Problem 2548: Maximum Price to Fill a Bag

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

, weight

i

]

denotes the price and weight of the

i

th

item, respectively.

You are also given a

positive

integer

capacity

.

Each item can be divided into two items with ratios

part1

and

part2

, where

part1 + part2 == 1

.

The weight of the first item is

weight

i

* part1

and the price of the first item is

price

i

* part1

.

Similarly, the weight of the second item is

weight

i

* part2

and the price of the second item is

price

i

* part2

.

Return

the maximum total price

to fill a bag of capacity

capacity

with given items

. If it is impossible to fill a bag return

-1

. Answers within

10

-5

of the

actual answer

will be considered accepted.

Example 1:

Input:

items = [[50,1],[10,8]], capacity = 5

Output:

55.00000

Explanation:

We divide the 2

nd

item into two parts with part1 = 0.5 and part2 = 0.5. The price and weight of the 1

st

item are 5, 4. And similarly, the price and the weight of the 2

nd

item are 5, 4. The array items after operation becomes [[50,1],[5,4],[5,4]]. To fill a bag with capacity 5 we take the 1

st

element with a price of 50 and the 2

nd

element with a price of 5. It can be proved that 55.0 is the maximum total price that we can achieve.

Example 2:

Input:

items = [[100,30]], capacity = 50

Output:

-1.00000

Explanation:

It is impossible to fill a bag with the given item.

Constraints:

1 <= items.length <= 10

5

items[i].length == 2

1 <= price

i

, weight

i

<= 10

4

1 <= capacity <= 10

Code Snippets

C++:

```
class Solution {
public:
double maxPrice(vector<vector<int>>& items, int capacity) {

}
};
```

Java:

```
class Solution {
public double maxPrice(int[][] items, int capacity) {

}
}
```

Python3:

```
class Solution:
def maxPrice(self, items: List[List[int]], capacity: int) -> float:
```

Python:

```
class Solution(object):
def maxPrice(self, items, capacity):
"""
:type items: List[List[int]]
:type capacity: int
:rtype: float
"""
```

JavaScript:

```
/**
* @param {number[][]} items
* @param {number} capacity
```

```
* @return {number}
*/
var maxPrice = function(items, capacity) {

};
```

TypeScript:

```
function maxPrice(items: number[][], capacity: number): number {

};
```

C#:

```
public class Solution {
public double MaxPrice(int[][] items, int capacity) {

}
}
```

C:

```
double maxPrice(int** items, int itemsSize, int* itemsColSize, int capacity)
{

}
```

Go:

```
func maxPrice(items [][]int, capacity int) float64 {

}
```

Kotlin:

```
class Solution {
fun maxPrice(items: Array<IntArray>, capacity: Int): Double {

}
}
```

Swift:

```
class Solution {
func maxPrice(_ items: [[Int]], _ capacity: Int) -> Double {

}
}
```

Rust:

```
impl Solution {
pub fn max_price(items: Vec<Vec<i32>>, capacity: i32) -> f64 {

}
}
```

Ruby:

```
# @param {Integer[][]} items
# @param {Integer} capacity
# @return {Float}
def max_price(items, capacity)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $items
* @param Integer $capacity
* @return Float
*/
function maxPrice($items, $capacity) {

}
}
```

Dart:

```
class Solution {
double maxPrice(List<List<int>> items, int capacity) {

}
```

```
}
```

Scala:

```
object Solution {
def maxPrice(items: Array[Array[Int]], capacity: Int): Double = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_price(items :: [[integer]], capacity :: integer) :: float
def max_price(items, capacity) do

end
end
```

Erlang:

```
-spec max_price(Items :: [[integer()]], Capacity :: integer()) -> float().
max_price(Items, Capacity) ->
.
```

Racket:

```
(define/contract (max-price items capacity)
(-> (listof (listof exact-integer?)) exact-integer? flonum?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Price to Fill a Bag
* Difficulty: Medium
* 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:
double maxPrice(vector<vector<int>>& items, int capacity) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Price to Fill a Bag
* Difficulty: Medium
* 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 double maxPrice(int[][] items, int capacity) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Price to Fill a Bag
Difficulty: Medium
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 maxPrice(self, items: List[List[int]], capacity: int) -> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxPrice(self, items, capacity):
"""
:type items: List[List[int]]
:type capacity: int
:rtype: float
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Price to Fill a Bag
* Difficulty: Medium
* 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[][]} items
* @param {number} capacity
* @return {number}
*/
var maxPrice = function(items, capacity) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Price to Fill a Bag
```

```
* Difficulty: Medium
* 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 maxPrice(items: number[][], capacity: number): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Price to Fill a Bag
* Difficulty: Medium
* 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 double MaxPrice(int[][] items, int capacity) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Price to Fill a Bag
* Difficulty: Medium
* 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
*/
```

```
double maxPrice(int** items, int itemsSize, int* itemsColSize, int capacity)
{

}
```

Go Solution:

```
// Problem: Maximum Price to Fill a Bag
// Difficulty: Medium
// 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 maxPrice(items [][]int, capacity int) float64 {

}
```

Kotlin Solution:

```
class Solution {
fun maxPrice(items: Array<IntArray>, capacity: Int): Double {

}
}
```

Swift Solution:

```
class Solution {
func maxPrice(_ items: [[Int]], _ capacity: Int) -> Double {

}
}
```

Rust Solution:

```
// Problem: Maximum Price to Fill a Bag
// Difficulty: Medium
// 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 max_price(items: Vec<Vec<i32>>, capacity: i32) -> f64 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} items
# @param {Integer} capacity
# @return {Float}
def max_price(items, capacity)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $items
* @param Integer $capacity
* @return Float
*/
function maxPrice($items, $capacity) {

}
}
```

Dart Solution:

```
class Solution {
double maxPrice(List<List<int>> items, int capacity) {

}
}
```

Scala Solution:

```
object Solution {
def maxPrice(items: Array[Array[Int]], capacity: Int): Double = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_price(items :: [[integer]], capacity :: integer) :: float
def max_price(items, capacity) do

end
end
```

Erlang Solution:

```
-spec max_price(Items :: [[integer()]], Capacity :: integer()) -> float().
max_price(Items, Capacity) ->
.
```

Racket Solution:

```
(define/contract (max-price items capacity)
(-> (listof (listof exact-integer?)) exact-integer? flonum?)
)
```