1672. Richest Customer Wealth

Easy Array Matrix

Leetcode Link

Problem Description

In this problem, we are dealing with a grid, called `accounts`, where each row represents a customer, and each column represents a bank. The value at `accounts[i][j]` indicates the amount of money the i^{th} customer has at the j^{th} bank. The task is to calculate the total wealth for each customer, which is the sum of money they have across all banks. After computing the wealth of all customers, we need to determine the wealth of the richest customer, which is simply the maximum total wealth among all customers.

Intuition

To solve this problem, the most straightforward approach is to consider each customer one by one and calculate their wealth by summing the amounts from all the banks they hold money in. Once we have this sum, representing a single customer's total wealth, we can compare it with the wealth of other customers.

The solution leverages Python's list comprehension and the `max` and `sum` functions as its core components. Here's how we break down the solution:

1. Use a list comprehension to iterate through each customer (represented by each row in the `accounts` grid).
2. For each customer, apply the `sum` function to add up the wealth across all their bank accounts.
3. Enclose this operation within the `max` function to find the maximum sum, which would represent the richest customer's wealth.

The crucial part of this solution is recognizing that we can perform the summation and comparison in a single line, utilizing Python's concise syntax to achieve an efficient and clean solution.

Solution Approach

The implementation of the solution uses very simple and powerful features of Python. Specifically, the solution relies on the following:

1. **List Comprehension**: This Python feature allows us to create a new list by iterating over each element of an existing list. In our case, we iterate over each customer (each row in the `accounts` grid) to calculate their wealth.

2. The **`sum` function**: This built-in Python function takes an iterable (like a list) and returns the sum of its elements. When we use it with each row of the `accounts` grid inside our list comprehension, it calculates the sum of all the bank account balances for each customer.

3. The **`max` function**: This is another built-in Python function that takes an iterable and returns the largest element. In our solution, it is used to find the maximum value within the list of wealth sums generated by the list comprehension.

Putting it all together, the algorithm for solving the problem can be described in the following steps:

- Iterate over the `accounts` grid with a list comprehension, taking each customer's accounts as a sub-list.
- For each sub-list (representing a customer's accounts), use the `sum` function to add up the amounts and calculate the customer's wealth.
- As the list comprehension executes, it generates a list of customers' wealths.
- Finally, apply the `max` function to the list of wealth sums to determine the maximum wealth, identifying the richest customer.

The final line of code that implements the solution is:

```
return max(sum(v) for v in accounts)
```

Here, `sum(v)` computes each customer's wealth, and `max(...)` finds the highest wealth from the list comprehension. The result returned by this line is the maximum wealth of the richest customer, which is what the problem asks for.

Example Walkthrough

Let's walk through an example to illustrate the solution approach mentioned above. Suppose we have the following `accounts` grid, where each row represents a customer's bank account balances and each column represents a different bank:

```
accounts = [
    [2, 8, 4],  # Customer 1
    [1, 5, 7],  # Customer 2
    [3, 9, 3]   # Customer 3
]
```

The wealth of each customer is calculated as follows:

- For Customer 1, the total wealth is `2 + 8 + 4 = 14`.
- For Customer 2, the total wealth is `1 + 5 + 7 = 13`.
- For Customer 3, the total wealth is `3 + 9 + 3 = 15`.

Now that we have calculated the total wealth for each customer, we need to identify the wealth of the richest customer. In this case:

- Customer 1 has a wealth of 14.
- Customer 2 has a wealth of 13.
- Customer 3 has a wealth of 15.

Therefore, Customer 3 is the richest, with a wealth of 15.

The Python code to solve this example using the solution approach would look like this:

```
# Define the accounts grid
accounts = [
    [2, 8, 4],  # Customer 1
    [1, 5, 7],  # Customer 2
    [3, 9, 3]   # Customer 3
]

# Use list comprehension to calculate each customer's wealth and find the maximum
richest_wealth = max(sum(v) for v in accounts)

# The result would be 15, which is the maximum wealth among all customers
print(richest_wealth)
```

When the code is executed, `sum(v)` calculates the wealth for each customer, and `max(...)` selects the maximum value from those calculations. The final output is `15`, which corresponds to the wealth of the richest customer according to our grid.

Python Solution

```
# Define a class called Solution as required by the LeetCode format.
class Solution:
    # This method calculates the maximum wealth across all customers.
    # 'accounts' is a list of lists, where each inner list represents the wealth (bank account balances) of a single customer across
    def maximumWealth(self, accounts: List[List[int]]) -> int:
        # Use a generator expression to compute the sum of each customer's wealth iteratively.
        # Then, apply the max() function to find the greatest total wealth across all customers.
        return max(sum(account) for account in accounts)
        # The max function is applied over the iterable produced by the generator expression.
        # The sum function calculates the total wealth of each customer by summing up their respective account balances.

```

Java Solution

```
class Solution {
    public int maximumWealth(int[][] accounts) {
        // Initialize the variable that will hold the maximum wealth found
        int maxWealth = 0;

        // Loop through each customer's account
        for (int[] customerAccounts : accounts) {
            // Sum the wealth of the current customer
            int customerWealth = 0;
            for (int accountBalance : customerAccounts) {
                customerWealth += accountBalance;
            }

            // Update maxWealth if the current customer's wealth is greater
            maxWealth = Math.max(maxWealth, customerWealth);
        }
        // Return the maximum wealth across all customers
        return maxWealth;
    }
}

```

C++ Solution

```
#include <vector> // Include vector
#include <numeric> // Include accumulate function

class Solution {
public:
    // Function to find the maximum wealth among all customers
    int maximumWealth(vector<vector<int>>& accounts) {
        int maximumWealth = 0; // Initialize the maximum wealth to zero

        // Loop through each customer's account
        for (const auto& account : accounts) {
            // Calculate the wealth of the current customer by summing the account balances
            // and update the maximumWealth if the current wealth is higher
            maximumWealth = max(maximumWealth, accumulate(account.begin(), account.end(), 0));
        }

        // Return the maximum wealth found
        return maximumWealth;
    }
};

```

Typescript Solution

```
/**
 * Computes the maximum wealth where wealth is defined as the sum of all bank accounts.
 * Iterates through each customer's array of bank accounts to calculate their wealth.
 *
 * @param accounts - A 2D array where each sub-array represents a customer's bank accounts.
 * @returns The maximum wealth found among all customers.
 */
function maximumWealth(accounts: number[][]): number {
    // Iterate through the array of accounts using 'reduce' to find the maximum wealth.
    // 'maxWealth' accumulates the maximum wealth encountered so far.
    // 'customerAccounts' represents the accounts for one customer.
    return accounts.reduce((maxWealth, customerAccounts) => {
        // Calculate the total wealth for the current customer by summing their account balances.
        // 'customerWealth' accumulates the sum of the customer's accounts.
        // 'accountBalance' represents the balance of one account.
        const customerWealth = customerAccounts.reduce((sumWealth, accountBalance) => sumWealth + accountBalance, 0);
        // Compare and return the greater value: either the current max wealth or the customer's wealth.
        return Math.max(maxWealth, customerWealth);
    }, 0); // Initialize the maximum wealth to 0, as it's the lowest possible wealth.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(m * n)`, where `m` is the number of customers (i.e., number of sub-arrays within `accounts`) and `n` is the number of banks (i.e., number of elements in each sub-array). This is because the code iterates through each customer's list of account balances once and sums the balances in each list.

Space Complexity

The space complexity of the code is `O(1)`. No additional space is used that is dependent on the input size; the variables used to compute the maximum wealth (such as the sum of values in an account) do not scale with the size of the input.