Problem 2462: Total Cost to Hire K Workers

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

costs

where

costs[i]

is the cost of hiring the

i

th

worker.

You are also given two integers

k

and

candidates

. We want to hire exactly

k

workers according to the following rules:

You will run

k

sessions and hire exactly one worker in each session.

In each hiring session, choose the worker with the lowest cost from either the first

candidates

workers or the last

candidates

workers. Break the tie by the smallest index.

For example, if

costs = [3,2,7,7,1,2]

and

candidates = 2

, then in the first hiring session, we will choose the

4

th

worker because they have the lowest cost

[

3,2

,7,7,

1

,2

]

.

In the second hiring session, we will choose

1

st

worker because they have the same lowest cost as

4

th

worker but they have the smallest index

[

3,

2

,7,

7,2

]

. Please note that the indexing may be changed in the process.

If there are fewer than candidates workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index.

A worker can only be chosen once.

Return

the total cost to hire exactly

k

workers.

Example 1:

Input:

costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4

Output:

11

Explanation:

We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [

17,12,10,2

,7,

2,11,20,8

]. The lowest cost is 2, and we break the tie by the smallest index, which is 3. The total cost = 0 + 2 = 2. - In the second hiring round we choose the worker from [

17,12,10,7

,

2,11,20,8

]. The lowest cost is 2 (index 4). The total cost = 2 + 2 = 4. - In the third hiring round we choose the worker from [

17,12,10,7,11,20,8

]. The lowest cost is 7 (index 3). The total cost = 4 + 7 = 11. Notice that the worker with index 3 was common in the first and last four workers. The total hiring cost is 11.

Example 2:

Input:

costs = [1,2,4,1], k = 3, candidates = 3

Output:

4

Explanation:

We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [

1,2,4,1

]. The lowest cost is 1, and we break the tie by the smallest index, which is 0. The total cost = 0 + 1 = 1. Notice that workers with index 1 and 2 are common in the first and last 3 workers. - In the second hiring round we choose the worker from [

2,4,1

]. The lowest cost is 1 (index 2). The total cost = 1 + 1 = 2. - In the third hiring round there are less than three candidates. We choose the worker from the remaining workers [

2,4

]. The lowest cost is 2 (index 0). The total cost = 2 + 2 = 4. The total hiring cost is 4.

Constraints:

1 <= costs.length <= 10

5

1 <= costs[i] <= 10

5

1 <= k, candidates <= costs.length

Code Snippets

C++:

```
class Solution {
public:
long long totalCost(vector<int>& costs, int k, int candidates) {

}
};
```

Java:

```
class Solution {
public long totalCost(int[] costs, int k, int candidates) {

}
}
```

Python3:

```
class Solution:
def totalCost(self, costs: List[int], k: int, candidates: int) -> int:
```

Python:

```
class Solution(object):
def totalCost(self, costs, k, candidates):
"""
:type costs: List[int]
:type k: int
:type candidates: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} costs
* @param {number} k
* @param {number} candidates
* @return {number}
*/
var totalCost = function(costs, k, candidates) {

};
```

TypeScript:

```
function totalCost(costs: number[], k: number, candidates: number): number {

};
```

C#:

```
public class Solution {
public long TotalCost(int[] costs, int k, int candidates) {

}
}
```

C:

```
long long totalCost(int* costs, int costsSize, int k, int candidates) {

}
```

Go:

```
func totalCost(costs []int, k int, candidates int) int64 {

}
```

Kotlin:

```
class Solution {
fun totalCost(costs: IntArray, k: Int, candidates: Int): Long {

}
}
```

Swift:

```
class Solution {
func totalCost(_ costs: [Int], _ k: Int, _ candidates: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn total_cost(costs: Vec<i32>, k: i32, candidates: i32) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} costs
# @param {Integer} k
# @param {Integer} candidates
# @return {Integer}
def total_cost(costs, k, candidates)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $costs
* @param Integer $k
* @param Integer $candidates
* @return Integer
*/
function totalCost($costs, $k, $candidates) {

}
}
```

Dart:

```
class Solution {
int totalCost(List<int> costs, int k, int candidates) {

}
}
```

Scala:

```
object Solution {
def totalCost(costs: Array[Int], k: Int, candidates: Int): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec total_cost(costs :: [integer], k :: integer, candidates :: integer) ::
integer
def total_cost(costs, k, candidates) do

end
end
```

Erlang:

```
-spec total_cost(Costs :: [integer()], K :: integer(), Candidates ::
integer()) -> integer().
total_cost(Costs, K, Candidates) ->
.
```

Racket:

```
(define/contract (total-cost costs k candidates)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Total Cost to Hire K Workers
* Difficulty: Medium
* Tags: array, queue, heap
*
* 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 totalCost(vector<int>& costs, int k, int candidates) {

}
};
```

Java Solution:

```
/**
* Problem: Total Cost to Hire K Workers
* Difficulty: Medium
* Tags: array, queue, heap
*
* 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 totalCost(int[] costs, int k, int candidates) {

}
}
```

Python3 Solution:

```
"""
Problem: Total Cost to Hire K Workers
Difficulty: Medium
Tags: array, queue, heap

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 totalCost(self, costs: List[int], k: int, candidates: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def totalCost(self, costs, k, candidates):
"""
:type costs: List[int]
:type k: int
:type candidates: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Total Cost to Hire K Workers
```

```
* Difficulty: Medium
* Tags: array, queue, heap
*
* 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[]} costs
* @param {number} k
* @param {number} candidates
* @return {number}
*/
var totalCost = function(costs, k, candidates) {

};
```

TypeScript Solution:

```
/**
* Problem: Total Cost to Hire K Workers
* Difficulty: Medium
* Tags: array, queue, heap
*
* 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 totalCost(costs: number[], k: number, candidates: number): number {

};
```

C# Solution:

```
/*
* Problem: Total Cost to Hire K Workers
* Difficulty: Medium
* Tags: array, queue, heap
*
```

```
* 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 TotalCost(int[] costs, int k, int candidates) {

}
}
```

C Solution:

```
/*
* Problem: Total Cost to Hire K Workers
* Difficulty: Medium
* Tags: array, queue, heap
*
* 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 totalCost(int* costs, int costsSize, int k, int candidates) {

}
```

Go Solution:

```
// Problem: Total Cost to Hire K Workers
// Difficulty: Medium
// Tags: array, queue, heap
//
// 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 totalCost(costs []int, k int, candidates int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun totalCost(costs: IntArray, k: Int, candidates: Int): Long {

}
}
```

Swift Solution:

```
class Solution {
func totalCost(_ costs: [Int], _ k: Int, _ candidates: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Total Cost to Hire K Workers
// Difficulty: Medium
// Tags: array, queue, heap
//
// 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 total_cost(costs: Vec<i32>, k: i32, candidates: i32) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} costs
# @param {Integer} k
# @param {Integer} candidates
# @return {Integer}
def total_cost(costs, k, candidates)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $costs
* @param Integer $k
* @param Integer $candidates
* @return Integer
*/
function totalCost($costs, $k, $candidates) {

}
}
```

Dart Solution:

```
class Solution {
int totalCost(List<int> costs, int k, int candidates) {

}
}
```

Scala Solution:

```
object Solution {
def totalCost(costs: Array[Int], k: Int, candidates: Int): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec total_cost(costs :: [integer], k :: integer, candidates :: integer) ::
integer
def total_cost(costs, k, candidates) do

end
end
```

Erlang Solution:

```
-spec total_cost(Costs :: [integer()], K :: integer(), Candidates ::
integer()) -> integer().
total_cost(Costs, K, Candidates) ->
.
```

Racket Solution:

```
(define/contract (total-cost costs k candidates)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?)
)
```