# 题目

编写一个高效的算法来搜索m x n矩阵matrix中的一个目标值target。该矩阵具有以下特性：

每行的元素从左到右升序排列。

每列的元素从上到下升序排列。

**示例:**

现有矩阵matrix如下：

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

给定target = 5，返回true。

给定target = 20，返回false。

**类似题目：**剑指offer 04。

# 分析

## 方法一：模拟/分治算法

**思路：**

从右上角开始（左下也可以），往左下角走，右上角那个元素往左元素递减，往下元素递增，一增一减大大降低了搜索的时间复杂度。

targe大了，往下搜

target小了，往左搜

**分析：**

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

if(matrix.size()==0)

return false;

int m=matrix.size(); //总行号

int n=matrix.at(0).size(); //总列号

int row = 0;

int col = n-1;

//可以使用右上角或左下角作为参考值

while(row<m && col>=0)

{

if(target < matrix[row][col])

{

col--;

}

else if(target >matrix[row][col])

{

row++;

}

else

{

return true;

}

}

return false;

}

};

**复杂度分析：**

时间复杂度：O(n+m)，因为最差情况是从右上搜到左下角。

空间复杂度：O(1)，因为这种方法只处理几个指针，所以它的内存占用是恒定的。

## 方法二：二分查找

思路：

1、利用行有序性：矩阵每行元素从左到右升序排列，因此每行可独立进行二分查找。

2、提前过滤无效行：

若当前行的第一个元素大于目标值，由于列也升序（下方行元素更大），后续所有行均不可能包含目标，直接终止搜索。

若当前行的最后一个元素小于目标值，当前行不可能包含目标，跳过该行。

3、逐行二分查找：对可能包含目标的行，使用二分查找快速定位目标值。

简言之，通过 “过滤无效行 + 每行二分” 的组合策略，在利用二分查找高效性的同时，避免对不可能包含目标的行进行无效搜索。

代码：

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

if (matrix.empty() || matrix[0].empty()) {

return false;

}

int m = matrix.size();

int n = matrix[0].size();

// 遍历每一行，对每行进行二分查找

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

// 先判断当前行是否可能包含目标值，优化查找

if (matrix[i][0] > target) {

// 当前行第一个元素就大于目标值，后续行更大，无需再查

break;

}

if (matrix[i][n-1] < target) {

// 当前行最后一个元素小于目标值，跳过当前行

continue;

}

// 对当前行进行二分查找

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (matrix[i][mid] == target) {

return true;

} else if (matrix[i][mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

}

return false;

}

};

**复杂度分析：**

**时间复杂度：**O(m × log n)

- 其中 m 为矩阵行数，n 为列数。

- 最坏情况下需遍历 m 行，每行二分查找的时间复杂度为 O (log n)，因此总时间为两者乘积。

- 实际运行中，由于 “提前过滤” 机制，往往无需遍历所有行，效率会更高。

**空间复杂度：**O(1)

仅使用常数个额外变量（如循环索引、二分查找的左右边界等），不随输入规模变化，属于原地算法。