Problem 1357: Apply Discount Every n Orders

Problem Information

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

Problem Description

There is a supermarket that is frequented by many customers. The products sold at the supermarket are represented as two parallel integer arrays

products

and

prices

, where the

i

th

product has an ID of

products[i]

and a price of

prices[i]

.

When a customer is paying, their bill is represented as two parallel integer arrays

product

and

amount

, where the

j

th

product they purchased has an ID of

product[j]

, and

amount[j]

is how much of the product they bought. Their subtotal is calculated as the sum of each

amount[j] * (price of the j

th

product)

.

The supermarket decided to have a sale. Every

n

th

customer paying for their groceries will be given a

percentage discount

. The discount amount is given by

discount

, where they will be given

discount

percent off their subtotal. More formally, if their subtotal is

bill

, then they would actually pay

bill * ((100 - discount) / 100)

.

Implement the

Cashier

class:

Cashier(int n, int discount, int[] products, int[] prices)

Initializes the object with

n

, the

discount

, and the

products

and their

prices

.

double getBill(int[] product, int[] amount)

Returns the final total of the bill with the discount applied (if any). Answers within

10

-5

of the actual value will be accepted.

Example 1:

Input

["Cashier","getBill","getBill","getBill","getBill","getBill","getBill","getBill"] [[3,50,[1,2,3,4,5,6,7],[100,200,300,400,300,200,100]],[[1,2],[1,2]],[[3,7],[10,10]],[[1,2,3,4,5,6,7],[1,1,1,1,1,1,1]],[[4],[10]],[[7,3],[10,10]],[[7,5,3,1,6,4,2],[10,10,10,9,9,9,7]],[[2,3,5],[5,3,2]]]

Output

[null,500.0,4000.0,800.0,4000.0,4000.0,7350.0,2500.0]

Explanation

Cashier cashier = new Cashier(3,50,[1,2,3,4,5,6,7],[100,200,300,400,300,200,100]); cashier.getBill([1,2],[1,2]); // return 500.0. 1

st

customer, no discount. // bill = 1 * 100 + 2 * 200 = 500. cashier.getBill([3,7],[10,10]); // return 4000.0. 2

nd

customer, no discount. // bill = 10 * 300 + 10 * 100 = 4000.
cashier.getBill([1,2,3,4,5,6,7],[1,1,1,1,1,1,1]); // return 800.0. 3

rd

customer, 50% discount. // Original bill = 1600 // Actual bill = 1600 * ((100 - 50) / 100) = 800.
cashier.getBill([4],[10]); // return 4000.0. 4

th

customer, no discount. cashier.getBill([7,3],[10,10]); // return 4000.0. 5

th

customer, no discount. cashier.getBill([7,5,3,1,6,4,2],[10,10,10,9,9,9,7]); // return 7350.0. 6

th

customer, 50% discount. // Original bill = 14700, but with // Actual bill = 14700 * ((100 - 50) / 100) = 7350. cashier.getBill([2,3,5],[5,3,2]); // return 2500.0. 7

th

customer, no discount.

Constraints:

1 <= n <= 10

4

0 <= discount <= 100

1 <= products.length <= 200

prices.length == products.length

1 <= products[i] <= 200

1 <= prices[i] <= 1000

The elements in

products

are

unique

.

1 <= product.length <= products.length

amount.length == product.length

product[j]

exists in

products

.

1 <= amount[j] <= 1000

The elements of

product

are

unique

.

At most

1000

calls will be made to

getBill

.

Answers within

10

-5

of the actual value will be accepted.

Code Snippets

C++:

```
class Cashier {
public:
Cashier(int n, int discount, vector<int>& products, vector<int>& prices) {

}

double getBill(vector<int> product, vector<int> amount) {

}
};

/**
* Your Cashier object will be instantiated and called as such:
* Cashier* obj = new Cashier(n, discount, products, prices);
* double param_1 = obj->getBill(product,amount);
*/
```

Java:

```
class Cashier {

public Cashier(int n, int discount, int[] products, int[] prices) {
```

```
}

public double getBill(int[] product, int[] amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* Cashier obj = new Cashier(n, discount, products, prices);
* double param_1 = obj.getBill(product,amount);
*/
```

Python3:

```
class Cashier:

def __init__(self, n: int, discount: int, products: List[int], prices:
List[int]):


def getBill(self, product: List[int], amount: List[int]) -> float:



# Your Cashier object will be instantiated and called as such:
# obj = Cashier(n, discount, products, prices)
# param_1 = obj.getBill(product,amount)
```

Python:

```
class Cashier(object):

def __init__(self, n, discount, products, prices):
"""
:type n: int
:type discount: int
:type products: List[int]
:type prices: List[int]
"""
```

```
def getBill(self, product, amount):
"""
:type product: List[int]
:type amount: List[int]
:rtype: float
"""



# Your Cashier object will be instantiated and called as such:
# obj = Cashier(n, discount, products, prices)
# param_1 = obj.getBill(product,amount)
```

JavaScript:

```
/**
* @param {number} n
* @param {number} discount
* @param {number[]} products
* @param {number[]} prices
*/
var Cashier = function(n, discount, products, prices) {

};

/**
* @param {number[]} product
* @param {number[]} amount
* @return {number}
*/
Cashier.prototype.getBill = function(product, amount) {

};

/**
* Your Cashier object will be instantiated and called as such:
* var obj = new Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

TypeScript:

```
class Cashier {
constructor(n: number, discount: number, products: number[], prices:
number[]) {

}

getBill(product: number[], amount: number[]): number {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* var obj = new Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

C#:

```
public class Cashier {

public Cashier(int n, int discount, int[] products, int[] prices) {

}

public double GetBill(int[] product, int[] amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* Cashier obj = new Cashier(n, discount, products, prices);
* double param_1 = obj.GetBill(product,amount);
*/
```

C:

```

```

```
typedef struct {

} Cashier;


Cashier* cashierCreate(int n, int discount, int* products, int productsSize,
int* prices, int pricesSize) {

}

double cashierGetBill(Cashier* obj, int* product, int productSize, int*
amount, int amountSize) {

}

void cashierFree(Cashier* obj) {

}

/**
* Your Cashier struct will be instantiated and called as such:
* Cashier* obj = cashierCreate(n, discount, products, productsSize, prices,
pricesSize);
* double param_1 = cashierGetBill(obj, product, productSize, amount,
amountSize);

* cashierFree(obj);
*/
```

Go:

```
type Cashier struct {

}


func Constructor(n int, discount int, products []int, prices []int) Cashier {

}


func (this *Cashier) GetBill(product []int, amount []int) float64 {
```

```
}


/**
* Your Cashier object will be instantiated and called as such:
* obj := Constructor(n, discount, products, prices);
* param_1 := obj.GetBill(product,amount);
*/
```

Kotlin:

```
class Cashier(n: Int, discount: Int, products: IntArray, prices: IntArray) {

fun getBill(product: IntArray, amount: IntArray): Double {

}

}

/**
* Your Cashier object will be instantiated and called as such:
* var obj = Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

Swift:

```
class Cashier {

init(_ n: Int, _ discount: Int, _ products: [Int], _ prices: [Int]) {

}

func getBill(_ product: [Int], _ amount: [Int]) -> Double {

}
}

/**
* Your Cashier object will be instantiated and called as such:
```

```
* let obj = Cashier(n, discount, products, prices)
* let ret_1: Double = obj.getBill(product, amount)
*/
```

Rust:

```
struct Cashier {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Cashier {

fn new(n: i32, discount: i32, products: Vec<i32>, prices: Vec<i32>) -> Self {

}

fn get_bill(&self, product: Vec<i32>, amount: Vec<i32>) -> f64 {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* let obj = Cashier::new(n, discount, products, prices);
* let ret_1: f64 = obj.get_bill(product, amount);
*/
```

Ruby:

```
class Cashier

=begin
:type n: Integer
:type discount: Integer
:type products: Integer[]
:type prices: Integer[]
=end
```

```
def initialize(n, discount, products, prices)

end


=begin
:type product: Integer[]
:type amount: Integer[]
:rtype: Float
=end
def get_bill(product, amount)

end



end

# Your Cashier object will be instantiated and called as such:
# obj = Cashier.new(n, discount, products, prices)
# param_1 = obj.get_bill(product, amount)
```

PHP:

```
class Cashier {
/**
* @param Integer $n
* @param Integer $discount
* @param Integer[] $products
* @param Integer[] $prices
*/
function __construct($n, $discount, $products, $prices) {

}

/**
* @param Integer[] $product
* @param Integer[] $amount
* @return Float
*/
function getBill($product, $amount) {

}
```

```
}

/**
* Your Cashier object will be instantiated and called as such:
* $obj = Cashier($n, $discount, $products, $prices);
* $ret_1 = $obj->getBill($product, $amount);
*/
```

Dart:

```
class Cashier {

Cashier(int n, int discount, List<int> products, List<int> prices) {

}

double getBill(List<int> product, List<int> amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* Cashier obj = Cashier(n, discount, products, prices);
* double param1 = obj.getBill(product,amount);
*/
```

Scala:

```
class Cashier(_n: Int, _discount: Int, _products: Array[Int], _prices:
Array[Int]) {

def getBill(product: Array[Int], amount: Array[Int]): Double = {

}

}

/**
* Your Cashier object will be instantiated and called as such:
* val obj = new Cashier(n, discount, products, prices)
* val param_1 = obj.getBill(product,amount)
```

```
*/
```

Elixir:

```
defmodule Cashier do
@spec init_(n :: integer, discount :: integer, products :: [integer], prices
:: [integer]) :: any
def init_(n, discount, products, prices) do

end

@spec get_bill(product :: [integer], amount :: [integer]) :: float
def get_bill(product, amount) do

end
end

# Your functions will be called as such:
# Cashier.init_(n, discount, products, prices)
# param_1 = Cashier.get_bill(product, amount)

# Cashier.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang:

```
-spec cashier_init_(N :: integer(), Discount :: integer(), Products ::
[integer()], Prices :: [integer()]) -> any().
cashier_init_(N, Discount, Products, Prices) ->
.

-spec cashier_get_bill(Product :: [integer()], Amount :: [integer()]) ->
float().
cashier_get_bill(Product, Amount) ->
.


%% Your functions will be called as such:
%% cashier_init_(N, Discount, Products, Prices),
%% Param_1 = cashier_get_bill(Product, Amount),

%% cashier_init_ will be called before every test case, in which you can do
```

```
some necessary initializations.
```

Racket:

```
(define cashier%
(class object%
(super-new)

; n : exact-integer?
; discount : exact-integer?
; products : (listof exact-integer?)
; prices : (listof exact-integer?)
(init-field
n
discount
products
prices)

; get-bill : (listof exact-integer?) (listof exact-integer?) -> flonum?
(define/public (get-bill product amount)
)))

;; Your cashier% object will be instantiated and called as such:
;; (define obj (new cashier% [n n] [discount discount] [products products]
[prices prices]))
;; (define param_1 (send obj get-bill product amount))
```

Solutions

C++ Solution:

```
/*
* Problem: Apply Discount Every n Orders
* Difficulty: Medium
* Tags: array, hash
*
* 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 Cashier {
public:
Cashier(int n, int discount, vector<int>& products, vector<int>& prices) {

}

double getBill(vector<int> product, vector<int> amount) {

}
};

/**
* Your Cashier object will be instantiated and called as such:
* Cashier* obj = new Cashier(n, discount, products, prices);
* double param_1 = obj->getBill(product,amount);
*/
```

Java Solution:

```
/**
* Problem: Apply Discount Every n Orders
* Difficulty: Medium
* Tags: array, hash
*
* 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 Cashier {

public Cashier(int n, int discount, int[] products, int[] prices) {

}

public double getBill(int[] product, int[] amount) {

}
}

/**
```

```
* Your Cashier object will be instantiated and called as such:
* Cashier obj = new Cashier(n, discount, products, prices);
* double param_1 = obj.getBill(product,amount);
*/
```

Python3 Solution:

```
"""
Problem: Apply Discount Every n Orders
Difficulty: Medium
Tags: array, hash

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 Cashier:

def __init__(self, n: int, discount: int, products: List[int], prices:
List[int]):



def getBill(self, product: List[int], amount: List[int]) -> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Cashier(object):

def __init__(self, n, discount, products, prices):
"""
:type n: int
:type discount: int
:type products: List[int]
:type prices: List[int]
"""



def getBill(self, product, amount):
```

```
"""
:type product: List[int]
:type amount: List[int]
:rtype: float
"""



# Your Cashier object will be instantiated and called as such:
# obj = Cashier(n, discount, products, prices)
# param_1 = obj.getBill(product,amount)
```

JavaScript Solution:

```
/**
* Problem: Apply Discount Every n Orders
* Difficulty: Medium
* Tags: array, hash
*
* 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} n
* @param {number} discount
* @param {number[]} products
* @param {number[]} prices
*/
var Cashier = function(n, discount, products, prices) {

};

/**
* @param {number[]} product
* @param {number[]} amount
* @return {number}
*/
Cashier.prototype.getBill = function(product, amount) {
```

```
};

/**
* Your Cashier object will be instantiated and called as such:
* var obj = new Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

TypeScript Solution:

```
/**
* Problem: Apply Discount Every n Orders
* Difficulty: Medium
* Tags: array, hash
*
* 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 Cashier {
constructor(n: number, discount: number, products: number[], prices:
number[]) {

}

getBill(product: number[], amount: number[]): number {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* var obj = new Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

C# Solution:

```
/*
* Problem: Apply Discount Every n Orders
```

```
* Difficulty: Medium
* Tags: array, hash
*
* 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 Cashier {

public Cashier(int n, int discount, int[] products, int[] prices) {

}

public double GetBill(int[] product, int[] amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* Cashier obj = new Cashier(n, discount, products, prices);
* double param_1 = obj.GetBill(product,amount);
*/
```

C Solution:

```
/*
* Problem: Apply Discount Every n Orders
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/




typedef struct {
```

```
} Cashier;


Cashier* cashierCreate(int n, int discount, int* products, int productsSize,
int* prices, int pricesSize) {

}

double cashierGetBill(Cashier* obj, int* product, int productSize, int*
amount, int amountSize) {

}

void cashierFree(Cashier* obj) {

}

/**
* Your Cashier struct will be instantiated and called as such:
* Cashier* obj = cashierCreate(n, discount, products, productsSize, prices,
pricesSize);
* double param_1 = cashierGetBill(obj, product, productSize, amount,
amountSize);

* cashierFree(obj);
*/
```

Go Solution:

```
// Problem: Apply Discount Every n Orders
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type Cashier struct {

}
```

```
func Constructor(n int, discount int, products []int, prices []int) Cashier {

}


func (this *Cashier) GetBill(product []int, amount []int) float64 {

}


/**
* Your Cashier object will be instantiated and called as such:
* obj := Constructor(n, discount, products, prices);
* param_1 := obj.GetBill(product,amount);
*/
```

Kotlin Solution:

```
class Cashier(n: Int, discount: Int, products: IntArray, prices: IntArray) {

fun getBill(product: IntArray, amount: IntArray): Double {

}

}

/**
* Your Cashier object will be instantiated and called as such:
* var obj = Cashier(n, discount, products, prices)
* var param_1 = obj.getBill(product,amount)
*/
```

Swift Solution:

```
class Cashier {

init(_ n: Int, _ discount: Int, _ products: [Int], _ prices: [Int]) {
```

```
}

func getBill(_ product: [Int], _ amount: [Int]) -> Double {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* let obj = Cashier(n, discount, products, prices)
* let ret_1: Double = obj.getBill(product, amount)
*/
```

Rust Solution:

```
// Problem: Apply Discount Every n Orders
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct Cashier {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Cashier {

fn new(n: i32, discount: i32, products: Vec<i32>, prices: Vec<i32>) -> Self {

}

fn get_bill(&self, product: Vec<i32>, amount: Vec<i32>) -> f64 {

}
```

```
}

/**
* Your Cashier object will be instantiated and called as such:
* let obj = Cashier::new(n, discount, products, prices);
* let ret_1: f64 = obj.get_bill(product, amount);
*/
```

Ruby Solution:

```
class Cashier

=begin
:type n: Integer
:type discount: Integer
:type products: Integer[]
:type prices: Integer[]
=end
def initialize(n, discount, products, prices)

end


=begin
:type product: Integer[]
:type amount: Integer[]
:rtype: Float
=end
def get_bill(product, amount)

end


end

# Your Cashier object will be instantiated and called as such:
# obj = Cashier.new(n, discount, products, prices)
# param_1 = obj.get_bill(product, amount)
```

PHP Solution:

```
class Cashier {
/**
* @param Integer $n
* @param Integer $discount
* @param Integer[] $products
* @param Integer[] $prices
*/
function __construct($n, $discount, $products, $prices) {

}

/**
* @param Integer[] $product
* @param Integer[] $amount
* @return Float
*/
function getBill($product, $amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
* $obj = Cashier($n, $discount, $products, $prices);
* $ret_1 = $obj->getBill($product, $amount);
*/
```

Dart Solution:

```
class Cashier {

Cashier(int n, int discount, List<int> products, List<int> prices) {

}

double getBill(List<int> product, List<int> amount) {

}
}

/**
* Your Cashier object will be instantiated and called as such:
```

```
* Cashier obj = Cashier(n, discount, products, prices);
* double param1 = obj.getBill(product,amount);
*/
```

Scala Solution:

```
class Cashier(_n: Int, _discount: Int, _products: Array[Int], _prices:
Array[Int]) {

def getBill(product: Array[Int], amount: Array[Int]): Double = {

}

}

/**
* Your Cashier object will be instantiated and called as such:
* val obj = new Cashier(n, discount, products, prices)
* val param_1 = obj.getBill(product,amount)
*/
```

Elixir Solution:

```
defmodule Cashier do
@spec init_(n :: integer, discount :: integer, products :: [integer], prices
:: [integer]) :: any
def init_(n, discount, products, prices) do

end

@spec get_bill(product :: [integer], amount :: [integer]) :: float
def get_bill(product, amount) do

end
end

# Your functions will be called as such:
# Cashier.init_(n, discount, products, prices)
# param_1 = Cashier.get_bill(product, amount)

# Cashier.init_ will be called before every test case, in which you can do
```

```
some necessary initializations.
```

Erlang Solution:

```
-spec cashier_init_(N :: integer(), Discount :: integer(), Products ::
[integer()], Prices :: [integer()]) -> any().
cashier_init_(N, Discount, Products, Prices) ->
.

-spec cashier_get_bill(Product :: [integer()], Amount :: [integer()]) ->
float().
cashier_get_bill(Product, Amount) ->
.


%% Your functions will be called as such:
%% cashier_init_(N, Discount, Products, Prices),
%% Param_1 = cashier_get_bill(Product, Amount),

%% cashier_init_ will be called before every test case, in which you can do
some necessary initializations.
```

Racket Solution:

```
(define cashier%
(class object%
(super-new)

; n : exact-integer?
; discount : exact-integer?
; products : (listof exact-integer?)
; prices : (listof exact-integer?)
(init-field
n
discount
products
prices)

; get-bill : (listof exact-integer?) (listof exact-integer?) -> flonum?
(define/public (get-bill product amount)
)))
```

```
;; Your cashier% object will be instantiated and called as such:
;; (define obj (new cashier% [n n] [discount discount] [products products]
[prices prices]))
;; (define param_1 (send obj get-bill product amount))
```