Problem 3108: Minimum Cost Walk in Weighted Graph

Problem Information

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

Problem Description

There is an undirected weighted graph with

n

vertices labeled from

0

to

n - 1

.

You are given the integer

n

and an array

edges

, where

edges[i] = [u

i

, v

i

, w

i

]

indicates that there is an edge between vertices

u

i

and

v

i

with a weight of

w

i

.

A walk on a graph is a sequence of vertices and edges. The walk starts and ends with a vertex, and each edge connects the vertex that comes before it and the vertex that comes after it. It's important to note that a walk may visit the same edge or vertex more than once.

The

cost

of a walk starting at node

u

and ending at node

v

is defined as the bitwise

AND

of the weights of the edges traversed during the walk. In other words, if the sequence of edge weights encountered during the walk is

w

0

, w

1

, w

2

, ..., w

k

, then the cost is calculated as

w

0

& w

1

& w

2

& ... & w

k

, where

&

denotes the bitwise

AND

operator.

You are also given a 2D array

query

, where

query[i] = [s

i

, t

i

]

. For each query, you need to find the minimum cost of the walk starting at vertex s

i

and ending at vertex

t

i

. If there exists no such walk, the answer is

-1

.

Return

the array

answer

, where

answer[i]

denotes the

minimum

cost of a walk for query

i

.

Example 1:

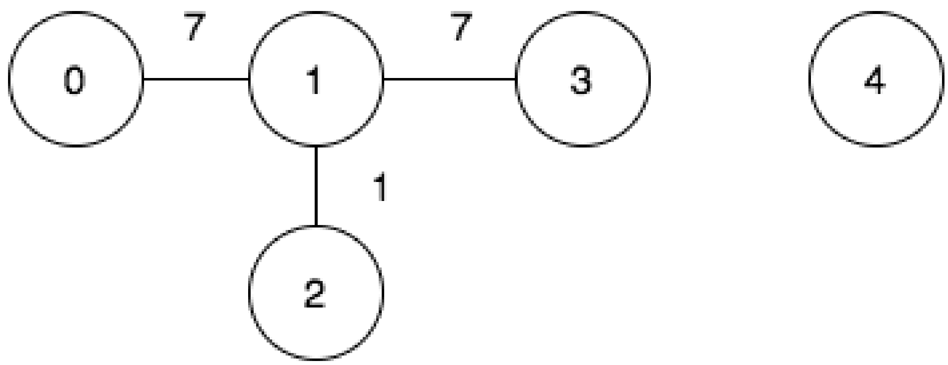
Input:

n = 5, edges = [[0,1,7],[1,3,7],[1,2,1]], query = [[0,3],[3,4]]

Output:

[1,-1]

Explanation:



To achieve the cost of 1 in the first query, we need to move on the following edges:

0->1

(weight 7),

1->2

(weight 1),

2->1

(weight 1),

1->3

(weight 7).

In the second query, there is no walk between nodes 3 and 4, so the answer is -1.

Example 2:

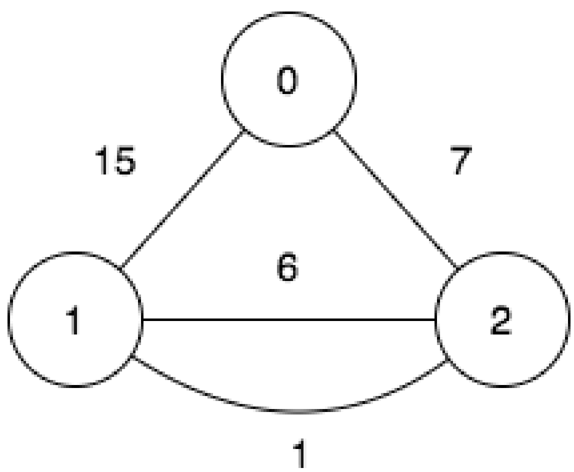
Input:

n = 3, edges = [[0,2,7],[0,1,15],[1,2,6],[1,2,1]], query = [[1,2]]

Output:

[0]

Explanation:



To achieve the cost of 0 in the first query, we need to move on the following edges:

1->2

(weight 1),

2->1

(weight 6),

1->2

(weight 1).

Constraints:

2 <= n <= 10

5

0 <= edges.length <= 10

5

edges[i].length == 3

0 <= u

i

, v

i

<= n - 1

u

i

!= v

i

0 <= w

i

<= 10

5

1 <= query.length <= 10

5

query[i].length == 2

0 <= s

i

, t

i

<= n - 1

s

i

!= t

i

Code Snippets

C++:

```
class Solution {
public:
vector<int> minimumCost(int n, vector<vector<int>>& edges,
vector<vector<int>>& query) {

}
};
```

Java:

```
class Solution {
public int[] minimumCost(int n, int[][] edges, int[][] query) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function minimumCost(n: number, edges: number[][], query: number[][]):
number[] {

};
```

C#:

```
public class Solution {
public int[] MinimumCost(int n, int[][] edges, int[][] query) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minimumCost(int n, int** edges, int edgesSize, int* edgesColSize, int**
query, int querySize, int* queryColSize, int* returnSize) {

}
```

Go:

```
func minimumCost(n int, edges [][]int, query [][]int) []int {

}
```

Kotlin:

```
class Solution {
fun minimumCost(n: Int, edges: Array<IntArray>, query: Array<IntArray>):
IntArray {

}
}
```

Swift:

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

}
}
```

Rust:

```
impl Solution {
pub fn minimum_cost(n: i32, edges: Vec<Vec<i32>>, query: Vec<Vec<i32>>) ->
Vec<i32> {

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
List<int> minimumCost(int n, List<List<int>> edges, List<List<int>> query) {

}
}
```

Scala:

```
object Solution {
def minimumCost(n: Int, edges: Array[Array[Int]], query: Array[Array[Int]]):
Array[Int] = {

}
}
```

Elixir:

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

end
end
```

Erlang:

```
-spec minimum_cost(N :: integer(), Edges :: [[integer()]], Query ::
[[integer()]]) -> [integer()].
minimum_cost(N, Edges, Query) ->
.
```

Racket:

```
(define/contract (minimum-cost n edges query)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<int> minimumCost(int n, vector<vector<int>>& edges,
vector<vector<int>>& query) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int[] minimumCost(int n, int[][] edges, int[][] query) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost Walk in Weighted Graph
Difficulty: Hard
Tags: array, graph

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def minimumCost(self, n: int, edges: List[List[int]], query: List[List[int]])
-> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} n
* @param {number[][]} edges
* @param {number[][]} query
* @return {number[]}
*/
var minimumCost = function(n, edges, query) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
function minimumCost(n: number, edges: number[][], query: number[][]):
number[] {

};
```

C# Solution:

```
/*
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[] MinimumCost(int n, int[][] edges, int[][] query) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost Walk in Weighted Graph
* Difficulty: Hard
* Tags: array, graph
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minimumCost(int n, int** edges, int edgesSize, int* edgesColSize, int**
query, int querySize, int* queryColSize, int* returnSize) {
```

```
}
```

Go Solution:

```
// Problem: Minimum Cost Walk in Weighted Graph
// Difficulty: Hard
// Tags: array, graph
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func minimumCost(n int, edges [][]int, query [][]int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun minimumCost(n: Int, edges: Array<IntArray>, query: Array<IntArray>):
IntArray {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Minimum Cost Walk in Weighted Graph
// Difficulty: Hard
// Tags: array, graph
//
// Approach: Use two pointers or sliding window technique
```

```
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn minimum_cost(n: i32, edges: Vec<Vec<i32>>, query: Vec<Vec<i32>>) ->
Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
List<int> minimumCost(int n, List<List<int>> edges, List<List<int>> query) {

}
```

```
}
```

Scala Solution:

```
object Solution {
def minimumCost(n: Int, edges: Array[Array[Int]], query: Array[Array[Int]]):
Array[Int] = {

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

```
-spec minimum_cost(N :: integer(), Edges :: [[integer()]], Query ::
[[integer()]]) -> [integer()].
minimum_cost(N, Edges, Query) ->
.
```

Racket Solution:

```
(define/contract (minimum-cost n edges query)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```