# 数组

## 定义

元素类型相同，大小相等。

**数组元素访问时间为什么是常数时间？**

访问数组元素的时候，需要计算出该元素距离数组基址的偏移量，而为了计算偏移量，必须先做乘法，然后把乘法所得的结果，加到基址上面，已确定出该元素的内存地址。

整个过程涉及一次乘法与一次加法。由于两项操作都只需要花费常数级别的时间，因此访问数组元素，只需要常数级别时间。

注：数组的具体实现可以参考C++ Array和JDK代码。

## 分类

在数组定义中，可以将元素数量指定为一个常量表达式，或者在特定情况下，指定为涉及变量的表达式。采用这两种方式定义的数组分别被称为固定长度数组（fixed-length）和长度可变（variable-length）数组。

### 固定长度数组

固定长度数组可以具有任意存储类别：可以将它们定义在所有函数的外面或语句块的里面，并且可以使用或不使用存储类修饰符 static。唯一的限制是数组不能作为函数参数。一个传入函数的数组参数需要被转换为指向数组第一个元素的指针。

下面 4 种数组定义方式都是合法的：

int a[10]; // a有外部链接

static int b[10]; // b有静态存储周期和文件作用域

void func()

{

static int c[10]; // c有静态存储周期和块作用域

int d[10]; // d有动态存储周期

/\* ... \*/

}

### 长度可变数组/动态数组

动态数组能够改变长度的数据结构。

有一种简单的实现方式，是先分配固定的数组空间，等到数组填满时，再多分配一倍的空间。同理，如果数组现存的元素还不到总量的1/2，那就把数组大小减半。

如果一个数组具有动态存储周期（也就是说，如果在语句块内定义数组，并且没有 static 修饰符），那么 C99 也允许把非常量表达式作为元素数量来定义该数组。这样的数组被称为长度可变数组（variable-length array）/动态数组。

而且，长度可变数组的名称必须是普通的标识符。长度可变数组不能作为结构或联合的成员。在下面的示例中，只有 vla 数组的定义是合法的：

void func( int n )

{

int vla[2\*n]; // 合法：存储周期为动态的

static int e[n]; // 非法：长度可变数组不可有静态存储周期

struct S { int f[n]; }; // 非法：f不是一个普通标识符

/\* ... \*/

}

与其他动态变量一样，每次程序流进入包含长度可变数组定义的语句块时，都会重新创建这个长度可变数组。因此，在每次实例化时，数组都可以有不同的长度。然而，一旦被创建，即便是长度可变数组，在它的当前存储周期内也不能改变数组长度。

动态对象被存储在栈中，当程序流离开对象所在的语句块时，动态对象的空间就会被释放。因此，只有对小的、临时的数组，定义长度可变数组才比较合理。如想动态地创建大型数组，通常应该使用标准函数 malloc和 calloc来显式地分配内存空间。

这种数组的存储周期会持续到程序结束，也可以调用函数 free（）来主动地释放被占用的内存空间。

例如：

#include <stdio.h>

#include <stdlib.h>

int main()

{

int n1,i;

int \*array;

printf("请输入所要创建的一维动态数组的长度：");

scanf("%d",&n1);

array=(int\*)calloc(n1,sizeof(int));

for(i=0;i<n1;i++)

{

printf("%d\t",array[i]);

}

printf("\n");

for(i=0;i<n1;i++)

{

array[i]=i+1;

printf("%d\t",array[i]);

}

free(array);//释放第一维指针

return 0;

## 操作

### 定义

struct Array

{

int\* pBase; //存储的是数组第一个元素的地址

int len; //数组所能容纳的最大元素的个数

int cnt; //当前数组有效元素的个数

int increment; //自动增长因子

}

### 初始化

void init\_array(struct Array \*pArr, int length)

{

pArr->pBase = (int \*)malloc(sizeof(int) \* length);

if (NULL == pArr->pBase)

{

exit(-1);

}

else

{

pArr->length = length;

pArr->count = 0;

}

return;

}

### 判空

bool is\_empty(struct Array\* pArr)

{

if (0 == pArr->count)

return true;

else

return false;

}

### 追加

需要判断数组是否满。

### 插入

判断长度是否达到最大值。

### 删除

## 特点

### 优点

1. 简单易用；
2. 可以迅速访问其元素（只需耗费常数级别的时间）

### 缺点

1. 大小固定；
2. 必须一次分配一整块内存；
3. 按照位置来插入元素是一种较为复杂的操作。

# 算法

## 前缀和

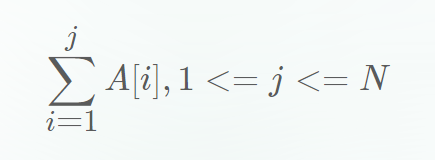
前缀和可以简单理解为「数列的前n项的和」，是一种重要的预处理方式，能大大降低查询的时间复杂度。

C++ 标准库中实现了前缀和函数 std::partial\_sum，定义于头文件<numeric>中。

## 树状数组

树状数组或二叉索引树（Binary Indexed Tree），又以其发明者命名为Fenwick树。其初衷是解决数据压缩里的累积频率的计算问题，现多用于高效计算数列的前缀和、区间和。

它可以以O(logn)的时间得到任意前缀和：



并同时支持在O(logn)时间内支持动态单点值的修改。

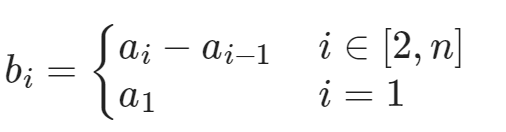
空间复杂度O(n)。

## 后缀数组

## 差分

差分是一种和前缀和相对的策略，可以当做是求和的逆运算。

这种策略的定义是令：



# 应用

数组的应用主要包括：

1. 数组排序
2. 数组查找
3. 数组删除
4. 数组合并/组合
5. 二维数组

## 查找

数组相关的查找，包括：两数和差运算，出现一次/重复/超过一半/缺失的数字（一般考虑哈希表、排序、位运算、set[仅出现1次的场景]），最值（最大/最小/局部最值/K值）。

### 两数之和

注：LeetCode 1

### 搜索插入位置

注：LeetCode 35

### 只出现一次的数字/数组中的查找出现一次的数

题目要求：一个整数数组中有一个元素出现了一次，其他元素都出现了两次，使用最小的时间复杂度找出出现一次的数。

注：LeetCode 136

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

一个数组中只有一个数字出现了一次，其他都是出现了两次

找到这个出现一次的数字

\*/

//将整个数组中的整数进行异或 最后得到的结果就是所要找的数

//两个相等的数异或操作得到的是0

//0和另外一个不等的数异或操作得到的是另外一个数

//将数组中所有整数异或操作，得到的就是想要找的数

int SingleNumber(vector<int>& vec)

{

if(vec.size()<=0)

return 0;

int i;

int value = vec[0];

for(i=1;i<vec.size();i++)

value ^= vec[i];

return value;

}

/\*

如果一个数组中有两个数出现了一次，其他都是出现了两次，这两个数分别是什么

\*/

void SingleNumberII(vector<int>& a,int& pN1,int& pN2)

//入参：待输入参数a，2个输出参数pN1,pN2

{

int i,j,temp;

//计算这两个数的异或结果（计算两个出现1次的数异或操作）

temp =0;

for(i=0;i<a.size();i++)

temp ^= a[i];

//找到异或结果中第一个为1的位

//（依据这个位将大数组分割成2个，这样就与前面的情况一致了）

for(j=0;j<sizeof(int)\*8;j++)

if(((temp >> j)&1) ==1)

break;

//第j位为1，说明这两个数字在第j位上不同，根据这个来进行分组

pN1 =0;

pN2 =0;

for(i=0;i<a.size();i++)

if(((a[i] >> j )&1) ==0)

pN1 ^= a[i];

else

pN2 ^= a[i];

}

/\*

一个数组中只有一个数出现了一次，其他都是出现了三次，找出这个数

\*/

//同样根据位来判断

int SingleNumberIII(vector<int>& vec)

{

int i,j;

int low=0,high=vec.size()-1;

int bit =1;

int flag =0;

//通过某一位的值将数组分为两部分，其中一部分包含了待找的数，另一

部分不包含

while(low <= high)

{

if(low == high)

return vec[low];

i = low-1;

for(j = low;j<= high;j++)

{

if((vec[j]&bit) == 0)

{

i++;

swap(vec[i],vec[j]);

}

}

if(i >= low)

{

if((i-low+1)%3 == 0)

low = i+1;

else

high =i;

}

bit = bit<<1;

}

return 0;

}

int main()

{

int array[]={12,23,12,45,56,45,23,78,78,78,12,23,45};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

//上述是用C语言定义数组，然后拷贝到C++的vector中

int fir,sec;

cout<<SingleNumberIII(vec);

// cout<<fir<<endl<<sec<<endl;

return 0;

}

拓展：

如果有两个数均出现了一次，其他都出现了两次，如何查找这两个数？

如果一个数组中有一个数出现了一次，其他数都出现了三次，如何找到出现一次的数？

### 数组中重复的数字

题目要求：在一个长度为n的数组里的所有数字都在0到n-1的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字3或者3。

注：剑指offer P261

### 多数元素

注：LeetCode 169

### 数组中出现次数超过一半的数字

题目要求：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。

注：剑指offer P163

### 统计位数为偶数的数字

**题目：**给你一个整数数组 nums，请你返回其中位数为 偶数 的数字的个数。

示例 1：

输入：nums = [12,345,2,6,7896]

输出：2

解释：

12 是 2 位数字（位数为偶数）

345 是 3 位数字（位数为奇数）

2 是 1 位数字（位数为奇数）

6 是 1 位数字 位数为奇数）

7896 是 4 位数字（位数为偶数）

因此只有 12 和 7896 是位数为偶数的数字

示例 2：

输入：nums = [555,901,482,1771]

输出：1

解释：

只有 1771 是位数为偶数的数字。

**分析：**

**方法一：枚举 + 字符串**

我们枚举数组 nums 中的整数，并依次判断每个整数 x 是否包含偶数个数字。

一种简单的方法是使用语言内置的整数转字符串函数，将 x 转换为字符串后，判断其长度是否为偶数即可。

**代码：**

class Solution {

public:

int findNumbers(vector<int>& nums) {

int ans = 0;

for (int num: nums) {

if (to\_string(num).size() % 2 == 0) {

++ans;

}

}

return ans;

}

};

**复杂度分析：**

时间复杂度：O(N)，其中N是数组nums的长度。这里假设将整数转换为字符串的时间复杂度为O(1)。

空间复杂度：O(1)。

### 丢失的数字

注：LeetCode 268

### 第一个缺失的整数

题目：给定一个数组A[0…N-1]，找到从1开始，第一个不再数组中的正整数。如3,5,1,2，-3,7,14,8，输出4。

分析：

思路：将找到的元素放到正确的位置上，如果最终发现某个元素一直没有找到，则该元素即为所求。

循环不变式：如果某命题初始为真，且每次更改后仍然保持该命题为真，则若干次更改后命题仍然为真。

为表述方便，下面的算法描述从1开始数。

假定前i-1个数已经找到，并且一次存放在A[1,2,…,i-1]中，继续考察A[i]：

1. 若A[i]<i且A[i]≥1，则A[1,2,…,i-1]中已经出现过，可以直接丢弃

若A[i]为负，则更应该丢弃它

1. 若A[i]>i且A[i]≤N，则A[i]应该位于后面的位置上，则将A[A[i]]和A[i]交换

若A[i]≥N，由于缺失数据一定小于N，则A[i]丢弃

1. 若A[i]=i，则A[i]位于正确的位置上，则i加1，循环不变式扩大，继续比较后面的元素。

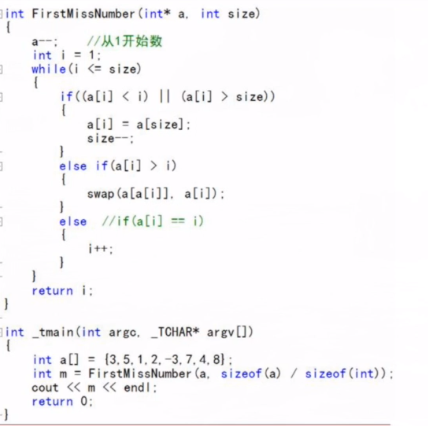
综上，算法描述为：

1. 若A[i]<i或者A[i]>N，则丢弃A[i]
2. 若A[i]>i，则将A[A[i]]和A[i]交换
3. 若A[i]=i，i++，继续比较后面的元素

思考：如何快速丢弃A[i]？

将A[N]赋值给A[i]，然后N减1。

**代码：**



### 从1到n整数中1出现的次数

题目要求：输入一个整数n，求从1到n这n个整数的十进制表示中1出现的次数。例如输入12，从1到12这些整数中包含1的数字有1，10，11和12，1一共出现了5次。

注：剑指offer P174

### 寻找循环数组最小值

题目：假定一个排序数组以某个未知元素为支点做了旋转，如：原数组0 1 2 4 5 6 7旋转后得到4 5 6 7 0 1 2。请找出旋转后数组的最小值。假定数组中没有重复的数字。

分析：

代码：

### 第K大问题

题目要求：在一个整数数组中，查找该数组中第K大的元素，时间复杂度能达到O(n)吗？

代码：

#include <iostream>

#include <stdlib.h>

#include <algorithm>

#include <vector>

using namespace std;

//查找一个数组中的第K大的元素

int Find\_K(int a[],int low,int high,int k)

{

if(k<=0)

return -1;

if(k>high-low+1)

return -1;

//随机选择一个支点

int pivot = low+rand()%(high-low+1);

swap(a[low],a[pivot]);

int m = low;

int count = 1;

//每遍历一次，把较大的放到数组的左边

for(int i=low+1;i<=high;i++)

{

if(a[i] > a[low])

{

swap(a[++m],a[i]);

count++; //比关键点大的数的个数为count+1

}

}

swap(a[m],a[low]);// 将关键点放在左右两部分的分界点

if(count > k)

return Find\_K(a,low,m-1,k);

else if(count < k)

return Find\_K(a,m+1,high,k-count);

else

return m;

}

//查找最小的K个数

void FindMinTopK(vector<int>& vec,int k)

{

vector<int> heap(vec.begin(),vec.begin()+k);

make\_heap(heap.begin(),heap.end());

int i;

//开始处理剩余的数据

for(i =k;i<vec.size();i++)

{

if(vec[i]<heap[0])

{

pop\_heap(heap.begin(),heap.end());

heap.pop\_back();

heap.push\_back(vec[i]);

push\_heap(heap.begin(),heap.end());

}

}

for(i=0;i<heap.size();i++)

cout<<heap[i]<<endl;

}

int main()

{

int a[]={5,15,4,8,2,3,9,10};

int r = Find\_K(a,0,sizeof(a)/sizeof(int)-1,3);

// cout<<(r==-1? r : a[r])<<endl;

int array[]={5,3,6,7,4,2,1,9,8,10};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

FindMinTopK(vec,4);

return 0;

}

拓展：

top K问题！

### 最小的k个数

题目要求：输入n个整数，找出其中最小的k个数。例如输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

注：剑指offer P167

### 第三大的数

**题目：**给定一个非空数组，返回此数组中第三大的数。如果不存在，则返回数组中最大的数。要求算法时间复杂度必须是O(n)。

示例 1:

输入: [3, 2, 1]

输出: 1

解释: 第三大的数是 1.

示例 2:

输入: [1, 2]

输出: 2

解释: 第三大的数不存在, 所以返回最大的数 2 .

示例 3:

输入: [2, 2, 3, 1]

输出: 1

解释: 注意，要求返回第三大的数，是指第三大且唯一出现的数。

存在两个值为2的数，它们都排第二。

**分析：**

**方法一：set**

有点类似于第K大的数，只需要维护一个k大小的数组即可。

利用set中元素的有序性和唯一性，将元素放入set中，若set的size不小于3输出倒数第三个元素；若set的size小于3，输出最后一个元素。

**代码：**

**方法一：**

class Solution {

public:

int thirdMax(vector<int>& nums)

{

set<int> s(nums.begin(), nums.end());

auto it = s.end();

it--;

if(s.size() >= 3){

it--;

it--;

}

return \*it;

}

};

### 局部最大值

**题目：**给定一个无重复元素的数组A[0…N-1]，求找到一个该数组的局部最大值。规定：在数组边界外的值无穷小。即A[0]>A[-1]，A[N-1]>A[N]。

**分析：**

显然，遍历一遍可以找到全局最大值，而全局最大值显然是局部最大值。

可否有更快的方法？

若子数组Array[from,…,to]满足：

Array[from]>Array[from-1]

Array[to]>Array[to+1]

称该子数组为“高原数组”。

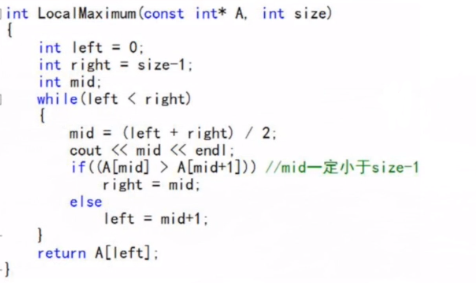
若高原数组长度为1，则该高原数组的元素为局部最大值。

使用索引left、right分别指向数组首尾，根据定义，该数组为高原数组。

1. 求中点mid=(left+right)/2；
2. A[mid]>A[mid+1]，子数组A[left…mid]为高原数组，丢弃后半段right=mid
3. A[mid+1]>A[mid]，子数组A[mid…right]高原数组，丢弃前半段：left=mid+1
4. 递归直至left==right

时间复杂度为O(logN)。

**代码：**



## 排序

### 数组中的逆序对

题目要求：一个数组中，如果一个元素的值比它右边的值大，那么这一对元素构成逆序对，求一个乱序数组中逆序对的数目。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

unsigned int CountInversions(vector<unsigned int>& a);

unsigned int MergeSortCount(vector<unsigned int>& a,int left,

int right,vector<unsigned int>& temp);

unsigned int MergeCount(vector<unsigned int>& a,int left,int mid,

int right,vector<unsigned int>& temp);

int main()

{

int array[]={4,2,10,3,5,7};

vector<unsigned int> a(array,array+sizeof(array)/sizeof(int));

unsigned int result = CountInversions(a);

cout <<result<<endl;

return 0;

}

unsigned int MergeCount(vector<unsigned int>& a,int left,int mid,

int right,vector<unsigned int>& temp)

{

int i = left;

int j= mid+1;

int k = left;

int count = 0;

//开始查找逆序对的个数

while( i<= mid && j<=right)

{

if(a[i]<= a[j])

temp[k++] = a[i++];

else

{

temp[k++]=a[j++];

count+= mid-i+1;

}

}

while(i<=mid) temp[k++]= a[i++];

while(j<= right) temp[k++] = a[j++];

for(i=left;i<=right;i++)

a[i]= temp[i];

return count;

}

unsigned int MergeSortCount(vector<unsigned int>& a,int left,

int right,vector<unsigned int>& temp)

{

if(left >= right) return 0;

int mid = (left+right)/2;

unsigned int Inverleft = MergeSortCount(a,left,mid,temp);

unsigned int Inverright = MergeSortCount(a,mid+1,right,temp);

unsigned int InverSum = MergeCount(a,left,mid,right,temp);

return Inverleft+Inverright+InverSum;

}

unsigned int CountInversions(vector<unsigned int>& a)

{

int n= a.size();

vector<unsigned int> temp(a.begin(),a.end());

unsigned int ans = MergeSortCount(a,0,n-1,temp);

return ans;

}

### 调整数组顺序使奇数位于偶数前面

题目要求：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有偶数位于数组的后半部分。

注：剑指offer P102，LeetCode 905，LeetCode922

## 合并

### 两个数组的交集

注：LeetCode 349

### 两个数组的交集II

注：LeetCode 350

### 合并两个有序数组/合并排序的数组

**题目：**给定两个排序后的数组A和B，其中A的末端有足够的缓冲空间容纳B。编写一个方法，将B合并入A并排序。

初始化A和B的元素数量分别为m和n。

注：LeetCode 88

示例:

输入:

A = [1,2,3,0,0,0], m = 3

B = [2,5,6], n = 3

输出: [1,2,2,3,5,6]

说明:

A.length == n + m

**分析：**

**方法1: 直接合并后排序**

最直观的方法是先将数组 B 放进数组 A 的尾部，然后直接对整个数组进行排序。

**方法 2: 双指针**

没有利用数组 A 与 B 已经被排序的性质。为了利用这一性质，我们可以使用双指针方法。这一方法将两个数组看作队列，每次从两个数组头部取出比较小的数字放到结果中。

**代码：**

**方法一：**

class Solution {

public:

void merge(vector<int>& A, int m, vector<int>& B, int n) {

for (int i = 0; i != n; ++i)

A[m + i] = B[i];

sort(A.begin(), A.end());

}

};

**方法二：**

class Solution {

public:

void merge(vector<int>& A, int m, vector<int>& B, int n) {

int pa = 0, pb = 0;

int sorted[m + n];

int cur;

while (pa < m || pb < n) {

if (pa == m)

cur = B[pb++]; //第一个执行到末尾则取另一个数组的值

else if (pb == n)

cur = A[pa++];

else if (A[pa] < B[pb])

cur = A[pa++];

else

cur = B[pb++];

sorted[pa + pb - 1] = cur;

}

for (int i = 0; i != m + n; ++i)

A[i] = sorted[i];

}

};

另一种写法：

class Solution {

public:

void merge(vector<int>& A, int m, vector<int>& B, int n) {

int pa = 0;

int pb = 0;

int cur = 0;

vector<int> tmpVec(m+n);

while(pa<m || pb <n)

{

if(pa == m)

{

cur = B[pb++];

}

else if(pb == n)

{

cur = A[pa++];

}

else if(A[pa] < B[pb])

{

cur = A[pa++];

}

else

{

cur = B[pb++];

}

tmpVec[pa+pb-1] = cur;

}

for(size\_t i=0;i<tmpVec.size();i++)

{

A[i] = tmpVec[i];

}

}

};

## 删除

### 删除有序数组中的重复项

注：Leetcode 26

### 删除有序数组中的重复项 II

注：Leetcode 80

### 移除元素

题目：给你一个数组nums 和一个值val，你需要原地移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用O(1)额外空间并 原地 修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。 注：Leetcode 27

**分析：**

**方法一：快慢指针**

**代码：**

class Solution {

public:

int removeElement(vector<int>& nums, int val) {

int i = 0;

for(int j=0;j<nums.size();j++)

{

if(nums.at(j) != val)

{

nums[i] = nums[j];

i++;

}

}

return i;

}

};

### 删除某些元素后的数组均值

注：Leetcode 1619

### 删除一个元素使数组严格递增

注：Leetcode 1909

### 删除最短的子数组使剩余数组有序

注：Leetcode 1574

### 通过删除字母匹配到字典里最长单词

注：Leetcode 524

## 排列组合

### 连续子数组的最大和

题目要求：输入一个整型数组，数组里有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。

注：剑指offer P171

### 把数组排成最小的数

题目要求：输入一个正整数数组，把数组中所有数字拼接起来排成一个数，打印能拼接处的所有数字中最小的一个。例如输入数组{3,32,321}，则打印出这3个数字能排成的最小数字321323。

注：剑指offer P177

### 构造乘积数组

题目要求：给定一个数组A[0,1,…,n-1]，请构造一个数组B[0,1,…,n-1]，其中B中的元素B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]，不能使用除法。

注：剑指offer P263

### 最大间隔

### 字符串全排列

## 运算

### 加1

注：LeetCode 66

### 有序数组的平方

## 矩阵/二维数组

二维数组/矩阵的定义如下：

vector<vector<int>> matrix;

### 遍历

#### 螺旋矩阵

给你一个m行n列的矩阵matrix，请按照顺时针螺旋顺序，返回矩阵中的所有元素。

注：Leetcode 54

分析：

#### 螺旋矩阵II

给你一个正整数n，生成一个包含1到n2所有元素，且元素按顺时针顺序螺旋排列的n x n正方形矩阵matrix。

注：Leetcode 59

#### 螺旋矩阵III

注：Leetcode 885

#### 螺旋矩阵IV

注：Leetcode 2326

#### 对角线遍历

注：Leetcode 498

#### 矩阵对角线元素的和

注：Leetcode 1572

#### 矩阵中的局部最大值

注：Leetcode 2373

#### 矩阵中的和

注：Leetcode 2679

#### 最小路径和

给定一个包含非负整数的m×n网格grid，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

注：Leetcode 64

分析：

动态规划。

### 查找

#### 二维数组中的查找

题目要求：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

注：剑指offer P38

#### 找出缺失和重复的数字

给你一个下标从0开始的二维整数矩阵grid，大小为n \* n ，其中的值在 [1, n2] 范围内。除了a出现两次，b 缺失之外，每个整数都恰好出现一次。

任务是找出重复的数字a和缺失的数字b。

返回一个下标从0开始、长度为2的整数数组ans，其中ans[0]等于a，ans[1]等于b。

注：Leetcode 2965

分析：

使用数组遍历或者哈希表统计。

#### 矩阵中的幸运数

给你一个 m \* n 的矩阵，矩阵中的数字 各不相同 。请你按 任意 顺序返回矩阵中的所有幸运数。

幸运数是指矩阵中满足同时下列两个条件的元素：

在同一行的所有元素中最小

在同一列的所有元素中最大

注：Leetcode 1380

分析：

#### 统计有序矩阵中的负数

给你一个m \* n的矩阵grid，矩阵中的元素无论是按行还是按列，都以非递增顺序排列。请你统计并返回grid中负数的数目。

注：Leetcode 1351

分析：

#### 寻找目标值 - 二维数组

注：LCR 121. 寻找目标值 - 二维数组

分析：

#### 单词搜索

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

注：Leetcode 79

分析：

#### 搜索二维矩阵

给你一个满足下述两条属性的m x n整数矩阵：

每行中的整数从左到右按非严格递增顺序排列。

每行的第一个整数大于前一行的最后一个整数。

给你一个整数 target ，如果 target 在矩阵中，返回 true ；否则，返回 false 。

注：Leetcode 74

分析：

采用模拟或二分查找。

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();

int row = 0;

int col = n - 1;

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

if (matrix[row][col] == target) {

return true;

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

col--;

} else {

row++;

}

}

return false;

}

};

#### 搜索二维矩阵II

编写一个高效的算法来搜索mxn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。

注：Leetcode 240

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;

}

};

#### 排序矩阵查找

注：面试题 10.09. 排序矩阵查找

分析：

### 修改

#### 将一维数组转变成二维数组

给你一个下标从0开始的一维整数数组original和两个整数m和n。你需要使用original中所有元素创建一个m行n列的二维数组。

original中下标从0到n - 1（都包含）的元素构成二维数组的第一行，下标从n到2 \* n - 1（都包含）的元素构成二维数组的第二行，依此类推。

请你根据上述过程返回一个 m x n 的二维数组。如果无法构成这样的二维数组，请你返回一个空的二维数组。

注：Leetcode 2022

分析：

#### 设置二维数组中的元素值

题目要求：在一个二维数组中，如果某一个元素的值为0，那么就将该元素所在的行与列中的所有元素都赋值为0，你能达到的最小空间复杂度是多少？

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

在一个二维数组中，如果某一个元素的值为0 将其所在的行和列所有元素都置为0，尽可能小的空间复杂度

使用某一行列存储即可，不需要存储全部行列，其他行列根据第1行1列设置

\*/

void SetZero(vector<vector<int> >& matrix)

{

bool firstLine = false;

bool firstRow = false;

//判断第一行是否有值为0 的元素

for(int i=0;i<matrix[0].size();i++)

{

if(matrix[0][i] == 0)

{

firstLine = true;

break;

}

}

//判断第一列是否有0的元素

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

{

if(matrix[i][0] == 0)

{

firstRow = true;

break;

}

}

//先处理第一行和第一列

for(int i=1;i<matrix.size();i++)

{

for(int j=1;j<matrix[0].size();j++)

{

if(matrix[i][j] == 0)

{

matrix[i][0] =0;

matrix[0][j] =0;

}

}

}

//根据第一行和第一列的元素标记，更新其他元素

for(int i=1;i<matrix.size();i++)

{

for(int j=1;j<matrix[0].size();j++)

{

if(matrix[i][0]==0 || matrix[0][j] ==0)

{

matrix[i][j] =0;

}

}

}

//更新第一行

if(firstLine)

{

for(int i=0;i<matrix[0].size();i++)

matrix[0][i] =0;

}

//更新第一列

if(firstRow)

{

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

matrix[i][0] = 0;

}

}

#### 删除每行中的最大值

给你一个 m x n 大小的矩阵 grid ，由若干正整数组成。

1、执行下述操作，直到 grid 变为空矩阵：

2、从每一行删除值最大的元素。如果存在多个这样的值，删除其中任何一个。

将删除元素中的最大值与答案相加。

注意：每执行一次操作，矩阵中列的数据就会减1。

返回执行上述操作后的答案。

注：Leetcode 2500

分析：

排序后处理。

#### 旋转图像

给定一个n×n的二维矩阵matrix表示一个图像。请你将图像顺时针旋转90度。

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

注：Leetcode 48

分析：

先对矩阵做沿主对角线的翻转（其实就是矩阵的转置），原地完成；接着再对操作后的矩阵进行根据列序号倒排。

#### 矩阵置零

给定一个m x n的矩阵，如果一个元素为0，则将其所在行和列的所有元素都设为0。请使用原地算法。

注：Leetcode 73

分析：

采用模拟实现。

class Solution {

public:

void setZeroes(vector<vector<int>>& matrix) {

int m = matrix.size();

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

vector<int> row(m), col(n);

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

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

if (matrix[i][j] == 0) {

row[i] = col[j] = true;

}

}

}

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

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

if (row[i] || col[j]) {

matrix[i][j] = 0;

}

}

}

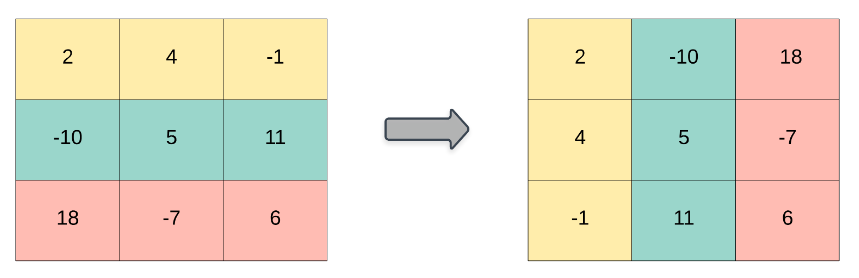
}

};

#### 转置矩阵

给你一个二维整数数组matrix，返回matrix的转置矩阵。

矩阵的转置是指将矩阵的主对角线翻转，交换矩阵的行索引与列索引。



注：Leetcode 867

分析：

直接使用转置矩阵模拟。

class Solution {

public:

vector<vector<int>> transpose(vector<vector<int>>& matrix) {

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

vector<vector<int>> transposed(n, vector<int>(m));

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

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

transposed[j][i] = matrix[i][j];

}

}

return transposed;

}

};

## 零子数组