Problem 3544: Subtree Inversion Sum

Problem Information

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

Problem Description

You are given an undirected tree rooted at node

0

, with

n

nodes numbered from 0 to

n - 1

. The tree is represented by a 2D integer array

edges

of length

n - 1

, where

edges[i] = [u

i

, v

i

]

indicates an edge between nodes

u

i

and

v

i

.

You are also given an integer array

nums

of length

n

, where

nums[i]

represents the value at node

i

, and an integer

k

.

You may perform

inversion operations

on a subset of nodes subject to the following rules:

Subtree Inversion Operation:

When you invert a node, every value in the

subtree

rooted at that node is multiplied by -1.

Distance Constraint on Inversions:

You may only invert a node if it is "sufficiently far" from any other inverted node.

Specifically, if you invert two nodes

a

and

b

such that one is an ancestor of the other (i.e., if

LCA(a, b) = a

or

LCA(a, b) = b

), then the distance (the number of edges on the unique path between them) must be at least

k

.

Return the

maximum

possible

sum

of the tree's node values after applying

inversion operations

.

Example 1:

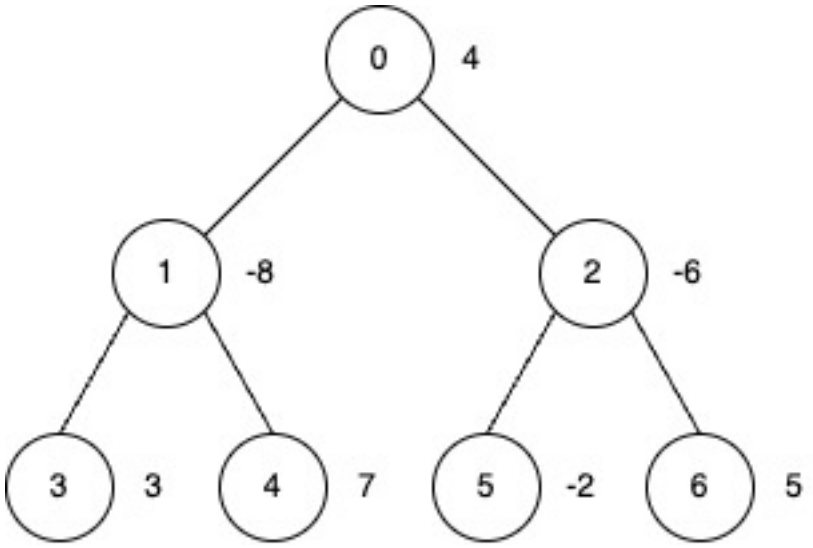
Input:

edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]], nums = [4,-8,-6,3,7,-2,5], k = 2

Output:

27

Explanation:



Apply inversion operations at nodes 0, 3, 4 and 6.

The final

nums

array is

[-4, 8, 6, 3, 7, 2, 5]

, and the total sum is 27.

Example 2:

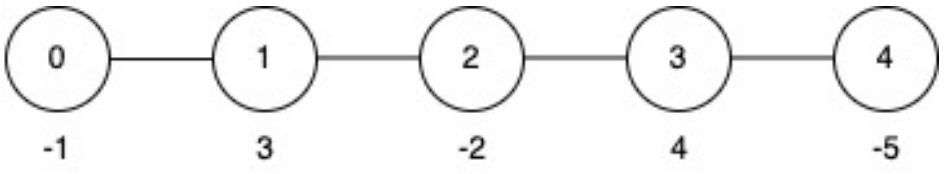
Input:

edges = [[0,1],[1,2],[2,3],[3,4]], nums = [-1,3,-2,4,-5], k = 2

Output:

9

Explanation:



Apply the inversion operation at node 4.

The final

nums

array becomes

[-1, 3, -2, 4, 5]

, and the total sum is 9.

Example 3:

Input:

edges = [[0,1],[0,2]], nums = [0,-1,-2], k = 3

Output:

3

Explanation:

Apply inversion operations at nodes 1 and 2.

Constraints:

2 <= n <= 5 * 10

4

edges.length == n - 1

edges[i] = [u

i

, v

i

]

0 <= u

i

, v

i

< n

nums.length == n

-5 * 10

4

<= nums[i] <= 5 * 10

4

1 <= k <= 50

The input is generated such that

edges

represents a valid tree.

Code Snippets

C++:

```
class Solution {
public:
long long subtreeInversionSum(vector<vector<int>>& edges, vector<int>& nums,
int k) {

}
};
```

Java:

```
class Solution {
public long subtreeInversionSum(int[][] edges, int[] nums, int k) {

}
}
```

Python3:

```
class Solution:
def subtreeInversionSum(self, edges: List[List[int]], nums: List[int], k:
int) -> int:
```

Python:

```
class Solution(object):
def subtreeInversionSum(self, edges, nums, k):
"""
:type edges: List[List[int]]
:type nums: List[int]
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} edges
* @param {number[]} nums
* @param {number} k
```

```
 * @return {number}
 */
var subtreeInversionSum = function(edges, nums, k) {

};
```

TypeScript:

```
function subtreeInversionSum(edges: number[][], nums: number[], k: number):
number {

};
```

C#:

```
public class Solution {
public long SubtreeInversionSum(int[][] edges, int[] nums, int k) {

}
}
```

C:

```
long long subtreeInversionSum(int** edges, int edgesSize, int* edgesColSize,
int* nums, int numsSize, int k) {

}
```

Go:

```
func subtreeInversionSum(edges [][]int, nums []int, k int) int64 {

}
```

Kotlin:

```
class Solution {
fun subtreeInversionSum(edges: Array<IntArray>, nums: IntArray, k: Int): Long
{

}
}
```

Swift:

```
class Solution {
func subtreeInversionSum(_ edges: [[Int]], _ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn subtree_inversion_sum(edges: Vec<Vec<i32>>, nums: Vec<i32>, k: i32) ->
i64 {

}
}
```

Ruby:

```
# @param {Integer[][]} edges
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def subtree_inversion_sum(edges, nums, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $edges
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function subtreeInversionSum($edges, $nums, $k) {

}
}
```

Dart:

```
class Solution {
int subtreeInversionSum(List<List<int>> edges, List<int> nums, int k) {

}
}
```

Scala:

```
object Solution {
def subtreeInversionSum(edges: Array[Array[Int]], nums: Array[Int], k: Int):
Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec subtree_inversion_sum(edges :: [[integer]], nums :: [integer], k ::
integer) :: integer
def subtree_inversion_sum(edges, nums, k) do

end
end
```

Erlang:

```
-spec subtree_inversion_sum(Edges :: [[integer()]], Nums :: [integer()], K ::
integer()) -> integer().
subtree_inversion_sum(Edges, Nums, K) ->
.
```

Racket:

```
(define/contract (subtree-inversion-sum edges nums k)
(-> (listof (listof exact-integer?)) (listof exact-integer?) exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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:
long long subtreeInversionSum(vector<vector<int>>& edges, vector<int>& nums,
int k) {

}
};
```

Java Solution:

```
/**
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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 long subtreeInversionSum(int[][] edges, int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Subtree Inversion Sum
Difficulty: Hard
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 subtreeInversionSum(self, edges: List[List[int]], nums: List[int], k:
int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def subtreeInversionSum(self, edges, nums, k):
"""
:type edges: List[List[int]]
:type nums: List[int]
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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[][]} edges
* @param {number[]} nums
```

```
* @param {number} k
* @return {number}
*/
var subtreeInversionSum = function(edges, nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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 subtreeInversionSum(edges: number[][], nums: number[], k: number):
number {

};
```

C# Solution:

```
/*
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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 long SubtreeInversionSum(int[][] edges, int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Subtree Inversion Sum
* Difficulty: Hard
* 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
*/

long long subtreeInversionSum(int** edges, int edgesSize, int* edgesColSize,
int* nums, int numsSize, int k) {

}
```

Go Solution:

```
// Problem: Subtree Inversion Sum
// Difficulty: Hard
// 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 subtreeInversionSum(edges [][]int, nums []int, k int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun subtreeInversionSum(edges: Array<IntArray>, nums: IntArray, k: Int): Long
{

}
}
```

Swift Solution:

```
class Solution {
func subtreeInversionSum(_ edges: [[Int]], _ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Subtree Inversion Sum
// Difficulty: Hard
// 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 subtree_inversion_sum(edges: Vec<Vec<i32>>, nums: Vec<i32>, k: i32) ->
i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} edges
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def subtree_inversion_sum(edges, nums, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $edges
* @param Integer[] $nums
* @param Integer $k
* @return Integer
```

```
*/
function subtreeInversionSum($edges, $nums, $k) {

}
}
```

Dart Solution:

```
class Solution {
int subtreeInversionSum(List<List<int>> edges, List<int> nums, int k) {

}
}
```

Scala Solution:

```
object Solution {
def subtreeInversionSum(edges: Array[Array[Int]], nums: Array[Int], k: Int):
Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec subtree_inversion_sum(edges :: [[integer]], nums :: [integer], k ::
integer) :: integer
def subtree_inversion_sum(edges, nums, k) do

end
end
```

Erlang Solution:

```
-spec subtree_inversion_sum(Edges :: [[integer()]], Nums :: [integer()], K ::
integer()) -> integer().
subtree_inversion_sum(Edges, Nums, K) ->
.
```

Racket Solution:

```
(define/contract (subtree-inversion-sum edges nums k)
(-> (listof (listof exact-integer?)) (listof exact-integer?) exact-integer?
exact-integer?)
)
```