Problem 2611: Mice and Cheese

Problem Information

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

Problem Description

There are two mice and

n

different types of cheese, each type of cheese should be eaten by exactly one mouse.

A point of the cheese with index

i

(

0-indexed

) is:

reward1[i]

if the first mouse eats it.

reward2[i]

if the second mouse eats it.

You are given a positive integer array

reward1

, a positive integer array

reward2

, and a non-negative integer

k

.

Return

the maximum

points the mice can achieve if the first mouse eats exactly

k

types of cheese.

Example 1:

Input:

reward1 = [1,1,3,4], reward2 = [4,4,1,1], k = 2

Output:

15

Explanation:

In this example, the first mouse eats the 2

nd

(0-indexed) and the 3

rd

types of cheese, and the second mouse eats the 0

th

and the 1

st

types of cheese. The total points are 4 + 4 + 3 + 4 = 15. It can be proven that 15 is the maximum total points that the mice can achieve.

Example 2:

Input:

reward1 = [1,1], reward2 = [1,1], k = 2

Output:

2

Explanation:

In this example, the first mouse eats the 0

th

(0-indexed) and 1

st

types of cheese, and the second mouse does not eat any cheese. The total points are 1 + 1 = 2. It can be proven that 2 is the maximum total points that the mice can achieve.

Constraints:

1 <= n == reward1.length == reward2.length <= 10

5

1 <= reward1[i], reward2[i] <= 1000

0 <= k <= n

Code Snippets

C++:

```
class Solution {
public:
int miceAndCheese(vector<int>& reward1, vector<int>& reward2, int k) {

}
};
```

Java:

```
class Solution {
public int miceAndCheese(int[] reward1, int[] reward2, int k) {

}
}
```

Python3:

```
class Solution:
def miceAndCheese(self, reward1: List[int], reward2: List[int], k: int) ->
int:
```

Python:

```
class Solution(object):
def miceAndCheese(self, reward1, reward2, k):
"""
:type reward1: List[int]
:type reward2: List[int]
:type k: int
:rtype: int
```

```
"""
```

JavaScript:

```
/**
* @param {number[]} reward1
* @param {number[]} reward2
* @param {number} k
* @return {number}
*/
var miceAndCheese = function(reward1, reward2, k) {

};
```

TypeScript:

```
function miceAndCheese(reward1: number[], reward2: number[], k: number):
number {

};
```

C#:

```
public class Solution {
public int MiceAndCheese(int[] reward1, int[] reward2, int k) {

}
}
```

C:

```
int miceAndCheese(int* reward1, int reward1Size, int* reward2, int
reward2Size, int k) {

}
```

Go:

```
func miceAndCheese(reward1 []int, reward2 []int, k int) int {

}
```

Kotlin:

```
class Solution {
fun miceAndCheese(reward1: IntArray, reward2: IntArray, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func miceAndCheese(_ reward1: [Int], _ reward2: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn mice_and_cheese(reward1: Vec<i32>, reward2: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} reward1
# @param {Integer[]} reward2
# @param {Integer} k
# @return {Integer}
def mice_and_cheese(reward1, reward2, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $reward1
* @param Integer[] $reward2
* @param Integer $k
* @return Integer
```

```
*/
function miceAndCheese($reward1, $reward2, $k) {

}
}
```

Dart:

```
class Solution {
int miceAndCheese(List<int> reward1, List<int> reward2, int k) {

}
}
```

Scala:

```
object Solution {
def miceAndCheese(reward1: Array[Int], reward2: Array[Int], k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec mice_and_cheese(reward1 :: [integer], reward2 :: [integer], k ::
integer) :: integer
def mice_and_cheese(reward1, reward2, k) do

end
end
```

Erlang:

```
-spec mice_and_cheese(Reward1 :: [integer()], Reward2 :: [integer()], K ::
integer()) -> integer().
mice_and_cheese(Reward1, Reward2, K) ->
.
```

Racket:

```
(define/contract (mice-and-cheese reward1 reward2 k)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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:
int miceAndCheese(vector<int>& reward1, vector<int>& reward2, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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 int miceAndCheese(int[] reward1, int[] reward2, int k) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Mice and Cheese
Difficulty: Medium
Tags: array, greedy, sort, 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 miceAndCheese(self, reward1: List[int], reward2: List[int], k: int) ->
int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def miceAndCheese(self, reward1, reward2, k):
"""
:type reward1: List[int]
:type reward2: List[int]
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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[]} reward1
* @param {number[]} reward2
* @param {number} k
* @return {number}
*/
var miceAndCheese = function(reward1, reward2, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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 miceAndCheese(reward1: number[], reward2: number[], k: number):
number {

};
```

C# Solution:

```
/*
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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 int MiceAndCheese(int[] reward1, int[] reward2, int k) {

}
}
```

C Solution:

```
/*
* Problem: Mice and Cheese
* Difficulty: Medium
* Tags: array, greedy, sort, 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
*/

int miceAndCheese(int* reward1, int reward1Size, int* reward2, int
reward2Size, int k) {

}
```

Go Solution:

```
// Problem: Mice and Cheese
// Difficulty: Medium
// Tags: array, greedy, sort, 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 miceAndCheese(reward1 []int, reward2 []int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun miceAndCheese(reward1: IntArray, reward2: IntArray, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func miceAndCheese(_ reward1: [Int], _ reward2: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Mice and Cheese
// Difficulty: Medium
// Tags: array, greedy, sort, 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 mice_and_cheese(reward1: Vec<i32>, reward2: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} reward1
# @param {Integer[]} reward2
# @param {Integer} k
# @return {Integer}
def mice_and_cheese(reward1, reward2, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $reward1
* @param Integer[] $reward2
* @param Integer $k
* @return Integer
*/
function miceAndCheese($reward1, $reward2, $k) {

}
}
```

Dart Solution:

```
class Solution {
int miceAndCheese(List<int> reward1, List<int> reward2, int k) {

}
}
```

Scala Solution:

```
object Solution {
def miceAndCheese(reward1: Array[Int], reward2: Array[Int], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec mice_and_cheese(reward1 :: [integer], reward2 :: [integer], k ::
integer) :: integer
def mice_and_cheese(reward1, reward2, k) do

end
end
```

Erlang Solution:

```
-spec mice_and_cheese(Reward1 :: [integer()], Reward2 :: [integer()], K ::
integer()) -> integer().
mice_and_cheese(Reward1, Reward2, K) ->
.
```

Racket Solution:

```
(define/contract (mice-and-cheese reward1 reward2 k)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```