Problem 3540: Minimum Time to Visit All Houses

Problem Information

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

Problem Description

You are given two integer arrays

forward

and

backward

, both of size

n

. You are also given another integer array

queries

.

There are

n

houses

arranged in a circle

. The houses are connected via roads in a special arrangement:

For all

$0 <= i <= n - 2$

, house

i

is connected to house

$i + 1$

via a road with length

forward[i]

meters. Additionally, house

$n - 1$

is connected back to house 0 via a road with length

forward[n - 1]

meters, completing the circle.

For all

$1 <= i <= n - 1$

, house

i

is connected to house

i - 1

via a road with length

backward[i]

meters. Additionally, house 0 is connected back to house

n - 1

via a road with length

backward[0]

meters, completing the circle.

You can walk at a pace of

one

meter per second. Starting from house 0, find the

minimum

time taken to visit each house in the order specified by

queries

.

Return the

minimum

total time taken to visit the houses.

Example 1:

Input:

forward = [1,4,4], backward = [4,1,2], queries = [1,2,0,2]

Output:

12

Explanation:

The path followed is

0

(0)

→

1

(1)

→

2

(5)

→

1

(7)

→

0

(8)

→

2

(12)

.

Note:

The notation used is

node

(total time)

,

→

represents forward road, and

→

represents backward road.

Example 2:

Input:

forward = [1,1,1,1], backward = [2,2,2,2], queries = [1,2,3,0]

Output:

4

Explanation:

The path travelled is

0

→

1

→

2

→

3

→

0

. Each step is in the forward direction and requires 1 second.

Constraints:

2 <= n <= 10

5

n == forward.length == backward.length

1 <= forward[i], backward[i] <= 10

5

1 <= queries.length <= 10

5

0 <= queries[i] < n

queries[i] != queries[i + 1]

queries[0]

is not 0.

Code Snippets

C++:

```
class Solution {
public:
long long minTotalTime(vector<int>& forward, vector<int>& backward,
vector<int>& queries) {

}
};
```

Java:

```
class Solution {
public long minTotalTime(int[] forward, int[] backward, int[] queries) {

}
}
```

Python3:

```
class Solution:
def minTotalTime(self, forward: List[int], backward: List[int], queries:
List[int]) -> int:
```

Python:

```
class Solution(object):
def minTotalTime(self, forward, backward, queries):
"""
:type forward: List[int]
:type backward: List[int]
:type queries: List[int]
```

```
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} forward
* @param {number[]} backward
* @param {number[]} queries
* @return {number}
*/
var minTotalTime = function(forward, backward, queries) {

};
```

TypeScript:

```
function minTotalTime(forward: number[], backward: number[], queries:
number[]): number {

};
```

C#:

```
public class Solution {
public long MinTotalTime(int[] forward, int[] backward, int[] queries) {

}
}
```

C:

```
long long minTotalTime(int* forward, int forwardSize, int* backward, int
backwardSize, int* queries, int queriesSize) {

}
```

Go:

```
func minTotalTime(forward []int, backward []int, queries []int) int64 {

}
```

Kotlin:

```
class Solution {
fun minTotalTime(forward: IntArray, backward: IntArray, queries: IntArray):
Long {

}
}
```

Swift:

```
class Solution {
func minTotalTime(_ forward: [Int], _ backward: [Int], _ queries: [Int]) ->
Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_total_time(forward: Vec<i32>, backward: Vec<i32>, queries:
Vec<i32>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} forward
# @param {Integer[]} backward
# @param {Integer[]} queries
# @return {Integer}
def min_total_time(forward, backward, queries)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $forward
```

```
* @param Integer[] $backward
* @param Integer[] $queries
* @return Integer
*/
function minTotalTime($forward, $backward, $queries) {

}
}
```

Dart:

```
class Solution {
int minTotalTime(List<int> forward, List<int> backward, List<int> queries) {

}
}
```

Scala:

```
object Solution {
def minTotalTime(forward: Array[Int], backward: Array[Int], queries:
Array[Int]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_total_time(forward :: [integer], backward :: [integer], queries ::
[integer]) :: integer
def min_total_time(forward, backward, queries) do

end
end
```

Erlang:

```
-spec min_total_time(Forward :: [integer()], Backward :: [integer()], Queries
:: [integer()]) -> integer().
min_total_time(Forward, Backward, Queries) ->
.
```

Racket:

```
(define/contract (min-total-time forward backward queries)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
*
* 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:
long long minTotalTime(vector<int>& forward, vector<int>& backward,
vector<int>& queries) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
*
* 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 long minTotalTime(int[] forward, int[] backward, int[] queries) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Time to Visit All Houses
Difficulty: Medium
Tags: array

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 minTotalTime(self, forward: List[int], backward: List[int], queries:
List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minTotalTime(self, forward, backward, queries):
"""
:type forward: List[int]
:type backward: List[int]
:type queries: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
```

```
*
* 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[]} forward
* @param {number[]} backward
* @param {number[]} queries
* @return {number}
*/
var minTotalTime = function(forward, backward, queries) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
*
* 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 minTotalTime(forward: number[], backward: number[], queries:
number[]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
*
* 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 long MinTotalTime(int[] forward, int[] backward, int[] queries) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Time to Visit All Houses
* Difficulty: Medium
* Tags: array
*
* 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
*/

long long minTotalTime(int* forward, int forwardSize, int* backward, int
backwardSize, int* queries, int queriesSize) {

}
```

Go Solution:

```
// Problem: Minimum Time to Visit All Houses
// Difficulty: Medium
// Tags: array
//
// 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 minTotalTime(forward []int, backward []int, queries []int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minTotalTime(forward: IntArray, backward: IntArray, queries: IntArray):
Long {

}
}
```

Swift Solution:

```
class Solution {
func minTotalTime(_ forward: [Int], _ backward: [Int], _ queries: [Int]) ->
Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Time to Visit All Houses
// Difficulty: Medium
// Tags: array
//
// 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 min_total_time(forward: Vec<i32>, backward: Vec<i32>, queries:
Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} forward
# @param {Integer[]} backward
# @param {Integer[]} queries
# @return {Integer}
def min_total_time(forward, backward, queries)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $forward
* @param Integer[] $backward
* @param Integer[] $queries
* @return Integer
*/
function minTotalTime($forward, $backward, $queries) {

}
}
```

Dart Solution:

```
class Solution {
int minTotalTime(List<int> forward, List<int> backward, List<int> queries) {

}
}
```

Scala Solution:

```
object Solution {
def minTotalTime(forward: Array[Int], backward: Array[Int], queries:
Array[Int]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_total_time(forward :: [integer], backward :: [integer], queries ::
[integer]) :: integer
def min_total_time(forward, backward, queries) do
```

```
end
end
```

Erlang Solution:

```
-spec min_total_time(Forward :: [integer()], Backward :: [integer()], Queries
:: [integer()]) -> integer().
min_total_time(Forward, Backward, Queries) ->
.
```

Racket Solution:

```
(define/contract (min-total-time forward backward queries)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
exact-integer?)
)
```