Problem 3418: Maximum Amount of Money Robot Can Earn

Problem Information

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

Problem Description

You are given an

m x n

grid. A robot starts at the top-left corner of the grid

(0, 0)

and wants to reach the bottom-right corner

(m - 1, n - 1)

. The robot can move either right or down at any point in time.

The grid contains a value

coins[i][j]

in each cell:

If

coins[i][j] >= 0

, the robot gains that many coins.

If

coins[i][j] < 0

, the robot encounters a robber, and the robber steals the

absolute

value of

coins[i][j]

coins.

The robot has a special ability to

neutralize robbers

in at most

2 cells

on its path, preventing them from stealing coins in those cells.

Note:

The robot's total coins can be negative.

Return the

maximum

profit the robot can gain on the route.

Example 1:

Input:

coins = [[0,1,-1],[1,-2,3],[2,-3,4]]

Output:

8

Explanation:

An optimal path for maximum coins is:

Start at

(0, 0)

with

0

coins (total coins =

0

).

Move to

(0, 1)

, gaining

1

coin (total coins =

0 + 1 = 1

).

Move to

(1, 1)

, where there's a robber stealing

2

coins. The robot uses one neutralization here, avoiding the robbery (total coins =

1

).

Move to

(1, 2)

, gaining

3

coins (total coins =

1 + 3 = 4

).

Move to

(2, 2)

, gaining

4

coins (total coins =

4 + 4 = 8

).

Example 2:

Input:

coins = [[10,10,10],[10,10,10]]

Output:

40

Explanation:

An optimal path for maximum coins is:

Start at

(0, 0)

with

10

coins (total coins =

10

).

Move to

(0, 1)

, gaining

10

coins (total coins =

10 + 10 = 20

).

Move to

(0, 2)

, gaining another

10

coins (total coins =

20 + 10 = 30

).

Move to

(1, 2)

, gaining the final

10

coins (total coins =

30 + 10 = 40

).

Constraints:

m == coins.length

n == coins[i].length

1 <= m, n <= 500

-1000 <= coins[i][j] <= 1000

Code Snippets

C++:

```
class Solution {
public:
int maximumAmount(vector<vector<int>>& coins) {

}
};
```

Java:

```
class Solution {
public int maximumAmount(int[][] coins) {

}
}
```

Python3:

```
class Solution:
def maximumAmount(self, coins: List[List[int]]) -> int:
```

Python:

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

JavaScript:

```
/**
* @param {number[][]} coins
```

```
* @return {number}
*/
var maximumAmount = function(coins) {

};
```

TypeScript:

```
function maximumAmount(coins: number[][]): number {

};
```

C#:

```
public class Solution {
public int MaximumAmount(int[][] coins) {

}
}
```

C:

```
int maximumAmount(int** coins, int coinsSize, int* coinsColSize) {

}
```

Go:

```
func maximumAmount(coins [][]int) int {

}
```

Kotlin:

```
class Solution {
fun maximumAmount(coins: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func maximumAmount(_ coins: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn maximum_amount(coins: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} coins
# @return {Integer}
def maximum_amount(coins)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $coins
* @return Integer
*/
function maximumAmount($coins) {

}
}
```

Dart:

```
class Solution {
int maximumAmount(List<List<int>> coins) {

}
}
```

Scala:

```
object Solution {
def maximumAmount(coins: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_amount(coins :: [[integer]]) :: integer
def maximum_amount(coins) do

end
end
```

Erlang:

```
-spec maximum_amount(Coins :: [[integer()]]) -> integer().
maximum_amount(Coins) ->
.
```

Racket:

```
(define/contract (maximum-amount coins)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
class Solution {
public:
int maximumAmount(vector<vector<int>>& coins) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int maximumAmount(int[][] coins) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Amount of Money Robot Can Earn
Difficulty: Medium
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def maximumAmount(self, coins: List[List[int]]) -> int:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[][]} coins
* @return {number}
*/
var maximumAmount = function(coins) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

function maximumAmount(coins: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int MaximumAmount(int[][] coins) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Amount of Money Robot Can Earn
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int maximumAmount(int** coins, int coinsSize, int* coinsColSize) {

}
```

Go Solution:

```
// Problem: Maximum Amount of Money Robot Can Earn
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func maximumAmount(coins [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maximumAmount(coins: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func maximumAmount(_ coins: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Amount of Money Robot Can Earn
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn maximum_amount(coins: Vec<Vec<i32>>) -> i32 {

}
```

```
}
```

Ruby Solution:

```
# @param {Integer[][]} coins
# @return {Integer}
def maximum_amount(coins)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $coins
* @return Integer
*/
function maximumAmount($coins) {

}
}
```

Dart Solution:

```
class Solution {
int maximumAmount(List<List<int>> coins) {

}
}
```

Scala Solution:

```
object Solution {
def maximumAmount(coins: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_amount(coins :: [[integer]]) :: integer
def maximum_amount(coins) do

end
end
```

Erlang Solution:

```
-spec maximum_amount(Coins :: [[integer()]]) -> integer().
maximum_amount(Coins) ->
.
```

Racket Solution:

```
(define/contract (maximum-amount coins)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```