Problem 2050: Parallel Courses III

Problem Information

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

Problem Description

You are given an integer

n

, which indicates that there are

n

courses labeled from

1

to

n

. You are also given a 2D integer array

relations

where

relations[j] = [prevCourse

j

, nextCourse

j

]

denotes that course

prevCourse

j

has to be completed

before

course

nextCourse

j

(prerequisite relationship). Furthermore, you are given a

0-indexed

integer array

time

where

time[i]

denotes how many

months

it takes to complete the

(i+1)

th

course.

You must find the

minimum

number of months needed to complete all the courses following these rules:

You may start taking a course at

any time

if the prerequisites are met.

Any number of courses

can be taken at the

same time

.

Return

the

minimum

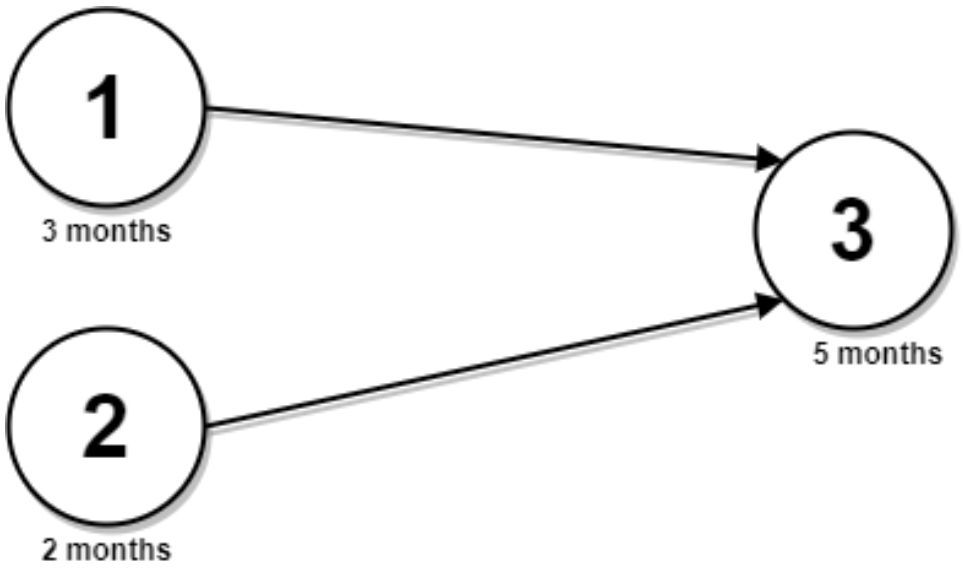
number of months needed to complete all the courses

.

Note:

The test cases are generated such that it is possible to complete every course (i.e., the graph is a directed acyclic graph).

Example 1:



Input:

n = 3, relations = [[1,3],[2,3]], time = [3,2,5]

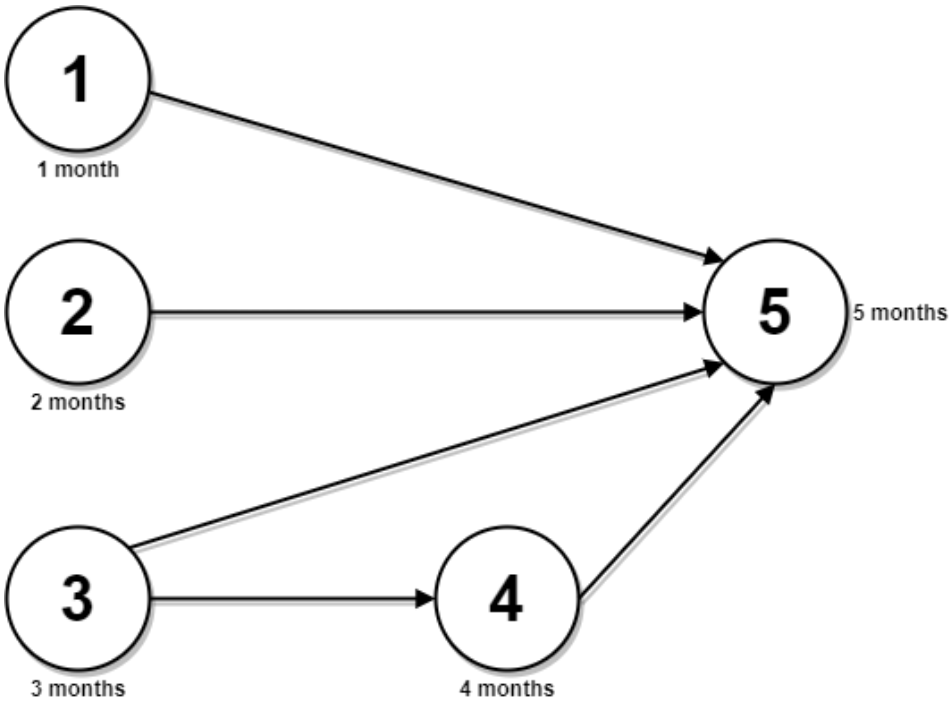
Output:

8

Explanation:

The figure above represents the given graph and the time required to complete each course. We start course 1 and course 2 simultaneously at month 0. Course 1 takes 3 months and course 2 takes 2 months to complete respectively. Thus, the earliest time we can start course 3 is at month 3, and the total time required is 3 + 5 = 8 months.

Example 2:



Input:

n = 5, relations = [[1,5],[2,5],[3,5],[3,4],[4,5]], time = [1,2,3,4,5]

Output:

12

Explanation:

The figure above represents the given graph and the time required to complete each course. You can start courses 1, 2, and 3 at month 0. You can complete them after 1, 2, and 3 months respectively. Course 4 can be taken only after course 3 is completed, i.e., after 3 months. It is completed after 3 + 4 = 7 months. Course 5 can be taken only after courses 1, 2, 3, and 4 have been completed, i.e., after max(1,2,3,7) = 7 months. Thus, the minimum time needed to complete all the courses is 7 + 5 = 12 months.

Constraints:

1 <= n <= 5 * 10

4

0 <= relations.length <= min(n * (n - 1) / 2, 5 * 10

4

)

relations[j].length == 2

1 <= prevCourse

j

, nextCourse

j

<= n

prevCourse

j

!= nextCourse

j

All the pairs

[prevCourse

j

, nextCourse

j

]

are

unique

.

time.length == n

1 <= time[i] <= 10

4

The given graph is a directed acyclic graph.

Code Snippets

C++:

```
class Solution {
public:
int minimumTime(int n, vector<vector<int>>& relations, vector<int>& time) {

}
};
```

Java:

```
class Solution {
public int minimumTime(int n, int[][] relations, int[] time) {

}
}
```

Python3:

```
class Solution:
def minimumTime(self, n: int, relations: List[List[int]], time: List[int]) ->
int:
```

Python:

```
class Solution(object):
def minimumTime(self, n, relations, time):
"""
:type n: int
:type relations: List[List[int]]
:type time: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} relations
* @param {number[]} time
* @return {number}
*/
var minimumTime = function(n, relations, time) {

};
```

TypeScript:

```
function minimumTime(n: number, relations: number[][], time: number[]):
number {

};
```

C#:

```
public class Solution {
public int MinimumTime(int n, int[][] relations, int[] time) {

}
}
```

C:

```
int minimumTime(int n, int** relations, int relationsSize, int*
relationsColSize, int* time, int timeSize) {

}
```

Go:

```
func minimumTime(n int, relations [][]int, time []int) int {

}
```

Kotlin:

```
class Solution {
fun minimumTime(n: Int, relations: Array<IntArray>, time: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func minimumTime(_ n: Int, _ relations: [[Int]], _ time: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_time(n: i32, relations: Vec<Vec<i32>>, time: Vec<i32>) -> i32
{

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} relations
# @param {Integer[]} time
# @return {Integer}
def minimum_time(n, relations, time)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $relations
* @param Integer[] $time
* @return Integer
*/
function minimumTime($n, $relations, $time) {

}
}
```

Dart:

```
class Solution {
int minimumTime(int n, List<List<int>> relations, List<int> time) {

}
}
```

Scala:

```
object Solution {
def minimumTime(n: Int, relations: Array[Array[Int]], time: Array[Int]): Int
= {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_time(n :: integer, relations :: [[integer]], time :: [integer])
:: integer
def minimum_time(n, relations, time) do

end
end
```

Erlang:

```
-spec minimum_time(N :: integer(), Relations :: [[integer()]], Time ::
[integer()]) -> integer().
minimum_time(N, Relations, Time) ->
.
```

Racket:

```
(define/contract (minimum-time n relations time)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Parallel Courses III
* Difficulty: Hard
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int minimumTime(int n, vector<vector<int>>& relations, vector<int>& time) {

}
};
```

Java Solution:

```
/**
* Problem: Parallel Courses III
* Difficulty: Hard
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
```

```
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int minimumTime(int n, int[][] relations, int[] time) {

}
}
```

Python3 Solution:

```
"""
Problem: Parallel Courses III
Difficulty: Hard
Tags: array, graph, dp, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def minimumTime(self, n: int, relations: List[List[int]], time: List[int]) ->
int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumTime(self, n, relations, time):
"""
:type n: int
:type relations: List[List[int]]
:type time: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Parallel Courses III
* Difficulty: Hard
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @param {number[][]} relations
* @param {number[]} time
* @return {number}
*/
var minimumTime = function(n, relations, time) {

};
```

TypeScript Solution:

```
/**
* Problem: Parallel Courses III
* Difficulty: Hard
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function minimumTime(n: number, relations: number[][], time: number[]):
number {

};
```

C# Solution:

```
/*
* Problem: Parallel Courses III
* Difficulty: Hard
```

```
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int MinimumTime(int n, int[][] relations, int[] time) {

}
}
```

C Solution:

```
/*
* Problem: Parallel Courses III
* Difficulty: Hard
* Tags: array, graph, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int minimumTime(int n, int** relations, int relationsSize, int*
relationsColSize, int* time, int timeSize) {

}
```

Go Solution:

```
// Problem: Parallel Courses III
// Difficulty: Hard
// Tags: array, graph, dp, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func minimumTime(n int, relations [][]int, time []int) int {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun minimumTime(n: Int, relations: Array<IntArray>, time: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func minimumTime(_ n: Int, _ relations: [[Int]], _ time: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Parallel Courses III
// Difficulty: Hard
// Tags: array, graph, dp, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn minimum_time(n: i32, relations: Vec<Vec<i32>>, time: Vec<i32>) -> i32
{

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} relations
# @param {Integer[]} time
```

```
# @return {Integer}
def minimum_time(n, relations, time)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $relations
* @param Integer[] $time
* @return Integer
*/
function minimumTime($n, $relations, $time) {

}
}
```

Dart Solution:

```
class Solution {
int minimumTime(int n, List<List<int>> relations, List<int> time) {

}
}
```

Scala Solution:

```
object Solution {
def minimumTime(n: Int, relations: Array[Array[Int]], time: Array[Int]): Int
= {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_time(n :: integer, relations :: [[integer]], time :: [integer])
```

```
:: integer
def minimum_time(n, relations, time) do

end
end
```

Erlang Solution:

```
-spec minimum_time(N :: integer(), Relations :: [[integer()]], Time ::
[integer()]) -> integer().
minimum_time(N, Relations, Time) ->
.
```

Racket Solution:

```
(define/contract (minimum-time n relations time)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer?)
)
```