

Problem 879: Profitable Schemes

Problem Information

Difficulty: Hard

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

There is a group of

n

members, and a list of various crimes they could commit. The

i

th

crime generates a

profit[i]

and requires

group[i]

members to participate in it. If a member participates in one crime, that member can't participate in another crime.

Let's call a

profitable scheme

any subset of these crimes that generates at least

minProfit

profit, and the total number of members participating in that subset of crimes is at most

n

Return the number of schemes that can be chosen. Since the answer may be very large,

return it modulo

10

9

+ 7

Example 1:

Input:

$n = 5$, $\text{minProfit} = 3$, $\text{group} = [2,2]$, $\text{profit} = [2,3]$

Output:

2

Explanation:

To make a profit of at least 3, the group could either commit crimes 0 and 1, or just crime 1. In total, there are 2 schemes.

Example 2:

Input:

$n = 10$, $\text{minProfit} = 5$, $\text{group} = [2,3,5]$, $\text{profit} = [6,7,8]$

Output:

7

Explanation:

To make a profit of at least 5, the group could commit any crimes, as long as they commit one. There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2), and (0,1,2).

Constraints:

$1 \leq n \leq 100$

$0 \leq \text{minProfit} \leq 100$

$1 \leq \text{group.length} \leq 100$

$1 \leq \text{group}[i] \leq 100$

$\text{profit.length} == \text{group.length}$

$0 \leq \text{profit}[i] \leq 100$

Code Snippets

C++:

```
class Solution {
public:
    int profitableSchemes(int n, int minProfit, vector<int>& group, vector<int>& profit) {
        }
    };
}
```

Java:

```
class Solution {  
    public int profitableSchemes(int n, int minProfit, int[] group, int[] profit)  
    {  
  
    }  
}
```

Python3:

```
class Solution:  
    def profitableSchemes(self, n: int, minProfit: int, group: List[int], profit: List[int]) -> int:
```

Python:

```
class Solution(object):  
    def profitableSchemes(self, n, minProfit, group, profit):  
        """  
        :type n: int  
        :type minProfit: int  
        :type group: List[int]  
        :type profit: List[int]  
        :rtype: int  
        """
```

JavaScript:

```
/**  
 * @param {number} n  
 * @param {number} minProfit  
 * @param {number[]} group  
 * @param {number[]} profit  
 * @return {number}  
 */  
var profitableSchemes = function(n, minProfit, group, profit) {  
  
};
```

TypeScript:

```
function profitableSchemes(n: number, minProfit: number, group: number[],  
profit: number[]): number {
```

```
};
```

C#:

```
public class Solution {  
    public int ProfitableSchemes(int n, int minProfit, int[] group, int[] profit)  
    {  
  
    }  
}
```

C:

```
int profitableSchemes(int n, int minProfit, int* group, int groupSize, int*  
profit, int profitSize) {  
  
}
```

Go:

```
func profitableSchemes(n int, minProfit int, group []int, profit []int) int {  
  
}
```

Kotlin:

```
class Solution {  
    fun profitableSchemes(n: Int, minProfit: Int, group: IntArray, profit:  
        IntArray): Int {  
  
    }  
}
```

Swift:

```
class Solution {  
    func profitableSchemes(_ n: Int, _ minProfit: Int, _ group: [Int], _ profit:  
        [Int]) -> Int {  
  
    }  
}
```

Rust:

```
impl Solution {
    pub fn profitable_schemes(n: i32, min_profit: i32, group: Vec<i32>, profit: Vec<i32>) -> i32 {
        }
    }
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer} min_profit
# @param {Integer[]} group
# @param {Integer[]} profit
# @return {Integer}
def profitable_schemes(n, min_profit, group, profit)

end
```

PHP:

```
class Solution {

    /**
     * @param Integer $n
     * @param Integer $minProfit
     * @param Integer[] $group
     * @param Integer[] $profit
     * @return Integer
     */
    function profitableSchemes($n, $minProfit, $group, $profit) {

    }
}
```

Dart:

```
class Solution {
    int profitableSchemes(int n, int minProfit, List<int> group, List<int> profit) {
    }
}
```

```
}
```

Scala:

```
object Solution {  
    def profitableSchemes(n: Int, minProfit: Int, group: Array[Int], profit: Array[Int]): Int = {  
  
    }  
}
```

Elixir:

```
defmodule Solution do  
  @spec profitable_schemes(n :: integer, min_profit :: integer, group :: [integer], profit :: [integer]) :: integer  
  def profitable_schemes(n, min_profit, group, profit) do  
  
  end  
end
```

Erlang:

```
-spec profitable_schemes(N :: integer(), MinProfit :: integer(), Group :: [integer()], Profit :: [integer()]) -> integer().  
profitable_schemes(N, MinProfit, Group, Profit) ->  
. . .
```

Racket:

```
(define/contract (profitable-schemes n minProfit group profit)  
  (-> exact-integer? exact-integer? (listof exact-integer?) (listof  
    exact-integer?) exact-integer?)  
)
```

Solutions

C++ Solution:

```

/*
 * Problem: Profitable Schemes
 * 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 profitableSchemes(int n, int minProfit, vector<int>& group, vector<int>& profit) {
        }
    };
}

```

Java Solution:

```

/**
 * Problem: Profitable Schemes
 * 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 profitableSchemes(int n, int minProfit, int[] group, int[] profit)
{
}
}

```

Python3 Solution:

```

"""
Problem: Profitable Schemes
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 profitableSchemes(self, n: int, minProfit: int, group: List[int], profit: List[int]) -> int:  
        # TODO: Implement optimized solution  
        pass
```

Python Solution:

```
class Solution(object):  
    def profitableSchemes(self, n, minProfit, group, profit):  
        """  
        :type n: int  
        :type minProfit: int  
        :type group: List[int]  
        :type profit: List[int]  
        :rtype: int  
        """
```

JavaScript Solution:

```
/**  
 * Problem: Profitable Schemes  
 * 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} n  
 * @param {number} minProfit  
 * @param {number[]} group
```

```

* @param {number[]} profit
* @return {number}
*/
var profitableSchemes = function(n, minProfit, group, profit) {
};


```

TypeScript Solution:

```

/** 
 * Problem: Profitable Schemes
 * 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 profitableSchemes(n: number, minProfit: number, group: number[], profit: number[]): number {
}


```

C# Solution:

```

/*
 * Problem: Profitable Schemes
 * 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 ProfitableSchemes(int n, int minProfit, int[] group, int[] profit)
    {
    }
}
```

```
}
```

C Solution:

```
/*
 * Problem: Profitable Schemes
 * 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 profitableSchemes(int n, int minProfit, int* group, int groupSize, int*
profit, int profitSize) {

}
```

Go Solution:

```
// Problem: Profitable Schemes
// 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 profitableSchemes(n int, minProfit int, group []int, profit []int) int {

}
```

Kotlin Solution:

```
class Solution {
    fun profitableSchemes(n: Int, minProfit: Int, group: IntArray, profit:
IntArray): Int {
        }
    }
}
```

Swift Solution:

```
class Solution {  
    func profitableSchemes(_ n: Int, _ minProfit: Int, _ group: [Int], _ profit: [Int]) -> Int {  
  
    }  
}
```

Rust Solution:

```
// Problem: Profitable Schemes  
// 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 profitable_schemes(n: i32, min_profit: i32, group: Vec<i32>, profit: Vec<i32>) -> i32 {  
  
    }  
}
```

Ruby Solution:

```
# @param {Integer} n  
# @param {Integer} min_profit  
# @param {Integer[]} group  
# @param {Integer[]} profit  
# @return {Integer}  
def profitable_schemes(n, min_profit, group, profit)  
  
end
```

PHP Solution:

```
class Solution {  
  
    /**
```

```

* @param Integer $n
* @param Integer $minProfit
* @param Integer[] $group
* @param Integer[] $profit
* @return Integer
*/
function profitableSchemes($n, $minProfit, $group, $profit) {

}
}

```

Dart Solution:

```

class Solution {
int profitableSchemes(int n, int minProfit, List<int> group, List<int>
profit) {

}
}

```

Scala Solution:

```

object Solution {
def profitableSchemes(n: Int, minProfit: Int, group: Array[Int], profit:
Array[Int]): Int = {

}
}

```

Elixir Solution:

```

defmodule Solution do
@spec profitable_schemes(n :: integer, min_profit :: integer, group :: [integer],
profit :: [integer]) :: integer
def profitable_schemes(n, min_profit, group, profit) do

end
end

```

Erlang Solution:

```
-spec profitable_schemes(N :: integer(), MinProfit :: integer(), Group :: [integer()], Profit :: [integer()]) -> integer().  
profitable_schemes(N, MinProfit, Group, Profit) ->  
. 
```

Racket Solution:

```
(define/contract (profitable-schemes n minProfit group profit)  
(-> exact-integer? exact-integer? (listof exact-integer?) (listof  
exact-integer?) exact-integer?)  
) 
```