Problem 558: Logical OR of Two Binary Grids Represented as Quad-Trees

Problem Information

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

Problem Description

A Binary Matrix is a matrix in which all the elements are either

0

or

1

.

Given

quadTree1

and

quadTree2

.

quadTree1

represents a

n * n

binary matrix and

quadTree2

represents another

n * n

binary matrix.

Return

a Quad-Tree

representing the

n * n

binary matrix which is the result of

logical bitwise OR

of the two binary matrixes represented by

quadTree1

and

quadTree2

.

Notice that you can assign the value of a node to

True

or

False

when

isLeaf

is

False

, and both are

accepted

in the answer.

A Quad-Tree is a tree data structure in which each internal node has exactly four children. Besides, each node has two attributes:

val

: True if the node represents a grid of 1's or False if the node represents a grid of 0's.

isLeaf

: True if the node is leaf node on the tree or False if the node has the four children.

class Node { public boolean val; public boolean isLeaf; public Node topLeft; public Node topRight; public Node bottomLeft; public Node bottomRight; }

We can construct a Quad-Tree from a two-dimensional area using the following steps:

If the current grid has the same value (i.e all

1's

or all

0's

) set

isLeaf

True and set

val

to the value of the grid and set the four children to Null and stop.

If the current grid has different values, set

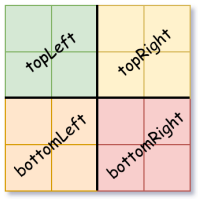
isLeaf

to False and set

val

to any value and divide the current grid into four sub-grids as shown in the photo.

Recurse for each of the children with the proper sub-grid.



If you want to know more about the Quad-Tree, you can refer to the

wiki

.

Quad-Tree format:

The input/output represents the serialized format of a Quad-Tree using level order traversal, where

null

signifies a path terminator where no node exists below.

It is very similar to the serialization of the binary tree. The only difference is that the node is represented as a list

[isLeaf, val]

.

If the value of

isLeaf

or

val

is True we represent it as

1

in the list

[isLeaf, val]

and if the value of

isLeaf

or

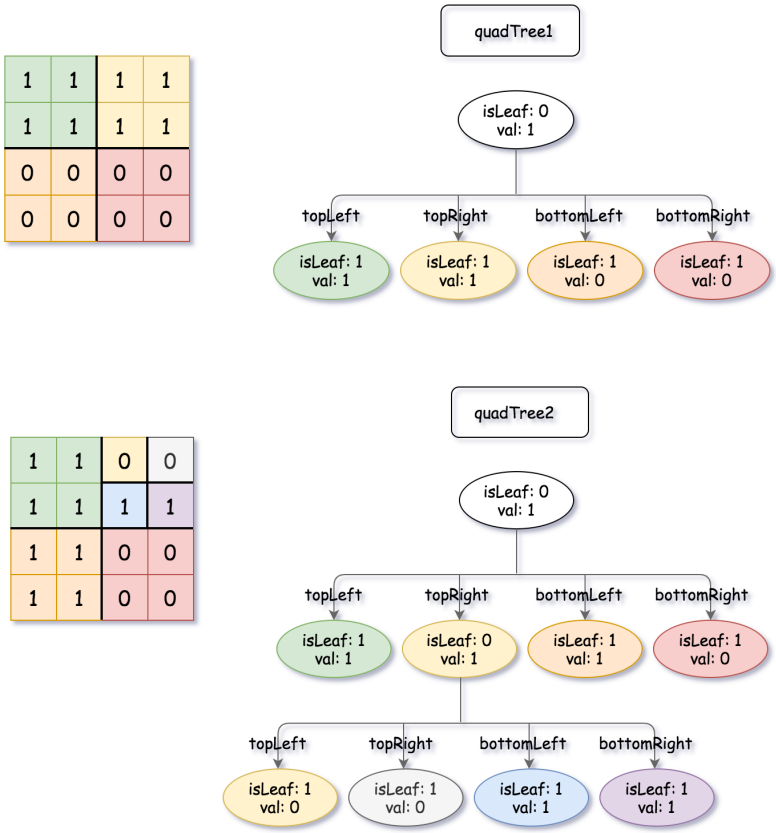
val

is False we represent it as

0

.

Example 1:



Input:

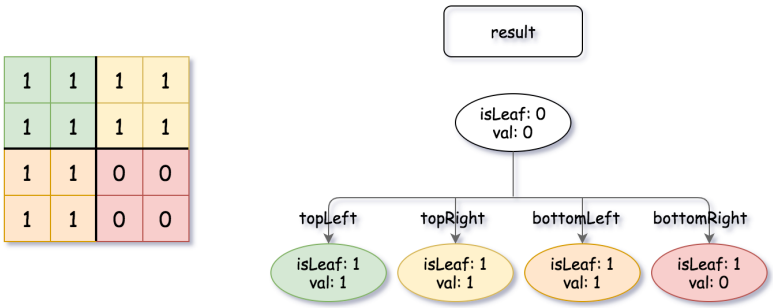
quadTree1 = [[0,1],[1,1],[1,1],[1,0],[1,0]] , quadTree2 = [[0,1],[1,1],[0,1],[1,1],[1,0],null,null,null,null,[1,0],[1,0],[1,1],[1,1]]

Output:

[[0,0],[1,1],[1,1],[1,1],[1,0]]

Explanation:

quadTree1 and quadTree2 are shown above. You can see the binary matrix which is represented by each Quad-Tree. If we apply logical bitwise OR on the two binary matrices we get the binary matrix below which is represented by the result Quad-Tree. Notice that the binary matrices shown are only for illustration, you don't have to construct the binary matrix to get the result tree.



Example 2:

Input:

quadTree1 = [[1,0]], quadTree2 = [[1,0]]

Output:

[[1,0]]

Explanation:

Each tree represents a binary matrix of size 1*1. Each matrix contains only zero. The resulting matrix is of size 1*1 with also zero.

Constraints:

quadTree1

and

quadTree2

are both

valid

Quad-Trees each representing a

n * n

grid.

n == 2

x

where

0 <= x <= 9

.

Code Snippets

C++:

```
/*
// Definition for a QuadTree node.
class Node {
public:
bool val;
bool isLeaf;
Node* topLeft;
Node* topRight;
Node* bottomLeft;
Node* bottomRight;

Node() {
val = false;
isLeaf = false;
topLeft = NULL;
topRight = NULL;
bottomLeft = NULL;
bottomRight = NULL;
}

Node(bool _val, bool _isLeaf) {
val = _val;
isLeaf = _isLeaf;
topLeft = NULL;
```

```
topRight = NULL;
bottomLeft = NULL;
bottomRight = NULL;
}

Node(bool _val, bool _isLeaf, Node* _topLeft, Node* _topRight, Node*
_bottomLeft, Node* _bottomRight) {
val = _val;
isLeaf = _isLeaf;
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
}
};
*/

class Solution {
public:
Node* intersect(Node* quadTree1, Node* quadTree2) {

}
};
```

Java:

```
/*
// Definition for a QuadTree node.
class Node {
public boolean val;
public boolean isLeaf;
public Node topLeft;
public Node topRight;
public Node bottomLeft;
public Node bottomRight;

public Node() {}

public Node(boolean _val,boolean _isLeaf,Node _topLeft,Node _topRight,Node
_bottomLeft,Node _bottomRight) {
val = _val;
isLeaf = _isLeaf;
```

```
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
}
};
*/

class Solution {
public Node intersect(Node quadTree1, Node quadTree2) {

}
}
```

Python3:

```
"""
# Definition for a QuadTree node.
class Node:
def __init__(self, val, isLeaf, topLeft, topRight, bottomLeft, bottomRight):
self.val = val
self.isLeaf = isLeaf
self.topLeft = topLeft
self.topRight = topRight
self.bottomLeft = bottomLeft
self.bottomRight = bottomRight
"""

class Solution:
def intersect(self, quadTree1: 'Node', quadTree2: 'Node') -> 'Node':
```

Python:

```
"""
# Definition for a QuadTree node.
class Node(object):
def __init__(self, val, isLeaf, topLeft, topRight, bottomLeft, bottomRight):
self.val = val
self.isLeaf = isLeaf
self.topLeft = topLeft
self.topRight = topRight
self.bottomLeft = bottomLeft
```

```
self.bottomRight = bottomRight
"""

class Solution(object):
def intersect(self, quadTree1, quadTree2):
"""
:type quadTree1: Node
:type quadTree2: Node
:rtype: Node
"""
```

JavaScript:

```
/**
* // Definition for a QuadTree node.
* function _Node(val,isLeaf,topLeft,topRight,bottomLeft,bottomRight) {
* this.val = val;
* this.isLeaf = isLeaf;
* this.topLeft = topLeft;
* this.topRight = topRight;
* this.bottomLeft = bottomLeft;
* this.bottomRight = bottomRight;
* };
*/

/**
* @param {_Node} quadTree1
* @param {_Node} quadTree2
* @return {_Node}
*/
var intersect = function(quadTree1, quadTree2) {

};
```

TypeScript:

```
/**
* Definition for _Node.
* class _Node {
* val: boolean
* isLeaf: boolean
* topLeft: _Node | null
```

```
* topRight: _Node | null
* bottomLeft: _Node | null
* bottomRight: _Node | null
* constructor(val?: boolean, isLeaf?: boolean, topLeft?: _Node, topRight?:
_Node, bottomLeft?: _Node, bottomRight?: _Node) {
* this.val = (val===undefined ? false : val)
* this.isLeaf = (isLeaf===undefined ? false : isLeaf)
* this.topLeft = (topLeft===undefined ? null : topLeft)
* this.topRight = (topRight===undefined ? null : topRight)
* this.bottomLeft = (bottomLeft===undefined ? null : bottomLeft)
* this.bottomRight = (bottomRight===undefined ? null : bottomRight)
* }
* }
*/


function intersect(quadTree1: _Node | null, quadTree2: _Node | null): _Node |
null {

};
```

C#:

```
/*
// Definition for a QuadTree node.
public class Node {
public bool val;
public bool isLeaf;
public Node topLeft;
public Node topRight;
public Node bottomLeft;
public Node bottomRight;

public Node(){}
public Node(bool _val,bool _isLeaf,Node _topLeft,Node _topRight,Node
_bottomLeft,Node _bottomRight) {
val = _val;
isLeaf = _isLeaf;
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
```

```
}
}
*/

public class Solution {
public Node Intersect(Node quadTree1, Node quadTree2) {

}
}
```

Go:

```
/**
* Definition for a QuadTree node.
* type Node struct {
* Val bool
* IsLeaf bool
* TopLeft *Node
* TopRight *Node
* BottomLeft *Node
* BottomRight *Node
* }
*/

func intersect(quadTree1 *Node, quadTree2 *Node) *Node {

}
```

Kotlin:

```
/**
* Definition for a QuadTree node.
* class Node(var `val`: Boolean, var isLeaf: Boolean) {
* var topLeft: Node? = null
* var topRight: Node? = null
* var bottomLeft: Node? = null
* var bottomRight: Node? = null
* }
*/

class Solution {
fun intersect(quadTree1: Node?, quadTree2: Node?): Node? {
```

```
}
}
```

Swift:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Bool
* public var isLeaf: Bool
* public var topLeft: Node?
* public var topRight: Node?
* public var bottomLeft: Node?
* public var bottomRight: Node?
* public init(_ val: Bool, _ isLeaf: Bool) {
* self.val = val
* self.isLeaf = isLeaf
* self.topLeft = nil
* self.topRight = nil
* self.bottomLeft = nil
* self.bottomRight = nil
* }
* }
*/

class Solution {
func intersect(_ quadTree1: Node?, _ quadTree2: Node?) -> Node? {

}
}
```

Ruby:

```
# Definition for a QuadTree node.
# class Node
# attr_accessor :val, :isLeaf, :topLeft, :topRight, :bottomLeft, :bottomRight
# def initialize(val=false, isLeaf=false, topLeft=nil, topRight=nil,
bottomLeft=nil, bottomRight=nil)
# @val = val
# @isLeaf = isLeaf
# @topLeft = topLeft
```

```
# @topRight = topRight
# @bottomLeft = bottomLeft
# @bottomRight = bottomRight
# end
# end

# @param {Node} quadTree1
# @param {Node} quadTree2
# @return {Node}
def intersect(quadTree1, quadTree2)

end
```

PHP:

```
/**
* Definition for a QuadTree node.
* class Node {
* public $val = null;
* public $isLeaf = null;
* public $topLeft = null;
* public $topRight = null;
* public $bottomLeft = null;
* public $bottomRight = null;
* function __construct($val, $isLeaf) {
* $this->val = $val;
* $this->isLeaf = $isLeaf;
* $this->topLeft = null;
* $this->topRight = null;
* $this->bottomLeft = null;
* $this->bottomRight = null;
* }
* }
*/

class Solution {
/**
* @param Node $quadTree1
* @param Node $quadTree2
* @return Node
*/
function intersect($quadTree1, $quadTree2) {
```

```
}
}
```

Scala:

```
/**
* Definition for a QuadTree node.
* class Node(var _value: Boolean, var _isLeaf: Boolean) {
* var value: Int = _value
* var isLeaf: Boolean = _isLeaf
* var topLeft: Node = null
* var topRight: Node = null
* var bottomLeft: Node = null
* var bottomRight: Node = null
* }
*/

object Solution {
def intersect(quadTree1: Node, quadTree2: Node): Node = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
* Difficulty: Medium
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a QuadTree node.
```

```
class Node {
public:
bool val;
bool isLeaf;
Node* topLeft;
Node* topRight;
Node* bottomLeft;
Node* bottomRight;

Node() {
val = false;
isLeaf = false;
topLeft = NULL;
topRight = NULL;
bottomLeft = NULL;
bottomRight = NULL;
}

Node(bool _val, bool _isLeaf) {
val = _val;
isLeaf = _isLeaf;
topLeft = NULL;
topRight = NULL;
bottomLeft = NULL;
bottomRight = NULL;
}

Node(bool _val, bool _isLeaf, Node* _topLeft, Node* _topRight, Node*
_bottomLeft, Node* _bottomRight) {
val = _val;
isLeaf = _isLeaf;
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
}
};
*/

class Solution {
public:
Node* intersect(Node* quadTree1, Node* quadTree2) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
* Difficulty: Medium
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a QuadTree node.
class Node {
public boolean val;
public boolean isLeaf;
public Node topLeft;
public Node topRight;
public Node bottomLeft;
public Node bottomRight;

public Node() {
// TODO: Implement optimized solution
return 0;
}

public Node(boolean _val,boolean _isLeaf,Node _topLeft,Node _topRight,Node
_bottomLeft,Node _bottomRight) {
val = _val;
isLeaf = _isLeaf;
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
}
};
```

```
*/

class Solution {
public Node intersect(Node quadTree1, Node quadTree2) {

}
}
```

Python3 Solution:

```
"""
Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
Difficulty: Medium
Tags: tree

Approach: DFS or BFS traversal
Time Complexity: O(n) where n is number of nodes
Space Complexity: O(h) for recursion stack where h is height
"""

"""
# Definition for a QuadTree node.
class Node:
def __init__(self, val, isLeaf, topLeft, topRight, bottomLeft, bottomRight):
self.val = val
self.isLeaf = isLeaf
self.topLeft = topLeft
self.topRight = topRight
self.bottomLeft = bottomLeft
self.bottomRight = bottomRight
"""

class Solution:
def intersect(self, quadTree1: 'Node', quadTree2: 'Node') -> 'Node':
# TODO: Implement optimized solution
pass
```

Python Solution:

```
"""
# Definition for a QuadTree node.
```

```
class Node(object):
def __init__(self, val, isLeaf, topLeft, topRight, bottomLeft, bottomRight):
self.val = val
self.isLeaf = isLeaf
self.topLeft = topLeft
self.topRight = topRight
self.bottomLeft = bottomLeft
self.bottomRight = bottomRight
"""

class Solution(object):
def intersect(self, quadTree1, quadTree2):
"""
:type quadTree1: Node
:type quadTree2: Node
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
* Difficulty: Medium
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* // Definition for a QuadTree node.
* function _Node(val,isLeaf,topLeft,topRight,bottomLeft,bottomRight) {
* this.val = val;
* this.isLeaf = isLeaf;
* this.topLeft = topLeft;
* this.topRight = topRight;
* this.bottomLeft = bottomLeft;
* this.bottomRight = bottomRight;
* };
*/
```

```
/**
* @param {_Node} quadTree1
* @param {_Node} quadTree2
* @return {_Node}
*/
var intersect = function(quadTree1, quadTree2) {

};
```

TypeScript Solution:

```
/**
* Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
* Difficulty: Medium
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for _Node.
* class _Node {
* val: boolean
* isLeaf: boolean
* topLeft: _Node | null
* topRight: _Node | null
* bottomLeft: _Node | null
* bottomRight: _Node | null
* constructor(val?: boolean, isLeaf?: boolean, topLeft?: _Node, topRight?:
_Node, bottomLeft?: _Node, bottomRight?: _Node) {
* this.val = (val===undefined ? false : val)
* this.isLeaf = (isLeaf===undefined ? false : isLeaf)
* this.topLeft = (topLeft===undefined ? null : topLeft)
* this.topRight = (topRight===undefined ? null : topRight)
* this.bottomLeft = (bottomLeft===undefined ? null : bottomLeft)
* this.bottomRight = (bottomRight===undefined ? null : bottomRight)
* }
* }
```

```
*/


function intersect(quadTree1: _Node | null, quadTree2: _Node | null): _Node |
null {

};
```

C# Solution:

```
/*
* Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
* Difficulty: Medium
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a QuadTree node.
public class Node {
public bool val;
public bool isLeaf;
public Node topLeft;
public Node topRight;
public Node bottomLeft;
public Node bottomRight;

public Node(){}
public Node(bool _val,bool _isLeaf,Node _topLeft,Node _topRight,Node
_bottomLeft,Node _bottomRight) {
val = _val;
isLeaf = _isLeaf;
topLeft = _topLeft;
topRight = _topRight;
bottomLeft = _bottomLeft;
bottomRight = _bottomRight;
}
}
```

```
*/

public class Solution {
public Node Intersect(Node quadTree1, Node quadTree2) {

}
}
```

Go Solution:

```
// Problem: Logical OR of Two Binary Grids Represented as Quad-Trees
// Difficulty: Medium
// Tags: tree
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

/**
* Definition for a QuadTree node.
* type Node struct {
* Val bool
* IsLeaf bool
* TopLeft *Node
* TopRight *Node
* BottomLeft *Node
* BottomRight *Node
* }
*/

func intersect(quadTree1 *Node, quadTree2 *Node) *Node {

}
```

Kotlin Solution:

```
/**
* Definition for a QuadTree node.
* class Node(var `val`: Boolean, var isLeaf: Boolean) {
* var topLeft: Node? = null
* var topRight: Node? = null
```

```
* var bottomLeft: Node? = null
* var bottomRight: Node? = null
* }
*/

class Solution {
fun intersect(quadTree1: Node?, quadTree2: Node?): Node? {

}
}
```

Swift Solution:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Bool
* public var isLeaf: Bool
* public var topLeft: Node?
* public var topRight: Node?
* public var bottomLeft: Node?
* public var bottomRight: Node?
* public init(_ val: Bool, _ isLeaf: Bool) {
* self.val = val
* self.isLeaf = isLeaf
* self.topLeft = nil
* self.topRight = nil
* self.bottomLeft = nil
* self.bottomRight = nil
* }
* }
*/

class Solution {
func intersect(_ quadTree1: Node?, _ quadTree2: Node?) -> Node? {

}
}
```

Ruby Solution:

```
# Definition for a QuadTree node.
# class Node
# attr_accessor :val, :isLeaf, :topLeft, :topRight, :bottomLeft, :bottomRight
# def initialize(val=false, isLeaf=false, topLeft=nil, topRight=nil,
bottomLeft=nil, bottomRight=nil)
# @val = val
# @isLeaf = isLeaf
# @topLeft = topLeft
# @topRight = topRight
# @bottomLeft = bottomLeft
# @bottomRight = bottomRight
# end
# end

# @param {Node} quadTree1
# @param {Node} quadTree2
# @return {Node}
def intersect(quadTree1, quadTree2)

end
```

PHP Solution:

```
/**
* Definition for a QuadTree node.
* class Node {
* public $val = null;
* public $isLeaf = null;
* public $topLeft = null;
* public $topRight = null;
* public $bottomLeft = null;
* public $bottomRight = null;
* function __construct($val, $isLeaf) {
* $this->val = $val;
* $this->isLeaf = $isLeaf;
* $this->topLeft = null;
* $this->topRight = null;
* $this->bottomLeft = null;
* $this->bottomRight = null;
* }
* }
*/
```

```
class Solution {
/**
* @param Node $quadTree1
* @param Node $quadTree2
* @return Node
*/
function intersect($quadTree1, $quadTree2) {

}
}
```

Scala Solution:

```
/**
* Definition for a QuadTree node.
* class Node(var _value: Boolean, var _isLeaf: Boolean) {
* var value: Int = _value
* var isLeaf: Boolean = _isLeaf
* var topLeft: Node = null
* var topRight: Node = null
* var bottomLeft: Node = null
* var bottomRight: Node = null
* }
*/

object Solution {
def intersect(quadTree1: Node, quadTree2: Node): Node = {

}
}
```