Random Flip Matrix - Documentation

Table of Contents

1. Problem Statement

You are given an m x n binary matrix initialized with all 0s. Design an algorithm that:

- flip(): Randomly selects an index [i, j] such that matrix[i][j] == 0, and flips it to 1.
- reset(): Resets all values in the matrix back to 0.

All available (0-valued) positions should have equal probability of being chosen. You must optimize for time, space, and minimal random calls.

2. Intuition

Instead of maintaining the actual matrix, we can flatten it into a 1D list of size m * n and manage random selection with a hash map that acts like a dynamic shuffle array.

3. Key Observations

- A 2D matrix can be treated as a 1D array using the formula: i * n + j.
- Tracking which positions are flipped can be done with a hash map to simulate swaps without mutating an actual list.
- Once a position is flipped, it should not be picked again.

4. Approach

- Initialization:
  - total = m * n → total number of elements.
  - Use a hash map map to store simulated swaps.
- flip():
  - Choose a random index from [0, available - 1].
  - Get the actual position from map if exists, else use the index itself.
  - Simulate removing this index by mapping it to the last available index.
- reset():
  - Clear the map and reset available = total.

5. Edge Cases

- Repeated calls to flip() should not return the same cell until reset() is called.
- Calling reset() should make all indices available again.
- Supports large matrices efficiently (m or n up to 10^4).

6. Complexity Analysis

 Time Complexity

- flip(): O(1) – constant time hash lookup and math.
- reset(): O(1) – just clears the map and resets a counter.

💾 Space Complexity

- O(K) where K is the number of flipped cells (max 1000, as per constraints).

7. Alternative Approaches

❌ Brute Force

- Maintain the actual 2D matrix and scan for zeros.
- Time: O(mn) for each flip() call.
- Not feasible for large m, n.

✅ Reservoir Sampling (Less Practical Here)

- Inefficient without full knowledge of zero positions.

The hash map shuffle approach is superior in this context.

8. Test Cases

✅ Basic Functional Test

```
solution = Solution(3, 1)
print(solution.flip())  # One of [0,0], [1,0], [2,0]
print(solution.flip())  # One of remaining
print(solution.flip())  # The last one
solution.reset()
print(solution.flip())  # All are again valid
```

✅ Edge Case (Large Matrix)

```
solution = Solution(10000, 1)
print(solution.flip())  # Should return [x, 0] where x ∈ [0, 9999]
```

9. Final Thoughts

This problem is an excellent example of using hash maps to simulate shuffle operations efficiently. The approach is space-efficient, fast, and meets all the constraints of random uniformity and low overhead—ideal for high-performance systems dealing with large-scale matrices.