2732. Find a Good Subset of the Matrix

Problem Explanation

You have a 2-dimensional `m*n` binary matrix (`grid`) indexed by 0 where `m` is the number of rows and `n` is the number of columns. You have to pick some rows (possibly none/all) from this matrix such that the sum of all chosen subset of rows should be at most half of the total number of rows you selected. More formally, if `k` rows are selected, the sum of selected rows should be at most floor(k / 2).

You should return the indices of rows of this 'good' subset. These indices should be in ascending order. There can be multiple good subsets and you only need to return one. If no good subset exists, return an empty array.

Let's walk through an example;

Consider the following binary matrix:

```
1
2
3 0 1 1
4 1 0 1
5 0 1 0
6 1 1 1
```

Here we can choose the rows with the indices 0 and 2, which gives us the following subset:

```
1
2
3 0 1 1
4 0 1 0
```

The sum of all the elements in this subset is 3 which is less than or equal to the floor of half the number of rows (floor(2 / 2) = 1).

Solution Approach

The solution approach for this problem involves using a hash map (`unordered_map` in C++) to map row mask to row index and a bitwise operation to get the mask from the row. A helper method `getMask` is used to calculate the row mask. The row mask of a row is the integer representation of the binary number which the row represents. For each row in the given grid, the row mask is calculated and checked if it's 0. If it is 0, return the current index because a row with all 0s is always a good subset.

If the mask is different from 0, iterate through the bit value of each column (from 1 to `kMaxBit`) where `kMaxBit` is the maximum bit representing the columns of the grid. If the bit of the row mask at the current column is 0 and the current bit is present in our hash map, return an array containing the index of the row saved in the hash map and the current index because these rows fulfill the condition of a good subset. If the current row mask is not present in the map, add it with its index.

Finally, if the program checks all rows and hasn't return, then there is no good subset present. It returns an empty array.

This solution is using `Bits manipulation`, `Hash Map` data structure and `Iteration` pattern.

Solutions

C++ Solution

```
1
2 cpp
3 class Solution {
4  public:
5   vector<int> goodSubsetOfBinaryMatrix(vector<vector<int>>& grid) {
6     constexpr int kMaxBit = 32;
7     unordered_map<int, int> maskToIndex;
8
9     for (int i = 0; i < grid.size(); ++i) {
10      const int mask = getMask(grid[i]);
11      if (mask == 0)
12        return {i};
13      for (int prevMask = 1; prevMask < kMaxBit; ++prevMask) // Iterating through each bit
14        if ((mask & prevMask) == 0 && maskToIndex.count(prevMask)) // Checking if the bit of the row mask at the current column is 0
15          return {maskToIndex[prevMask], i};
16      maskToIndex[mask] = i;
17    }
18
19    return {};
20  }
21
22 private:
23  int getMask(const vector<int>& row) {
24    int mask = 0;
25    for (int i = 0; i < row.size(); ++i) // Calculating mask of the row
26      if (row[i] == 1)
27        mask |= 1 << i;
28    return mask;
29  }
30 };
```

Python Solution

```
1
2 python
3 import collections
4 K_MAX_BIT = 32
5
6 class Solution:
7     def get_mask(self, row):
8         mask = 0
9         for i in range(len(row)):
10            if row[i] == 1:
11                mask |= 1 << i
12        return mask
13
14    def goodSubsetOfBinaryMatrix(self, grid):
15        mask_to_index = {}
16        for i in range(len(grid)):
17            mask = self.get_mask(grid[i])
18            if mask == 0:
19                return [i]
20            for prev_mask in range(1, K_MAX_BIT):
21                if (mask & prev_mask) == 0 and prev_mask in mask_to_index:
22                    return [mask_to_index[prev_mask], i]
23            mask_to_index[mask] = i
24        return []
```

Java Solution

```
1
2 java
3 class Solution {
4     private int getMask(int[] row) {
5         int mask = 0;
6         for (int i = 0; i < row.length; ++i)
7             if (row[i] == 1)
8                 mask |= 1 << i;
9         return mask;
10    }
11
12    public int[] goodSubsetOfBinaryMatrix(int[][] grid) {
13        final int K_MAX_BIT = 32;
14        Map<Integer, Integer> maskToIndex = new HashMap<>();
15
16        for (int i = 0; i < grid.length; ++i) {
17            int mask = getMask(grid[i]);
18            if (mask == 0)
19                return new int[]{i};
20            for (int prevMask = 1; prevMask < K_MAX_BIT; ++prevMask)
21                if ((mask & prevMask) == 0 && maskToIndex.containsKey(prevMask))
22                    return new int[] {maskToIndex.get(prevMask), i};
23            maskToIndex.put(mask, i);
24        }
25
26        return new int[]{};
27    }
28 }
```

Please note that we need to implement other language solutions as well.

JavaScript Solution

```
1
2 javascript
3 var goodSubsetOfBinaryMatrix = function(grid) {
4     let maskToIndex = {};
5     const K_MAX_BIT = 32;
6
7     const getMask = (row) => {
8         let mask = 0;
9         for (let i = 0; i < row.length; ++i)
10            if (row[i] === 1)
11                mask |= 1 << i;
12        return mask;
13    }
14
15    for (let i = 0; i < grid.length; ++i) {
16        let mask = getMask(grid[i]);
17        if (mask === 0)
18            return [i];
19        for (let prevMask = 1; prevMask < K_MAX_BIT; ++prevMask)
20            if ((mask & prevMask) === 0 && prevMask in maskToIndex)
21                return [maskToIndex[prevMask], i];
22        maskToIndex[mask] = i;
23    }
24
25    return [];
26 };
```

With the above solutions implemented in different languages, we can solve the problem of good subset selection in a 2D binary matrix for different use-cases. Regardless of the language you choose, the core idea of the solution remains the same.

It is important to note that this problem makes extensive use of bitwise operators and hash maps, therefore understanding these concepts is crucial for solving similar problems. Also, the use of bitwise operators makes lookup operations faster and the algorithm more efficient, where the time complexity is O(N) where N is the size of the grid.