Problem 2509: Cycle Length Queries in a Tree

Problem Information

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

Problem Description

You are given an integer

n

. There is a

complete binary tree

with

2

n

- 1

nodes. The root of that tree is the node with the value

1

, and every node with a value

val

in the range

[1, 2

n - 1

- 1]

has two children where:

The left node has the value

2 * val

, and

The right node has the value

2 * val + 1

.

You are also given a 2D integer array

queries

of length

m

, where

queries[i] = [a

i

, b

i

]

. For each query, solve the following problem:

Add an edge between the nodes with values

a

i

and

b

i

.

Find the length of the cycle in the graph.

Remove the added edge between nodes with values

a

i

and

b

i

.

Note

that:

A

cycle

is a path that starts and ends at the same node, and each edge in the path is visited only once.

The length of a cycle is the number of edges visited in the cycle.

There could be multiple edges between two nodes in the tree after adding the edge of the query.

Return

an array

answer

of length

m

where

answer[i]

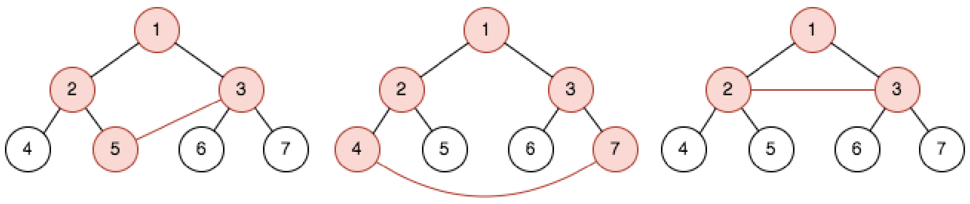
is the answer to the

i

th

query.

Example 1:



Input:

n = 3, queries = [[5,3],[4,7],[2,3]]

Output:

[4,5,3]

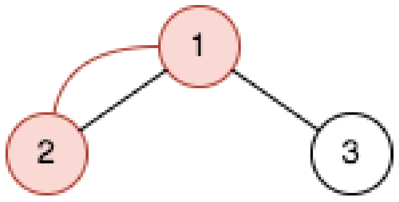
Explanation:

The diagrams above show the tree of 2

3

- 1 nodes. Nodes colored in red describe the nodes in the cycle after adding the edge. - After adding the edge between nodes 3 and 5, the graph contains a cycle of nodes [5,2,1,3]. Thus answer to the first query is 4. We delete the added edge and process the next query. - After adding the edge between nodes 4 and 7, the graph contains a cycle of nodes [4,2,1,3,7]. Thus answer to the second query is 5. We delete the added edge and process the next query. - After adding the edge between nodes 2 and 3, the graph contains a cycle of nodes [2,1,3]. Thus answer to the third query is 3. We delete the added edge.

Example 2:



Input:

n = 2, queries = [[1,2]]

Output:

[2]

Explanation:

The diagram above shows the tree of 2

2

- 1 nodes. Nodes colored in red describe the nodes in the cycle after adding the edge. - After adding the edge between nodes 1 and 2, the graph contains a cycle of nodes [2,1]. Thus answer for the first query is 2. We delete the added edge.

Constraints:

2 <= n <= 30

m == queries.length

1 <= m <= 10

5

queries[i].length == 2

1 <= a

i

, b

i

<= 2

n

- 1

a

i

!= b

i

Code Snippets

C++:

```
class Solution {
public:
vector<int> cycleLengthQueries(int n, vector<vector<int>>& queries) {

}
};
```

Java:

```
class Solution {
public int[] cycleLengthQueries(int n, int[][] queries) {

}
}
```

Python3:

```
class Solution:
def cycleLengthQueries(self, n: int, queries: List[List[int]]) -> List[int]:
```

Python:

```
class Solution(object):
def cycleLengthQueries(self, n, queries):
"""
:type n: int
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} queries
* @return {number[]}
*/
var cycleLengthQueries = function(n, queries) {

};
```

TypeScript:

```
function cycleLengthQueries(n: number, queries: number[][]): number[] {

};
```

C#:

```
public class Solution {
public int[] CycleLengthQueries(int n, int[][] queries) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* cycleLengthQueries(int n, int** queries, int queriesSize, int*
queriesColSize, int* returnSize) {

}
```

Go:

```
func cycleLengthQueries(n int, queries [][]int) []int {

}
```

Kotlin:

```
class Solution {
fun cycleLengthQueries(n: Int, queries: Array<IntArray>): IntArray {

}

}
```

Swift:

```
class Solution {
func cycleLengthQueries(_ n: Int, _ queries: [[Int]]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn cycle_length_queries(n: i32, queries: Vec<Vec<i32>>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} queries
# @return {Integer[]}
def cycle_length_queries(n, queries)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $queries
* @return Integer[]
*/
function cycleLengthQueries($n, $queries) {

}
}
```

Dart:

```
class Solution {
List<int> cycleLengthQueries(int n, List<List<int>> queries) {

}
}
```

Scala:

```
object Solution {
def cycleLengthQueries(n: Int, queries: Array[Array[Int]]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec cycle_length_queries(n :: integer, queries :: [[integer]]) :: [integer]
def cycle_length_queries(n, queries) do

end
end
```

Erlang:

```
-spec cycle_length_queries(N :: integer(), Queries :: [[integer()]]) ->
[integer()].
cycle_length_queries(N, Queries) ->
.
```

Racket:

```
(define/contract (cycle-length-queries n queries)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Cycle Length Queries in 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:
vector<int> cycleLengthQueries(int n, vector<vector<int>>& queries) {

}
};
```

Java Solution:

```
/**
* Problem: Cycle Length Queries in 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[] cycleLengthQueries(int n, int[][] queries) {

}
}
```

Python3 Solution:

```
"""
Problem: Cycle Length Queries in 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 cycleLengthQueries(self, n: int, queries: List[List[int]]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def cycleLengthQueries(self, n, queries):
"""
:type n: int
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Cycle Length Queries in 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} n
* @param {number[][]} queries
* @return {number[]}
*/
var cycleLengthQueries = function(n, queries) {

};
```

TypeScript Solution:

```
/**
* Problem: Cycle Length Queries in 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 cycleLengthQueries(n: number, queries: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Cycle Length Queries in 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[] CycleLengthQueries(int n, int[][] queries) {

}
}
```

C Solution:

```
/*
* Problem: Cycle Length Queries in 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* cycleLengthQueries(int n, int** queries, int queriesSize, int*
queriesColSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Cycle Length Queries in 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 cycleLengthQueries(n int, queries [][]int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun cycleLengthQueries(n: Int, queries: Array<IntArray>): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func cycleLengthQueries(_ n: Int, _ queries: [[Int]]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Cycle Length Queries in 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 cycle_length_queries(n: i32, queries: Vec<Vec<i32>>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} queries
# @return {Integer[]}
def cycle_length_queries(n, queries)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $queries
* @return Integer[]
*/
function cycleLengthQueries($n, $queries) {

}
}
```

Dart Solution:

```
class Solution {
List<int> cycleLengthQueries(int n, List<List<int>> queries) {

}
}
```

Scala Solution:

```
object Solution {
def cycleLengthQueries(n: Int, queries: Array[Array[Int]]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec cycle_length_queries(n :: integer, queries :: [[integer]]) :: [integer]
def cycle_length_queries(n, queries) do

end
end
```

Erlang Solution:

```
-spec cycle_length_queries(N :: integer(), Queries :: [[integer()]]) ->
[integer()].
cycle_length_queries(N, Queries) ->
.
```

Racket Solution:

```
(define/contract (cycle-length-queries n queries)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?))
)
```