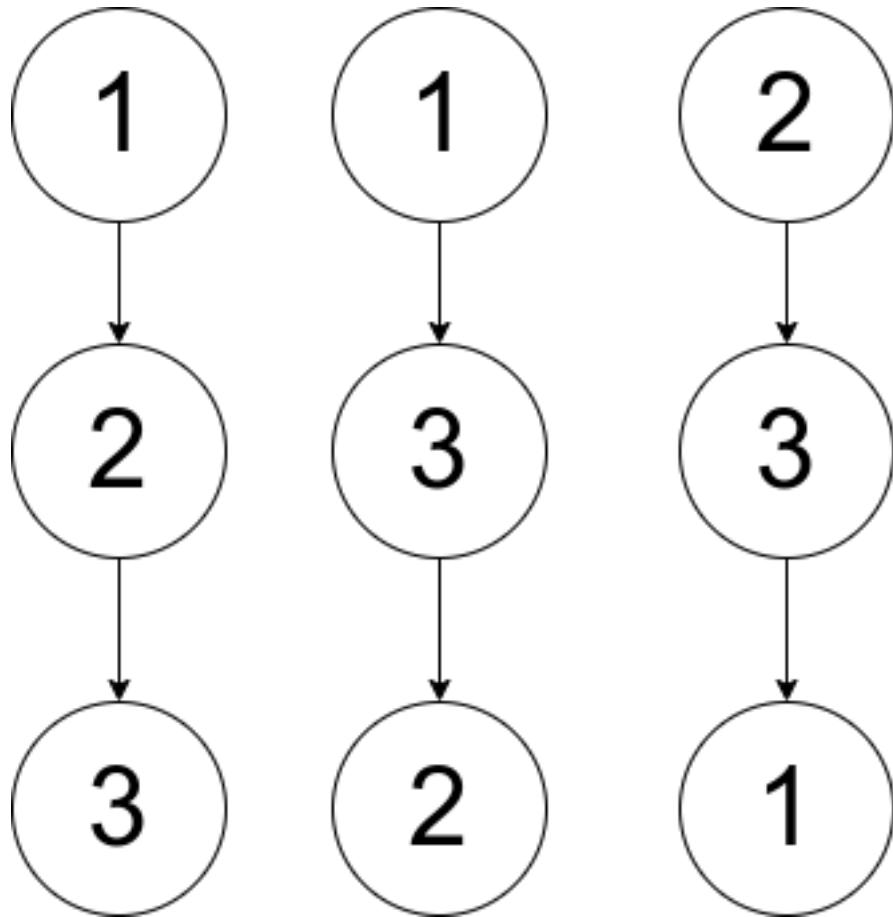
Output:

1

Explanation:

There is exactly one valid rooted tree, which is shown in the above figure.

Example 2:



Input:

```
pairs = [[1,2],[2,3],[1,3]]
```

Output:

2

Explanation:

There are multiple valid rooted trees. Three of them are shown in the above figures.

Example 3:

Input:

```
pairs = [[1,2],[2,3],[2,4],[1,5]]
```

Output:

0

Explanation:

There are no valid rooted trees.

Constraints:

$1 \leq \text{pairs.length} \leq 10$

5

$1 \leq x$

i

$< y$

i

$\leq 500$

The elements in

pairs

are unique.

## Code Snippets

### C++:

```
class Solution {  
public:  
    int checkWays(vector<vector<int>>& pairs) {  
  
    }  
};
```

### Java:

```
class Solution {  
    public int checkWays(int[][] pairs) {  
  
    }  
}
```

### Python3:

```
class Solution:  
    def checkWays(self, pairs: List[List[int]]) -> int:
```

### Python:

```
class Solution(object):  
    def checkWays(self, pairs):  
        """  
        :type pairs: List[List[int]]  
        :rtype: int  
        """
```

### JavaScript:

```
/**  
 * @param {number[][]} pairs  
 * @return {number}  
 */  
var checkWays = function(pairs) {  
  
};
```

**TypeScript:**

```
function checkWays(pairs: number[][]): number {  
}  
};
```

**C#:**

```
public class Solution {  
    public int CheckWays(int[][] pairs) {  
  
    }  
}
```

**C:**

```
int checkWays(int** pairs, int pairsSize, int* pairsColSize) {  
  
}
```

**Go:**

```
func checkWays(pairs [[[int]]]) int {  
  
}
```

**Kotlin:**

```
class Solution {  
    fun checkWays(pairs: Array<IntArray>): Int {  
  
    }  
}
```

**Swift:**

```
class Solution {  
    func checkWays(_ pairs: [[Int]]) -> Int {  
  
    }  
}
```

**Rust:**

```
impl Solution {  
    pub fn check_ways(pairs: Vec<Vec<i32>>) -> i32 {  
        }  
        }  
}
```

### Ruby:

```
# @param {Integer[][]} pairs  
# @return {Integer}  
def check_ways(pairs)  
  
end
```

### PHP:

```
class Solution {  
  
    /**  
     * @param Integer[][] $pairs  
     * @return Integer  
     */  
    function checkWays($pairs) {  
  
    }  
}
```

### Dart:

```
class Solution {  
    int checkWays(List<List<int>> pairs) {  
  
    }  
}
```

### Scala:

```
object Solution {  
    def checkWays(pairs: Array[Array[Int]]): Int = {  
  
    }  
}
```

### Elixir:

```
defmodule Solution do
  @spec check_ways(pairs :: [[integer]]) :: integer
  def check_ways(pairs) do
    end
  end
```

### Erlang:

```
-spec check_ways(Pairs :: [[integer()]]) -> integer().
check_ways(Pairs) ->
  .
```

### Racket:

```
(define/contract (check-ways pairs)
  (-> (listof (listof exact-integer?)) exact-integer?))
```

## Solutions

### C++ Solution:

```
/*
 * Problem: Number Of Ways To Reconstruct 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 checkWays(vector<vector<int>>& pairs) {

  }
};
```

### Java Solution:

```
/**  
 * Problem: Number Of Ways To Reconstruct 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 checkWays(int[][][] pairs) {  
        // Implementation  
    }  
}
```

### Python3 Solution:

```
"""  
Problem: Number Of Ways To Reconstruct 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 checkWays(self, pairs: List[List[int]]) -> int:  
        # TODO: Implement optimized solution  
        pass
```

### Python Solution:

```
class Solution(object):  
    def checkWays(self, pairs):  
        """  
        :type pairs: List[List[int]]  
        :rtype: int
```

```
"""
```

### JavaScript Solution:

```
/**  
 * Problem: Number Of Ways To Reconstruct 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[][]} pairs  
 * @return {number}  
 */  
var checkWays = function(pairs) {  
  
};
```

### TypeScript Solution:

```
/**  
 * Problem: Number Of Ways To Reconstruct 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 checkWays(pairs: number[][]): number {  
  
};
```

### C# Solution:

```

/*
 * Problem: Number Of Ways To Reconstruct 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 CheckWays(int[][][] pairs) {
        return 0;
    }
}

```

## C Solution:

```

/*
 * Problem: Number Of Ways To Reconstruct 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
 */

int checkWays(int** pairs, int pairsSize, int* pairsColSize) {
    return 0;
}

```

## Go Solution:

```

// Problem: Number Of Ways To Reconstruct 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 checkWays(pairs [][]int) int {  
}
```

## Kotlin Solution:

```
class Solution {  
    fun checkWays(pairs: Array<IntArray>): Int {  
        ...  
    }  
}
```

## **Swift Solution:**

```
class Solution {
    func checkWays(_ pairs: [[Int]]) -> Int {
}
}
```

## Rust Solution:

```
// Problem: Number Of Ways To Reconstruct 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 check_ways(pairs: Vec<Vec<i32>>) -> i32 {
        }

    }
}
```

## Ruby Solution:

```
# @param {Integer[][]} pairs
# @return {Integer}
def check_ways(pairs)
```

```
end
```

### PHP Solution:

```
class Solution {  
  
    /**  
     * @param Integer[][] $pairs  
     * @return Integer  
     */  
    function checkWays($pairs) {  
  
    }  
}
```

### Dart Solution:

```
class Solution {  
int checkWays(List<List<int>> pairs) {  
  
}  
}
```

### Scala Solution:

```
object Solution {  
def checkWays(pairs: Array[Array[Int]]): Int = {  
  
}  
}
```

### Elixir Solution:

```
defmodule Solution do  
@spec check_ways([integer]) :: integer  
def check_ways(pairs) do  
  
end  
end
```

### Erlang Solution:

```
-spec check_ways(Pairs :: [[integer()]])) -> integer().  
check_ways(Pairs) ->  
.
```

### Racket Solution:

```
(define/contract (check-ways pairs)  
(-> (listof (listof exact-integer?)) exact-integer?)  
)
```