Problem 1895: Largest Magic Square

Problem Information

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

Problem Description

A

k x k

magic square

is a

k x k

grid filled with integers such that every row sum, every column sum, and both diagonal sums are

all equal

. The integers in the magic square

do not have to be distinct

. Every

1 x 1

grid is trivially a

magic square

.

Given an

m x n

integer

grid

, return

the

size

(i.e., the side length

k

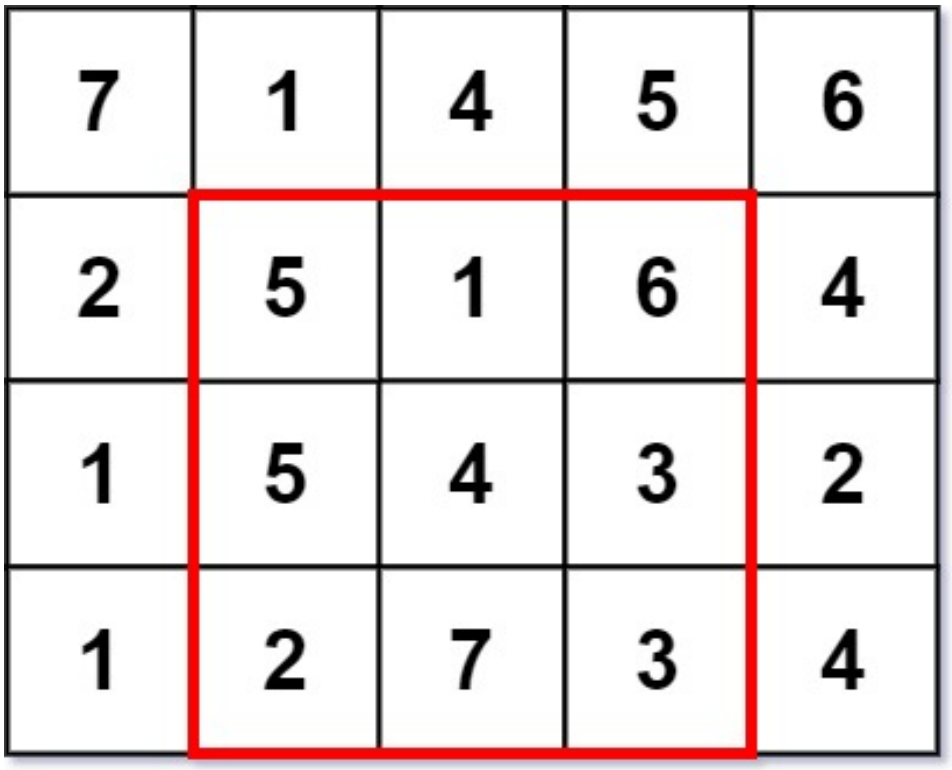
) of the

largest magic square

that can be found within this grid

.

Example 1:



Input:

grid = [[7,1,4,5,6],[2,5,1,6,4],[1,5,4,3,2],[1,2,7,3,4]]

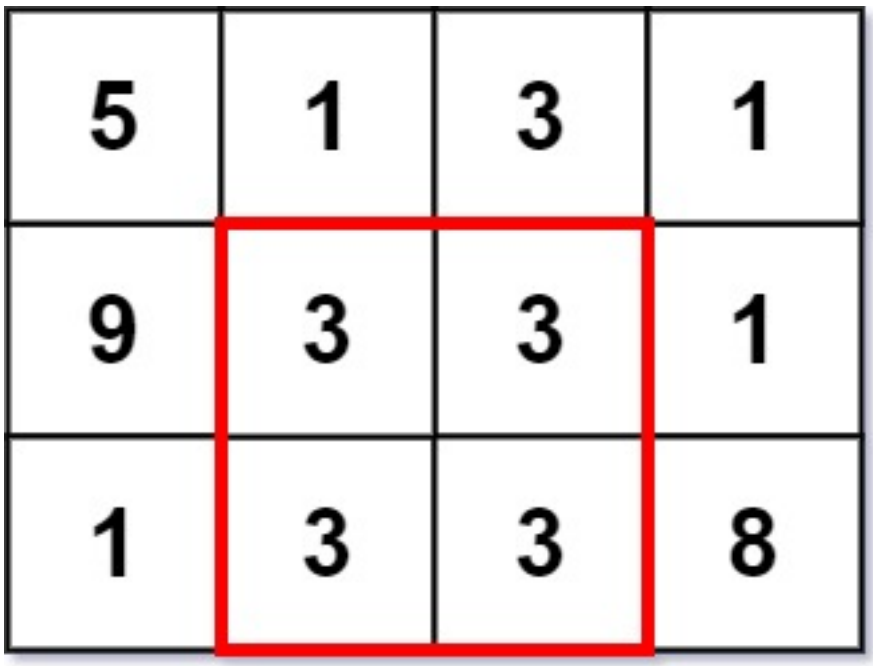
Output:

3

Explanation:

The largest magic square has a size of 3. Every row sum, column sum, and diagonal sum of this magic square is equal to 12. - Row sums: 5+1+6 = 5+4+3 = 2+7+3 = 12 - Column sums: 5+5+2 = 1+4+7 = 6+3+3 = 12 - Diagonal sums: 5+4+3 = 6+4+2 = 12

Example 2:



Input:

grid = [[5,1,3,1],[9,3,3,1],[1,3,3,8]]

Output:

2

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 50

1 <= grid[i][j] <= 10

6

Code Snippets

C++:

```
class Solution {
public:
int largestMagicSquare(vector<vector<int>>& grid) {

}
};
```

Java:

```
class Solution {
public int largestMagicSquare(int[][] grid) {

}
}
```

Python3:

```
class Solution:
def largestMagicSquare(self, grid: List[List[int]]) -> int:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function largestMagicSquare(grid: number[][]): number {

};
```

C#:

```
public class Solution {
public int LargestMagicSquare(int[][] grid) {

}
}
```

C:

```
int largestMagicSquare(int** grid, int gridSize, int* gridColSize) {

}
```

Go:

```
func largestMagicSquare(grid [][]int) int {

}
```

Kotlin:

```
class Solution {
fun largestMagicSquare(grid: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func largestMagicSquare(_ grid: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn largest_magic_square(grid: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} grid
# @return {Integer}
def largest_magic_square(grid)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $grid
* @return Integer
*/
function largestMagicSquare($grid) {

}
}
```

Dart:

```
class Solution {
int largestMagicSquare(List<List<int>> grid) {

}
}
```

Scala:

```
object Solution {
def largestMagicSquare(grid: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec largest_magic_square(grid :: [[integer]]) :: integer
def largest_magic_square(grid) do

end
end
```

Erlang:

```
-spec largest_magic_square(Grid :: [[integer()]]) -> integer().
largest_magic_square(Grid) ->
.
```

Racket:

```
(define/contract (largest-magic-square grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Largest Magic Square
* Difficulty: Medium
* 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 largestMagicSquare(vector<vector<int>>& grid) {

}
};
```

Java Solution:

```
/**
* Problem: Largest Magic Square
* Difficulty: Medium
* 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 largestMagicSquare(int[][] grid) {

}
}
```

Python3 Solution:

```
"""
Problem: Largest Magic Square
Difficulty: Medium
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 largestMagicSquare(self, grid: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

```
"""
```

JavaScript Solution:

```
/**
* Problem: Largest Magic Square
* Difficulty: Medium
* 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[][]} grid
* @return {number}
*/
var largestMagicSquare = function(grid) {

};
```

TypeScript Solution:

```
/**
* Problem: Largest Magic Square
* Difficulty: Medium
* 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 largestMagicSquare(grid: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Largest Magic Square
* Difficulty: Medium
* 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 LargestMagicSquare(int[][] grid) {

}
}
```

C Solution:

```
/*
* Problem: Largest Magic Square
* Difficulty: Medium
* 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 largestMagicSquare(int** grid, int gridSize, int* gridColSize) {

}
```

Go Solution:

```
// Problem: Largest Magic Square
// Difficulty: Medium
// 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 largestMagicSquare(grid [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun largestMagicSquare(grid: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func largestMagicSquare(_ grid: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Largest Magic Square
// Difficulty: Medium
// 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 largest_magic_square(grid: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} grid
# @return {Integer}
def largest_magic_square(grid)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $grid
* @return Integer
*/
function largestMagicSquare($grid) {

}
}
```

Dart Solution:

```
class Solution {
int largestMagicSquare(List<List<int>> grid) {

}
}
```

Scala Solution:

```
object Solution {
def largestMagicSquare(grid: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec largest_magic_square(grid :: [[integer]]) :: integer
def largest_magic_square(grid) do

end
end
```

Erlang Solution:

```
-spec largest_magic_square(Grid :: [[integer()]]) -> integer().
largest_magic_square(Grid) ->
.
```

Racket Solution:

```
(define/contract (largest-magic-square grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```