Problem 1000: Minimum Cost to Merge Stones

Problem Information

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

Problem Description

There are

n

piles of

stones

arranged in a row. The

i

th

pile has

stones[i]

stones.

A move consists of merging exactly

k

consecutive

piles into one pile, and the cost of this move is equal to the total number of stones in these

k

piles.

Return

the minimum cost to merge all piles of stones into one pile

. If it is impossible, return

-1

.

Example 1:

Input:

stones = [3,2,4,1], k = 2

Output:

20

Explanation:

We start with [3, 2, 4, 1]. We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1]. We merge [4, 1] for a cost of 5, and we are left with [5, 5]. We merge [5, 5] for a cost of 10, and we are left with [10]. The total cost was 20, and this is the minimum possible.

Example 2:

Input:

stones = [3,2,4,1], k = 3

Output:

-1

Explanation:

After any merge operation, there are 2 piles left, and we can't merge anymore. So the task is impossible.

Example 3:

Input:

stones = [3,5,1,2,6], k = 3

Output:

25

Explanation:

We start with [3, 5, 1, 2, 6]. We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6]. We merge [3, 8, 6] for a cost of 17, and we are left with [17]. The total cost was 25, and this is the minimum possible.

Constraints:

n == stones.length

1 <= n <= 30

1 <= stones[i] <= 100

2 <= k <= 30

Code Snippets

C++:

```
class Solution {
public:
int mergeStones(vector<int>& stones, int k) {

}
};
```

Java:

```
class Solution {
public int mergeStones(int[] stones, int k) {

}
}
```

Python3:

```
class Solution:
def mergeStones(self, stones: List[int], k: int) -> int:
```

Python:

```
class Solution(object):
def mergeStones(self, stones, k):
"""
:type stones: List[int]
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} stones
* @param {number} k
* @return {number}
*/
var mergeStones = function(stones, k) {

};
```

TypeScript:

```
function mergeStones(stones: number[], k: number): number {

};
```

C#:

```
public class Solution {
public int MergeStones(int[] stones, int k) {

}
}
```

C:

```
int mergeStones(int* stones, int stonesSize, int k) {

}
```

Go:

```
func mergeStones(stones []int, k int) int {

}
```

Kotlin:

```
class Solution {
fun mergeStones(stones: IntArray, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func mergeStones(_ stones: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn merge_stones(stones: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} stones
# @param {Integer} k
# @return {Integer}
def merge_stones(stones, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $stones
* @param Integer $k
* @return Integer
*/
function mergeStones($stones, $k) {

}
}
```

Dart:

```
class Solution {
int mergeStones(List<int> stones, int k) {

}
}
```

Scala:

```
object Solution {
def mergeStones(stones: Array[Int], k: Int): Int = {

}
```

```
}
```

Elixir:

```
defmodule Solution do
@spec merge_stones(stones :: [integer], k :: integer) :: integer
def merge_stones(stones, k) do

end
end
```

Erlang:

```
-spec merge_stones(Stones :: [integer()], K :: integer()) -> integer().
merge_stones(Stones, K) ->
.
```

Racket:

```
(define/contract (merge-stones stones k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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 mergeStones(vector<int>& stones, int k) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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 mergeStones(int[] stones, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Merge Stones
Difficulty: Hard
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 mergeStones(self, stones: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def mergeStones(self, stones, k):
"""
:type stones: List[int]
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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[]} stones
* @param {number} k
* @return {number}
*/
var mergeStones = function(stones, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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 mergeStones(stones: number[], k: number): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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 MergeStones(int[] stones, int k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost to Merge Stones
* Difficulty: Hard
* 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 mergeStones(int* stones, int stonesSize, int k) {

}
```

Go Solution:

```
// Problem: Minimum Cost to Merge Stones
// Difficulty: Hard
```

```
// 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 mergeStones(stones []int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun mergeStones(stones: IntArray, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func mergeStones(_ stones: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Cost to Merge Stones
// Difficulty: Hard
// 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 merge_stones(stones: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} stones
# @param {Integer} k
# @return {Integer}
def merge_stones(stones, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $stones
* @param Integer $k
* @return Integer
*/
function mergeStones($stones, $k) {

}
}
```

Dart Solution:

```
class Solution {
int mergeStones(List<int> stones, int k) {

}
}
```

Scala Solution:

```
object Solution {
def mergeStones(stones: Array[Int], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec merge_stones(stones :: [integer], k :: integer) :: integer
def merge_stones(stones, k) do

end
end
```

Erlang Solution:

```
-spec merge_stones(Stones :: [integer()], K :: integer()) -> integer().
merge_stones(Stones, K) ->
.
```

Racket Solution:

```
(define/contract (merge-stones stones k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```