## 2412. Minimum Money Required Before Transactions

<u>Greedy</u> <u>Array</u>

Sorting

### **Problem Description**

Hard

In this problem, we're given an array transactions that represents a series of transactions. Each transaction is described by two values: a cost and a cashback. The cost is the amount of money that must be paid to perform the transaction, and the cashback is the amount that is returned after the transaction is completed. Our goal is to find the minimum amount of money (money) required before any transaction is made so that it's possible to complete all transactions in any order. The key condition is that at any point, money must be greater than or equal to the cost of the transaction being performed, and after each transaction, money updates to money - cost + cashback. Intuition

than any other transaction types. Therefore, one needs to ensure they have enough money to handle the worst-case scenario for these transactions. Here's the intuition to solve the problem:

Calculate the minimum amount of money needed to ensure that you always have enough to cover the cost for the riskier

Coming up with a solution involves understanding that there are certain transactions that are riskier in terms of running out of

money. Specifically, transactions where cost is greater than cashback are risky because they deplete the pool of money more

transactions. This is done by finding the sum of the differences between cost and cashback for these transactions sum(max(0, a - b) for a, b in transactions).

- Keep track of the maximum additional money required on top of this sum, which comes from the cashback (or cost if cashback is greater) of the transactions. This step includes the consideration for the transaction with the highest cashback that would minimize the amount of money needed upfront.
- The solution iterates through each transaction to calculate these values. The initial sum represents the minimum starting money required to perform all riskier transactions one after another, without the cashback from any transaction dichotomizing the sum. The 'ans' being the maximum helps to keep a running maximum of what that minimum starting money might be considering the

By the end of the iteration, if a transaction's cost is greater than its cashback, the maximum of the sum plus cashback becomes a potential answer. Otherwise, the maximum of the sum plus cost is considered for the less risky transactions. The final ans value is the minimum amount of money required to start with to complete all transactions in any order.

cashbacks (or cost, whichever is lower for each of the less risky transactions).

Solution Approach The solution uses a simple linear scan algorithm, which is efficient since it only needs to iterate through the transactions once. It performs two key calculations:

It first calculates the sum of all the positive differences between cost and cashback for each transaction. This is because, for

transactions where cost is greater than cashback, you will lose some money, and you need to have enough in reserve to cover

#### these losses. The calculation is done with a generator expression in Python:

s = sum(max(0, a - b)) for a, b in transactions) Here, max(0, a - b) ensures that we only sum positive differences, as we do not need extra money upfront for transactions

- where cashback is greater or equal to cost. The code then determines the maximum additional money required to ensure that transactions can be completed in any order. It iterates through the transactions, comparing each cost and cashback: ans = 0
- For riskier transactions (a > b), the code considers the possibility of doing this transaction first when the reserve s is at its full. After paying a (cost), you receive b (cashback), so you need at least s + b to start with to do this transaction without

```
For transactions that are not riskier (a <= b), because they refund equal or more than their cost, doing them first would only
   require enough money to cover the cost, which is a. Thus, for each transaction, you check if s + a (cost) is a new maximum
   requirement to start with.
The final result in ans is the minimum money required before any transaction that allows you to complete all of them regardless of
```

• Transaction 1: cost = 5, cashback = 2

• Transaction 2: cost = 3, cashback = 3

• Transaction 3: cost = 6, cashback = 1

• For Transaction 2: cost - cashback = 3 - 3 = 0 (no positive difference)

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

# Calculate the initial sum required to cover all negative cash flows,

# i.e., where the cost of a transaction is higher than the cashback

# Initialize maximum additional money required at the start as zero

# If transaction cost is lower or equal to cashback,

for a, b in transactions:

ans = max(ans, s + b)

ans = max(ans, s + a)

if a > b:

else:

going broke.

**Example Walkthrough** Let's walk through a small example to better understand the solution approach. Suppose we have the following transactions:

order. The solution is efficient because it has a time complexity of O(n), where n is the number of transactions, and does not

require any additional complex data structures, making use of only iteration and comparison to arrive at the solution.

We need to find the minimum amount of money (money) required to complete all these transactions in any order. First, we calculate the sum of all the positive differences between cost and cashback. In our example: • For Transaction 1: cost - cashback = 5 - 2 = 3 (positive difference)

#### For Transaction 3: cost - cashback = 6 - 1 = 5 (positive difference) We sum these up to get the total sum s:

Solution Implementation

if cost > cashback:

else:

# the current transaction

# the current transaction

long long minimumMoney(vector<vector<int>>& transactions) {

costSum += max(0, transaction[0] - transaction[1]);

// Determine the minimum initial money needed before any of the transactions

long long costSum = 0, minInitialMoney = 0;

for (auto& transaction : transactions) {

for (auto& transaction : transactions) {

if (transaction[0] > transaction[1]) {

return max\_additional\_money\_required

class Solution:

• Transaction 1: Since cost > cashback, we consider s + cashback which is 8 + 2 = 10. We update ans to 10 as it's greater than the current ans (0).

• Transaction 3: Since cost > cashback, we consider s + cashback which is 8 + 1 = 9. ans remains 11 as it's greater than 9.

Next, we iterate over each transaction to decide the maximum additional money required. We start with ans = 0:

sum = 3 (from Transaction 1) + 0 (from Transaction 2) + 5 (from Transaction 3) = 8

before starting any transactions to be able to complete all of them in any order.

total negative\_cash\_flow = sum(max(0, cost - cashback)) for cost, cashback in transactions)

# If transaction cost is higher than cashback, calculate additional money

# required by considering the total negative cash flow and cashback of

max\_additional\_money\_required = max(max\_additional\_money\_required,

# calculate by considering total negative cash flow and cost of

max\_additional\_money\_required = max(max\_additional\_money\_required,

**Python** from typing import List

• Transaction 2: Since cost <= cashback, we consider s + cost which is 8 + 3 = 11. We update ans to 11 as it's greater than the current ans (10).

After considering each transaction, the final answer ans is 11. This is the minimum amount of money we need to have upfront

max\_additional\_money\_required = 0 # Iterate through all transactions to determine the maximum additional money # required at the start to not end up with a negative balance at any point for cost, cashback in transactions:

total\_negative\_cash\_flow + cashback)

total\_negative\_cash\_flow + cost)

```
# Return the maximum additional money that is required at the beginning
```

Java

class Solution {

} else {

from typing import List

else:

**Time Complexity** 

class Solution:

return minInitialMoney;

max\_additional\_money\_required = 0

# the current transaction

return max\_additional\_money\_required

**Time and Space Complexity** 

public:

```
class Solution {
    public long minimumMoney(int[][] transactions) {
       // Initialize a sum variable to hold the total amount of initial money we need
       // without considering cashback.
        long totalWithoutCashback = 0;
       // Iterate over each transaction.
        for (int[] transaction : transactions) {
           // To not go negative at the start of a transaction, we need to have enough money
           // that even if we don't get the cashback immediately, we won't go bankrupt.
           // Hence, we add the net loss of each transaction to the initial money required.
            totalWithoutCashback += Math.max(0, transaction[0] - transaction[1]);
       // Initialize a variable to store the answer, which is the final minimum money needed.
        long minMoneyRequired = 0;
       // Iterate over each transaction again to find the peak money needed at any transaction.
        for (int[] transaction : transactions) {
            if (transaction[0] > transaction[1]) {
               // If the cost is greater than the cashback, the peak money will be the total
               // without considering cashback plus the cashback of this transaction.
                // This ensures we have enough during the transaction and after getting the cashback.
               minMoneyRequired = Math.max(minMoneyRequired, totalWithoutCashback + transaction[1]);
            } else {
               // If the cashback is greater than or equal to the cost, we just need the cost of
                // this transaction as the peak money, on top of the initially calculated money,
                // because we'll get back as much or more than we spend.
               minMoneyRequired = Math.max(minMoneyRequired, totalWithoutCashback + transaction[0]);
       // Return the maximum of money we need at any point which will be enough to start all transactions.
       return minMoneyRequired;
```

```
// For transactions that cost more than you get back, add back the money obtained from that transaction
               minInitialMoney = max(minInitialMoney, costSum + transaction[1]);
           } else {
               // For transactions that cost less or equal to what you get back, take the maximum of the cost
               minInitialMoney = max(minInitialMoney, costSum + transaction[0]);
       // Return the calculated minimum initial money required to complete all transactions
       return minInitialMoney;
TypeScript
function minimumMoney(transactions: number[][]): number {
   // Initialize the sum of costs and the answer, which will store the minimum initial money required
   let costSum: number = 0;
    let minInitialMoney: number = 0;
   // Calculate the sum of extra costs needed after transactions that cost more than you get back
   for (const transaction of transactions) {
       costSum += Math.max(0, transaction[0] - transaction[1]);
   // Determine the minimum initial money needed before any of the transactions
   for (const transaction of transactions) {
       if (transaction[0] > transaction[1]) {
           // For transactions that cost more than what you get back, add back the money obtained from that transaction
           minInitialMoney = Math.max(minInitialMoney, costSum + transaction[1]);
```

// For transactions that cost less than or equal to what you get back, take the maximum of the cost

total\_negative\_cash\_flow + cashback)

total negative cash flow + cost)

// Initialize the sum of costs and the answer which will store the minimum initial money required

// Calculate the sum of extra costs needed after transactions that cost more than you get back

```
# Iterate through all transactions to determine the maximum additional money
# required at the start to not end up with a negative balance at any point
for cost, cashback in transactions:
    if cost > cashback:
       # If transaction cost is higher than cashback, calculate additional money
       # required by considering the total negative cash flow and cashback of
        # the current transaction
       max_additional_money_required = max(max_additional_money_required,
```

# calculate by considering total negative cash flow and cost of

max\_additional\_money\_required = max(max\_additional\_money\_required,

total\_negative\_cash\_flow = sum(max(0, cost - cashback) for cost, cashback in transactions)

# Calculate the initial sum required to cover all negative cash flows,

# i.e., where the cost of a transaction is higher than the cashback

# Initialize maximum additional money required at the start as zero

# If transaction cost is lower or equal to cashback,

# Return the maximum additional money that is required at the beginning

The given code consists of two main parts which contribute to the time complexity:

minInitialMoney = Math.max(minInitialMoney, costSum + transaction[0]);

// Return the calculated minimum initial money required to complete all transactions

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

The first part is the sum operation with a generator expression: s = sum(max(0, a - b)) for a, b in transactions)

```
complexity of O(N), where N is the number of transactions.
The second part is a loop that iterates over all transactions again to determine the maximum money required:
```

else: ans = max(ans, s + a)

```
This loop runs for each transaction, making it also O(N).
Since these two parts run sequentially, the overall time complexity is the sum of both parts, which is O(N) + O(N), resulting in a
total time complexity of O(N).
```

for a, b in transactions:

ans = max(ans, s + b)

if a > b:

**Space Complexity** 

For each transaction in the list transactions, it calculates the maximum between 0 and a - b. This operation has a time

# The space complexity of the code is O(1), disregarding the input size. This is because the space used by the variables s and ans

is constant and does not depend on the size of the input list transactions. The generator expression also does not create an intermediate list, which keeps the space complexity low.