Problem 2813: Maximum Elegance of a K-Length Subsequence

Problem Information

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

Problem Description

You are given a

0-indexed

2D integer array

items

of length

n

and an integer

k

.

items[i] = [profit

i

, category

i

]

, where

profit

i

and

category

i

denote the profit and category of the

i

th

item respectively.

Let's define the

elegance

of a

subsequence

of

items

as

total_profit + distinct_categories

2

, where

total_profit

is the sum of all profits in the subsequence, and

distinct_categories

is the number of

distinct

categories from all the categories in the selected subsequence.

Your task is to find the

maximum elegance

from all subsequences of size

k

in

items

.

Return

an integer denoting the maximum elegance of a subsequence of

items

with size exactly

k

.

Note:

A subsequence of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order.

Example 1:

Input:

items = [[3,2],[5,1],[10,1]], k = 2

Output:

17

Explanation:

In this example, we have to select a subsequence of size 2. We can select items[0] = [3,2] and items[2] = [10,1]. The total profit in this subsequence is 3 + 10 = 13, and the subsequence contains 2 distinct categories [2,1]. Hence, the elegance is 13 + 2

2

= 17, and we can show that it is the maximum achievable elegance.

Example 2:

Input:

items = [[3,1],[3,1],[2,2],[5,3]], k = 3

Output:

19

Explanation:

In this example, we have to select a subsequence of size 3. We can select items[0] = [3,1], items[2] = [2,2], and items[3] = [5,3]. The total profit in this subsequence is 3 + 2 + 5 = 10, and the subsequence contains 3 distinct categories [1,2,3]. Hence, the elegance is 10 + 3

2

= 19, and we can show that it is the maximum achievable elegance.

Example 3:

Input:

items = [[1,1],[2,1],[3,1]], k = 3

Output:

7

Explanation:

In this example, we have to select a subsequence of size 3. We should select all the items. The total profit will be 1 + 2 + 3 = 6, and the subsequence contains 1 distinct category [1]. Hence, the maximum elegance is 6 + 1

2

= 7.

Constraints:

1 <= items.length == n <= 10

5

items[i].length == 2

items[i][0] == profit

i

items[i][1] == category

i

1 <= profit

i

<= 10

9

1 <= category

i

<= n

1 <= k <= n

Code Snippets

C++:

```
class Solution {
public:
long long findMaximumElegance(vector<vector<int>>& items, int k) {

}
};
```

Java:

```
class Solution {
public long findMaximumElegance(int[][] items, int k) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function findMaximumElegance(items: number[][], k: number): number {

};
```

C#:

```
public class Solution {
public long FindMaximumElegance(int[][] items, int k) {

}
}
```

C:

```
long long findMaximumElegance(int** items, int itemsSize, int* itemsColSize,
int k) {

}
```

Go:

```
func findMaximumElegance(items [][]int, k int) int64 {

}
```

Kotlin:

```
class Solution {
fun findMaximumElegance(items: Array<IntArray>, k: Int): Long {

}
}
```

Swift:

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

}
}
```

Rust:

```
impl Solution {
pub fn find_maximum_elegance(items: Vec<Vec<i32>>, k: i32) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[][]} items
# @param {Integer} k
# @return {Integer}
def find_maximum_elegance(items, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $items
* @param Integer $k
* @return Integer
*/
function findMaximumElegance($items, $k) {

}
}
```

Dart:

```
class Solution {
int findMaximumElegance(List<List<int>> items, int k) {

}
}
```

Scala:

```
object Solution {
def findMaximumElegance(items: Array[Array[Int]], k: Int): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_maximum_elegance(items :: [[integer]], k :: integer) :: integer
def find_maximum_elegance(items, k) do

end
end
```

Erlang:

```
-spec find_maximum_elegance(Items :: [[integer()]], K :: integer()) ->
integer().
```

```
find_maximum_elegance(Items, K) ->
.
```

Racket:

```
(define/contract (find-maximum-elegance items k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
long long findMaximumElegance(vector<vector<int>>& items, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
```

```
*/

class Solution {
public long findMaximumElegance(int[][] items, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Elegance of a K-Length Subsequence
Difficulty: Hard
Tags: array, greedy, hash, sort, stack, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def findMaximumElegance(self, items: List[List[int]], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[][]} items
* @param {number} k
* @return {number}
*/
var findMaximumElegance = function(items, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function findMaximumElegance(items: number[][], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
```

```
*/

public class Solution {
public long FindMaximumElegance(int[][] items, int k) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Elegance of a K-Length Subsequence
* Difficulty: Hard
* Tags: array, greedy, hash, sort, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

long long findMaximumElegance(int** items, int itemsSize, int* itemsColSize,
int k) {

}
```

Go Solution:

```
// Problem: Maximum Elegance of a K-Length Subsequence
// Difficulty: Hard
// Tags: array, greedy, hash, sort, stack, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func findMaximumElegance(items [][]int, k int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun findMaximumElegance(items: Array<IntArray>, k: Int): Long {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Elegance of a K-Length Subsequence
// Difficulty: Hard
// Tags: array, greedy, hash, sort, stack, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn find_maximum_elegance(items: Vec<Vec<i32>>, k: i32) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} items
# @param {Integer} k
# @return {Integer}
def find_maximum_elegance(items, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $items
* @param Integer $k
* @return Integer
*/
function findMaximumElegance($items, $k) {

}
}
```

Dart Solution:

```
class Solution {
int findMaximumElegance(List<List<int>> items, int k) {

}
}
```

Scala Solution:

```
object Solution {
def findMaximumElegance(items: Array[Array[Int]], k: Int): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_maximum_elegance(items :: [[integer]], k :: integer) :: integer
def find_maximum_elegance(items, k) do

end
end
```

Erlang Solution:

```
-spec find_maximum_elegance(Items :: [[integer()]], K :: integer()) ->
integer().
find_maximum_elegance(Items, K) ->
```

```
.
```

Racket Solution:

```
(define/contract (find-maximum-elegance items k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?)
)
```