Problem 2706: Buy Two Chocolates

Problem Information

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

Problem Description

You are given an integer array

prices

representing the prices of various chocolates in a store. You are also given a single integer

money

, which represents your initial amount of money.

You must buy

exactly

two chocolates in such a way that you still have some

non-negative

leftover money. You would like to minimize the sum of the prices of the two chocolates you buy.

Return

the amount of money you will have leftover after buying the two chocolates

. If there is no way for you to buy two chocolates without ending up in debt, return

money

. Note that the leftover must be non-negative.

Example 1:

Input:

prices = [1,2,2], money = 3

Output:

0

Explanation:

Purchase the chocolates priced at 1 and 2 units respectively. You will have 3 - 3 = 0 units of money afterwards. Thus, we return 0.

Example 2:

Input:

prices = [3,2,3], money = 3

Output:

3

Explanation:

You cannot buy 2 chocolates without going in debt, so we return 3.

Constraints:

2 <= prices.length <= 50

1 <= prices[i] <= 100

1 <= money <= 100

Code Snippets

C++:

```
class Solution {
public:
int buyChoco(vector<int>& prices, int money) {

}
};
```

Java:

```
class Solution {
public int buyChoco(int[] prices, int money) {

}
}
```

Python3:

```
class Solution:
def buyChoco(self, prices: List[int], money: int) -> int:
```

Python:

```
class Solution(object):
def buyChoco(self, prices, money):
"""
:type prices: List[int]
:type money: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} prices
* @param {number} money
```

```
 * @return {number}
 */
var buyChoco = function(prices, money) {

};
```

TypeScript:

```
function buyChoco(prices: number[], money: number): number {

};
```

C#:

```
public class Solution {
public int BuyChoco(int[] prices, int money) {

}
}
```

C:

```
int buyChoco(int* prices, int pricesSize, int money) {

}
```

Go:

```
func buyChoco(prices []int, money int) int {

}
```

Kotlin:

```
class Solution {
fun buyChoco(prices: IntArray, money: Int): Int {

}
}
```

Swift:

```
class Solution {
func buyChoco(_ prices: [Int], _ money: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn buy_choco(prices: Vec<i32>, money: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} prices
# @param {Integer} money
# @return {Integer}
def buy_choco(prices, money)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $prices
* @param Integer $money
* @return Integer
*/
function buyChoco($prices, $money) {

}
}
```

Dart:

```
class Solution {
int buyChoco(List<int> prices, int money) {

}
```

```
}
```

Scala:

```
object Solution {
def buyChoco(prices: Array[Int], money: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec buy_choco(prices :: [integer], money :: integer) :: integer
def buy_choco(prices, money) do

end
end
```

Erlang:

```
-spec buy_choco(Prices :: [integer()], Money :: integer()) -> integer().
buy_choco(Prices, Money) ->
.
```

Racket:

```
(define/contract (buy-choco prices money)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Buy Two Chocolates
* Difficulty: Easy
* Tags: array, greedy, sort
*
```

```
* 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 buyChoco(vector<int>& prices, int money) {

}
};
```

Java Solution:

```
/**
* Problem: Buy Two Chocolates
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 buyChoco(int[] prices, int money) {

}
}
```

Python3 Solution:

```
"""
Problem: Buy Two Chocolates
Difficulty: Easy
Tags: array, greedy, sort

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 buyChoco(self, prices: List[int], money: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def buyChoco(self, prices, money):
"""
:type prices: List[int]
:type money: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Buy Two Chocolates
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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[]} prices
* @param {number} money
* @return {number}
*/
var buyChoco = function(prices, money) {

};
```

TypeScript Solution:

```
/**
* Problem: Buy Two Chocolates
```

```
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 buyChoco(prices: number[], money: number): number {

};
```

C# Solution:

```
/*
* Problem: Buy Two Chocolates
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 BuyChoco(int[] prices, int money) {

}
}
```

C Solution:

```
/*
* Problem: Buy Two Chocolates
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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
*/
```

```
int buyChoco(int* prices, int pricesSize, int money) {

}
```

Go Solution:

```
// Problem: Buy Two Chocolates
// Difficulty: Easy
// Tags: array, greedy, sort
//
// 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 buyChoco(prices []int, money int) int {

}
```

Kotlin Solution:

```
class Solution {
fun buyChoco(prices: IntArray, money: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func buyChoco(_ prices: [Int], _ money: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Buy Two Chocolates
// Difficulty: Easy
// Tags: array, greedy, sort
//
```

```
// 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 buy_choco(prices: Vec<i32>, money: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} prices
# @param {Integer} money
# @return {Integer}
def buy_choco(prices, money)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $prices
* @param Integer $money
* @return Integer
*/
function buyChoco($prices, $money) {

}
}
```

Dart Solution:

```
class Solution {
int buyChoco(List<int> prices, int money) {

}
}
```

Scala Solution:

```
object Solution {
def buyChoco(prices: Array[Int], money: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec buy_choco(prices :: [integer], money :: integer) :: integer
def buy_choco(prices, money) do

end
end
```

Erlang Solution:

```
-spec buy_choco(Prices :: [integer()], Money :: integer()) -> integer().
buy_choco(Prices, Money) ->
.
```

Racket Solution:

```
(define/contract (buy-choco prices money)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```