Problem 2642: Design Graph With Shortest Path Calculator

Problem Information

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

Problem Description

There is a

directed weighted

graph that consists of

n

nodes numbered from

0

to

n - 1

. The edges of the graph are initially represented by the given array

edges

where

edges[i] = [from

i

, to

i

, edgeCost

i

]

meaning that there is an edge from

from

i

to

to

i

with the cost

edgeCost

i

.

Implement the

Graph

class:

Graph(int n, int[][] edges)

initializes the object with

n

nodes and the given edges.

addEdge(int[] edge)

adds an edge to the list of edges where

edge = [from, to, edgeCost]

. It is guaranteed that there is no edge between the two nodes before adding this one.

int shortestPath(int node1, int node2)

returns the

minimum

cost of a path from

node1

to

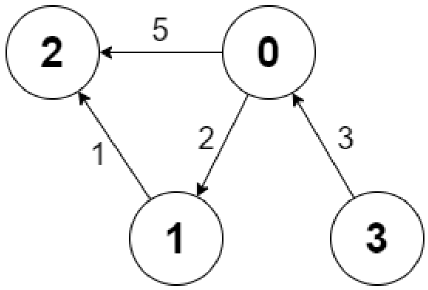
node2

. If no path exists, return

-1

. The cost of a path is the sum of the costs of the edges in the path.

Example 1:



Input

["Graph", "shortestPath", "shortestPath", "addEdge", "shortestPath"] [[4, [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]]], [3, 2], [0, 3], [[1, 3, 4]], [0, 3]]

Output

[null, 6, -1, null, 6]

Explanation

Graph g = new Graph(4, [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]]); g.shortestPath(3, 2); // return 6. The shortest path from 3 to 2 in the first diagram above is 3 -> 0 -> 1 -> 2 with a total cost of 3 + 2 + 1 = 6. g.shortestPath(0, 3); // return -1. There is no path from 0 to 3. g.addEdge([1, 3, 4]); // We add an edge from node 1 to node 3, and we get the second diagram above. g.shortestPath(0, 3); // return 6. The shortest path from 0 to 3 now is 0 -> 1 -> 3 with a total cost of 2 + 4 = 6.

Constraints:

1 <= n <= 100

0 <= edges.length <= n * (n - 1)

edges[i].length == edge.length == 3

0 <= from

i

, to

i

, from, to, node1, node2 <= n - 1

1 <= edgeCost

i

, edgeCost <= 10

6

There are no repeated edges and no self-loops in the graph at any point.

At most

100

calls will be made for

addEdge

.

At most

100

calls will be made for

shortestPath

.

Code Snippets

C++:

```
class Graph {
public:
Graph(int n, vector<vector<int>>& edges) {

}

void addEdge(vector<int> edge) {

}

int shortestPath(int node1, int node2) {

}
};

/**
* Your Graph object will be instantiated and called as such:
* Graph* obj = new Graph(n, edges);
* obj->addEdge(edge);
* int param_2 = obj->shortestPath(node1,node2);
*/
```

Java:

```
class Graph {

public Graph(int n, int[][] edges) {

}

public void addEdge(int[] edge) {

}

public int shortestPath(int node1, int node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = new Graph(n, edges);
* obj.addEdge(edge);
```

```
* int param_2 = obj.shortestPath(node1,node2);
*/
```

Python3:

```
class Graph:

def __init__(self, n: int, edges: List[List[int]]):



def addEdge(self, edge: List[int]) -> None:



def shortestPath(self, node1: int, node2: int) -> int:




# Your Graph object will be instantiated and called as such:
# obj = Graph(n, edges)
# obj.addEdge(edge)
# param_2 = obj.shortestPath(node1,node2)
```

Python:

```
class Graph(object):

def __init__(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
"""



def addEdge(self, edge):
"""
:type edge: List[int]
:rtype: None
"""



def shortestPath(self, node1, node2):
"""
```

```
:type node1: int
:type node2: int
:rtype: int
"""



# Your Graph object will be instantiated and called as such:
# obj = Graph(n, edges)
# obj.addEdge(edge)
# param_2 = obj.shortestPath(node1,node2)
```

JavaScript:

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

};

/**
* @param {number[]} edge
* @return {void}
*/
Graph.prototype.addEdge = function(edge) {

};

/**
* @param {number} node1
* @param {number} node2
* @return {number}
*/
Graph.prototype.shortestPath = function(node1, node2) {

};

/**
* Your Graph object will be instantiated and called as such:
* var obj = new Graph(n, edges)
```

```
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

TypeScript:

```
class Graph {
constructor(n: number, edges: number[][]) {

}

addEdge(edge: number[]): void {

}

shortestPath(node1: number, node2: number): number {

}
}

/**
* Your Graph object will be instantiated and called as such:
* var obj = new Graph(n, edges)
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

C#:

```
public class Graph {

public Graph(int n, int[][] edges) {

}

public void AddEdge(int[] edge) {

}

public int ShortestPath(int node1, int node2) {

}
```

```
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = new Graph(n, edges);
* obj.AddEdge(edge);
* int param_2 = obj.ShortestPath(node1,node2);
*/
```

C:

```
typedef struct {

} Graph;


Graph* graphCreate(int n, int** edges, int edgesSize, int* edgesColSize) {

}

void graphAddEdge(Graph* obj, int* edge, int edgeSize) {

}

int graphShortestPath(Graph* obj, int node1, int node2) {

}

void graphFree(Graph* obj) {

}

/**
* Your Graph struct will be instantiated and called as such:
* Graph* obj = graphCreate(n, edges, edgesSize, edgesColSize);
* graphAddEdge(obj, edge, edgeSize);

* int param_2 = graphShortestPath(obj, node1, node2);
```

```
* graphFree(obj);
*/
```

Go:

```
type Graph struct {

}


func Constructor(n int, edges [][]int) Graph {

}


func (this *Graph) AddEdge(edge []int) {

}


func (this *Graph) ShortestPath(node1 int, node2 int) int {

}


/**
* Your Graph object will be instantiated and called as such:
* obj := Constructor(n, edges);
* obj.AddEdge(edge);
* param_2 := obj.ShortestPath(node1,node2);
*/
```

Kotlin:

```
class Graph(n: Int, edges: Array<IntArray>) {

fun addEdge(edge: IntArray) {

}

fun shortestPath(node1: Int, node2: Int): Int {
```

```
}

}

/**
* Your Graph object will be instantiated and called as such:
* var obj = Graph(n, edges)
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

Swift:

```
class Graph {

init(_ n: Int, _ edges: [[Int]]) {

}

func addEdge(_ edge: [Int]) {

}

func shortestPath(_ node1: Int, _ node2: Int) -> Int {

}
}

/**
* Your Graph object will be instantiated and called as such:
* let obj = Graph(n, edges)
* obj.addEdge(edge)
* let ret_2: Int = obj.shortestPath(node1, node2)
*/
```

Rust:

```
struct Graph {

}
```

```
/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Graph {

fn new(n: i32, edges: Vec<Vec<i32>>) -> Self {

}

fn add_edge(&self, edge: Vec<i32>) {

}

fn shortest_path(&self, node1: i32, node2: i32) -> i32 {

}
}

/**
* Your Graph object will be instantiated and called as such:
* let obj = Graph::new(n, edges);
* obj.add_edge(edge);
* let ret_2: i32 = obj.shortest_path(node1, node2);
*/
```

Ruby:

```
class Graph

=begin
:type n: Integer
:type edges: Integer[][]
=end
def initialize(n, edges)

end



=begin
:type edge: Integer[]
```

```
:rtype: Void
=end
def add_edge(edge)

end



=begin
:type node1: Integer
:type node2: Integer
:rtype: Integer
=end
def shortest_path(node1, node2)

end



end

# Your Graph object will be instantiated and called as such:
# obj = Graph.new(n, edges)
# obj.add_edge(edge)
# param_2 = obj.shortest_path(node1, node2)
```

PHP:

```
class Graph {
/**
* @param Integer $n
* @param Integer[][] $edges
*/
function __construct($n, $edges) {

}

/**
* @param Integer[] $edge
* @return NULL
*/
function addEdge($edge) {

}
```

```
/**
* @param Integer $node1
* @param Integer $node2
* @return Integer
*/
function shortestPath($node1, $node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* $obj = Graph($n, $edges);
* $obj->addEdge($edge);
* $ret_2 = $obj->shortestPath($node1, $node2);
*/
```

Dart:

```
class Graph {

Graph(int n, List<List<int>> edges) {

}

void addEdge(List<int> edge) {

}

int shortestPath(int node1, int node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = Graph(n, edges);
* obj.addEdge(edge);
* int param2 = obj.shortestPath(node1,node2);
*/
```

Scala:

```
class Graph(_n: Int, _edges: Array[Array[Int]]) {

def addEdge(edge: Array[Int]): Unit = {

}

def shortestPath(node1: Int, node2: Int): Int = {

}

}

/**
* Your Graph object will be instantiated and called as such:
* val obj = new Graph(n, edges)
* obj.addEdge(edge)
* val param_2 = obj.shortestPath(node1,node2)
*/
```

Elixir:

```
defmodule Graph do
@spec init_(n :: integer, edges :: [[integer]]) :: any
def init_(n, edges) do

end

@spec add_edge(edge :: [integer]) :: any
def add_edge(edge) do

end

@spec shortest_path(node1 :: integer, node2 :: integer) :: integer
def shortest_path(node1, node2) do

end
end

# Your functions will be called as such:
# Graph.init_(n, edges)
```

```
# Graph.add_edge(edge)
# param_2 = Graph.shortest_path(node1, node2)

# Graph.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang:

```
-spec graph_init_(N :: integer(), Edges :: [[integer()]]) -> any().
graph_init_(N, Edges) ->
.

-spec graph_add_edge(Edge :: [integer()]) -> any().
graph_add_edge(Edge) ->
.

-spec graph_shortest_path(Node1 :: integer(), Node2 :: integer()) ->
integer().
graph_shortest_path(Node1, Node2) ->
.



%% Your functions will be called as such:
%% graph_init_(N, Edges),
%% graph_add_edge(Edge),
%% Param_2 = graph_shortest_path(Node1, Node2),

%% graph_init_ will be called before every test case, in which you can do
some necessary initializations.
```

Racket:

```
(define graph%
(class object%
(super-new)

; n : exact-integer?
; edges : (listof (listof exact-integer?))
(init-field
n
edges)
```

```
; add-edge : (listof exact-integer?) -> void?
(define/public (add-edge edge)
)
; shortest-path : exact-integer? exact-integer? -> exact-integer?
(define/public (shortest-path node1 node2)
)))

;; Your graph% object will be instantiated and called as such:
;; (define obj (new graph% [n n] [edges edges]))
;; (send obj add-edge edge)
;; (define param_2 (send obj shortest-path node1 node2))
```

Solutions

C++ Solution:

```
/*
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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 Graph {
public:
Graph(int n, vector<vector<int>>& edges) {

}

void addEdge(vector<int> edge) {

}

int shortestPath(int node1, int node2) {

}
};
```

```
/**
* Your Graph object will be instantiated and called as such:
* Graph* obj = new Graph(n, edges);
* obj->addEdge(edge);
* int param_2 = obj->shortestPath(node1,node2);
*/
```

Java Solution:

```
/**
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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 Graph {

public Graph(int n, int[][] edges) {

}

public void addEdge(int[] edge) {

}

public int shortestPath(int node1, int node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = new Graph(n, edges);
* obj.addEdge(edge);
* int param_2 = obj.shortestPath(node1,node2);
*/
```

Python3 Solution:

```
"""
Problem: Design Graph With Shortest Path Calculator
Difficulty: Hard
Tags: array, graph, queue, heap

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 Graph:

def __init__(self, n: int, edges: List[List[int]]):



def addEdge(self, edge: List[int]) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Graph(object):

def __init__(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
"""



def addEdge(self, edge):
"""
:type edge: List[int]
:rtype: None
"""



def shortestPath(self, node1, node2):
"""
:type node1: int
```

```
:type node2: int
:rtype: int
"""



# Your Graph object will be instantiated and called as such:
# obj = Graph(n, edges)
# obj.addEdge(edge)
# param_2 = obj.shortestPath(node1,node2)
```

JavaScript Solution:

```
/**
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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
*/
var Graph = function(n, edges) {

};

/**
* @param {number[]} edge
* @return {void}
*/
Graph.prototype.addEdge = function(edge) {

};

/**
* @param {number} node1
```

```
* @param {number} node2
* @return {number}
*/
Graph.prototype.shortestPath = function(node1, node2) {

};

/**
* Your Graph object will be instantiated and called as such:
* var obj = new Graph(n, edges)
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

TypeScript Solution:

```
/**
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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 Graph {
constructor(n: number, edges: number[][]) {

}

addEdge(edge: number[]): void {

}

shortestPath(node1: number, node2: number): number {

}
}

/**
```

```
* Your Graph object will be instantiated and called as such:
* var obj = new Graph(n, edges)
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

C# Solution:

```
/*
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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 Graph {

public Graph(int n, int[][] edges) {

}

public void AddEdge(int[] edge) {

}

public int ShortestPath(int node1, int node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = new Graph(n, edges);
* obj.AddEdge(edge);
* int param_2 = obj.ShortestPath(node1,node2);
*/
```

C Solution:

```c
/*
* Problem: Design Graph With Shortest Path Calculator
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* 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
*/




typedef struct {

} Graph;


Graph* graphCreate(int n, int** edges, int edgesSize, int* edgesColSize) {

}

void graphAddEdge(Graph* obj, int* edge, int edgeSize) {

}

int graphShortestPath(Graph* obj, int node1, int node2) {

}

void graphFree(Graph* obj) {

}

/**
* Your Graph struct will be instantiated and called as such:
* Graph* obj = graphCreate(n, edges, edgesSize, edgesColSize);
* graphAddEdge(obj, edge, edgeSize);

* int param_2 = graphShortestPath(obj, node1, node2);

* graphFree(obj);
*/
```

Go Solution:

```go
// Problem: Design Graph With Shortest Path Calculator
// Difficulty: Hard
// Tags: array, graph, queue, heap
//
// 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

type Graph struct {

}


func Constructor(n int, edges [][]int) Graph {

}


func (this *Graph) AddEdge(edge []int) {

}


func (this *Graph) ShortestPath(node1 int, node2 int) int {

}


/**
* Your Graph object will be instantiated and called as such:
* obj := Constructor(n, edges);
* obj.AddEdge(edge);
* param_2 := obj.ShortestPath(node1,node2);
*/
```

Kotlin Solution:

```kotlin
class Graph(n: Int, edges: Array<IntArray>) {

fun addEdge(edge: IntArray) {
```

```
}

fun shortestPath(node1: Int, node2: Int): Int {

}

}

/**
* Your Graph object will be instantiated and called as such:
* var obj = Graph(n, edges)
* obj.addEdge(edge)
* var param_2 = obj.shortestPath(node1,node2)
*/
```

Swift Solution:

```
class Graph {

init(_ n: Int, _ edges: [[Int]]) {

}

func addEdge(_ edge: [Int]) {

}

func shortestPath(_ node1: Int, _ node2: Int) -> Int {

}
}

/**
* Your Graph object will be instantiated and called as such:
* let obj = Graph(n, edges)
* obj.addEdge(edge)
* let ret_2: Int = obj.shortestPath(node1, node2)
*/
```

Rust Solution:

```
// Problem: Design Graph With Shortest Path Calculator
// Difficulty: Hard
// Tags: array, graph, queue, heap
//
// 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

struct Graph {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Graph {

fn new(n: i32, edges: Vec<Vec<i32>>) -> Self {

}

fn add_edge(&self, edge: Vec<i32>) {

}

fn shortest_path(&self, node1: i32, node2: i32) -> i32 {

}
}

/**
* Your Graph object will be instantiated and called as such:
* let obj = Graph::new(n, edges);
* obj.add_edge(edge);
* let ret_2: i32 = obj.shortest_path(node1, node2);
*/
```

Ruby Solution:

```
class Graph

=begin
:type n: Integer
:type edges: Integer[][]
=end
def initialize(n, edges)

end



=begin
:type edge: Integer[]
:rtype: Void
=end
def add_edge(edge)

end



=begin
:type node1: Integer
:type node2: Integer
:rtype: Integer
=end
def shortest_path(node1, node2)

end



end

# Your Graph object will be instantiated and called as such:
# obj = Graph.new(n, edges)
# obj.add_edge(edge)
# param_2 = obj.shortest_path(node1, node2)
```

PHP Solution:

```
class Graph {
/**
* @param Integer $n
```

```
* @param Integer[][] $edges
*/
function __construct($n, $edges) {

}

/**
* @param Integer[] $edge
* @return NULL
*/
function addEdge($edge) {

}

/**
* @param Integer $node1
* @param Integer $node2
* @return Integer
*/
function shortestPath($node1, $node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* $obj = Graph($n, $edges);
* $obj->addEdge($edge);
* $ret_2 = $obj->shortestPath($node1, $node2);
*/
```

Dart Solution:

```
class Graph {

Graph(int n, List<List<int>> edges) {

}

void addEdge(List<int> edge) {
```

```
}

int shortestPath(int node1, int node2) {

}
}

/**
* Your Graph object will be instantiated and called as such:
* Graph obj = Graph(n, edges);
* obj.addEdge(edge);
* int param2 = obj.shortestPath(node1,node2);
*/
```

Scala Solution:

```
class Graph(_n: Int, _edges: Array[Array[Int]]) {

def addEdge(edge: Array[Int]): Unit = {

}

def shortestPath(node1: Int, node2: Int): Int = {

}

}

/**
* Your Graph object will be instantiated and called as such:
* val obj = new Graph(n, edges)
* obj.addEdge(edge)
* val param_2 = obj.shortestPath(node1,node2)
*/
```

Elixir Solution:

```
defmodule Graph do
@spec init_(n :: integer, edges :: [[integer]]) :: any
def init_(n, edges) do
```

```
end

@spec add_edge(edge :: [integer]) :: any
def add_edge(edge) do

end

@spec shortest_path(node1 :: integer, node2 :: integer) :: integer
def shortest_path(node1, node2) do

end
end

# Your functions will be called as such:
# Graph.init_(n, edges)
# Graph.add_edge(edge)
# param_2 = Graph.shortest_path(node1, node2)

# Graph.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang Solution:

```
-spec graph_init_(N :: integer(), Edges :: [[integer()]]) -> any().
graph_init_(N, Edges) ->
.

-spec graph_add_edge(Edge :: [integer()]) -> any().
graph_add_edge(Edge) ->
.

-spec graph_shortest_path(Node1 :: integer(), Node2 :: integer()) ->
integer().
graph_shortest_path(Node1, Node2) ->
.



%% Your functions will be called as such:
%% graph_init_(N, Edges),
%% graph_add_edge(Edge),
%% Param_2 = graph_shortest_path(Node1, Node2),
```

```
%% graph_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket Solution:

```
(define graph%
(class object%
(super-new)

; n : exact-integer?
; edges : (listof (listof exact-integer?))
(init-field
n
edges)

; add-edge : (listof exact-integer?) -> void?
(define/public (add-edge edge)
)
; shortest-path : exact-integer? exact-integer? -> exact-integer?
(define/public (shortest-path node1 node2)
)))

;; Your graph% object will be instantiated and called as such:
;; (define obj (new graph% [n n] [edges edges]))
;; (send obj add-edge edge)
;; (define param_2 (send obj shortest-path node1 node2))
```