Problem 1975: Maximum Matrix Sum

Problem Information

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

Problem Description

You are given an

n x n

integer

matrix

. You can do the following operation

any

number of times:

Choose any two

adjacent

elements of

matrix

and

multiply

each of them by

-1

.

Two elements are considered

adjacent

if and only if they share a

border

.

Your goal is to

maximize

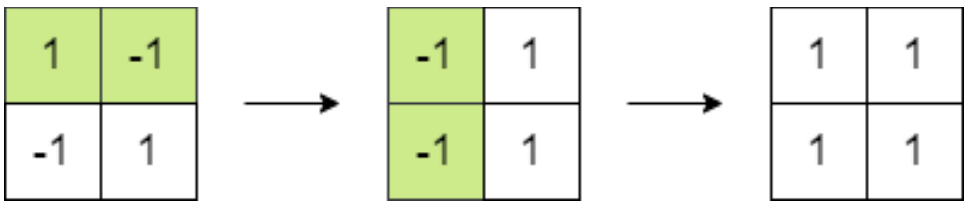
the summation of the matrix's elements. Return

the

maximum

sum of the matrix's elements using the operation mentioned above.

Example 1:



Input:

matrix = [[1,-1],[-1,1]]

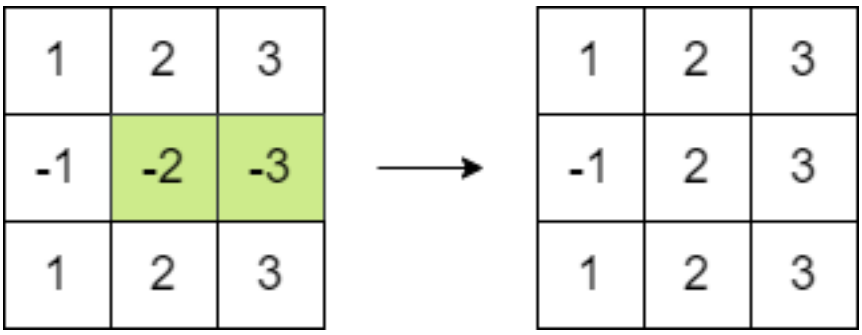
Output:

4

Explanation:

We can follow the following steps to reach sum equals 4: - Multiply the 2 elements in the first row by -1. - Multiply the 2 elements in the first column by -1.

Example 2:



Input:

matrix = [[1,2,3],[-1,-2,-3],[1,2,3]]

Output:

16

Explanation:

We can follow the following step to reach sum equals 16: - Multiply the 2 last elements in the second row by -1.

Constraints:

n == matrix.length == matrix[i].length

2 <= n <= 250

-10

5

<= matrix[i][j] <= 10

5

Code Snippets

C++:

```
class Solution {
public:
long long maxMatrixSum(vector<vector<int>>& matrix) {

}
};
```

Java:

```
class Solution {
public long maxMatrixSum(int[][] matrix) {

}
}
```

Python3:

```
class Solution:
def maxMatrixSum(self, matrix: List[List[int]]) -> int:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function maxMatrixSum(matrix: number[][]): number {

};
```

C#:

```
public class Solution {
public long MaxMatrixSum(int[][] matrix) {

}
}
```

C:

```
long long maxMatrixSum(int** matrix, int matrixSize, int* matrixColSize) {

}
```

Go:

```
func maxMatrixSum(matrix [][]int) int64 {

}
```

Kotlin:

```
class Solution {
fun maxMatrixSum(matrix: Array<IntArray>): Long {

}
}
```

Swift:

```
class Solution {
func maxMatrixSum(_ matrix: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn max_matrix_sum(matrix: Vec<Vec<i32>>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[][]} matrix
# @return {Integer}
def max_matrix_sum(matrix)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $matrix
* @return Integer
*/
function maxMatrixSum($matrix) {

}
}
```

Dart:

```
class Solution {
int maxMatrixSum(List<List<int>> matrix) {

}
}
```

Scala:

```
object Solution {
def maxMatrixSum(matrix: Array[Array[Int]]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_matrix_sum(matrix :: [[integer]]) :: integer
def max_matrix_sum(matrix) do

end
end
```

Erlang:

```
-spec max_matrix_sum(Matrix :: [[integer()]]) -> integer().
max_matrix_sum(Matrix) ->
.
```

Racket:

```
(define/contract (max-matrix-sum matrix)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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 maxMatrixSum(vector<vector<int>>& matrix) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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 maxMatrixSum(int[][] matrix) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Matrix Sum
Difficulty: Medium
Tags: array, greedy

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 maxMatrixSum(self, matrix: List[List[int]]) -> int:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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[][]} matrix
* @return {number}
*/
var maxMatrixSum = function(matrix) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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 maxMatrixSum(matrix: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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 MaxMatrixSum(int[][] matrix) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Matrix Sum
* Difficulty: Medium
* Tags: array, greedy
*
* 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 maxMatrixSum(int** matrix, int matrixSize, int* matrixColSize) {

}
```

Go Solution:

```
// Problem: Maximum Matrix Sum
// Difficulty: Medium
// Tags: array, greedy
//
// 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 maxMatrixSum(matrix [][]int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun maxMatrixSum(matrix: Array<IntArray>): Long {

}
}
```

Swift Solution:

```
class Solution {
func maxMatrixSum(_ matrix: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Matrix Sum
// Difficulty: Medium
// Tags: array, greedy
//
// 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 max_matrix_sum(matrix: Vec<Vec<i32>>) -> i64 {

}
```

```
}
```

Ruby Solution:

```
# @param {Integer[][]} matrix
# @return {Integer}
def max_matrix_sum(matrix)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $matrix
* @return Integer
*/
function maxMatrixSum($matrix) {

}
}
```

Dart Solution:

```
class Solution {
int maxMatrixSum(List<List<int>> matrix) {

}
}
```

Scala Solution:

```
object Solution {
def maxMatrixSum(matrix: Array[Array[Int]]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_matrix_sum(matrix :: [[integer]]) :: integer
def max_matrix_sum(matrix) do

end
end
```

Erlang Solution:

```
-spec max_matrix_sum(Matrix :: [[integer()]]) -> integer().
max_matrix_sum(Matrix) ->
.
```

Racket Solution:

```
(define/contract (max-matrix-sum matrix)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```