Problem 3594: Minimum Time to Transport All Individuals

Problem Information

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

Problem Description

You are given

n

individuals at a base camp who need to cross a river to reach a destination using a single boat. The boat can carry at most

k

people at a time. The trip is affected by environmental conditions that vary

cyclically

over

m

stages.

Each stage

j

has a speed multiplier

mul[j]

:

If

mul[j] > 1

, the trip slows down.

If

mul[j] < 1

, the trip speeds up.

Each individual

i

has a rowing strength represented by

time[i]

, the time (in minutes) it takes them to cross alone in neutral conditions.

Rules:

A group

g

departing at stage

j

takes time equal to the

maximum

time[i]

among its members, multiplied by

mul[j]

minutes to reach the destination.

After the group crosses the river in time

d

, the stage advances by

floor(d) % m

steps.

If individuals are left behind, one person must return with the boat. Let

r

be the index of the returning person, the return takes

time[r] × mul[current_stage]

, defined as

return_time

, and the stage advances by

floor(return_time) % m

.

Return the

minimum

total time required to transport all individuals. If it is not possible to transport all individuals to the destination, return

-1

.

Example 1:

Input:

n = 1, k = 1, m = 2, time = [5], mul = [1.0,1.3]

Output:

5.00000

Explanation:

Individual 0 departs from stage 0, so crossing time =

5 × 1.00 = 5.00

minutes.

All team members are now at the destination. Thus, the total time taken is

5.00

minutes.

Example 2:

Input:

n = 3, k = 2, m = 3, time = [2,5,8], mul = [1.0,1.5,0.75]

Output:

14.50000

Explanation:

The optimal strategy is:

Send individuals 0 and 2 from the base camp to the destination from stage 0. The crossing time is

max(2, 8) × mul[0] = 8 × 1.00 = 8.00

minutes. The stage advances by

floor(8.00) % 3 = 2

, so the next stage is

(0 + 2) % 3 = 2

.

Individual 0 returns alone from the destination to the base camp from stage 2. The return time is

2 × mul[2] = 2 × 0.75 = 1.50

minutes. The stage advances by

floor(1.50) % 3 = 1

, so the next stage is

(2 + 1) % 3 = 0

.

Send individuals 0 and 1 from the base camp to the destination from stage 0. The crossing time is

max(2, 5) × mul[0] = 5 × 1.00 = 5.00

minutes. The stage advances by

floor(5.00) % 3 = 2

, so the final stage is

(0 + 2) % 3 = 2

.

All team members are now at the destination. The total time taken is

8.00 + 1.50 + 5.00 = 14.50

minutes.

Example 3:

Input:

n = 2, k = 1, m = 2, time = [10,10], mul = [2.0,2.0]

Output:

-1.00000

Explanation:

Since the boat can only carry one person at a time, it is impossible to transport both individuals as one must always return. Thus, the answer is

-1.00

.

Constraints:

1 <= n == time.length <= 12

1 <= k <= 5

1 <= m <= 5

1 <= time[i] <= 100

m == mul.length

0.5 <= mul[i] <= 2.0

Code Snippets

C++:

```
class Solution {
public:
double minTime(int n, int k, int m, vector<int>& time, vector<double>& mul) {

}
};
```

Java:

```
class Solution {
public double minTime(int n, int k, int m, int[] time, double[] mul) {

}
}
```

Python3:

```
class Solution:
def minTime(self, n: int, k: int, m: int, time: List[int], mul: List[float])
-> float:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function minTime(n: number, k: number, m: number, time: number[], mul:
number[]): number {

};
```

C#:

```
public class Solution {
public double MinTime(int n, int k, int m, int[] time, double[] mul) {

}
}
```

C:

```
double minTime(int n, int k, int m, int* time, int timeSize, double* mul, int
mulSize) {

}
```

Go:

```
func minTime(n int, k int, m int, time []int, mul []float64) float64 {

}
```

Kotlin:

```
class Solution {
fun minTime(n: Int, k: Int, m: Int, time: IntArray, mul: DoubleArray): Double
{

}
}
```

Swift:

```
class Solution {
func minTime(_ n: Int, _ k: Int, _ m: Int, _ time: [Int], _ mul: [Double]) ->
Double {

}
}
```

Rust:

```
impl Solution {
pub fn min_time(n: i32, k: i32, m: i32, time: Vec<i32>, mul: Vec<f64>) -> f64
{

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer} k
# @param {Integer} m
```

```
# @param {Integer[]} time
# @param {Float[]} mul
# @return {Float}
def min_time(n, k, m, time, mul)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer $k
* @param Integer $m
* @param Integer[] $time
* @param Float[] $mul
* @return Float
*/
function minTime($n, $k, $m, $time, $mul) {

}
}
```

Dart:

```
class Solution {
double minTime(int n, int k, int m, List<int> time, List<double> mul) {

}
}
```

Scala:

```
object Solution {
def minTime(n: Int, k: Int, m: Int, time: Array[Int], mul: Array[Double]):
Double = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_time(n :: integer, k :: integer, m :: integer, time :: [integer],
mul :: [float]) :: float
def min_time(n, k, m, time, mul) do

end
end
```

Erlang:

```
-spec min_time(N :: integer(), K :: integer(), M :: integer(), Time ::
[integer()], Mul :: [float()]) -> float().
min_time(N, K, M, Time, Mul) ->
.
```

Racket:

```
(define/contract (min-time n k m time mul)
(-> exact-integer? exact-integer? exact-integer? (listof exact-integer?)
(listof flonum?) flonum?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
* 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:
double minTime(int n, int k, int m, vector<int>& time, vector<double>& mul) {

}
```

```
};
```

Java Solution:

```
/**
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
* 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 double minTime(int n, int k, int m, int[] time, double[] mul) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Time to Transport All Individuals
Difficulty: Hard
Tags: array, graph, dp, queue, heap

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 minTime(self, n: int, k: int, m: int, time: List[int], mul: List[float])
-> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
* 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} k
* @param {number} m
* @param {number[]} time
* @param {number[]} mul
* @return {number}
*/
var minTime = function(n, k, m, time, mul) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
```

```
* 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 minTime(n: number, k: number, m: number, time: number[], mul:
number[]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
* 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 double MinTime(int n, int k, int m, int[] time, double[] mul) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Time to Transport All Individuals
* Difficulty: Hard
* Tags: array, graph, dp, queue, heap
*
* 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
*/

double minTime(int n, int k, int m, int* time, int timeSize, double* mul, int
```

```
mulSize) {

}
```

Go Solution:

```
// Problem: Minimum Time to Transport All Individuals
// Difficulty: Hard
// Tags: array, graph, dp, queue, heap
//
// 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 minTime(n int, k int, m int, time []int, mul []float64) float64 {

}
```

Kotlin Solution:

```
class Solution {
fun minTime(n: Int, k: Int, m: Int, time: IntArray, mul: DoubleArray): Double
{

}
}
```

Swift Solution:

```
class Solution {
func minTime(_ n: Int, _ k: Int, _ m: Int, _ time: [Int], _ mul: [Double]) ->
Double {

}
}
```

Rust Solution:

```
// Problem: Minimum Time to Transport All Individuals
// Difficulty: Hard
// Tags: array, graph, dp, queue, heap
//
```

```
// 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 min_time(n: i32, k: i32, m: i32, time: Vec<i32>, mul: Vec<f64>) -> f64
{

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer} k
# @param {Integer} m
# @param {Integer[]} time
# @param {Float[]} mul
# @return {Float}
def min_time(n, k, m, time, mul)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer $k
* @param Integer $m
* @param Integer[] $time
* @param Float[] $mul
* @return Float
*/
function minTime($n, $k, $m, $time, $mul) {

}
}
```

Dart Solution:

```
class Solution {
double minTime(int n, int k, int m, List<int> time, List<double> mul) {

}
}
```

Scala Solution:

```
object Solution {
def minTime(n: Int, k: Int, m: Int, time: Array[Int], mul: Array[Double]):
Double = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_time(n :: integer, k :: integer, m :: integer, time :: [integer],
mul :: [float]) :: float
def min_time(n, k, m, time, mul) do

end
end
```

Erlang Solution:

```
-spec min_time(N :: integer(), K :: integer(), M :: integer(), Time ::
[integer()], Mul :: [float()]) -> float().
min_time(N, K, M, Time, Mul) ->
.
```

Racket Solution:

```
(define/contract (min-time n k m time mul)
(-> exact-integer? exact-integer? exact-integer? (listof exact-integer?)
(listof flonum?) flonum?)
)
```