Problem 3502: Minimum Cost to Reach Every Position

Problem Information

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

Problem Description

You are given an integer array

cost

of size

n

. You are currently at position

n

(at the end of the line) in a line of

n + 1

people (numbered from 0 to

n

).

You wish to move forward in the line, but each person in front of you charges a specific amount to swap

places. The cost to swap with person

i

is given by

cost[i]

.

You are allowed to swap places with people as follows:

If they are in front of you, you

must

pay them

cost[i]

to swap with them.

If they are behind you, they can swap with you for free.

Return an array

answer

of size

n

, where

answer[i]

is the

minimum

total cost to reach each position

i

in the line

.

Example 1:

Input:

cost = [5,3,4,1,3,2]

Output:

[5,3,3,1,1,1]

Explanation:

We can get to each position in the following way:

i = 0

. We can swap with person 0 for a cost of 5.

i =

1

. We can swap with person 1 for a cost of 3.

i = 2

. We can swap with person 1 for a cost of 3, then swap with person 2 for free.

i = 3

. We can swap with person 3 for a cost of 1.

i = 4

. We can swap with person 3 for a cost of 1, then swap with person 4 for free.

i = 5

. We can swap with person 3 for a cost of 1, then swap with person 5 for free.

Example 2:

Input:

cost = [1,2,4,6,7]

Output:

[1,1,1,1,1]

Explanation:

We can swap with person 0 for a cost of

1, then we will be able to reach any position

i

for free.

Constraints:

1 <= n == cost.length <= 100

1 <= cost[i] <= 100

Code Snippets

C++:

```
class Solution {
public:
vector<int> minCosts(vector<int>& cost) {

}
};
```

Java:

```
class Solution {
public int[] minCosts(int[] cost) {

}
}
```

Python3:

```
class Solution:
def minCosts(self, cost: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def minCosts(self, cost):
"""
:type cost: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} cost
* @return {number[]}
*/
var minCosts = function(cost) {

};
```

TypeScript:

```
function minCosts(cost: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] MinCosts(int[] cost) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minCosts(int* cost, int costSize, int* returnSize) {

}
```

Go:

```
func minCosts(cost []int) []int {

}
```

Kotlin:

```
class Solution {
fun minCosts(cost: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func minCosts(_ cost: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn min_costs(cost: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} cost
# @return {Integer[]}
def min_costs(cost)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $cost
* @return Integer[]
*/
function minCosts($cost) {

}
}
```

Dart:

```
class Solution {
List<int> minCosts(List<int> cost) {

}
}
```

Scala:

```
object Solution {
def minCosts(cost: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_costs(cost :: [integer]) :: [integer]
def min_costs(cost) do

end
end
```

Erlang:

```
-spec min_costs(Cost :: [integer()]) -> [integer()].
min_costs(Cost) ->
.
```

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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:
vector<int> minCosts(vector<int>& cost) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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 int[] minCosts(int[] cost) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Reach Every Position
Difficulty: Easy
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 minCosts(self, cost: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minCosts(self, cost):
"""
:type cost: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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[]} cost
* @return {number[]}
*/
var minCosts = function(cost) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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 minCosts(cost: number[]): number[] {
};
```

C# Solution:

```
/*
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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 int[] MinCosts(int[] cost) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost to Reach Every Position
* Difficulty: Easy
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* minCosts(int* cost, int costSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Minimum Cost to Reach Every Position
// Difficulty: Easy
// 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 minCosts(cost []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun minCosts(cost: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func minCosts(_ cost: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Minimum Cost to Reach Every Position
// Difficulty: Easy
// 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_costs(cost: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} cost
# @return {Integer[]}
def min_costs(cost)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $cost
* @return Integer[]
*/
function minCosts($cost) {

}
}
```

Dart Solution:

```
class Solution {
List<int> minCosts(List<int> cost) {

}
}
```

Scala Solution:

```
object Solution {
def minCosts(cost: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_costs(cost :: [integer]) :: [integer]
def min_costs(cost) do

end
end
```

Erlang Solution:

```
-spec min_costs(Cost :: [integer()]) -> [integer()].
min_costs(Cost) ->
.
```

Racket Solution:

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