<https://leetcode.com/problems/maximum-rows-covered-by-columns/description/>

You are given an m x n binary matrix matrix and an integer numSelect.

Your goal is to select exactly numSelect **distinct**columns from matrix such that you cover as many rows as possible.

A row is considered **covered** if all the 1's in that row are also part of a column that you have selected. If a row does not have any 1s, it is also considered covered.

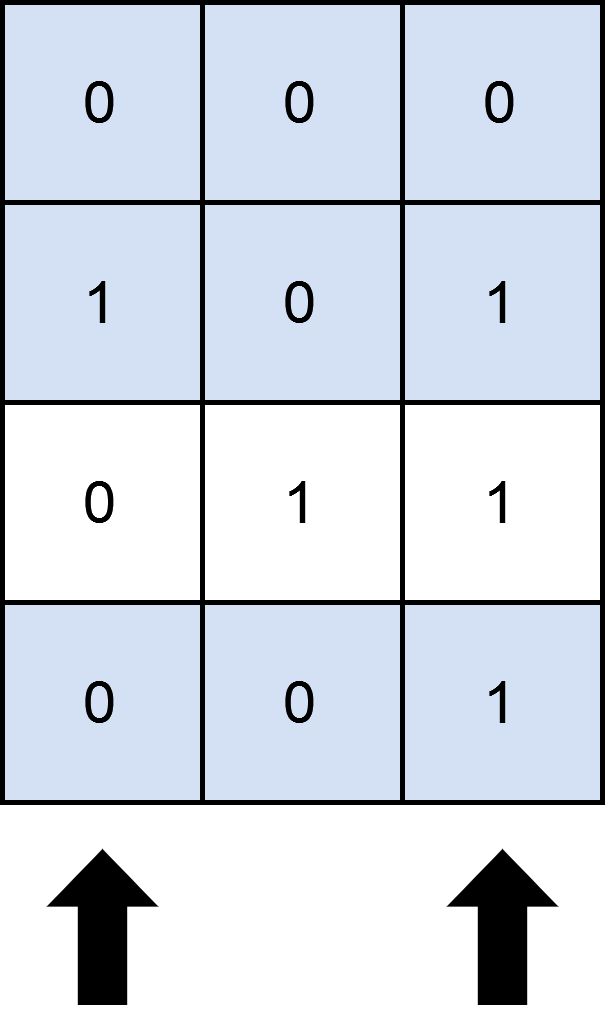
More formally, let us consider selected = {c1, c2, ...., cnumSelect} as the set of columns selected by you. A row i is **covered** by selected if:w

For each cell where matrix[i][j] == 1, the column j is in selected.

Or, no cell in row i has a value of 1.

Return the **maximum** number of rows that can be **covered** by a set of numSelect columns.

**Example 1:**



**Input:** matrix = [[0,0,0],[1,0,1],[0,1,1],[0,0,1]], numSelect = 2

**Output:** 3

**Explanation:**

One possible way to cover 3 rows is shown in the diagram above.

We choose s = {0, 2}.

- Row 0 is covered because it has no occurrences of 1.

- Row 1 is covered because the columns with value 1, i.e. 0 and 2 are present in s.

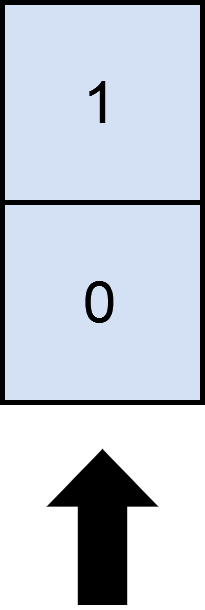
- Row 2 is not covered because matrix[2][1] == 1 but 1 is not present in s.

- Row 3 is covered because matrix[2][2] == 1 and 2 is present in s.

Thus, we can cover three rows.

Note that s = {1, 2} will also cover 3 rows, but it can be shown that no more than three rows can be covered.

**Example 2:**



**Input:** matrix = [[1],[0]], numSelect = 1

**Output:** 2

**Explanation:**

Selecting the only column will result in both rows being covered since the entire matrix is selected.

**Constraints:**

m == matrix.length

n == matrix[i].length

1 <= m, n <= 12

matrix[i][j] is either 0 or 1.

1 <= numSelect <= n

**Attempt 1: 2025-08-25**

**Solution 1: Bit Manipulation (180 min)**

class Solution {

    public int maximumRows(int[][] matrix, int numSelect) {

        int m = matrix.length;

        int n = matrix[0].length;

        // Step 1: Convert each row to a bitmask

        // Each row is represented as an integer where each bit indicates

        // whether the corresponding column has a 1 in that row

        int[] rowMasks = new int[m];

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(matrix[i][j] == 1) {

                    // Set the j-th bit in rowMasks[i]

                    // Example: if j=2, 1 << 2 = 4 (binary: 100)

                    // rowMasks[i] |= 4 sets the 3rd bit (0-indexed)

                    rowMasks[i] |= (1 << j);

                }

            }

        }

        int maxRows = 0;

        // Total possible column combinations: 2^n

        int totalMasks = 1 << n;

        // Step 2: Iterate through all possible column combinations

        for(int mask = 0; mask < totalMasks; mask++) {

            // Skip combinations that don't have exactly numSelect columns chosen

            if(Integer.bitCount(mask) != numSelect) {

                continue;

            }

            // Step 3: Count how many rows are covered by this column combination

            int count = 0;

            for(int rowMask : rowMasks) {

                // Check if all 1s in the row are within the selected columns

                // (rowMask & mask) extracts only the bits that are both in rowMask and mask

                // If this equals rowMask, it means all 1s in the row are covered by selected columns

                if((rowMask & mask) == rowMask) {

                    count++;

                }

            }

            maxRows = Math.max(maxRows, count);

        }

        return maxRows;

    }

}

Time Complexity: O(2^n \* m)

Space Complexity: O(m)

**Solution 2: Backtracking (30 min)**

class Solution {

    int maxRows = 0;

    public int maximumRows(int[][] matrix, int numSelect) {

        List<Integer> selectedColumns = new ArrayList<>();

        helper(matrix, numSelect, selectedColumns, 0);

        return maxRows;

    }

    private void helper(int[][] matrix, int numSelect, List<Integer> selectedColumns, int index) {

        if(selectedColumns.size() == numSelect) {

            int covered = countCoveredRows(matrix, selectedColumns);

            maxRows = Math.max(maxRows, covered);

            return;

        }

        for(int i = index; i < matrix[0].length; i++) {

            selectedColumns.add(i);

            helper(matrix, numSelect, selectedColumns, i + 1);

            selectedColumns.remove(selectedColumns.size() - 1);

        }

    }

    private int countCoveredRows(int[][] matrix, List<Integer> selectedColumns) {

        int count = 0;

        for(int i = 0; i < matrix.length; i++) {

            boolean covered = true;

            for(int j = 0; j < matrix[0].length; j++) {

                if(matrix[i][j] == 1 && !selectedColumns.contains(j)) {

                    covered = false;

                    break;

                }

            }

            if(covered) {

                count++;

            }

        }

        return count;

    }

}

Time Complexity: O(C(n, k) \* m \* n)

Space Complexity: O(n) for recursion stack

**Refer to Deepseek**

To solve this problem, we need to determine the maximum number of rows in a binary matrix that are covered by a selected set of columns. A row is considered covered if all the 1s in that row are within the selected columns.

**Approach**

**Problem Analysis**: The problem requires selecting a fixed number of columns such that as many rows as possible have all their 1s within these selected columns. Given the constraints (columns up to 12), we can use a bitmask approach to efficiently check all possible combinations of columns.

**Bitmask Representation**: Each row in the matrix can be represented as a bitmask where each bit indicates the presence of a 1 in the corresponding column. Similarly, each possible selection of columns can be represented as a bitmask.

**Check Coverage**: For each possible combination of columns (bitmask with exactly numSelect bits set), we check how many rows have their bitmask entirely contained within the selected columns bitmask. This is done using bitwise operations to ensure efficiency.

**Optimization**: By iterating over all possible bitmasks and counting the covered rows for each valid combination, we can find the maximum number of covered rows.

**Solution Code**

class Solution {

public int maximumRows(int[][] matrix, int numSelect) {

int m = matrix.length;

int n = matrix[0].length;

// Step 1: Convert each row to a bitmask

// Each row is represented as an integer where each bit indicates

// whether the corresponding column has a 1 in that row

int[] rowMasks = new int[m];

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (matrix[i][j] == 1) {

// Set the j-th bit in rowMasks[i]

// Example: if j=2, 1 << 2 = 4 (binary: 100)

// rowMasks[i] |= 4 sets the 3rd bit (0-indexed)

rowMasks[i] |= (1 << j);

}

}

}

int maxRows = 0;

int totalMasks = 1 << n; // Total possible column combinations: 2^n

// Step 2: Iterate through all possible column combinations

for (int mask = 0; mask < totalMasks; mask++) {

// Skip combinations that don't have exactly numSelect columns chosen

if (Integer.bitCount(mask) != numSelect) {

continue;

}

// Step 3: Count how many rows are covered by this column combination

int count = 0;

for (int rowMask : rowMasks) {

// Check if all 1s in the row are within the selected columns

// (rowMask & mask) extracts only the bits that are both in rowMask and mask

// If this equals rowMask, it means all 1s in the row are covered by selected columns

if ((rowMask & mask) == rowMask) {

count++;

}

}

// Update maximum count

if (count > maxRows) {

maxRows = count;

}

}

return maxRows;

}

}

**Explanation**

**Initialization**: The code first converts each row of the matrix into a bitmask where each bit represents whether the corresponding column has a 1 in that row.

**Bitmask Iteration**: It then iterates over all possible bitmasks (each representing a combination of columns). For each bitmask with exactly numSelect bits set, it checks how many rows are covered by verifying if the row's bitmask is a subset of the current column bitmask.

**Result Calculation**: The maximum number of covered rows across all valid bitmasks is tracked and returned as the result.

This approach efficiently leverages bitwise operations to check all possible column combinations, ensuring optimal performance given the problem constraints. The complexity is O(2^n \* m), which is feasible since n is at most 12.

**Bit Operation Explanation**

**1. Creating Row Bitmasks:**

rowMasks[i] |= (1 << j);

1 << j creates a number with only the j-th bit set

Example: j = 3 → 1 << 3 = 8 (binary: 1000)

|= (bitwise OR assignment) sets the j-th bit in rowMasks[i]

If rowMasks[i] was 5 (binary: 0101) and we set bit 2:

5 | (1 << 2) = 5 | 4 = 5 + 4 = 9 (binary: 1101)

**2. Checking Column Combinations:**

if (Integer.bitCount(mask) != numSelect)

Integer.bitCount(mask) counts how many bits are set to 1 in mask

This ensures we only consider combinations with exactly numSelect columns

**3. Checking Row Coverage:**

if ((rowMask & mask) == rowMask)

rowMask & mask performs bitwise AND:

Keeps only the bits that are set in both rowMask and mask

If the result equals rowMask, it means:

All 1s in the row (bits set in rowMask) are also selected in the column combination (bits set in mask)

In other words: the selected columns cover all 1s in this row

**Example:**

Row: [1, 0, 1, 1] → rowMask = 1101 (binary) = 13 (decimal)

Selected columns: [0, 2] (columns 0 and 2) → mask = 0101 (binary) = 5 (decimal)

rowMask & mask = 1101 & 0101 = 0101 ≠ 1101 → **Not covered**

Selected columns: [0, 2, 3] → mask = 1101 (binary) = 13 (decimal)

rowMask & mask = 1101 & 1101 = 1101 = rowMask → **Covered**

This bit manipulation approach efficiently checks all possible column combinations using fast bitwise operations, making it optimal for the problem constraints.

**Besides bitmask solution, do you know how to finish as Backtracking ?**

While the bitmask solution is efficient for this problem due to the constraints (columns up to 12), here's a backtracking approach that explores all combinations of selected columns:

**Backtracking Approach**

**Generate Combinations**: Use backtracking to generate all possible combinations of numSelect columns

**Check Coverage**: For each combination, check how many rows are fully covered (all 1s are within selected columns)

**Track Maximum**: Keep track of the maximum number of covered rows

**Solution Code**

class Solution {

private int maxRows = 0;

public int maximumRows(int[][] matrix, int numSelect) {

int n = matrix[0].length;

List<Integer> selected = new ArrayList<>();

backtrack(matrix, numSelect, 0, selected);

return maxRows;

}

private void backtrack(int[][] matrix, int numSelect, int start, List<Integer> selected) {

if (selected.size() == numSelect) {

// Check how many rows are covered by these selected columns

int covered = countCoveredRows(matrix, selected);

maxRows = Math.max(maxRows, covered);

return;

}

for (int i = start; i < matrix[0].length; i++) {

selected.add(i);

backtrack(matrix, numSelect, i + 1, selected);

selected.remove(selected.size() - 1);

}

}

private int countCoveredRows(int[][] matrix, List<Integer> selectedColumns) {

int count = 0;

for (int i = 0; i < matrix.length; i++) {

boolean covered = true;

for (int j = 0; j < matrix[0].length; j++) {

if (matrix[i][j] == 1 && !selectedColumns.contains(j)) {

covered = false;

break;

}

}

if (covered) {

count++;

}

}

return count;

}

}

**Optimized Backtracking with Pruning**

class Solution {

private int maxRows = 0;

public int maximumRows(int[][] matrix, int numSelect) {

int n = matrix[0].length;

boolean[] selected = new boolean[n];

backtrack(matrix, numSelect, 0, selected, 0);

return maxRows;

}

private void backtrack(int[][] matrix, int numSelect, int start, boolean[] selected, int countSelected) {

if (countSelected == numSelect) {

int covered = countCoveredRows(matrix, selected);

maxRows = Math.max(maxRows, covered);

return;

}

if (start >= selected.length) return;

// Prune if remaining columns + current selected < numSelect needed

if (selected.length - start < numSelect - countSelected) {

return;

}

// Include current column

selected[start] = true;

backtrack(matrix, numSelect, start + 1, selected, countSelected + 1);

selected[start] = false;

// Exclude current column

backtrack(matrix, numSelect, start + 1, selected, countSelected);

}

private int countCoveredRows(int[][] matrix, boolean[] selected) {

int count = 0;

for (int i = 0; i < matrix.length; i++) {

boolean covered = true;

for (int j = 0; j < matrix[0].length; j++) {

if (matrix[i][j] == 1 && !selected[j]) {

covered = false;

break;

}

}

if (covered) {

count++;

}

}

return count;

}

}

**Explanation**

**Backtracking Function**:

Generates all combinations of numSelect columns using recursion

For each valid combination, checks how many rows are covered

Uses pruning to avoid unnecessary computations

**Coverage Check**:

For each row, verifies if all 1s are in the selected columns

If a row has a 1 in a non-selected column, it's not covered

**Optimizations**:

Uses boolean array instead of list for better performance

Adds pruning to skip impossible combinations early

Tracks selected count to avoid recalculating

**Comparison with Bitmask Solution**

|  |  |  |
| --- | --- | --- |
| Aspect | Bitmask Solution | Backtracking Solution |
| Time Complexity | O(2^n \* m) | O(C(n, k) \* m \* n) |
| Space Complexity | O(m) | O(n) for recursion stack |
| Readability | More concise | More explicit |
| Performance | Faster (bit operations) | Slower (but acceptable for n=12) |

For n=12, both approaches are feasible since C(12, k) is manageable (maximum C(12,6)=924 combinations). The bitmask solution is generally preferred for its efficiency, but backtracking provides a more intuitive approach for those unfamiliar with bit manipulation.

**Refer to**

[L473.Matchsticks to Square (Ref.L2397)](note://WEB803bb8fa17c70b8a004a7117d62b41b1)