Problem 519: Random Flip Matrix

Problem Information

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

Problem Description

There is an

m x n

binary grid

matrix

with all the values set

0

initially. Design an algorithm to randomly pick an index

(i, j)

where

matrix[i][j] == 0

and flips it to

1

. All the indices

(i, j)

where

matrix[i][j] == 0

should be equally likely to be returned.

Optimize your algorithm to minimize the number of calls made to the

built-in

random function of your language and optimize the time and space complexity.

Implement the

Solution

class:

Solution(int m, int n)

Initializes the object with the size of the binary matrix

m

and

n

.

int[] flip()

Returns a random index

[i, j]

of the matrix where

matrix[i][j] == 0

and flips it to

1

.

void reset()

Resets all the values of the matrix to be

0

.

Example 1:

Input

["Solution", "flip", "flip", "flip", "reset", "flip"] [[3, 1], [], [], [], [], []]

Output

[null, [1, 0], [2, 0], [0, 0], null, [2, 0]]

Explanation

Solution solution = new Solution(3, 1); solution.flip(); // return [1, 0], [0,0], [1,0], and [2,0] should be equally likely to be returned. solution.flip(); // return [2, 0], Since [1,0] was returned, [2,0] and [0,0] solution.flip(); // return [0, 0], Based on the previously returned indices, only [0,0] can be returned. solution.reset(); // All the values are reset to 0 and can be returned. solution.flip(); // return [2, 0], [0,0], [1,0], and [2,0] should be equally likely to be returned.

Constraints:

1 <= m, n <= 10

4

There will be at least one free cell for each call to

flip

.

At most

1000

calls will be made to

flip

and

reset

.

Code Snippets

C++:

```
class Solution {
public:
Solution(int m, int n) {

}

vector<int> flip() {

}

void reset() {

}
};
```

```
/**
* Your Solution object will be instantiated and called as such:
* Solution* obj = new Solution(m, n);
* vector<int> param_1 = obj->flip();
* obj->reset();
*/
```

Java:

```
class Solution {

public Solution(int m, int n) {

}

public int[] flip() {

}

public void reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(m, n);
* int[] param_1 = obj.flip();
* obj.reset();
*/
```

Python3:

```
class Solution:

def __init__(self, m: int, n: int):



def flip(self) -> List[int]:



def reset(self) -> None:
```

```
# Your Solution object will be instantiated and called as such:
# obj = Solution(m, n)
# param_1 = obj.flip()
# obj.reset()
```

Python:

```
class Solution(object):

def __init__(self, m, n):
"""
:type m: int
:type n: int
"""



def flip(self):
"""
:rtype: List[int]
"""



def reset(self):
"""
:rtype: None
"""




# Your Solution object will be instantiated and called as such:
# obj = Solution(m, n)
# param_1 = obj.flip()
# obj.reset()
```

JavaScript:

```
/**
* @param {number} m
* @param {number} n
```

```
*/
var Solution = function(m, n) {

};

/**
* @return {number[]}
*/
Solution.prototype.flip = function() {

};

/**
* @return {void}
*/
Solution.prototype.reset = function() {

};

/**
* Your Solution object will be instantiated and called as such:
* var obj = new Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

TypeScript:

```
class Solution {
constructor(m: number, n: number) {

}

flip(): number[] {

}

reset(): void {

}
}
```

```
/**
* Your Solution object will be instantiated and called as such:
* var obj = new Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

C#:

```
public class Solution {

public Solution(int m, int n) {

}

public int[] Flip() {

}

public void Reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(m, n);
* int[] param_1 = obj.Flip();
* obj.Reset();
*/
```

C:

```
typedef struct {

} Solution;


Solution* solutionCreate(int m, int n) {
```

```
}

int* solutionFlip(Solution* obj, int* retSize) {

}

void solutionReset(Solution* obj) {

}

void solutionFree(Solution* obj) {

}

/**
* Your Solution struct will be instantiated and called as such:
* Solution* obj = solutionCreate(m, n);
* int* param_1 = solutionFlip(obj, retSize);

* solutionReset(obj);

* solutionFree(obj);
*/
```

Go:

```
type Solution struct {

}



func Constructor(m int, n int) Solution {

}



func (this *Solution) Flip() []int {

}
```

```
func (this *Solution) Reset() {

}



/**
* Your Solution object will be instantiated and called as such:
* obj := Constructor(m, n);
* param_1 := obj.Flip();
* obj.Reset();
*/
```

Kotlin:

```
class Solution(m: Int, n: Int) {

fun flip(): IntArray {

}

fun reset() {

}

}

/**
* Your Solution object will be instantiated and called as such:
* var obj = Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

Swift:

```
class Solution {

init(_ m: Int, _ n: Int) {

}
```

```
func flip() -> [Int] {

}

func reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution(m, n)
* let ret_1: [Int] = obj.flip()
* obj.reset()
*/
```

Rust:

```
struct Solution {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Solution {

fn new(m: i32, n: i32) -> Self {

}

fn flip(&self) -> Vec<i32> {

}

fn reset(&self) {

}
}
```

```
/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution::new(m, n);
* let ret_1: Vec<i32> = obj.flip();
* obj.reset();
*/
```

Ruby:

```
class Solution

=begin
:type m: Integer
:type n: Integer
=end
def initialize(m, n)

end



=begin
:rtype: Integer[]
=end
def flip()

end



=begin
:rtype: Void
=end
def reset()

end



end

# Your Solution object will be instantiated and called as such:
# obj = Solution.new(m, n)
# param_1 = obj.flip()
# obj.reset()
```

PHP:

```
class Solution {
/**
* @param Integer $m
* @param Integer $n
*/
function __construct($m, $n) {

}

/**
* @return Integer[]
*/
function flip() {

}

/**
* @return NULL
*/
function reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* $obj = Solution($m, $n);
* $ret_1 = $obj->flip();
* $obj->reset();
*/
```

Dart:

```
class Solution {

Solution(int m, int n) {

}

List<int> flip() {
```

```
}

void reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = Solution(m, n);
* List<int> param1 = obj.flip();
* obj.reset();
*/
```

Scala:

```
class Solution(_m: Int, _n: Int) {

def flip(): Array[Int] = {

}

def reset(): Unit = {

}

}

/**
* Your Solution object will be instantiated and called as such:
* val obj = new Solution(m, n)
* val param_1 = obj.flip()
* obj.reset()
*/
```

Elixir:

```
defmodule Solution do
@spec init_(m :: integer, n :: integer) :: any
def init_(m, n) do
```

```
end

@spec flip() :: [integer]
def flip() do

end

@spec reset() :: any
def reset() do

end
end

# Your functions will be called as such:
# Solution.init_(m, n)
# param_1 = Solution.flip()
# Solution.reset()

# Solution.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang:

```
-spec solution_init_(M :: integer(), N :: integer()) -> any().
solution_init_(M, N) ->
.

-spec solution_flip() -> [integer()].
solution_flip() ->
.

-spec solution_reset() -> any().
solution_reset() ->
.


%% Your functions will be called as such:
%% solution_init_(M, N),
%% Param_1 = solution_flip(),
%% solution_reset(),

%% solution_init_ will be called before every test case, in which you can do
```

```
some necessary initializations.
```

Racket:

```
(define solution%
(class object%
(super-new)

; m : exact-integer?
; n : exact-integer?
(init-field
m
n)

; flip : -> (listof exact-integer?)
(define/public (flip)
)
; reset : -> void?
(define/public (reset)
)))

;; Your solution% object will be instantiated and called as such:
;; (define obj (new solution% [m m] [n n]))
;; (define param_1 (send obj flip))
;; (send obj reset)
```

Solutions

C++ Solution:

```
/*
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class Solution {
```

```
public:
Solution(int m, int n) {

}

vector<int> flip() {

}

void reset() {

}
};

/**
* Your Solution object will be instantiated and called as such:
* Solution* obj = new Solution(m, n);
* vector<int> param_1 = obj->flip();
* obj->reset();
*/
```

Java Solution:

```
/**
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class Solution {

public Solution(int m, int n) {

}

public int[] flip() {
```

```
}

public void reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(m, n);
* int[] param_1 = obj.flip();
* obj.reset();
*/
```

Python3 Solution:

```
"""
Problem: Random Flip Matrix
Difficulty: Medium
Tags: math, hash

Approach: Use hash map for O(1) lookups
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(n) for hash map
"""

class Solution:

def __init__(self, m: int, n: int):



def flip(self) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):

def __init__(self, m, n):
"""
```

```
:type m: int
:type n: int
"""


def flip(self):
"""
:rtype: List[int]
"""


def reset(self):
"""
:rtype: None
"""



# Your Solution object will be instantiated and called as such:
# obj = Solution(m, n)
# param_1 = obj.flip()
# obj.reset()
```

JavaScript Solution:

```
/**
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} m
* @param {number} n
*/
var Solution = function(m, n) {
```

```
};

/**
* @return {number[]}
*/
Solution.prototype.flip = function() {

};

/**
* @return {void}
*/
Solution.prototype.reset = function() {

};

/**
* Your Solution object will be instantiated and called as such:
* var obj = new Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

TypeScript Solution:

```
/**
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class Solution {
constructor(m: number, n: number) {

}

flip(): number[] {
```

```
}

reset(): void {

}
}

/**
* Your Solution object will be instantiated and called as such:
* var obj = new Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

C# Solution:

```
/*
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

public class Solution {

public Solution(int m, int n) {

}

public int[] Flip() {

}

public void Reset() {

}
}
```

```
/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(m, n);
* int[] param_1 = obj.Flip();
* obj.Reset();
*/
```

C Solution:

```
/*
* Problem: Random Flip Matrix
* Difficulty: Medium
* Tags: math, hash
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/




typedef struct {

} Solution;


Solution* solutionCreate(int m, int n) {

}

int* solutionFlip(Solution* obj, int* retSize) {

}

void solutionReset(Solution* obj) {

}

void solutionFree(Solution* obj) {
```

```
}

/**
* Your Solution struct will be instantiated and called as such:
* Solution* obj = solutionCreate(m, n);
* int* param_1 = solutionFlip(obj, retSize);

* solutionReset(obj);

* solutionFree(obj);
*/
```

Go Solution:

```
// Problem: Random Flip Matrix
// Difficulty: Medium
// Tags: math, hash
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

type Solution struct {

}


func Constructor(m int, n int) Solution {

}


func (this *Solution) Flip() []int {

}


func (this *Solution) Reset() {

}
```

```
/**
* Your Solution object will be instantiated and called as such:
* obj := Constructor(m, n);
* param_1 := obj.Flip();
* obj.Reset();
*/
```

Kotlin Solution:

```
class Solution(m: Int, n: Int) {

fun flip(): IntArray {

}

fun reset() {

}

}

/**
* Your Solution object will be instantiated and called as such:
* var obj = Solution(m, n)
* var param_1 = obj.flip()
* obj.reset()
*/
```

Swift Solution:

```
class Solution {

init(_ m: Int, _ n: Int) {

}

func flip() -> [Int] {
```

```
}

func reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution(m, n)
* let ret_1: [Int] = obj.flip()
* obj.reset()
*/
```

Rust Solution:

```
// Problem: Random Flip Matrix
// Difficulty: Medium
// Tags: math, hash
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

struct Solution {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Solution {

fn new(m: i32, n: i32) -> Self {

}

fn flip(&self) -> Vec<i32> {
```

```
}

fn reset(&self) {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution::new(m, n);
* let ret_1: Vec<i32> = obj.flip();
* obj.reset();
*/
```

Ruby Solution:

```
class Solution

=begin
:type m: Integer
:type n: Integer
=end
def initialize(m, n)

end


=begin
:rtype: Integer[]
=end
def flip()

end


=begin
:rtype: Void
=end
def reset()

end
```

```
end

# Your Solution object will be instantiated and called as such:
# obj = Solution.new(m, n)
# param_1 = obj.flip()
# obj.reset()
```

PHP Solution:

```
class Solution {
/**
* @param Integer $m
* @param Integer $n
*/
function __construct($m, $n) {

}

/**
* @return Integer[]
*/
function flip() {

}

/**
* @return NULL
*/
function reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* $obj = Solution($m, $n);
* $ret_1 = $obj->flip();
* $obj->reset();
*/
```

Dart Solution:

```
class Solution {

Solution(int m, int n) {

}

List<int> flip() {

}

void reset() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = Solution(m, n);
* List<int> param1 = obj.flip();
* obj.reset();
*/
```

Scala Solution:

```
class Solution(_m: Int, _n: Int) {

def flip(): Array[Int] = {

}

def reset(): Unit = {

}

}

/**
* Your Solution object will be instantiated and called as such:
* val obj = new Solution(m, n)
* val param_1 = obj.flip()
```

```
* obj.reset()
*/
```

Elixir Solution:

```
defmodule Solution do
@spec init_(m :: integer, n :: integer) :: any
def init_(m, n) do

end

@spec flip() :: [integer]
def flip() do

end

@spec reset() :: any
def reset() do

end
end

# Your functions will be called as such:
# Solution.init_(m, n)
# param_1 = Solution.flip()
# Solution.reset()

# Solution.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang Solution:

```
-spec solution_init_(M :: integer(), N :: integer()) -> any().
solution_init_(M, N) ->
.

-spec solution_flip() -> [integer()].
solution_flip() ->
.

-spec solution_reset() -> any().
```

```
solution_reset() ->
.



%% Your functions will be called as such:
%% solution_init_(M, N),
%% Param_1 = solution_flip(),
%% solution_reset(),

%% solution_init_ will be called before every test case, in which you can do
some necessary initializations.
```

Racket Solution:

```
(define solution%
(class object%
(super-new)

; m : exact-integer?
; n : exact-integer?
(init-field
m
n)

; flip : -> (listof exact-integer?)
(define/public (flip)
)
; reset : -> void?
(define/public (reset)
)))

;; Your solution% object will be instantiated and called as such:
;; (define obj (new solution% [m m] [n n]))
;; (define param_1 (send obj flip))
;; (send obj reset)
```