Problem 3693: Climbing Stairs II

Problem Information

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

Problem Description

You are climbing a staircase with

n + 1

steps, numbered from 0 to

n

.

You are also given a

1-indexed

integer array

costs

of length

n

, where

costs[i]

is the cost of step

i

.

From step

i

, you can jump

only

to step

i + 1

,

i + 2

, or

i + 3

. The cost of jumping from step

i

to step

j

is defined as:

costs[j] + (j - i)

2

You start from step 0 with

cost = 0

.

Return the

minimum

total cost to reach step

n

.

Example 1:

Input:

n = 4, costs = [1,2,3,4]

Output:

13

Explanation:

One optimal path is

$0 \rightarrow 1 \rightarrow 2 \rightarrow 4$

Jump

Cost Calculation

Cost

$0 \rightarrow 1$

costs[1] + (1 - 0)

2

= 1 + 1

2

$1 \rightarrow 2$

costs[2] + (2 - 1)

2

= 2 + 1

3

$2 \rightarrow 4$

costs[4] + (4 - 2)

2

= 4 + 4

8

Thus, the minimum total cost is

2 + 3 + 8 = 13

Example 2:

Input:

n = 4, costs = [5,1,6,2]

Output:

11

Explanation:

One optimal path is

$0 \rightarrow 2 \rightarrow 4$

Jump

Cost Calculation

Cost

$0 \rightarrow 2$

costs[2] + (2 - 0)

2

= 1 + 4

5

$2 \rightarrow 4$

costs[4] + (4 - 2)

2

= 2 + 4

6

Thus, the minimum total cost is

5 + 6 = 11

Example 3:

Input:

n = 3, costs = [9,8,3]

Output:

12

Explanation:

The optimal path is

0 $\rightarrow$ 3

with total cost =

costs[3] + (3 - 0)

2

= 3 + 9 = 12

Constraints:

1 <= n == costs.length <= 10

5

1 <= costs[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
int climbStairs(int n, vector<int>& costs) {

}
};
```

Java:

```
class Solution {
public int climbStairs(int n, int[] costs) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

```
};
```

TypeScript:

```
function climbStairs(n: number, costs: number[]): number {

};
```

C#:

```
public class Solution {
public int ClimbStairs(int n, int[] costs) {

}
}
```

C:

```
int climbStairs(int n, int* costs, int costsSize) {

}
```

Go:

```
func climbStairs(n int, costs []int) int {

}
```

Kotlin:

```
class Solution {
fun climbStairs(n: Int, costs: IntArray): Int {

}
}
```

Swift:

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

}
```

```
}
```

Rust:

```
impl Solution {
pub fn climb_stairs(n: i32, costs: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[]} costs
# @return {Integer}
def climb_stairs(n, costs)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $costs
* @return Integer
*/
function climbStairs($n, $costs) {

}
}
```

Dart:

```
class Solution {
int climbStairs(int n, List<int> costs) {

}
}
```

Scala:

```
object Solution {
def climbStairs(n: Int, costs: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec climb_stairs(n :: integer, costs :: [integer]) :: integer
def climb_stairs(n, costs) do

end
end
```

Erlang:

```
-spec climb_stairs(N :: integer(), Costs :: [integer()]) -> integer().
climb_stairs(N, Costs) ->
.
```

Racket:

```
(define/contract (climb-stairs n costs)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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:
int climbStairs(int n, vector<int>& costs) {

}
};
```

Java Solution:

```
/**
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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 int climbStairs(int n, int[] costs) {

}
}
```

Python3 Solution:

```
"""
Problem: Climbing Stairs II
Difficulty: Medium
Tags: array, dp

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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[]} costs
* @return {number}
*/
var climbStairs = function(n, costs) {

};
```

TypeScript Solution:

```
/**
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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 climbStairs(n: number, costs: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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 int ClimbStairs(int n, int[] costs) {

}
}
```

C Solution:

```
/*
* Problem: Climbing Stairs II
* Difficulty: Medium
* Tags: array, dp
*
* 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
*/

int climbStairs(int n, int* costs, int costsSize) {

}
```

Go Solution:

```
// Problem: Climbing Stairs II
// Difficulty: Medium
// Tags: array, dp
//
// 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 climbStairs(n int, costs []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun climbStairs(n: Int, costs: IntArray): Int {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Climbing Stairs II
// Difficulty: Medium
// Tags: array, dp
//
// 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 climb_stairs(n: i32, costs: Vec<i32>) -> i32 {

}
```

```
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[]} costs
# @return {Integer}
def climb_stairs(n, costs)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $costs
* @return Integer
*/
function climbStairs($n, $costs) {

}
}
```

Dart Solution:

```
class Solution {
int climbStairs(int n, List<int> costs) {

}
}
```

Scala Solution:

```
object Solution {
def climbStairs(n: Int, costs: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec climb_stairs(n :: integer, costs :: [integer]) :: integer
def climb_stairs(n, costs) do

end
end
```

Erlang Solution:

```
-spec climb_stairs(N :: integer(), Costs :: [integer()]) -> integer().
climb_stairs(N, Costs) ->
.
```

Racket Solution:

```
(define/contract (climb-stairs n costs)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```