Problem 638: Shopping Offers

Problem Information

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

Problem Description

In LeetCode Store, there are

n

items to sell. Each item has a price. However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.

You are given an integer array

price

where

price[i]

is the price of the

i

th

item, and an integer array

needs

where

needs[i]

is the number of pieces of the

i

th

item you want to buy.

You are also given an array

special

where

special[i]

is of size

n + 1

where

special[i][j]

is the number of pieces of the

j

th

item in the

i

th

offer and

special[i][n]

(i.e., the last integer in the array) is the price of the

i

th

offer.

Return

the lowest price you have to pay for exactly certain items as given, where you could make optimal use of the special offers

. You are not allowed to buy more items than you want, even if that would lower the overall price. You could use any of the special offers as many times as you want.

Example 1:

Input:

price = [2,5], special = [[3,0,5],[1,2,10]], needs = [3,2]

Output:

14

Explanation:

There are two kinds of items, A and B. Their prices are $2 and $5 respectively. In special offer 1, you can pay $5 for 3A and 0B In special offer 2, you can pay $10 for 1A and 2B. You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $4 for 2A.

Example 2:

Input:

price = [2,3,4], special = [[1,1,0,4],[2,2,1,9]], needs = [1,2,1]

Output:

11

Explanation:

The price of A is $2, and $3 for B, $4 for C. You may pay $4 for 1A and 1B, and $9 for 2A ,2B and 1C. You need to buy 1A ,2B and 1C, so you may pay $4 for 1A and 1B (special offer #1), and $3 for 1B, $4 for 1C. You cannot add more items, though only $9 for 2A ,2B and 1C.

Constraints:

n == price.length == needs.length

1 <= n <= 6

0 <= price[i], needs[i] <= 10

1 <= special.length <= 100

special[i].length == n + 1

0 <= special[i][j] <= 50

The input is generated that at least one of

special[i][j]

is non-zero for

0 <= j <= n - 1

.

Code Snippets

C++:

```
class Solution {
public:
int shoppingOffers(vector<int>& price, vector<vector<int>>& special,
vector<int>& needs) {

}
};
```

Java:

```
class Solution {
public int shoppingOffers(List<Integer> price, List<List<Integer>> special,
List<Integer> needs) {

}
}
```

Python3:

```
class Solution:
def shoppingOffers(self, price: List[int], special: List[List[int]], needs:
List[int]) -> int:
```

Python:

```
class Solution(object):
def shoppingOffers(self, price, special, needs):
"""
:type price: List[int]
:type special: List[List[int]]
:type needs: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} price
* @param {number[][]} special
* @param {number[]} needs
* @return {number}
```

```
*/
var shoppingOffers = function(price, special, needs) {

};
```

TypeScript:

```
function shoppingOffers(price: number[], special: number[][], needs:
number[]): number {

};
```

C#:

```
public class Solution {
public int ShoppingOffers(IList<int> price, IList<IList<int>> special,
IList<int> needs) {

}
}
```

C:

```
int shoppingOffers(int* price, int priceSize, int** special, int specialSize,
int* specialColSize, int* needs, int needsSize) {

}
```

Go:

```
func shoppingOffers(price []int, special [][]int, needs []int) int {

}
```

Kotlin:

```
class Solution {
fun shoppingOffers(price: List<Int>, special: List<List<Int>>, needs:
List<Int>): Int {

}
}
```

Swift:

```
class Solution {
func shoppingOffers(_ price: [Int], _ special: [[Int]], _ needs: [Int]) ->
Int {

}
}
```

Rust:

```
impl Solution {
pub fn shopping_offers(price: Vec<i32>, special: Vec<Vec<i32>>, needs:
Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} price
# @param {Integer[][]} special
# @param {Integer[]} needs
# @return {Integer}
def shopping_offers(price, special, needs)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $price
* @param Integer[][] $special
* @param Integer[] $needs
* @return Integer
*/
function shoppingOffers($price, $special, $needs) {

}
}
```

Dart:

```
class Solution {
int shoppingOffers(List<int> price, List<List<int>> special, List<int> needs)
{

}
}
```

Scala:

```
object Solution {
def shoppingOffers(price: List[Int], special: List[List[Int]], needs:
List[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec shopping_offers(price :: [integer], special :: [[integer]], needs ::
[integer]) :: integer
def shopping_offers(price, special, needs) do

end
end
```

Erlang:

```
-spec shopping_offers(Price :: [integer()], Special :: [[integer()]], Needs
:: [integer()]) -> integer().
shopping_offers(Price, Special, Needs) ->
.
```

Racket:

```
(define/contract (shopping-offers price special needs)
(-> (listof exact-integer?) (listof (listof exact-integer?)) (listof
exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Shopping Offers
* Difficulty: Medium
* 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 shoppingOffers(vector<int>& price, vector<vector<int>>& special,
vector<int>& needs) {

}
};
```

Java Solution:

```
/**
* Problem: Shopping Offers
* Difficulty: Medium
* 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 shoppingOffers(List<Integer> price, List<List<Integer>> special,
List<Integer> needs) {

}
}
```

Python3 Solution:

```
"""
Problem: Shopping Offers
Difficulty: Medium
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 shoppingOffers(self, price: List[int], special: List[List[int]], needs:
List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def shoppingOffers(self, price, special, needs):
"""
:type price: List[int]
:type special: List[List[int]]
:type needs: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Shopping Offers
* Difficulty: Medium
* 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[]} price
* @param {number[][]} special
```

```
* @param {number[]} needs
* @return {number}
*/
var shoppingOffers = function(price, special, needs) {

};
```

TypeScript Solution:

```
/**
* Problem: Shopping Offers
* Difficulty: Medium
* 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 shoppingOffers(price: number[], special: number[][], needs:
number[]): number {

};
```

C# Solution:

```
/*
* Problem: Shopping Offers
* Difficulty: Medium
* 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 ShoppingOffers(IList<int> price, IList<IList<int>> special,
IList<int> needs) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Shopping Offers
* Difficulty: Medium
* 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 shoppingOffers(int* price, int priceSize, int** special, int specialSize,
int* specialColSize, int* needs, int needsSize) {

}
```

Go Solution:

```
// Problem: Shopping Offers
// Difficulty: Medium
// 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 shoppingOffers(price []int, special [][]int, needs []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun shoppingOffers(price: List<Int>, special: List<List<Int>>, needs:
List<Int>): Int {

}
}
```

Swift Solution:

```
class Solution {
func shoppingOffers(_ price: [Int], _ special: [[Int]], _ needs: [Int]) ->
Int {

}
}
```

Rust Solution:

```
// Problem: Shopping Offers
// Difficulty: Medium
// 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 shopping_offers(price: Vec<i32>, special: Vec<Vec<i32>>, needs:
Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} price
# @param {Integer[][]} special
# @param {Integer[]} needs
# @return {Integer}
def shopping_offers(price, special, needs)

end
```

PHP Solution:

```
class Solution {

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

```
* @param Integer[][] $special
* @param Integer[] $needs
* @return Integer
*/
function shoppingOffers($price, $special, $needs) {

}
}
```

Dart Solution:

```
class Solution {
int shoppingOffers(List<int> price, List<List<int>> special, List<int> needs)
{

}
}
```

Scala Solution:

```
object Solution {
def shoppingOffers(price: List[Int], special: List[List[Int]], needs:
List[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec shopping_offers(price :: [integer], special :: [[integer]], needs ::
[integer]) :: integer
def shopping_offers(price, special, needs) do

end
end
```

Erlang Solution:

```
-spec shopping_offers(Price :: [integer()], Special :: [[integer()]], Needs
:: [integer()]) -> integer().
```

```
shopping_offers(Price, Special, Needs) ->
.
```

Racket Solution:

```
(define/contract (shopping-offers price special needs)
(-> (listof exact-integer?) (listof (listof exact-integer?)) (listof
exact-integer?) exact-integer?)
)
```