Problem 2479: Maximum XOR of Two Non-Overlapping Subtrees

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] = [a

i

, b

i

]

indicates that there is an edge between nodes

a

i

and

b

i

in the tree. The root of the tree is the node labeled

0

.

Each node has an associated

value

. You are given an array

values

of length

n

, where

values[i]

is the

value

of the

i

th

node.

Select any two

non-overlapping

subtrees. Your

score

is the bitwise XOR of the sum of the values within those subtrees.

Return

the

maximum

possible

score

you can achieve

.

If it is impossible to find two nonoverlapping subtrees

, return

0

.

Note

that:

The

subtree

of a node is the tree consisting of that node and all of its descendants.

Two subtrees are

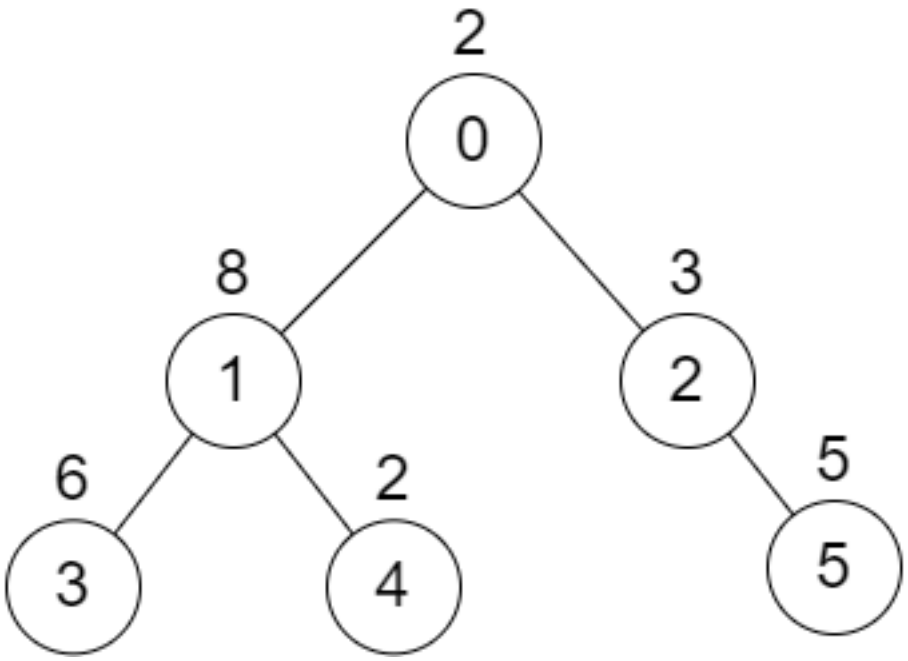
non-overlapping

if they do not share

any common

node.

Example 1:



Input:

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

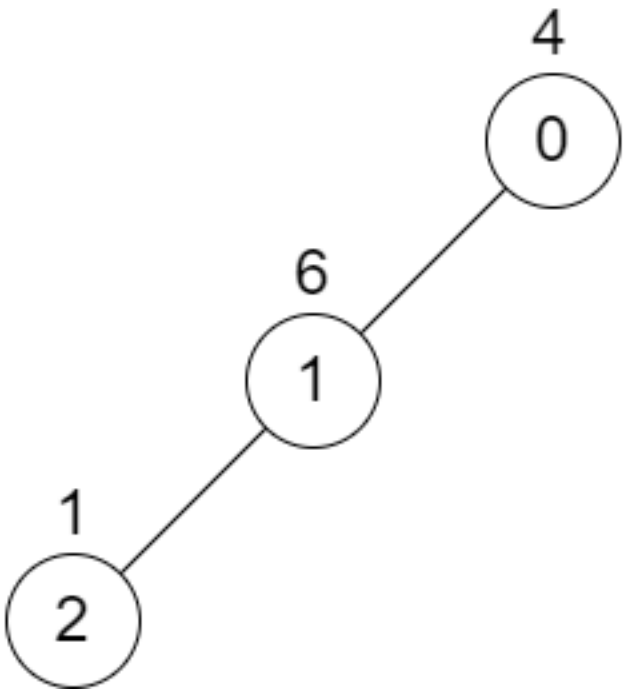
Output:

24

Explanation:

Node 1's subtree has sum of values 16, while node 2's subtree has sum of values 8, so choosing these nodes will yield a score of 16 XOR 8 = 24. It can be proved that is the maximum possible score we can obtain.

Example 2:



Input:

n = 3, edges = [[0,1],[1,2]], values = [4,6,1]

Output:

0

Explanation:

There is no possible way to select two non-overlapping subtrees, so we just return 0.

Constraints:

2 <= n <= 5 * 10

4

edges.length == n - 1

0 <= a

i

, b

i

< n

values.length == n

1 <= values[i] <= 10

9

It is guaranteed that

edges

represents a valid tree.

Code Snippets

C++:

```
class Solution {
public:
long long maxXor(int n, vector<vector<int>>& edges, vector<int>& values) {

}
};
```

Java:

```
class Solution {
public long maxXor(int n, int[][] edges, int[] values) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function maxXor(n: number, edges: number[][], values: number[]): number {

};
```

C#:

```
public class Solution {
public long MaxXor(int n, int[][] edges, int[] values) {

}
}
```

C:

```
long long maxXor(int n, int** edges, int edgesSize, int* edgesColSize, int*
values, int valuesSize) {

}
```

Go:

```
func maxXor(n int, edges [][]int, values []int) int64 {

}
```

Kotlin:

```
class Solution {
fun maxXor(n: Int, edges: Array<IntArray>, values: IntArray): Long {

}
}
```

Swift:

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

}
}
```

Rust:

```
impl Solution {
pub fn max_xor(n: i32, edges: Vec<Vec<i32>>, values: Vec<i32>) -> i64 {

}
}
```

Ruby:

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

```
end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
int maxXor(int n, List<List<int>> edges, List<int> values) {

}
}
```

Scala:

```
object Solution {
def maxXor(n: Int, edges: Array[Array[Int]], values: Array[Int]): Long = {

}
}
```

Elixir:

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

end
end
```

Erlang:

```
-spec max_xor(N :: integer(), Edges :: [[integer()]], Values :: [integer()])
-> integer().
max_xor(N, Edges, Values) ->
.
```

Racket:

```
(define/contract (max-xor n edges values)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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:
long long maxXor(int n, vector<vector<int>>& edges, vector<int>& values) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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 long maxXor(int n, int[][] edges, int[] values) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum XOR of Two Non-Overlapping Subtrees
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 maxXor(self, n: int, edges: List[List[int]], values: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```javascript
/**
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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} n
* @param {number[][]} edges
* @param {number[]} values
* @return {number}
*/
var maxXor = function(n, edges, values) {

};
```

TypeScript Solution:

```typescript
/**
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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 maxXor(n: number, edges: number[][], values: number[]): number {

};
```

C# Solution:

```csharp
/*
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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 long MaxXor(int n, int[][] edges, int[] values) {

}
}
```

C Solution:

```
/*
* Problem: Maximum XOR of Two Non-Overlapping Subtrees
* 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
*/

long long maxXor(int n, int** edges, int edgesSize, int* edgesColSize, int*
values, int valuesSize) {

}
```

Go Solution:

```
// Problem: Maximum XOR of Two Non-Overlapping Subtrees
// 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 maxXor(n int, edges [][]int, values []int) int64 {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun maxXor(n: Int, edges: Array<IntArray>, values: IntArray): Long {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum XOR of Two Non-Overlapping Subtrees
// 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 max_xor(n: i32, edges: Vec<Vec<i32>>, values: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

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

```
end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
int maxXor(int n, List<List<int>> edges, List<int> values) {

}
}
```

Scala Solution:

```
object Solution {
def maxXor(n: Int, edges: Array[Array[Int]], values: Array[Int]): Long = {

}
}
```

Elixir Solution:

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

```
end
end
```

Erlang Solution:

```
-spec max_xor(N :: integer(), Edges :: [[integer()]], Values :: [integer()])
-> integer().
max_xor(N, Edges, Values) ->
.
```

Racket Solution:

```
(define/contract (max-xor n edges values)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```