**Strivers-A2Z-DSA-Sheet-main\02.Binary Search\2D Arrays\5.Matrix_median.cpp**

```
/*QUESTION:

Given a row-wise sorted matrix of size R*C where R and C are always odd, find the median of the matrix.

Example:

Input:
R = 3, C = 3
M = [[1, 3, 5],
     [2, 6, 9],
     [3, 6, 9]]
Output: 5
Explanation: Sorting matrix elements gives us {1, 2, 3, 3, 5, 6, 6, 9, 9}. Hence, 5 is the median.

APPROACH:

To find the median of a row-wise sorted matrix, we can follow these steps:

1. Initialize two variables, `low` and `high`, to keep track of the minimum and maximum elements in the matrix.
2. Iterate through each row and update `low` with the minimum value of the first element in each row and `high` with the maximum value of the last element in each row.
3. Perform binary search between `low` and `high`.
4. For each iteration of binary search, count the number of elements in the matrix that are less than or equal to the mid value.
   - If the count is less than the desired median position, update `low` to mid + 1.
   - If the count is greater than or equal to the desired median position, update the answer with the mid value and update `high` to mid - 1.
5. Repeat steps 3-4 until `low` becomes greater than `high`.
6. Return the final answer as the median of the matrix.

CODE:*/

int median(vector<vector<int>>& matrix, int R, int C) {
    int low = INT_MAX;
    int high = INT_MIN;
    int opt_cnt = (R * C + 1) / 2;
    int ans = -1;

    for (int i = 0; i < R; i++) {
        low = min(low, matrix[i][0]);
        high = max(high, matrix[i][C - 1]);
    }

    while (low <= high) {
        int mid = low + (high - low) / 2;
        int cnt = 0;
        for (int i = 0; i < R; i++) {
            cnt += upper_bound(matrix[i].begin(), matrix[i].end(), mid) - matrix[i].begin();
        }
```

```
47          if (cnt < opt_cnt)
48              low = mid + 1;
49          else {
50              ans = mid;
51              high = mid - 1;
52          }
53      }
54
55      return ans;
56  }
57
58  /*
59  TIME COMPLEXITY: O(R * log(C) * log(range)), where R is the number of rows, C is the number
    of columns, and range is the difference between the minimum and maximum elements in the
    matrix.
60                  The algorithm performs binary search on each row, which takes O(log(C)) time,
    and the outer binary search iterates log(range) times.
61  SPACE COMPLEXITY: O(1) as the algorithm only uses a constant amount of additional space to
    store variables.
62  */
```