Problem 3643: Flip Square Submatrix Vertically

Problem Information

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

Problem Description

You are given an

m x n

integer matrix

grid

, and three integers

x

,

y

, and

k

.

The integers

x

and

y

represent the row and column indices of the

top-left

corner of a

square

submatrix and the integer

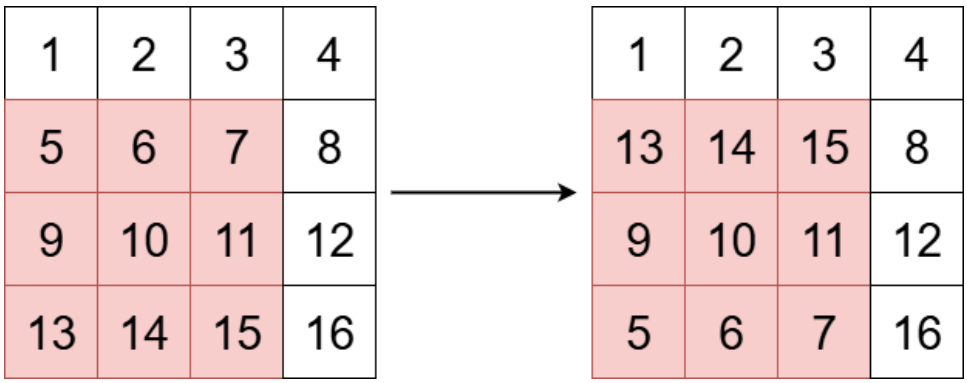
k

represents the size (side length) of the square submatrix.

Your task is to flip the submatrix by reversing the order of its rows vertically.

Return the updated matrix.

Example 1:



Input:

grid =

[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]

, x = 1, y = 0, k = 3

Output:

[[1,2,3,4],[13,14,15,8],[9,10,11,12],[5,6,7,16]]

Explanation:

The diagram above shows the grid before and after the transformation.

Example 2:

Input:

grid = [[3,4,2,3],[2,3,4,2]], x = 0, y = 2, k = 2

Output:

[[3,4,4,2],[2,3,2,3]]

Explanation:

The diagram above shows the grid before and after the transformation.

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 50

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

0 <= x < m

0 <= y < n

1 <= k <= min(m - x, n - y)

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> reverseSubmatrix(vector<vector<int>>& grid, int x, int y,
int k) {

}
};
```

Java:

```
class Solution {
public int[][] reverseSubmatrix(int[][] grid, int x, int y, int k) {

}
}
```

Python3:

```
class Solution:
def reverseSubmatrix(self, grid: List[List[int]], x: int, y: int, k: int) ->
List[List[int]]:
```

Python:

```
class Solution(object):
def reverseSubmatrix(self, grid, x, y, k):
"""
:type grid: List[List[int]]
:type x: int
:type y: int
```

```
:type k: int
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} grid
* @param {number} x
* @param {number} y
* @param {number} k
* @return {number[][]}
*/
var reverseSubmatrix = function(grid, x, y, k) {

};
```

TypeScript:

```
function reverseSubmatrix(grid: number[][], x: number, y: number, k: number):
number[][] {

};
```

C#:

```
public class Solution {
public int[][] ReverseSubmatrix(int[][] grid, int x, int y, int k) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** reverseSubmatrix(int** grid, int gridSize, int* gridColSize, int x, int
y, int k, int* returnSize, int** returnColumnSizes) {
```

```
}
```

Go:

```
func reverseSubmatrix(grid [][]int, x int, y int, k int) [][]int {

}
```

Kotlin:

```
class Solution {
fun reverseSubmatrix(grid: Array<IntArray>, x: Int, y: Int, k: Int):
Array<IntArray> {

}
}
```

Swift:

```
class Solution {
func reverseSubmatrix(_ grid: [[Int]], _ x: Int, _ y: Int, _ k: Int) ->
[[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn reverse_submatrix(grid: Vec<Vec<i32>>, x: i32, y: i32, k: i32) ->
Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} grid
# @param {Integer} x
# @param {Integer} y
# @param {Integer} k
```

```
# @return {Integer[][]}
def reverse_submatrix(grid, x, y, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $grid
* @param Integer $x
* @param Integer $y
* @param Integer $k
* @return Integer[][]
*/
function reverseSubmatrix($grid, $x, $y, $k) {

}
}
```

Dart:

```
class Solution {
List<List<int>> reverseSubmatrix(List<List<int>> grid, int x, int y, int k) {

}
}
```

Scala:

```
object Solution {
def reverseSubmatrix(grid: Array[Array[Int]], x: Int, y: Int, k: Int):
Array[Array[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec reverse_submatrix(grid :: [[integer]], x :: integer, y :: integer, k ::
```

```
integer) :: [[integer]]
def reverse_submatrix(grid, x, y, k) do

end
end
```

Erlang:

```
-spec reverse_submatrix(Grid :: [[integer()]], X :: integer(), Y ::
integer(), K :: integer()) -> [[integer()]].
reverse_submatrix(Grid, X, Y, K) ->
.
```

Racket:

```
(define/contract (reverse-submatrix grid x y k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer? (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Flip Square Submatrix Vertically
* 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:
vector<vector<int>> reverseSubmatrix(vector<vector<int>>& grid, int x, int y,
int k) {

}
};
```

Java Solution:

```
/**
* Problem: Flip Square Submatrix Vertically
* 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[][] reverseSubmatrix(int[][] grid, int x, int y, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Flip Square Submatrix Vertically
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 reverseSubmatrix(self, grid: List[List[int]], x: int, y: int, k: int) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def reverseSubmatrix(self, grid, x, y, k):
"""
:type grid: List[List[int]]
```

```
:type x: int
:type y: int
:type k: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Flip Square Submatrix Vertically
* 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[][]} grid
* @param {number} x
* @param {number} y
* @param {number} k
* @return {number[][]}
*/
var reverseSubmatrix = function(grid, x, y, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Flip Square Submatrix Vertically
* 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 reverseSubmatrix(grid: number[][], x: number, y: number, k: number):
number[][] {

};
```

C# Solution:

```
/*
* Problem: Flip Square Submatrix Vertically
* 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[][] ReverseSubmatrix(int[][] grid, int x, int y, int k) {

}
}
```

C Solution:

```
/*
* Problem: Flip Square Submatrix Vertically
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
```

```
int** reverseSubmatrix(int** grid, int gridSize, int* gridColSize, int x, int
y, int k, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Flip Square Submatrix Vertically
// 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 reverseSubmatrix(grid [][]int, x int, y int, k int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun reverseSubmatrix(grid: Array<IntArray>, x: Int, y: Int, k: Int):
Array<IntArray> {

}
}
```

Swift Solution:

```
class Solution {
func reverseSubmatrix(_ grid: [[Int]], _ x: Int, _ y: Int, _ k: Int) ->
[[Int]] {

}
}
```

Rust Solution:

```
// Problem: Flip Square Submatrix Vertically
// 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 reverse_submatrix(grid: Vec<Vec<i32>>, x: i32, y: i32, k: i32) ->
Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} grid
# @param {Integer} x
# @param {Integer} y
# @param {Integer} k
# @return {Integer[][]}
def reverse_submatrix(grid, x, y, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $grid
* @param Integer $x
* @param Integer $y
* @param Integer $k
* @return Integer[][]
*/
function reverseSubmatrix($grid, $x, $y, $k) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> reverseSubmatrix(List<List<int>> grid, int x, int y, int k) {

}
}
```

Scala Solution:

```
object Solution {
def reverseSubmatrix(grid: Array[Array[Int]], x: Int, y: Int, k: Int):
Array[Array[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec reverse_submatrix(grid :: [[integer]], x :: integer, y :: integer, k ::
integer) :: [[integer]]
def reverse_submatrix(grid, x, y, k) do

end
end
```

Erlang Solution:

```
-spec reverse_submatrix(Grid :: [[integer()]], X :: integer(), Y ::
integer(), K :: integer()) -> [[integer()]].
reverse_submatrix(Grid, X, Y, K) ->
.
```

Racket Solution:

```
(define/contract (reverse-submatrix grid x y k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer? (listof (listof exact-integer?)))
)
```