Problem 1672: Richest Customer Wealth

Problem Information

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

Problem Description

You are given an

m x n

integer grid

accounts

where

accounts[i][j]

is the amount of money the

i

th

customer has in the

j

th

bank. Return

the

wealth

that the richest customer has.

A customer's

wealth

is the amount of money they have in all their bank accounts. The richest customer is the customer that has the maximum

wealth

.

Example 1:

Input:

accounts = [[1,2,3],[3,2,1]]

Output:

6

Explanation

:

1st customer has wealth = 1 + 2 + 3 = 6

2nd customer has wealth = 3 + 2 + 1 = 6

Both customers are considered the richest with a wealth of 6 each, so return 6.

Example 2:

Input:

accounts = [[1,5],[7,3],[3,5]]

Output:

10

Explanation

: 1st customer has wealth = 6 2nd customer has wealth = 10 3rd customer has wealth = 8 The 2nd customer is the richest with a wealth of 10.

Example 3:

Input:

accounts = [[2,8,7],[7,1,3],[1,9,5]]

Output:

17

Constraints:

m == accounts.length

n == accounts[i].length

1 <= m, n <= 50

1 <= accounts[i][j] <= 100

Code Snippets

C++:

```
class Solution {
public:
int maximumWealth(vector<vector<int>>& accounts) {

}
};
```

Java:

```
class Solution {
public int maximumWealth(int[][] accounts) {

}
}
```

Python3:

```
class Solution:
def maximumWealth(self, accounts: List[List[int]]) -> int:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function maximumWealth(accounts: number[][]): number {

};
```

C#:

```
public class Solution {
public int MaximumWealth(int[][] accounts) {

}
}
```

C:

```
int maximumWealth(int** accounts, int accountsSize, int* accountsColSize) {

}
```

Go:

```
func maximumWealth(accounts [][]int) int {

}
```

Kotlin:

```
class Solution {
fun maximumWealth(accounts: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func maximumWealth(_ accounts: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn maximum_wealth(accounts: Vec<Vec<i32>>) -> i32 {
}
}
```

Ruby:

```
# @param {Integer[][]} accounts
# @return {Integer}
def maximum_wealth(accounts)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $accounts
* @return Integer
*/
function maximumWealth($accounts) {

}
}
```

Dart:

```
class Solution {
int maximumWealth(List<List<int>> accounts) {

}
}
```

Scala:

```
object Solution {
def maximumWealth(accounts: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_wealth(accounts :: [[integer]]) :: integer
def maximum_wealth(accounts) do

end
end
```

Erlang:

```
-spec maximum_wealth(Accounts :: [[integer()]]) -> integer().
maximum_wealth(Accounts) ->
.
```

Racket:

```
(define/contract (maximum-wealth accounts)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
* 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 maximumWealth(vector<vector<int>>& accounts) {

}
};
```

Java Solution:

```
/**
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
* 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 maximumWealth(int[][] accounts) {

}
}
```

Python3 Solution:

```
"""
Problem: Richest Customer Wealth
Difficulty: Easy
Tags: array

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 maximumWealth(self, accounts: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
* 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[][]} accounts
* @return {number}
*/
var maximumWealth = function(accounts) {

};
```

TypeScript Solution:

```
/**
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
* 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 maximumWealth(accounts: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
```

```
* 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 MaximumWealth(int[][] accounts) {

}
}
```

C Solution:

```
/*
* Problem: Richest Customer Wealth
* Difficulty: Easy
* Tags: array
*
* 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 maximumWealth(int** accounts, int accountsSize, int* accountsColSize) {

}
```

Go Solution:

```
// Problem: Richest Customer Wealth
// Difficulty: Easy
// Tags: array
//
// 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 maximumWealth(accounts [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maximumWealth(accounts: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func maximumWealth(_ accounts: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Richest Customer Wealth
// Difficulty: Easy
// Tags: array
//
// 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 maximum_wealth(accounts: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} accounts
# @return {Integer}
def maximum_wealth(accounts)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $accounts
* @return Integer
*/
function maximumWealth($accounts) {

}
}
```

Dart Solution:

```
class Solution {
int maximumWealth(List<List<int>> accounts) {

}
}
```

Scala Solution:

```
object Solution {
def maximumWealth(accounts: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_wealth(accounts :: [[integer]]) :: integer
def maximum_wealth(accounts) do

end
end
```

Erlang Solution:

```
-spec maximum_wealth(Accounts :: [[integer()]]) -> integer().
maximum_wealth(Accounts) ->
.
```

Racket Solution:

```
(define/contract (maximum-wealth accounts)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```