## 一、二维数组查找

#### 1.理解问题

在一个m行n列的二维数组中，每一行都按照从左到右递增的顺序排列，每一列都按照从上到下递增的顺序排列。请设计一个查找算法在该数组中查找每个元素，并分析算法性能。

输入：依次输入待查找的元素x，数组（n\*m）的规模n、m，数组内的所有元素

例如：5 5 5 0 0 0 0 0 0 1 1 1 1 2 3 3 4 4 3 3 3 3 3 4 4 5 6 6

输出：true

#### **2.**算法设计

方法(1) 每次将右上角的元素rightup作为比较对象

如果x=rightup直接返回true

如果x<rightup则删去右边一列（m--）

如果x>rightup则删去上面一行。

方法(2) 分治法：将矩阵从中心点划分为四部分，比较中心点与x

如果x=中心点直接返回true

如果x<中心点，则舍去右下角的矩阵元素

如果x>中心点，则舍去左上角的矩阵元素

方法(3) 贪心：将矩阵逆时针旋转45度，得到类似于二叉搜索树的结构，比根结点小的元素一定在左边，比根结点大的元素一定在右边。从根结点出发，

如果x=当前元素就返回true

如果x>当前元素就向右走

如果x<当前元素就向左走

#### 算法代码与分析

方法(1) 代码

#include <stdio.h>

int TwoDimensionSearch(int arr[][100],int n,int m,int x){

    if (n==0||m==0)

    {

        return -1;  // 数组为空

    }

    int row=0,column=m-1;

    //这种处理也包括单行或单列的基情况

    while(row<n&&column>=0){

        if (arr[row][column]==x)

        {

            return 1;  // 找到元素

        }else if (arr[row][column]<x)

        {

            row++;

        }else if (arr[row][column]>x)

        {

            column--;

        }

    }

    return 0;

}

int main(){

    int x;

    printf("Please input the number you want to find:\n");

    scanf("%d",&x);

    printf("Please input the size of matrix(n\*m) using n m:\n");

    int n,m;

    scanf("%d %d",&n,&m);

    printf("Please input the number of rows and columns:\n");

    int arr[100][100];

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < m; j++)

        {

            scanf("%d",&arr[i][j]);

        }

    }

    int res=TwoDimensionSearch(arr,n,m,x);

    if(res==1)printf("Found!\n");

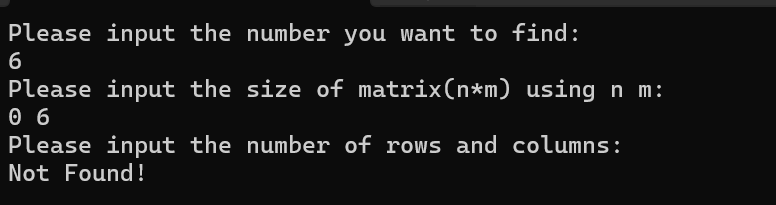
    else printf("Not Found!\n");

}

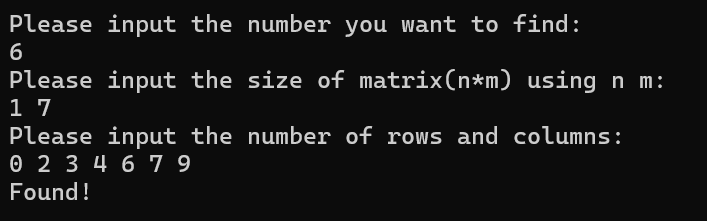
测试结果：

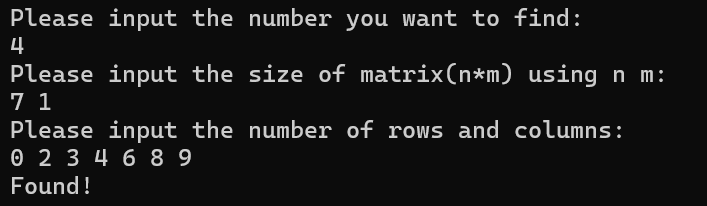
边界情况：

1. n或m有一个为0：

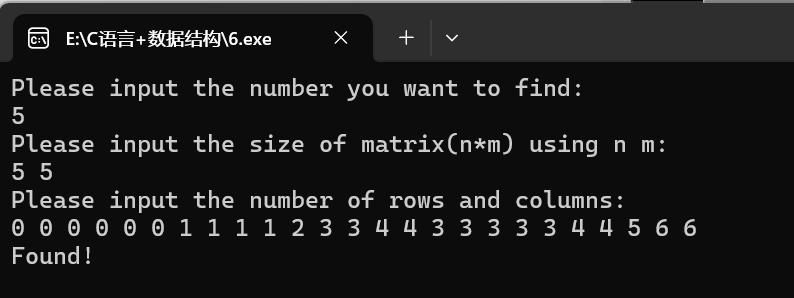


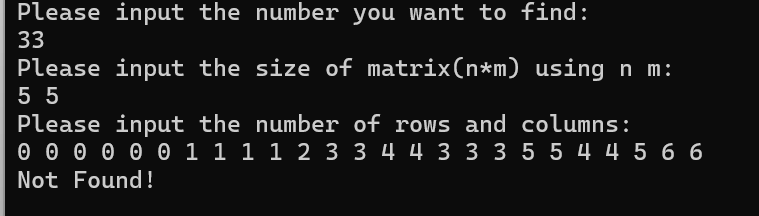
1. 只有一行或者只有一列





1. 其他情况





#### 性能分析

方法(1) 每次将右上角的元素rightup作为比较对象

如果x=rightup直接返回true

如果x<rightup则删去右边一列（m--）

如果x>rightup则删去上面一行。

该方法的主要时间复杂度在 while(row<n&&column>=0) 循环语句上面，时间复杂度为O（n+m）

方法(2) 分治法：将矩阵从中心点划分为四部分，比较中心点与x

如果x=中心点直接返回true

如果x<中心点，则舍去右下角的矩阵元素

如果x>中心点，则舍去左上角的矩阵元素

由master公式，T(N)=3\*T(n\*m/4)+1

因为d=0,a=3,b=4

0<log4 3

所以时间复杂度为O（（n\*m）^0.79）

方法(3) 贪心：将矩阵逆时针旋转45度，得到类似于二叉搜索树的结构，比根结点小的元素一定在左边，比根结点大的元素一定在右边。从根结点出发，

如果x=当前元素就返回true

如果x>当前元素就向右走

如果x<当前元素就向左走

该方法时间复杂度为O（n+m）

## 二、最大波谷

#### 1.理解问题

假设X[ 1 : n ] 是实数数组，l是一个固定的正整数（l<=n）。X的 l-子数组 是数组X的l连续元素的任意序列，l-子数组的波谷是该1-子数组中的最小值。给出算法（至少包括一个分治算法），对于输入一个数组X[ 1 : n ]和一个正整数l<=n，返回具有最大波谷的l子数组的起始位置，并分析算法的时间复杂度。

输入：依次输入数组大小n，固定的正整数l（l<=n），数组的每个元素

例如：10 3 1 8 2 3 4 10 11 6 9 1

输出：6

#### **2.**算法设计

**方法(1) 分治法：**

Divide:将数组从中间分成左右两部分，划分成两个新的子问题

Conquer:分别找左右两个子数组的maxValley，找跨越左右边界的maxValley

Merge:比较左问题、右问题、跨越边界的maxValley取最大值，返回该子数组的起始位置

**方法(2) 滑动窗口+单调队列优化：**

分治法最花费时间的地方是求区间最小值，是线性时间复杂度，但是因为是对有很多个重复元素的多个区间求最小值，能利用之前得到的信息优化这个过程。

优化求长为l的区间的最小值步骤：

按下列要求维护一个队列，对任意的新元素：

1. 检查队首元素编号，如果队首元素离开窗口，则弹出队首元素；
2. 检查队尾元素，如果新元素<=队尾元素，则弹出队尾元素，直到新元素>队尾元素
3. 在队尾插入新元素
4. 如果窗口长度满足要求，则与当前答案比较并记录
5. 最后得到波谷最大值并返回

**方法(3) ST表**

ST表：sparse table,又叫稀疏表，是用来处理区间最值查询的离线算法，用到了倍增的思想。某个区间查询问题能否适用ST表，关键在于进行的操作是否允许区间重叠。

例如max(a,b,c) = max{max(a,b),max(b,c)}就可以用ST表维护，区间和问题不能维护。

1. 构建ST表
2. 查询区间最值

**方法(4) 线段树**

线段树segment tree是一种树形结构，将每个长度不为1的区间划分为左右两个区间递归求解，把整个线段划分为一个树形结构，通过合并左右两区间信息来求得该区间的信息。这种数据结构可以方便的进行大部分区间操作。

本题中，我们设置每个树上的节点包含三个信息：左端点、右端点、所在区间最小值

Tree[p].l=Tree[lc(p)].l;

Tree[p].r=Tree[rc(p)].r;

Tree[p].minVal=min(Tree[lc(p)].minVal,Tree[rc(p)].minVal);

#### 3.算法代码与分析

#include<bits/stdc++.h>

using namespace std;

struct Element{

    int value;

    int index;

    Element(int v, int i) : value(v), index(i) {}

    bool operator<(const Element& other) const {

        // 如果值不同，则按值排序

        if (value != other.value) {

            return value > other.value; // 小于号表示最小堆

        }

        // 如果值相同，则按序号排序

        return index > other.index; // 小于号表示最小堆

    }

};

void printminHeap(priority\_queue<Element> minHeap){

    // 克隆一份优先队列

    priority\_queue<Element> clonedHeap = minHeap;

    // 打印队列所有元素

    cout << "minHeap priority queue:";

    while (!clonedHeap.empty()) {

        Element top = clonedHeap.top();

        cout <<  top.value << " ";

        clonedHeap.pop();

    }

    cout<<endl;

}

int findMaxWave(const vector<int>& nums, int l){

    int n = nums.size();

    int max\_wave = -1; // 最大波谷

    // 使用自定义类型创建最小堆

    priority\_queue<Element> minHeap;

    for(int i=0;i<l;i++)

        minHeap.push(Element(nums[i],i));

    printminHeap(minHeap);//显示一下

    for(int i=l;i<n;i++){

        // 移除队列中不在当前l-序列范围内的元素

        while(minHeap.top().index <= i -l  && !minHeap.empty())

            minHeap.pop();

        minHeap.push(Element(nums[i],i));

        printminHeap(minHeap);//显示一下

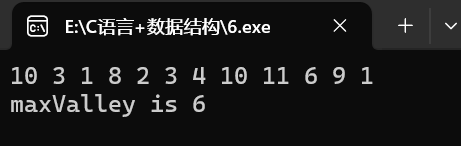
        if(minHeap.top().value > max\_wave)

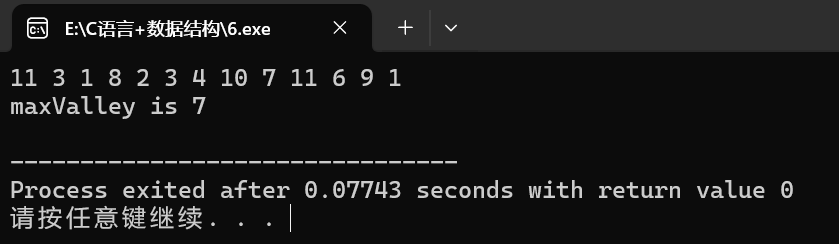
            max\_wave = minHeap.top().value;

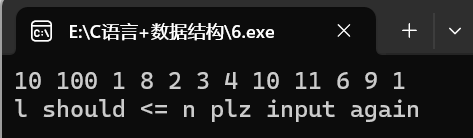
    }

    return max\_wave;

}







#### 4.性能分析

**方法(1) 分治法：**

Divide:将数组从中间分成左右两部分，划分成两个新的子问题

Conquer:分别找左右两个子数组的maxValley，找跨越左右边界的maxValley

Merge:比较左问题、右问题、跨越边界的maxValley取最大值，返回该子数组的起始位置

分治法和暴力枚举没有什么区别，还是检查了所有的l-子数组，时间复杂度为O（n^2）

**方法(2) 滑动窗口+单调队列优化：**

分治法最花费时间的地方是求区间最小值，是线性时间复杂度，但是因为是对有很多个重复元素的多个区间求最小值，能利用之前得到的信息优化这个过程。

该算法的每一步都是常数时间的维护，只需要线性的遍历一遍数组就可以求出答案，时间复杂度为O（n）

**方法(3) ST表**

时间复杂度为O（nlogn）

**方法(4) 线段树**

对于建树操作来说：叶子结点是单个序列元素，树是二叉树。

建树的时间复杂度不会超过O(N+N/2+N/4+....)=O（n）

对于查询操作：每次查询的深度不会大于树高，每一层不会访问超过四个节点

一次查询的时间复杂度不会超过O（logn）

总的时间复杂度为O(nlogn)