Problem 2532: Time to Cross a Bridge

Problem Information

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

Problem Description

There are

k

workers who want to move

n

boxes from the right (old) warehouse to the left (new) warehouse. You are given the two integers

n

and

k

, and a 2D integer array

time

of size

k x 4

where

time[i] = [right

i

, pick

i

, left

i

, put

i

]

.

The warehouses are separated by a river and connected by a bridge. Initially, all

k

workers are waiting on the left side of the bridge. To move the boxes, the

i

th

worker can do the following:

Cross the bridge to the right side in

right

i

minutes.

Pick a box from the right warehouse in

pick

i

minutes.

Cross the bridge to the left side in

left

i

minutes.

Put the box into the left warehouse in

put

i

minutes.

The

i

th

worker is

less efficient

than the j

th

worker if either condition is met:

left

i

+ right

i

> left

j

+ right

j

left

i

+ right

i

== left

j

+ right

j

and

i > j

The following rules regulate the movement of the workers through the bridge:

Only one worker can use the bridge at a time.

When the bridge is unused prioritize the

least efficient

worker (who have picked up the box) on the right side to cross. If not, prioritize the

least efficient

worker on the left side to cross.

If enough workers have already been dispatched from the left side to pick up all the remaining boxes,

no more

workers will be sent from the left side.

Return the

elapsed minutes

at which the last box reaches the

left side of the bridge

.

Example 1:

Input:

n = 1, k = 3, time = [[1,1,2,1],[1,1,3,1],[1,1,4,1]]

Output:

6

Explanation:

From 0 to 1 minutes: worker 2 crosses the bridge to the right. From 1 to 2 minutes: worker 2 picks up a box from the right warehouse. From 2 to 6 minutes: worker 2 crosses the bridge to the left. From 6 to 7 minutes: worker 2 puts a box at the left warehouse. The whole process ends after 7 minutes. We return 6 because the problem asks for the instance of time at which the last worker reaches the left side of the bridge.

Example 2:

Input:

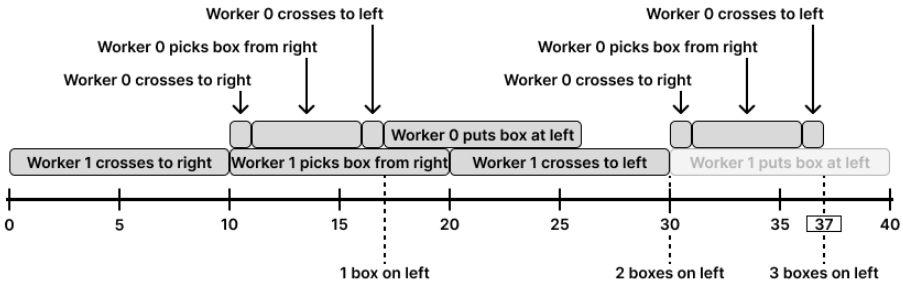
n = 3, k = 2, time =

[[1,5,1,8],[10,10,10,10]]

Output:

37

Explanation:



The last box reaches the left side at 37 seconds. Notice, how we

do not

put the last boxes down, as that would take more time, and they are already on the left with the workers.

Constraints:

1 <= n, k <= 10

4

time.length == k

time[i].length == 4

1 <= left

i

, pick

i

, right

i

, put

i

<= 1000

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int findCrossingTime(int n, int k, int[][] time) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function findCrossingTime(n: number, k: number, time: number[][]): number {

};
```

C#:

```
public class Solution {
public int FindCrossingTime(int n, int k, int[][] time) {

}
}
```

C:

```
int findCrossingTime(int n, int k, int** time, int timeSize, int*
timeColSize) {

}
```

Go:

```
func findCrossingTime(n int, k int, time [][]int) int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
```

```
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
int findCrossingTime(int n, int k, List<List<int>> time) {

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

```
-spec find_crossing_time(N :: integer(), K :: integer(), Time ::
[[integer()]]) -> integer().
find_crossing_time(N, K, Time) ->
.
```

Racket:

```
(define/contract (find-crossing-time n k time)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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 Solution {
public:
int findCrossingTime(int n, int k, vector<vector<int>>& time) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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 Solution {
public int findCrossingTime(int n, int k, int[][] time) {

}
}
```

Python3 Solution:

```
"""
Problem: Time to Cross a Bridge
Difficulty: Hard
Tags: array, 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 Solution:
def findCrossingTime(self, n: int, k: int, time: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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} k
* @param {number[][]} time
* @return {number}
*/
var findCrossingTime = function(n, k, time) {

};
```

TypeScript Solution:

```
/**
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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
*/
```

```
function findCrossingTime(n: number, k: number, time: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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 Solution {
public int FindCrossingTime(int n, int k, int[][] time) {

}
}
```

C Solution:

```
/*
* Problem: Time to Cross a Bridge
* Difficulty: Hard
* Tags: array, 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
*/

int findCrossingTime(int n, int k, int** time, int timeSize, int*
timeColSize) {

}
```

Go Solution:

```
// Problem: Time to Cross a Bridge
// Difficulty: Hard
// Tags: array, 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

func findCrossingTime(n int, k int, time [][]int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Time to Cross a Bridge
// Difficulty: Hard
// Tags: array, 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

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

}
```

```
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
int findCrossingTime(int n, int k, List<List<int>> time) {

}
}
```

Scala Solution:

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

}
```

```
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

```
-spec find_crossing_time(N :: integer(), K :: integer(), Time ::
[[integer()]]) -> integer().
find_crossing_time(N, K, Time) ->
.
```

Racket Solution:

```
(define/contract (find-crossing-time n k time)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```