Hash Table Medium Reservoir Sampling Math Randomized **Leetcode Link**

Problem Description

519. Random Flip Matrix

zero is flipped to one, it can no longer be selected.

In this LeetCode problem, we have a binary grid, matrix, which is initialized with all zeroes. The dimensions of the matrix are m x n. Our task is to design an algorithm that can randomly select an index (i, j) for which matrix[i][j] == 0, then flip that value to 1. It's crucial that each zero in the matrix is equally likely to be flipped. This means the selection process must be uniformly random. Once a

• Solution(int m, int n): This is the constructor that initializes the binary matrix with dimensions m x n.

and optimize both the time complexity (how fast the algorithm runs) and space complexity (how much memory it uses). We also need to implement a Solution class:

The problem requires us to solve this with a focus on efficiency: we should aim to use the random function as few times as possible

 int[] flip(): This function flips a random zero in the matrix to one and returns the index [i, j] of that element. void reset(): This function sets all the values in the matrix back to zero.

When trying to maintain the uniform random selection of zeros in the matrix while flipping and preventing a zero that's already been flipped to one from being flipped again, a direct approach of checking every time if an element is zero can lead to a large number of

- calls to the random function which is not efficient. To avoid this, we can use a mapping strategy.
- The intuition behind the solution is to simulate the matrix in a linear fashion rather than as a 2D grid. We can imagine that each cell of the matrix is an element in an array (linearly indexed from 0 to m*n - 1). The goal is to randomly pick an element from this array.
- An efficient way to do this is as follows: Randomly pick an index in the linear representation of the matrix.

Once it's picked, we need to ensure it's not selected again.

this index to the actual last index of the array. If the index has been picked before, we use the mapping to find the new, swapped index which we know is 0. • Each time we "flip" a cell, we decrement the range of indices we are picking from by one. This is because there's now one less 0 to choose from. • By storing a mapping that swaps the picked index with the last index in the range, we can continue to pick random indexes while

If it's the first time the index is picked and it's not in the mapping (which implies the cell is currently 0), we store the mapping of

This approach minimizes the number of calls to the random function, since each call results in a unique flip, and there's no need to

knowing they will always correspond to a 0 without having to verify each cell in the 2D grid. This is a space-efficient way to keep track of flipped indices.

Store the dimensions of the matrix (m and n) in self.m and self.n.

map entry points to the last unflipped index, otherwise it points to itself.

simply resets the two variables without the need to go through the matrix.

row and column numbers of the matrix and n is the number of columns.

Now, let's walk through the solution algorithm with this matrix:

 \circ Let's say we pick a random index x = 2.

% 3 = 2. So, matrix[0][2] is flipped from 0 to 1.

• We decrement self. total to 5.

Initialize an empty dictionary self.mp to hold the mappings.

the linear index of the zero value we will flip.

 \circ Calculate the total number of elements in the matrix (m * n) and store it in self.total.

Solution Approach The solution approach utilizes a hash map (or a dictionary in Python) to implement the mapping between the indices of the matrix

repeat random calls to find the next zero to flip. The use of mappings ensures efficient utilization of space and fast access times.

with each flip. Let's walk through the algorithm step by step: 1. Initialization (__init__ Method):

and the indices of a linear array that represents our matrix. This hash map is used to keep track of which indices have already been

flipped. Additionally, a variable to keep track of the total number of zeroes left unflipped in the matrix is used, which also reduces

• Decrement the self.total count since we are about to flip a zero to one, effectively reducing the count of zeroes by one. • Generate a random index x from 0 to self.total - 1. The number self.total acts as the range for the random numbers that corresponds to the number of zeroes left in the matrix.

2. The Flip Function (flip Method):

• Update the mapping: set self.mp[x] to be whatever is currently mapped to self.total, which is essentially the index of the last '0'. If self.total is not in the map, it simply maps to itself (since that is the current 'last index' that hasn't been flipped yet). Convert idx from a linear index to a 2D index (which corresponds to matrix indices (i, j)), and return it.

By following this approach, we avoid redundant random function calls and iterations over the matrix to check if an element is zero.

operations. The space complexity is also optimized because instead of storing the entire matrix, we only store the indices that have

Let's consider a small matrix of size 2x3, which means our matrix has 2 rows and 3 columns, or a 2 x 3 grid. Initially, all elements are

We can map this grid to a linear array from indices 0 to 5, with each cell's linear index calculated by i * n + j where (i, j) are the

been flipped; so at most the space will be 0(min(m*n, number of flips)). The reset method also operates in 0(1) time since it

The time complexity of the flip method is 0(1) on average since it requires a constant number of hash map and assignment

The crucial point is that the map holds a swap record of positions. If we've previously replaced the value at a position, then the

• Check if x is already in the mapping (self.mp). If it's not, use x; if it is, get the mapped value from self.mp. This value idx is

 \circ Reset self.total back to m * n, as we are resetting all values of the matrix back to zero. Clear the dictionary self.mp, because no indices will be flipped anymore.

Example Walkthrough

0, looking like this:

1. Initialization:

3. The Reset Function (reset Method):

1 0 0 0 2 0 0 0

Since 2 is not in self.mp, we use it directly. No mapping is needed yet.

• The matrix size is 2x3, so we initialize self.m = 2, self.n = 3, and thus self.total = self.m * self.n = 6. We also initialize an empty dictionary self.mp = {}, which will hold our mappings. 2. First Flip: We currently have self.total = 6, meaning there are 6 zeros available to flip.

3. Second Flip:

self.total is now 5.

• The matrix is now: 2 0 0 0

• We get matrix indices i = 2 // 3 = 0 and j = 2 % 3 = 2. But since matrix [0] [2] is already flipped to 1, we attempt to

update self.mp by making self.mp[2] point to self.mp[4], which is 5. This effectively swaps the picked index with one that

 \circ We determine the 2D matrix position of this index 2. We have i = idx // self.n = 2 // 3 = 0, and j = idx % self.n = 2

 \circ Suppose the random function gives us x = 4.

4. Third Flip:

- o Index 4 is not in self.mp, so we flip the cell at linear index 4 to 1, decrement self.total to 4, and update self.mp[4] = 5 to map to the last index of the current zero range. \circ In the 2D matrix, i = 4 // 3 = 1 and j = 4 % 3 = 1. So, matrix[1][1] changes to 1.
- 1 0 0 1 2 0 1 0

The matrix now looks like:

- Now self.total is 4. \circ We pick a random index x = 2. This time x is in self.mp. However, because we haven't remapped index 2 yet (since it was directly used the first time), we use it as is again, decrement self.total to 3.
- Since index 5 has not been flipped, we then use index 5 to flip in the matrix, which gives us i = 5 // 3 = 1 and j = 5 % 3 = 2. matrix[1][2] is flipped.

hasn't been used.

The updated matrix is:

2 0 1 1 5. Reset:

1 0 0 1

1 0 0 0

2 0 0 0

 If we want to reset the matrix, we set self.total back to 6 and clear self.mp. Our matrix is back to its initial state:

Python Solution

class Solution:

12

13

14

15

16

20

31

32

33

9

10

18

19

20

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

43

47

48

8

9

10

11

13

14

15

16

17

18

19

26

27

28

29

30

31

32

33

34

35

42

43

45

49

50

13

19 }

20

24

25

26

28

29

30

31

41 }

42

/**

*/

*/

- In summary, the linear mapping approach allows us to flip each zero in the matrix with an equal probability, while minimizing calls to the random function and avoiding checking all matrix cells multiple times. The use of a mapping dictionary efficiently tracks the cells that have been flipped, and the total variable ensures that the indices correspond to zeroes in our grid.
- from random import randint from typing import List
- 19 x = randint(0, self.total_cells) # Get the actual index to flip using the flip_map to handle collisions 20 21 idx = self.flip_map.get(x, x) # Map the current index to a new random index 23 self.flip_map[x] = self.flip_map.get(self.total_cells, self.total_cells) 24 # Convert the 1D index to 2D coordinates (row and column)

def reset(self) -> None:

35 # obj = Solution(m, n)

36 # param_1 = obj.flip()

37 # obj.reset()

Java Solution

1 import java.util.HashMap;

import java.util.Random;

private int rows;

private int cols;

private int totalCells;

public int[] flip() {

public void reset() {

totalCells = rows * cols;

* Solution obj = new Solution(m, n);

* int[] param_1 = obj.flip();

private Random randomGenerator = new Random();

private Map<Integer, Integer> flippedCellsMap = new HashMap<>();

// Flip a random cell and make sure not to flip the same cell again

int cellIndex = flippedCellsMap.getOrDefault(randomIndex, randomIndex);

// Find the actual cell index to be flipped, taking into account any previously flipped cells

// Number of rows in the matrix

int cellIndex = flippedCellsMap.count(randomIndex) ? flippedCellsMap[randomIndex] : randomIndex;

// Calculate and return the 'flipped' cell's coordinates as a vector containing row and column

* The following are the typical operations that will be performed on an instance of the Solution class:

// Typescript doesn't support public/private keywords without a class, so leaving it out

// Find the actual cell index to be flipped, taking into account any previously flipped cells

// Mark this cell as flipped by mapping it to the last available cell's index, effectively removing it from the pool of possible fl

// Mark this cell as flipped by mapping the chosen random index to the last available cell's index,

flippedCellsMap[randomIndex] = flippedCellsMap.count(totalCells) ? flippedCellsMap[totalCells] : totalCells;

std::mt19937 randomGenerator{rd()}; // Mersenne Twister random number generator

// Constructor with parameters for number of rows and columns of the matrix

Solution(int m, int n) : rows(m), cols(n), $totalCells(m * n) {}$

// Flip a random cell and ensure not to flip the same cell again

// effectively removing it from the pool of possible flips.

return {cellIndex / cols, cellIndex % cols};

std::unordered_map<int, int> flippedCellsMap; // Maps flipped cell index to a new index

// Number of columns in the matrix

// Total number of cells in the matrix

// Random device to seed the generator

flippedCellsMap.put(randomIndex, flippedCellsMap.getOrDefault(totalCells, totalCells));

// Mark this cell as flipped by mapping it to the last available cell's index, effectively removing it from the pool of possi

int randomIndex = randomGenerator.nextInt(totalCells--);

// return the 'flipped' cell's row and column as an array

// Reset the state of the Solution to allow all cells to be flipped again

return new int[] {cellIndex / cols, cellIndex % cols};

// Get a random index from the remaining cells

2 import java.util.Map;

class Solution {

self.flip map.clear()

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

self.total_cells = m * n

self.num_rows = m

self.num_cols = n

self.flip_map = {}

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

self.total_cells -= 1

Initialize the number of rows(m) and columns(n)

Calculate and store the total number of cells

return [idx // self.num_cols, idx % self.num_cols]

self.total cells = self.num rows * self.num cols

Your Solution object will be instantiated and called as such:

Reset the total number of available cells

Decrease the count of available cells

Generate a random index to flip

Clear the flip_map dictionary

Initialize a dictionary to keep track of flipped cell indices

- 11 12 // Constructor with parameters for number of rows and columns public Solution(int m, int n) { this.rows = m; 14 this.cols = n; 15 this.totalCells = m * n; 16 17
- flippedCellsMap.clear(); 37 38 39 40 /** 41 * The following are the typical operations that will be performed on an instance of the Solution class:

* obj.reset();

C++ Solution

1 #include <unordered map> 2 #include <random> #include <vector> class Solution {

private:

public:

int rows;

int cols;

int totalCells;

std::random_device rd;

std::vector<int> flip() {

- 20 // Get a random index from the remaining unflipped cells 21 std::uniform_int_distribution<> distrib(0, totalCells - 1); 22 int randomIndex = distrib(randomGenerator); 23 totalCells--; // Decrement the count of total cells as one will be flipped 24 25 // Find the actual cell index to be flipped, taking into account any previously flipped cells
- // Reset the state of the Solution to allow all cells to be flipped again 36 37 void reset() { totalCells = rows * cols; 38 39 flippedCellsMap.clear(); 40 41 };
- let rows: number; let cols: number; let totalCells: number; // Instantiate a Random number generator const randomGenerator: () => number = () => Math.floor(Math.random() * totalCells); 10

rows = m;

cols = n;

totalCells = m * n;

totalCells -= 1;

function flip(): number[] {

* Solution obj(m, n);

Typescript Solution

* obj.reset();

* std::vector<int> params = obj.flip();

// Define variables for rows, columns, and total cells

const flippedCellsMap: Map<number, number> = new Map<number, number>();

// Initialize the global state with number of rows and columns

// Flip a random cell and make sure not to flip the same cell again

const cellIndex: number = flippedCellsMap.get(randomIndex) ?? randomIndex;

flippedCellsMap.set(randomIndex, flippedCellsMap.get(totalCells) ?? totalCells);

function initializeSolution(m: number, n: number): void {

// Get a random index from the remaining cells

const randomIndex: number = randomGenerator();

// Define a map to keep track of flipped cells

32 // Return the 'flipped' cell's row and column as an array return [Math.floor(cellIndex / cols), cellIndex % cols]; 35 } 36 // Reset the state to allow all cells to be flipped again function reset(): void {

// Example of usage:

// reset();

totalCells = rows * cols;

flippedCellsMap.clear();

// initializeSolution(m, n);

Time and Space Complexity

// const param_1 = flip();

 For selecting a random index and flipping, the operations performed are constant time operations, including getting and setting values in a dictionary. Therefore, the time complexity of flip() is 0(1).

flip():

Time Complexity

• Clearing the hash map is an 0(1) operation assuming average case scenario where hash table operations have constant time complexity. However, if we consider the worse case scenario, where we have to traverse the hash map to clear it, it would have a time complexity of O(k) where k is the number of elements in the hash map. Since k can be at most the total number of flips

reset():

which is m * n, the worst case time complexity is 0(m * n). **Space Complexity**

• The hash map self.mp is used to keep track of the flipped indices, which can have at most m * n entries if all possible flips have occurred. Therefore, the maximum space complexity is 0(m * n) for storing these mappings.