Problem 2070: Most Beautiful Item for Each Query

Problem Information

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

Problem Description

You are given a 2D integer array

items

where

items[i] = [price

i

, beauty

i

]

denotes the

price

and

beauty

of an item respectively.

You are also given a

0-indexed

integer array

queries

. For each

queries[j]

, you want to determine the

maximum beauty

of an item whose

price

is

less than or equal

to

queries[j]

. If no such item exists, then the answer to this query is

0

.

Return

an array

answer

of the same length as

queries

where

answer[j]

is the answer to the

j

th

query

.

Example 1:

Input:

items = [[1,2],[3,2],[2,4],[5,6],[3,5]], queries = [1,2,3,4,5,6]

Output:

[2,4,5,5,6,6]

Explanation:

- For queries[0]=1, [1,2] is the only item which has price <= 1. Hence, the answer for this query is 2. - For queries[1]=2, the items which can be considered are [1,2] and [2,4]. The maximum beauty among them is 4. - For queries[2]=3 and queries[3]=4, the items which can be considered are [1,2], [3,2], [2,4], and [3,5]. The maximum beauty among them is 5. - For queries[4]=5 and queries[5]=6, all items can be considered. Hence, the answer for them is the maximum beauty of all items, i.e., 6.

Example 2:

Input:

items = [[1,2],[1,2],[1,3],[1,4]], queries = [1]

Output:

[4]

Explanation:

The price of every item is equal to 1, so we choose the item with the maximum beauty 4. Note that multiple items can have the same price and/or beauty.

Example 3:

Input:

items = [[10,1000]], queries = [5]

Output:

[0]

Explanation:

No item has a price less than or equal to 5, so no item can be chosen. Hence, the answer to the query is 0.

Constraints:

1 <= items.length, queries.length <= 10

5

items[i].length == 2

1 <= price

i

, beauty

i

, queries[j] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
vector<int> maximumBeauty(vector<vector<int>>& items, vector<int>& queries) {

}
};
```

Java:

```
class Solution {
public int[] maximumBeauty(int[][] items, int[] queries) {

}
}
```

Python3:

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

Python:

```
class Solution(object):
def maximumBeauty(self, items, queries):
"""
```

```
:type items: List[List[int]]
:type queries: List[int]
:rtype: List[int]
"""
```

JavaScript:

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

};
```

TypeScript:

```
function maximumBeauty(items: number[][], queries: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] MaximumBeauty(int[][] items, int[] queries) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maximumBeauty(int** items, int itemsSize, int* itemsColSize, int*
queries, int queriesSize, int* returnSize) {

}
```

Go:

```
func maximumBeauty(items [][]int, queries []int) []int {

}
```

Kotlin:

```
class Solution {
fun maximumBeauty(items: Array<IntArray>, queries: IntArray): IntArray {

}
}
```

Swift:

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

}
}
```

Rust:

```
impl Solution {
pub fn maximum_beauty(items: Vec<Vec<i32>>, queries: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[][]} items
# @param {Integer[]} queries
# @return {Integer[]}
def maximum_beauty(items, queries)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $items
```

```
* @param Integer[] $queries
* @return Integer[]
*/
function maximumBeauty($items, $queries) {

}
}
```

Dart:

```
class Solution {
List<int> maximumBeauty(List<List<int>> items, List<int> queries) {

}
}
```

Scala:

```
object Solution {
def maximumBeauty(items: Array[Array[Int]], queries: Array[Int]): Array[Int]
= {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_beauty(items :: [[integer]], queries :: [integer]) :: [integer]
def maximum_beauty(items, queries) do

end
end
```

Erlang:

```
-spec maximum_beauty(Items :: [[integer()]], Queries :: [integer()]) ->
[integer()].
maximum_beauty(Items, Queries) ->
.
```

Racket:

```
(define/contract (maximum-beauty items queries)
(-> (listof (listof exact-integer?)) (listof exact-integer?) (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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:
vector<int> maximumBeauty(vector<vector<int>>& items, vector<int>& queries) {

}
};
```

Java Solution:

```
/**
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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[] maximumBeauty(int[][] items, int[] queries) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Most Beautiful Item for Each Query
Difficulty: Medium
Tags: array, sort, search

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 maximumBeauty(self, items: List[List[int]], queries: List[int]) ->
List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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[][]} items
* @param {number[]} queries
* @return {number[]}
*/
var maximumBeauty = function(items, queries) {

};
```

TypeScript Solution:

```
/**
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 maximumBeauty(items: number[][], queries: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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[] MaximumBeauty(int[][] items, int[] queries) {
```

```
}
}
```

C Solution:

```
/*
* Problem: Most Beautiful Item for Each Query
* Difficulty: Medium
* Tags: array, sort, search
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maximumBeauty(int** items, int itemsSize, int* itemsColSize, int*
queries, int queriesSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Most Beautiful Item for Each Query
// Difficulty: Medium
// Tags: array, sort, search
//
// 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 maximumBeauty(items [][]int, queries []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun maximumBeauty(items: Array<IntArray>, queries: IntArray): IntArray {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Most Beautiful Item for Each Query
// Difficulty: Medium
// Tags: array, sort, search
//
// 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 maximum_beauty(items: Vec<Vec<i32>>, queries: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} items
# @param {Integer[]} queries
# @return {Integer[]}
def maximum_beauty(items, queries)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $items
* @param Integer[] $queries
* @return Integer[]
*/
function maximumBeauty($items, $queries) {

}
}
```

Dart Solution:

```
class Solution {
List<int> maximumBeauty(List<List<int>> items, List<int> queries) {

}
}
```

Scala Solution:

```
object Solution {
def maximumBeauty(items: Array[Array[Int]], queries: Array[Int]): Array[Int]
= {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_beauty(items :: [[integer]], queries :: [integer]) :: [integer]
def maximum_beauty(items, queries) do

end
end
```

Erlang Solution:

```
-spec maximum_beauty(Items :: [[integer()]], Queries :: [integer()]) ->
[integer()].
```

```
maximum_beauty(Items, Queries) ->
.
```

Racket Solution:

```
(define/contract (maximum-beauty items queries)
(-> (listof (listof exact-integer?)) (listof exact-integer?) (listof
exact-integer?))
)
```