Problem 2297: Jump Game VIII

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

nums

of length

n

. You are initially standing at index

0

. You can jump from index

i

to index

j

where

i < j

if:

nums[i] <= nums[j]

and

nums[k] < nums[i]

for all indexes

k

in the range

i < k < j

, or

nums[i] > nums[j]

and

nums[k] >= nums[i]

for all indexes

k

in the range

i < k < j

.

You are also given an integer array

costs

of length

n

where

costs[i]

denotes the cost of jumping

to

index

i

.

Return

the

minimum

cost to jump to the index

n - 1

.

Example 1:

Input:

nums = [3,2,4,4,1], costs = [3,7,6,4,2]

Output:

8

Explanation:

You start at index 0. - Jump to index 2 with a cost of costs[2] = 6. - Jump to index 4 with a cost of costs[4] = 2. The total cost is 8. It can be proven that 8 is the minimum cost needed. Two other possible paths are from index 0 -> 1 -> 4 and index 0 -> 2 -> 3 -> 4. These have a total cost of 9 and 12, respectively.

Example 2:

Input:

nums = [0,1,2], costs = [1,1,1]

Output:

2

Explanation:

Start at index 0. - Jump to index 1 with a cost of costs[1] = 1. - Jump to index 2 with a cost of costs[2] = 1. The total cost is 2. Note that you cannot jump directly from index 0 to index 2 because nums[0] <= nums[1].

Constraints:

n == nums.length == costs.length

1 <= n <= 10

5

0 <= nums[i], costs[i] <= 10

5

Code Snippets

C++:

```
class Solution {
public:
long long minCost(vector<int>& nums, vector<int>& costs) {

}
};
```

Java:

```
class Solution {
public long minCost(int[] nums, int[] costs) {

}
}
```

Python3:

```
class Solution:
def minCost(self, nums: List[int], costs: List[int]) -> int:
```

Python:

```
class Solution(object):
def minCost(self, nums, costs):
"""
:type nums: List[int]
:type costs: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number[]} costs
* @return {number}
*/
var minCost = function(nums, costs) {

};
```

TypeScript:

```
function minCost(nums: number[], costs: number[]): number {

};
```

C#:

```
public class Solution {
public long MinCost(int[] nums, int[] costs) {

}
}
```

C:

```
long long minCost(int* nums, int numsSize, int* costs, int costsSize) {

}
```

Go:

```
func minCost(nums []int, costs []int) int64 {

}
```

Kotlin:

```
class Solution {
fun minCost(nums: IntArray, costs: IntArray): Long {

}
}
```

Swift:

```
class Solution {
func minCost(_ nums: [Int], _ costs: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_cost(nums: Vec<i32>, costs: Vec<i32>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer[]} costs
# @return {Integer}
def min_cost(nums, costs)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer[] $costs
* @return Integer
*/
function minCost($nums, $costs) {

}
}
```

Dart:

```
class Solution {
int minCost(List<int> nums, List<int> costs) {

}
}
```

Scala:

```
object Solution {
def minCost(nums: Array[Int], costs: Array[Int]): Long = {

}

}
```

Elixir:

```
defmodule Solution do
@spec min_cost(nums :: [integer], costs :: [integer]) :: integer
def min_cost(nums, costs) do

end
end
```

Erlang:

```
-spec min_cost(Nums :: [integer()], Costs :: [integer()]) -> integer().
min_cost(Nums, Costs) ->
.
```

Racket:

```
(define/contract (min-cost nums costs)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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:
long long minCost(vector<int>& nums, vector<int>& costs) {
}
};
```

Java Solution:

```
/**
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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 long minCost(int[] nums, int[] costs) {

}
}
```

Python3 Solution:

```
"""
Problem: Jump Game VIII
Difficulty: Medium
Tags: array, graph, dp, stack

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 minCost(self, nums: List[int], costs: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minCost(self, nums, costs):
"""
:type nums: List[int]
:type costs: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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[]} nums
* @param {number[]} costs
* @return {number}
*/
var minCost = function(nums, costs) {

};
```

TypeScript Solution:

```
/**
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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 minCost(nums: number[], costs: number[]): number {
};
```

C# Solution:

```
/*
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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 long MinCost(int[] nums, int[] costs) {

}
}
```

C Solution:

```
/*
* Problem: Jump Game VIII
* Difficulty: Medium
* Tags: array, graph, dp, stack
*
* 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
*/

long long minCost(int* nums, int numsSize, int* costs, int costsSize) {

}
```

Go Solution:

```
// Problem: Jump Game VIII
// Difficulty: Medium
// Tags: array, graph, dp, stack
//
// 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 minCost(nums []int, costs []int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minCost(nums: IntArray, costs: IntArray): Long {

}
}
```

Swift Solution:

```
class Solution {
func minCost(_ nums: [Int], _ costs: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Jump Game VIII
// Difficulty: Medium
// Tags: array, graph, dp, stack
//
// 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_cost(nums: Vec<i32>, costs: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

```ruby
# @param {Integer[]} nums
# @param {Integer[]} costs
# @return {Integer}
def min_cost(nums, costs)

end
```

PHP Solution:

```php
class Solution {

/**
* @param Integer[] $nums
* @param Integer[] $costs
* @return Integer
*/
function minCost($nums, $costs) {

}
}
```

Dart Solution:

```dart
class Solution {
int minCost(List<int> nums, List<int> costs) {

}
}
```

Scala Solution:

```scala
object Solution {
def minCost(nums: Array[Int], costs: Array[Int]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_cost(nums :: [integer], costs :: [integer]) :: integer
def min_cost(nums, costs) do

end
end
```

Erlang Solution:

```
-spec min_cost(Nums :: [integer()], Costs :: [integer()]) -> integer().
min_cost(Nums, Costs) ->
.
```

Racket Solution:

```
(define/contract (min-cost nums costs)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```