Problem 2412: Minimum Money Required Before Transactions

Problem Information

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

Problem Description

You are given a

0-indexed

2D integer array

transactions

, where

transactions[i] = [cost

i

, cashback

i

]

.

The array describes transactions, where each transaction must be completed exactly once in

some order

. At any given moment, you have a certain amount of

money

. In order to complete transaction

i

,

money >= cost

i

must hold true. After performing a transaction,

money

becomes

money - cost

i

+ cashback

i

.

Return

the minimum amount of

money

required before any transaction so that all of the transactions can be completed

regardless of the order

of the transactions.

Example 1:

Input:

transactions = [[2,1],[5,0],[4,2]]

Output:

10

Explanation:

Starting with money = 10, the transactions can be performed in any order. It can be shown that starting with money < 10 will fail to complete all transactions in some order.

Example 2:

Input:

transactions = [[3,0],[0,3]]

Output:

3

Explanation:

- If transactions are in the order [[3,0],[0,3]], the minimum money required to complete the transactions is 3. - If transactions are in the order [[0,3],[3,0]], the minimum money required to complete the transactions is 0. Thus, starting with money = 3, the transactions can be performed in any order.

Constraints:

1 <= transactions.length <= 10

5

transactions[i].length == 2

0 <= cost

i

, cashback

i

<= 10

9

Code Snippets

C++:

```
class Solution {
public:
long long minimumMoney(vector<vector<int>>& transactions) {

}
};
```

Java:

```
class Solution {
public long minimumMoney(int[][] transactions) {

}
}
```

Python3:

```
class Solution:
def minimumMoney(self, transactions: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def minimumMoney(self, transactions):
"""
:type transactions: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} transactions
* @return {number}
*/
var minimumMoney = function(transactions) {

};
```

TypeScript:

```
function minimumMoney(transactions: number[][]): number {

};
```

C#:

```
public class Solution {
public long MinimumMoney(int[][] transactions) {

}
}
```

C:

```
long long minimumMoney(int** transactions, int transactionsSize, int*
transactionsColSize) {

}
```

Go:

```
func minimumMoney(transactions [][]int) int64 {

}
```

Kotlin:

```
class Solution {
fun minimumMoney(transactions: Array<IntArray>): Long {

}
}
```

Swift:

```
class Solution {
func minimumMoney(_ transactions: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_money(transactions: Vec<Vec<i32>>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[][]} transactions
# @return {Integer}
def minimum_money(transactions)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $transactions
* @return Integer
```

```
*/
function minimumMoney($transactions) {

}
}
```

Dart:

```
class Solution {
int minimumMoney(List<List<int>> transactions) {

}
}
```

Scala:

```
object Solution {
def minimumMoney(transactions: Array[Array[Int]]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_money(transactions :: [[integer]]) :: integer
def minimum_money(transactions) do

end
end
```

Erlang:

```
-spec minimum_money(Transactions :: [[integer()]]) -> integer().
minimum_money(Transactions) ->
.
```

Racket:

```
(define/contract (minimum-money transactions)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
* 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:
long long minimumMoney(vector<vector<int>>& transactions) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
* 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 long minimumMoney(int[][] transactions) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Money Required Before Transactions
Difficulty: Hard
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 minimumMoney(self, transactions: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumMoney(self, transactions):
"""
:type transactions: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
* 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[][]} transactions
* @return {number}
*/
var minimumMoney = function(transactions) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
* 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 minimumMoney(transactions: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
* 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 long MinimumMoney(int[][] transactions) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Money Required Before Transactions
* Difficulty: Hard
```

```
* 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
*/

long long minimumMoney(int** transactions, int transactionsSize, int*
transactionsColSize) {

}
```

Go Solution:

```
// Problem: Minimum Money Required Before Transactions
// Difficulty: Hard
// 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 minimumMoney(transactions [][]int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minimumMoney(transactions: Array<IntArray>): Long {

}
}
```

Swift Solution:

```
class Solution {
func minimumMoney(_ transactions: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Money Required Before Transactions
// Difficulty: Hard
// 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 minimum_money(transactions: Vec<Vec<i32>>) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} transactions
# @return {Integer}
def minimum_money(transactions)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $transactions
* @return Integer
*/
function minimumMoney($transactions) {

}
}
```

Dart Solution:

```
class Solution {
int minimumMoney(List<List<int>> transactions) {
```

```
}
}
```

Scala Solution:

```
object Solution {
def minimumMoney(transactions: Array[Array[Int]]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_money(transactions :: [[integer]]) :: integer
def minimum_money(transactions) do

end
end
```

Erlang Solution:

```
-spec minimum_money(Transactions :: [[integer()]]) -> integer().
minimum_money(Transactions) ->
.
```

Racket Solution:

```
(define/contract (minimum-money transactions)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```