写题理念

看到难题别怕，先慢慢拆解，看看难点是什么，然后拿东西去解决，不行再开始天马行空，别直接feeling启动了，一般的题目正常写都是可以解的，要分类好问题，看一下有哪几种情况。

多注意考虑边界情况，相加和问题，小心零，要多注意一下边界、特殊情况。

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

常用库：

#include algorithm

以下为常用函数

①、sort();【具有和快排一样的速度】

时间复杂度O (n*logn)

```
int a[5] = {4,2,1,3,5};
vector<int> b(a,a+5);
sort(a,a+5);//搭配数组  从小到大排序
sort(b.begin(),b.end());
```

写一个cmp函数 实现从大到小排序

```
#include <bits/stdc++.h>
using namespace std;

int cmp(int a, int b)
{
  return a > b;
  //蚂蚁感冒的正负数绝对值 return abs(a) < abs(b);
}
int main () {
    int a[5] = {4,2,1,3,5};
    vector<int> b(a,a+5);
    sort(a,a+5);//搭配数组  从小到大排序
    sort(b.begin(),b.end());//搭配容器      //从小到大
    for (int i = 0; i < 5; i ++) {
        cout << a[i] << " ";
    }
    cout << endl;

    for (auto x:b) {
        cout << x << " " ;
    }

    cout << endl;
    sort(b.begin(),b.end(),cmp); //从大到小
    for (auto x:b) {
        cout << x << " ";
    }
    return 0;
}
```

②__gcd 最大公约数

最大公约数小题

```
#include<cstdio>
#include<algorithm>
using namespace std;
int n,m;
int main()
{
    scanf("%d %d",&n,&m);
    int k=__gcd(n,m);//最大公约数
    printf("%d ",k);
    printf("%d", n * m / k); //最小公倍数
    return 0;
}
```

③max min

```
max(a,b);//返回最大值
min(a,b);//返回最小值
```

④swap

```
swap(a,b);//交换a和b
```

⑤lower_bound()与upper_bound() [二分查找]

时间复杂度O(log n)

使用之前一定要先排序

//使用方法

// 练习习题 https://leetcode.cn/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/?envType=study-plan&id=lcof&plan=lcof&plan_progress=x5qie1j7

//AC代码

```
class Solution {
public:
    int search(vector& nums, int target) {
        int l = lower_bound(nums.begin(), nums.end(), target) - nums.begin();
        int r = upper_bound(nums.begin(), nums.end(), target) - nums.begin();
        return r - l;
    }
};
```

⑥reverse 【倒置】

```
ector<int> v={1,2,3,4,5};
reverse(v.begin(),v.end());//v的值为5，4，3，2，1  倒置
```

⑦find

```
//在a中的从a.begin()（包括它）到a.end()（不包括它）的元素中查找10，
//若存在返回其在向量中的位置
  find(a.begin(),a.end(),10);
```

⑧、erase【删除】

```
//从c中删除迭代器p指定的元素，p必须指向c中的一个真实元素，不能等于c.end()
c.erase(p)
//从c中删除迭代器对b和e所表示的范围中的元素，返回e
c.erase(b,e)
```

#include cmath

1. 基本数学函数

函数功能示例abs(x)计算整数 x 的绝对值abs(-5) // 5fabs(x)计算浮点数 x 的绝对值fabs(-5.5) // 5.5fmod(x, y)计算 x 除以 y 的余数fmod(5.3, 2) // 1.3remainder(x, y)计算 x 除以 y 的余数remainder(5.5, 2) // 1.5fmax(x, y)返回 x 和 y 中的较大值fmax(3.5, 4.2) // 4.2fmin(x, y)返回 x 和 y 中的较小值fmin(3.5, 4.2) // 3.5hypot(x, y)计算 sqrt(x*x* + *yy*)hypot(3, 4) // 5

1. 指数和对数函数

函数功能示例exp(x)计算 e^x，e 为自然对数的底数exp(1) // 2.71828...log(x)计算 x 的自然对数log(2.71828) // 1log10(x)计算 x 的以 10 为底的对数log10(100) // 2pow(x, y)计算 x 的 y 次方pow(2, 3) // 8sqrt(x)计算 x 的平方根sqrt(16) // 4cbrt(x)计算 x 的立方根cbrt(27) // 3expm1(x)计算 e^x - 1expm1(1) // 1.71828...log1p(x)计算 log(1 + x)，适用于 x 接近 0 的情况log1p(0.00001) // 0.00001

1. 三角函数

函数功能示例sin(x)计算 x 的正弦值，x 以弧度为单位sin(3.14159 / 2) // 1cos(x)计算 x 的余弦值，x 以弧度为单位cos(3.14159) // -1tan(x)计算 x 的正切值，x 以弧度为单位tan(0) // 0asin(x)计算 x 的反正弦值，返回弧度asin(1) // 3.14159/2acos(x)计算 x 的反余弦值，返回弧度acos(-1) // 3.14159atan(x)计算 x 的反正切值，返回弧度atan(1) // 3.14159/4atan2(y, x)计算 y/x 的反正切值，返回弧度atan2(1, 1) // 3.14159/4

1. 取整和浮点数操作

函数功能示例ceil(x)返回不小于 x 的最小整数ceil(2.3) // 3floor(x)返回不大于 x 的最大整数floor(2.3) // 2trunc(x)返回去除小数部分的整数值trunc(2.8) // 2round(x)返回四舍五入到最接近的整数round(2.5) // 3lround(x)返回四舍五入到 long 类型lround(2.5) // 3llround(x)返回四舍五入到 long long 类型llround(2.5) // 3nearbyint(x)返回舍入到最接近整数（但不引发浮点异常）nearbyint(2.5) // 2rint(x)返回四舍五入到整数，符合当前舍入方式rint(2.5) // 3modf(x, &intpart)将 x 的整数和小数部分分离modf(2.3, &intpart)

一些经典模板（方法论）：

递归三部曲（深度优先）：

1. 确定递归函数的参数和返回值：确定在函数里要做什么处理。
2. 确定终止条件：先写终止条件！！操作系统也是用一个栈的结构来保存每一层递归的信息，如果递归没有终止，就会溢出。
3. 确认单纯递归的逻辑：确定每一层需要处理的逻辑。

模板题：

代码随想录二叉树遍历

广度优先：

```
vector<vector<int>> levelOrder(TreeNode* root)
{
        queue<TreeNode*> que;
        if (root != NULL) que.push(root);
        vector<vector<int>> result;
        while (!que.empty())
        {
            int size = que.size();
```

```
            vector<int> vec;// 这里一定要使用固定大小size，不要使用que.size()，因为
que.size是不断变化的
            for (int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                vec.push_back(node->val);
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
            }
            result.push_back(vec);
        }
        return result;
    }
```

while处理每一层，while里的for处理层内的每个节点。

贪心一般解题步骤

贪心算法一般分为如下四步：

- 将问题分解为若干个子问题
- 找出适合的贪心策略
- 求解每一个子问题的最优解
- 将局部最优解堆叠成全局最优解

回溯算法：

模板是：

```
先搞俩全局变量，储存单次的循环结果，另一个存总的结果
然后的回溯函数

backtraceing（）{
    if（终止条件）
    {
        存入result;
        return;
    }
    for（）
    {
        深度
        backtraceing（）；
        回溯
    }
}
```

其遍历过程就是：for循环横向遍历，递归纵向遍历，回溯不断调整结果集。

可以把他看作是一个树状结构，然后深入再广度

小知识

K--=k;

--k=k-1;

..别用x1，y1，这个在库函数里有，会重定义。。

注意有没有排序，可能题目的样例是排序好的，但是提交里不是，所以要仔细看清题目，看是否有明说排序好，没有的话最好还是自己先排序一下，

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

取平均数，写成下面这样样子，可以防止溢出。

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

1. **双引号（"a"）**：
    1. `"a"` 是一个字符串字面量，表示一个以空字符（`\0`）结尾的字符数组。
    2. 它属于 `const char[]` 类型，在内存中占用更多的空间，因为它包含终止的空字符。
    3. 字符串字面量是不可变的，这意味着你不能修改字符串中的字符。
    4. 例如，`"hello"` 是一个包含5个字符的字符串：`'h', 'e', 'l', 'l', 'o'`，后面跟着一个空字符。
2. **单引号（'a'）**：
    1. `'a'` 是一个字符字面量，表示单个字符。
    2. 它属于 `char` 类型，通常在内存中占用一个字节。
    3. 字符字面量可以用于表示单个字符，并且可以被修改（尽管在某些情况下，如字符串字面量内部的字符，它们是不可修改的）。

注意这个细节，例如：

```
record[s[i]-'a']++;
```

这个代码中"a"就不可以，只能用'a'。

函数参数是数组时，要&引用，不然只能传值进去。

erase 时间复杂度是O(n)

vector和string移除都用erase，要移除指定位置的元素，比如第i位的，vector.erase(vector.begin()+i);

移除一段范围内的元素

```
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.erase(vec.begin() + 1, vec.begin() + 3); // 移除第二个到第三个元素（值为2和3）
```

在这个例子中，`erase` 函数接受两个迭代器参数，分别指向要移除范围的开始和结束。不包括结束迭代器指向的元素。范围外的所有元素的迭代器、指针和引用都会变得无效。

sort

sort()并非只是普通的快速排序，除了对普通的快速排序进行优化，它还结合了插入排序和堆排序。根据不同的数量级别以及不同情况，能自动选用合适的排序方法。当数据量较大时采用快速排序，分段递归。一旦分段后的数据量小于某个阀值，为避免递归调用带来过大的额外负荷，便会改用插入排序。而如果递归层次过深，有出现最坏情况的倾向，还会改用堆排序。所以说sort()是一个比较灵活的函数，它也会根据我们数据的需要进行排序，所以我们就不用担心以上的问题了。对于大部分的排序需求，sort()都是可以满足的。

sort()基本使用方法

sort()函数可以对给定区间所有元素进行排序。它有三个参数sort(begin, end, cmp)，其中begin为指向待sort()的数组的第一个元素的指针，end为指向待sort()的数组的最后一个元素的下一个位置的指针，cmp参数为排序准则，cmp参数可以不写，如果不写的话，默认从小到大进行排序。如果我们想从大到小排序可以将cmp参数写为greater()就是对int数组进行排序，当然<>中我们也可以写double、long、float等等。如果我们需要按照其他的排序准则，那么就需要我们自己定义一个bool类型的函数来传入。比如我们对一个整型数组进行从大到小排序：

```
#include<iostream>
#include<algorithm>
using namespace std;

int main(){
        int num[10] = {6,5,9,1,2,8,7,3,4,0};
        sort(num,num+10,greater<int>());
        for(int i=0;i<10;i++){
                cout<<num[i]<<" ";
        }//输出结果:9 8 7 6 5 4 3 2 1 0
        return 0;

}
```

自定义排序准则

上面我们说到sort()函数可以自定义排序准则，以便满足不同的排序情况。使用sort()我们不仅仅可以从大到小排或者从小到大排，还可以按照一定的准则进行排序。比如说我们按照每个数的个位进行从大到小排序，我们就可以根据自己的需求来写一个函数作为排序的准则传入到sort()中。

我们可以将这个函数定义为：

```
bool cmp(int x,int y){
        return x % 10 > y % 10;
}
```

也可以对结构体排序

完整代码：

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;

struct Student{
        string name;
        int score;
        Student() {}
        Student(string n,int s):name(n),score(s) {}
};

bool cmp_score(Student x,Student y){
        return x.score > y.score;
}

int main(){
        Student stu[3];
        string n;
```

```
        int s;
        for(int i=0;i<3;i++){
                cin>>n>>s;
                stu[i] = Student(n,s);
        }

        sort(stu,stu+3,cmp_score);

        for(int i=0;i<3;i++){
                cout<<stu[i].name<<" "<<stu[i].score<<endl;
        }

        return 0;
}
```

关于值传递与引用传递

在以上的代码示例中使用了值传递，其实这并不是一种好的的做法，因为使用值传递每次调用函数时都会创建Student对象的副本，会增加额外的开销也会降低排序的效率。所以应该使用引用传递。使用引用传递的好处在于：

避免拷贝开销：值传递会创建参数的副本，对于大型对象或复杂数据结构，这可能涉及大量的内存分配和数据复制。引用传递避免了这些操作，因为它直接操作原始对象。

提高执行效率：由于避免了拷贝操作，函数调用的执行速度会更快，尤其是在处理大型数据或在需要频繁调用函数的情况下。

减少内存使用：**引用传递不涉及额外的内存分配，因此可以减少程序的内存占用。

允许修改原始数据：**引用传递允许函数直接修改原始数据，这在某些情况下非常有用，比如在排序函数中修改对象的内部状态。

保持数据一致性：引用传递确保函数内部对数据的任何修改都会反映到原始数据上，这有助于保持数据的一致性。

所以应该这样写。

```
bool cmp_score(const Student& x, const Student& y) {
    double average_x = (x.score[0] + x.score[1] + x.score[2] + x.score[3]) / 4;
    double average_y = (y.score[0] + y.score[1] + y.score[2] + y.score[3]) / 4;
    return average_x > average_y;
}
```

指针和引用的区别：

https://zhuanlan.zhihu.com/p/681617362

数据量太大的情况下，用scanf和printf，节省时间。

```
while (~scanf("%d%d", &a, &b))
 {
     int sum;
     if (a == 0) sum = p[b];
     else sum = p[b] - p[a - 1];
     printf("%d\n", sum);
 }
```

数组

二分查找：

bug：if（nums[mid]=target）。。。。。。。

二分法要注意区间的定义，先明确定义在去设计函数。

在这道题目中我们讲到了循环不变量原则，只有在循环中坚持对区间的定义，才能清楚的把握循环中的各种细节。

移除元素：

双指针法标准题目。

https://leetcode.cn/problems/remove-element/

思路不对，应该用双指针法，快指针去找非目标值的数字，慢指针直接覆盖原数组，得到新的数组，就是原地覆盖。

个人理解：遍历原始数组的速度一定大于覆盖的速度。

vector result(A.size(), 0);第一个参数是大小，第二个参数是初始化的值。

有序数组的平方：

https://leetcode.cn/problems/squares-of-a-sorted-array/description/

也是双指针。

双指针的理解：一个目的有两个点有矛盾冲突，或者有两个点有事情要做，然后就可以围绕这个点来建立双指针。**

长度最小的子数组：

双指针。

滑动窗口。

螺旋矩阵II

就是模拟，要注意边界的判断，主要是要去一个一个仔细写，复制的容易出问题。

质数表的构建

线性构建法，核心思想是：使得每一个数，都可以由它最小的质数因子去掉，

线性筛法,即是筛选掉所有合数,留下质数

我们知道合数可以由一个质数数与另一个合数相乘得到

而同时假设合数a=质数b×质数c×一个数d

令e=c × d,假设b ≥ e,e为合数,令f=d × b

a=f × c ,其中c

即比一个合数数大的质数和该合数的乘积可用一个更大的合数和比其小的质数相乘得到（你从小的质数开始筛选，那么我们就可以找出）

这也是if(!(i % prime[j]))break;的含义,这也是线性筛法算质数表的关键所在

例如遍历到9，9筛选到3就可以停止了，因为后续的数都是最大以3为最小质因数，例如94，*可以在后续遍历到18的时候去掉，*95在15的时候去掉。

```
#include <stdio.h>
#include <string.h>
#define MAXN 100000
#define MAXL 1000000
int prime[MAXN];
bool check[MAXL];

int main(void)
{
    int n, count;
    while (~scanf("%d", &n))
    {
        memset(check, 0, sizeof(check));
        count = 0;
        for (int i = 2; i <= n; i++)
        {
            if (!check[i])         //如果是素数
                prime[count++] = i;         //记录在数组中
            for (int j = 0; j < count; j++)
            {
                if (i*prime[j] > MAXL)         //判断是否越界
                    break; // 过大的时候跳出
                check[i*prime[j]] = 1; //如果check[i*prime[j]]是已知素数的整数倍那一定是合数
                if ( (i%prime[j]) == 0 ) // 如果i是一个合数，而且i % prime[j] == 0
                    break;
            }
        }
        for (int i = 0; i < count; i++)
            printf("%d\n", prime[i]);
    }
    return 0;
}
```

链表

链表的构造函数：

```
struct Node
{
    int val;
    node* next;
    Node(int val):val(val),next(nullptr){}
}
```

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

虚拟头节点

很好用很关键，这个可以让第一个节点一般化，不用单独对其进行考虑运算。

移除链表元素

c++没有自动清理内存，虽然leetcode能过，但是还是要手动清理一下好。

```
ListNode* tmp = node->next;
node->next=node->next->next;
delete tmp;
```

本题bug点：

while(head!=nullptr&&head->val==val)，应该要先判断存在不存在。

反转链表

有点烧脑，主要是流程得捋清楚，每个循环是什么样的。

第二遍看：感觉比第一次看简单的多，纠结久了头会乱掉，需要跳脱出来重新看一下。

链表相交

bug点：有个变量没有初始化到，要注意这个细节，中途加的变量，要保证赋值的语句一定执行到，比如：

```
int length=lenA;
        for(int i =0;i<diff;i++)
        {
            if(lenA>lenB)
            {
                curA=curA->next;
                length=lenB;
            }else{
                curB=curB->next;
                length=lenA;
            }
        }
```

如果这个代码中diff=0，他就不会执行这个for语句，这样就会导致length不被赋值

哈希表

构建哈希表的重点是

哈希函数

平方取中法

- **平方取中法**：先通过求关键字平方值的方式扩大相近数之间的差别，然后根据表长度取关键字平方值的中间几位数为哈希地址。

除留余数法

- **除留余数法**：假设哈希表的表长为 `m`，取一个不大于 `m` 但接近或等于 `m` 的质数 `p`，利用取模运算，将关键字转换为哈希地址。即：`Hash(key) = key % p`，其中 `p` 为不大于 `m` 的质数。

基数转换法

- **基数转换法**：将关键字看成另一种进制的数再转换成原来进制的数，然后选其中几位作为哈希地址。

但是肯定还是会发生冲突，冲突的解决办法有

开放地址法：指的是将哈希表中的「空地址」向处理冲突开放。当哈希表未满时，处理冲突时需要尝试另外的单元，直到找到空的单元为止。

简单的做法就是放在冲突地址的下一个单元。

链地址法（Chaining）：将具有相同哈希地址的元素（或记录）存储在同一个线性链表中。

算法部分

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

要判断用什么容器来做哈希表，

如果能用数组就用数组，数组的空间消耗，时间消耗都比较少。

数组：需要题目限制数值的大小，如果哈希值比较少、特别分散、跨度非常大，使用数组就造成空间的极大浪费。

集合：如果不用排序，而且数据不要重复就用unordered_set，这个查询、增减很快。

map映射：需要存俩值的情况用。

```
for (unordered_map<int, int>::iterator it = map.begin(); it != map.end(); it++)
{
}
```

map的遍历这样写。

两个数组的交集

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> result_set;
        unordered_set<int> nums1_set(nums1.begin(),nums1.end());
        //创建一个 unordered_set 的实例 nums_set，它包含了容器 nums1（可以是 vector、
array 或其他容器）中所有的元素。nums1.begin() 返回指向 nums1 第一个元素的迭代器，
nums1.end() 返回一个指向 nums1 结束位置的迭代器。unordered_set 的构造函数会使用这两个迭代
器来拷贝 nums1 中的所有元素到新创建的 unordered_set 中。
        for(int num:nums2)
        {
            if(nums1_set.find(num)!=nums1_set.end())//如果表中find不到某一个数，就
return set.end()迭代器
            {
                result_set.insert(num);
            }
        }
        return vector<int>(result_set.begin(),result_set.end());
    }
};
```

两数之和

为什么用哈希表：

思路主要是要找另一个数，我们有一个数，然后有这个数的和，做差去找另一个数即可，以这个思路我们就只需要遍历一遍就有结果了。

为什么会想到做差去找，因为和是已知的，要利用满所给的数据，尽可能做到不浪费信息。、

哈希表为什么用map：

因为他要返回数在数组里的索引，要存两个数据，也就是一对数据，所以要用map。

map基础知识：

这行代码通过迭代器 `iter` 访问 `map` 中元素的第二个元素，即值部分（`value`）。在 `map` 中，每个元素都是一个 `pair`，其中 `first` 是键（`key`），`second` 是值（`value`）。因此，`iter->second` 获取了与键 `target - nums[i]` 相关联的值。

字符串

字符串的库函数

```
    s.resize(s.size() + count * 5);
```

swap();

reverse();翻转字符串在#include 下

用好翻转函数，很多时候有奇效。

```
//string
reverse(str.begin(), str.end());
//vector
reverse(vec.begin(), vec.end());
```

1分割截取substr()

```
str.substr(7,3); // 从下标7开始截取子字符串，截取长度为3的字符串
```

2.查找指定子字符串find()

在字符串中查找指定子字符串,并返回其第一次出现的位置

```
size_t pos = str.find("World"); // 查找子字符串"World"的位置
```

3.替换字符串中的一部分replace()

```
str.replace(7, 5, "Universe"); // 替换从下标7开始的5个字符为"Universe"
```

4.在指定位置插入字符串insert()

```
str.insert(5, "Beautiful "); // 在下标5处插入字符串"Beautiful "
```

5.复制字符串(两种方法)

```
string str3(str1)//复制str到s
str3 = str1;// 复制 str1 到 str3
```

6.排序sort()

！！！需要头文件#include

```
string s = "12sklhfsabfskfb,a aghs 1425416 27638";
sort(s.begin(), s.end());
cout << s << endl;
```

//输出如下:

// ,111222344566678aaabbfffghhkklssss

7.删除erase()

使用erase()函数

```
string s="12345678";
s.erase(s.begin()+3);//使下标3的字符删掉
```

//输出

//1235678

删除后变为"1235678"，即把（默认第一个下标为0）下标3的字符删掉。

kmp

KMP的精髓所在就是前缀表。

https://blog.csdn.net/2301_77160836/article/details/143169087

关键思路要理顺，比如说到底是先验证能不能加，然后再加；还是先加再看能不能行，不行再回退。

栈和队列

基础知识

栈和队列不算容器，算容器适配器，他只是提供接口，相当于对底层容器进行包装，但是不规定具体使用那种底层容器，缺省情况下（也就是不指定的情况），栈和队列都是用deque实现的，这是一个双向队列。

这种数据结构，不提供迭代器，不允许遍历行为，接口只有两端。

单调队列

https://www.programmercarl.com/0239.%E6%BB%91%E5%8A%A8%E7%AA%97%E5%8F%A3%E6%9C%80%E5%A4%A7%E5%80%BC.html#%E7%AE%97%E6%B3%95%E5%85%AC%E5%BC%80%E8%AF%BE

因为这题是要弹出最大值，所以只需要维护保持队列中单调。

我们新建一个队列，然后记录窗口内的最大值和可能成为最大值的数（这一点很重要），因为最大值可能会在后续窗口滑动的时候弹出，所以要记录可能成为最大值的数，但是这样说太玄了，要怎么实现呢：

就是搞一个单调队列，在一个deque中保持单调，比如说我们确定左边最大，，然后对于新进来的数，就从从左边开始询问最右边的数是否比新数，直到队列为空或者最右边的数大于新数，然后就把新数排在最右边。

然后，，有数要脱离窗口时，只需要检查最左边（也就是最大的数）是不是要脱离的数，就算不是这个数也没啥，因为里面剩下的都是后面来的更大的数，他们会更晚走，前面进数的时候直接把比新数小的直接删去就是这个目的，在新数进来的时候，他的剩余存活时间一定是最长的，所以他的存活的时候那些被他删去的老数不可能成为最大值。

优先级队列的使用

```
#include <queue>

// 声明一个整型优先队列
priority_queue<int> pq;

// 声明一个自定义类型的优先队列，需要提供比较函数
//这个可以是struct也可以是class
struct compare {
    bool operator()(int a, int b) {
        return a > b; // 这里定义了最小堆
    }
};
priority_queue<int, vector<int>, compare> pq_min;
//这个定义中，第一个变量是储存类型，
//第二个是储存容器，
//第三个是比较函数，也可以用来定义是最大堆还是最小堆
```

栈的使用，栈其实是递归的一种实现结构。

二叉树

基础知识

完全二叉树：按数组顺序排，中间不能有缺口。

看代码随想录。

存储方式

一般是链式存储，就是和链表一样连在一起，顺序存储也可以

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

如果父节点的数组下标是 i，那么它的左孩子就是 i * 2 + 1，右孩子就是 i * 2 + 2。

二叉树的深度优先和广度优先

看代码随想录。

之前我们讲栈与队列的时候，就说过栈其实就是递归的一种实现结构，也就说前中后序遍历的逻辑其实都是可以借助栈使用递归的方式来实现的。

而广度优先遍历的实现一般使用队列来实现，这也是队列先进先出的特点所决定的，因为需要先进先出的结构，才能一层一层的来遍历二叉树。

这个也是栈、队列的应用场景。

定义

一定要会手写!!!每次看到手写一遍，主要是结构体的构造不熟悉

```
struct TreeNode
{
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int n):val(n),left(nullptr),right(nullptr){}
}
```

二叉树统一迭代法

没到null就是没到底，所以可以继续往下走， 如果碰到null，就是到底了，可以需要返回值了。

各种遍历的特点

后续遍历

服了找不到gif。。

后续遍历是从最底部往上遍历，因此和回溯关系非常紧密。

前序遍历

先遍历自身，节点本身好找。

二叉搜索树

遍历的方式，通过返回值来遍历、重构二叉树的结构：

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if (root == nullptr ) return nullptr;
        if (root->val < low) {
            TreeNode* right = trimBST(root->right, low, high); // 寻找符合区间[low,
high]的节点
            return right;
        }
        if (root->val > high) {
            TreeNode* left = trimBST(root->left, low, high); // 寻找符合区间[low,
high]的节点
            return left;
        }
        root->left = trimBST(root->left, low, high); // root->left接入符合条件的左孩
子
        root->right = trimBST(root->right, low, high); // root->right接入符合条件的
右孩子
        return root;
    }
```

```
};
```

随便挑的一题，重点是return返回的值是root->left所指向的节点，通过这个方式来重构二叉树，也能以此来进行遍历，如果需要换节点，就换返回值，正常情况就返回自己本身即可。

回溯算法

本质还是暴力，但是可控暴力，**主要可以自由次数的****for循环。**

模板是：

```
先搞俩全局变量，储存单次的循环结果，另一个存总的结果
然后的回溯函数

backtraceing（）{
    if（终止条件）
    {
        存入result;
        return；
    }
    for（）
    {
        深度
        backtraceing（）；
        回溯
    }
}
```

- 其遍历过程就是：for循环横向遍历，递归纵向遍历，回溯不断调整结果集。
- 可以把他看作是一个树状结构，然后深入再广度
- 树状结构其实就是有很多分支，抽象处理其实就是分支。
- 先构造完基础框架，在进行剪枝处理。

回溯法解决的问题

回溯法，一般可以解决如下几种问题：

- 组合问题：N个数里面按一定规则找出k个数的集合
- 切割问题：一个字符串按一定规则有几种切割方式
- 子集问题：一个N个数的集合里有多少符合条件的子集
- 排列问题：N个数按一定规则全排列，有几种排列方式
- 棋盘问题：N皇后，解数独等等

大部分的原因：因为不知道要循环几轮，要循环n-1轮，不可控，所以正常循环不行，得递归来深搜。

剪枝

剪枝精髓是：for循环在寻找起点的时候要有一个范围，如果这个起点到集合终止之间的元素已经不够 题目要求的k个元素了，就没有必要搜索了。

一开始可以先想想优化的办法，但是如果不能形成完整的解法的话，就老老实实写基础版，然后做剪枝。

去重

https://www.programmercarl.com/0040.%E7%BB%84%E5%90%88%E6%80%BB%E5%92%8CII.html#%E7%AE%97%E6%B3%95%E5%85%AC%E5%BC%80%E8%AF%BE

这题去重思路很重要，递归中去重一定得会这个，

个人理解：感觉有一点点像是高中c几取几和a几取几的区别，只遍历半轮，不让重复。

但是这题是组合问题，对于组合问题，本题的重点是集合可以有重复，但是组合不能重复，提取出来，也就是使用不能重复，对于不一样的数无所谓，重点在**1**同样的数要怎么处理，以及**2**如何在处理完这些数后，后面不会再碰到相同的数？由此我们可以想到需要排序，相当于我们把相同的数聚在一起了，**问题2**解决了，现在回到**问题1，**如何**处理相同**的数，让他**不重复出现在组合内**，他是组合，不能同时出现两个节点有两个相同元素，一个节点有两个相同元素，往下深是可能可以得到两个不同组合的，但是如果存在另一个节点有两个相同元素就会重复，所以直接断根，不如另一个节点产生。

贪心算法

基础逻辑就是每一阶段的局部最优，最后可以达到全局最优。

贪心一般解题步骤

贪心算法一般分为如下四步：

- 将问题分解为若干个子问题
- 找出适合的贪心策略
- 求解每一个子问题的最优解
- 将局部最优解堆叠成全局最优解

一些例题

代码随想录最经典的贪心逻辑，1每部取最大来得到最优解。

在遇到一个排列（之类的）有两个维度来管理，就需要先选择一者先管理，然后再去搞另一个，例题：https://www.programmercarl.com/0406.%E6%A0%B9%E6%8D%AE%E8%BA%AB%E9%AB%98%E9%87%8D%E5%BB%BA%E9%98%9F%E5%88%97.html#%E6%80%9D%E8%B7%AF

https://www.programmercarl.com/0135.%E5%88%86%E5%8F%91%E7%B3%96%E6%9E%9C.html#%E7%AE%97%E6%B3%95%E5%85%AC%E5%BC%80%E8%AF%BE

用好排序，排序过的这个逻辑可以解决很多问题，

动态规划dp

动态规划的解题步骤

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序 **很重要，可能是从前往后，也可能是从后往前**
5. 举例推导dp数组

一些同学可能想为什么要先确定递推公式，然后在考虑初始化呢？

因为一些情况是递推公式决定了dp数组要如何初始化！**

动态规划应该如何debug

把dp数组打印出来，看看究竟是不是按照自己思路推导，在写完dp数组后一定要模拟一下试试，保证没问题。

出bug了就问自己：

- 这道题目我举例推导状态转移公式了么？
- 我打印dp数组的日志了么？
- 打印出来了dp数组和我想的一样么？

模板题

代码随想录 经典爬楼梯，顺便搜一下面试问爬楼梯的。

面试题：

此时我就发现一个绝佳的大厂面试题，第一道题就是单纯的爬楼梯，然后看候选人的代码实现，如果把dp[0]的定义成1了，就可以发难了，为什么dp[0]一定要初始化为1，此时可能候选人就要强行给dp[0]应该是1找各种理由。那这就是一个考察点了，对dp[i]的定义理解的不深入。

然后可以继续发难，如果一步一个台阶，两个台阶，三个台阶，直到 m个台阶，有多少种方法爬到n阶楼顶。这道题目leetcode上并没有原题，绝对是考察候选人算法能力的绝佳好题。

这一连套问下来，候选人算法能力如何，面试官心里就有数了。

其实大厂面试最喜欢问题的就是这种简单题，然后慢慢变化，在小细节上考察候选人。

这道绝佳的面试题我没有用过，如果录友们有面试别人的需求，就把这个套路拿去吧。

我在通过一道面试题目，讲一讲递归算法的时间复杂度！ (opens new window)中，以我自己面试别人的真实经历，通过求x的n次方 这么简单的题目，就可以考察候选人对算法性能以及递归的理解深度，录友们可以看看，绝对有收获！

难题

代码随想录这题算法不难，主要是要如何把求二叉树数量转化为动态规划问题。

对数组的空间优化方式

https://www.programmercarl.com/0070.%E7%88%AC%E6%A5%BC%E6%A2%AF.html#%E6%80%9D%E8%B7%AF

好夸张，个人感觉主要的思路是节约没用的空间，数组前面的部分只是为了算后面的答案，所以就可以一直覆盖的来得到最后的答案。

01背包问题？？？？一题不会啊

最基础的逻辑就是这个dp递推式

```
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])
```

抽象化的问题就是有n个物品，不知道到底取那几样，但是每样都只能取一个。

滚动数组的话就是高效利用空间，把原本后面占用但是已经没用的空间优化掉，直接在现有的空间上覆盖更新。

但是这种重复更新的情况需要注意，要反过来，从大到小遍历，这样才不会一个东西放入好多遍。、、

碰到有n个物品，不知道到底取那几样，但是每样都只能取一个，（去往代价/价值的方向去靠）不一定，构造出一个01背包框架。

从例题讲变化维度

第一题：

分割组合问题，结果是使得两个子集元素和相等。

元素和相等->能否组合出一个子集等于总和的一半->背包框架已经有了

第二题：

1049.最后一块石头的重量II

这题难点在如何想到变成01背包组合问题，计算机思维还是不够。

直接化成分配问题，看看要怎么组合才能相差的最少->也就是算如果背包只有一半大小，我要怎么放才能最接近一半->背包

第三题

目标和

难点在于想法，也是计算机思维不够，每个物体有两个选择，都只能有一个物体（01背包框架）

有一点数论知识，如果遇到等式多的类似问题，可以先列几个式子找找感觉。

这题等式找到了直接秒了。

第四题

1 0

到这其实有感觉了，但是感觉O(knm）的时间复杂度有点怀疑自己三次方有点大了，但其实没错，2维的代价，1维的收益。写就行了。

图论

基础看代码随想录

邻接矩阵，适合稠密图

邻接表，适合稀疏图。

深搜（dfs）

代码框架：

用递归，框架基本上是一样的。

```
vector<vector<int>> result; // 保存符合条件的所有路径
vector<int> path; // 起点到终点的路径
void dfs(图，目前的节点/数据) {
    if (终止条件) {
        存放结果;
        return;
    }
    for (选择：本节点所连接的其他节点)
    {
        处理节点;
        dfs(图，选择的节点); // 递归
```

```
            回溯，撤销处理结果
        }
    }
```

广搜（bfs）

适合解决两个点之间的最短路径问题。二叉树的层序遍历就是广搜。

一般来说用队列，处理的可以简单一点。但是栈也是可以的。

```
int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1}; // 表示四个方向
// grid 是地图，也就是一个二维数组
// visited标记访问过的节点，不要重复访问
// x,y 表示开始搜索节点的下标
void bfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int x, int
y) {
    queue<pair<int, int>> que; // 定义队列
    que.push({x, y}); // 起始节点加入队列
    visited[x][y] = true; // 只要加入队列，立刻标记为访问过的节点
    while(!que.empty()) { // 开始遍历队列里的元素
        pair<int ,int> cur = que.front(); que.pop(); // 从队列取元素
        int curx = cur.first;int cury = cur.second; // 当前节点坐标
        for (int i = 0; i < 4; i++) { // 开始想当前节点的四个方向左右上下去遍历
            int nextx = curx + dir[i][0];
            int nexty = cury + dir[i][1]; // 获取周边四个方向的坐标
            if (nextx < 0 || nextx >= grid.size() || nexty < 0 || nexty >=
grid[0].size()) continue;  // 坐标越界了，直接跳过
            if (!visited[nextx][nexty]) { // 如果节点没被访问过
                que.push({nextx, nexty});  // 队列添加该节点为下一轮要遍历的节点
                visited[nextx][nexty] = true; // 只要加入队列立刻标记，避免重复访问
            }
        }
    }
}
```

并查集

用来快速判断连通性问题，大致思路就是在并查集里，让所有元素都指向他的根，然后通过元素是不是同根来判断连通性，主要特点是，（路径压缩）在每次查询或添加时，都要重构一下同根，为了保持每个元素都指向根节点。

代码模板

```
int n = 1005; // n根据题目中节点数量而定，一般比节点数量大一点就好
vector<int> father = vector<int> (n, 0); // C++里的一种数组结构
// 并查集初始化
void init() {
    for (int i = 0; i < n; ++i) {
        father[i] = i;
        }
}// 并查集里寻根的过程
int find(int u) {
    return u == father[u] ? u : father[u] = find(father[u]); // 路径压缩
}
// 判断 u 和 v是否找到同一个根
bool isSame(int u, int v) {
```

```
    u = find(u);
    v = find(v);
    return u == v;
}// 将v->u 这条边加入并查集
void join(int u, int v) {
    u = find(u); // 寻找u的根
    v = find(v); // 寻找v的根
    if (u == v) return ; // 如果发现根相同，则说明在一个集合，不用两个节点相连直接返回
    father[v] = u;
}
```

w位运算

一、位运算符

C++ 提供了按位与（&）、按位或（| ）、按位异或（^）、取反（~）、左移（<<）、右移（>>）这 6 种位运算符。 这些运算符只能用于整型操作数，即只能用于带符号或无符号的 char、short、int 与 long 类型。

（1）按位与运算符（&）

“a&b”是指将参加运算的两个整数a和b，按二进制位进行“与”运算。

运算规则：0&0=0; 0&1=0; 1&0=0; 1&1=1; 即：两位同时为“1”，结果才为“1”，否则为0

例如：3&5 即 0000 0011& 0000 0101 = 0000 0001 因此，3&5的值得1。

另，负数按补码形式参加按位与运算。

按位与&比较实用的例子：

1、比如我们经常要用的是否被2整除，一般都写成 if(n % 2 = = 0) 可以换成 if((n&1) = = 0)

2、按位与运算可以取出一个数中指定位。例如：要取出整数84从左边算起的第3、4、5、7、8位，只要执行84 & 59,因为84对应的二进制为01010100，59对应的二进制为00111011，01010100 & 00111011= 00010000 可知84从左边算起的第3、4、5、7、8位分别是0、1、0、0、0。

3、清零。如果想将一个单元清零，使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。

按位与应用举例1:

整数幂：判断一个数n ，是不是2的整数幂。比如：64 = 2^6,所以输出“yes”,而65无法表示成2的整数幂的形式，所以输出“NO”。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{  int n;
   cin>>n;
   if(n&(n-1))cout<<"NO";
   else cout<<"Yes";
}
```

按位与应用举例2:

计算一个数的二进制中1的个数:

算法1：通过与初始值为1的标志位进行与运算，判断最低位是否为1；然后将标志位左移，判断次低位是否为1；一直这样计算，直到将每一位都判断完毕。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{          int n = 0,num;
         unsigned int flag = 1;
         cin>>num;
          while(flag)
         {         if(num & flag) n++;
                  flag = flag << 1;
         }
         cout<<n;
}
```

算法2：还有一种方法，一个整数减一，可以得到该整数的最右边的1变为0，这个1右边的0变为1。对这个整数和整数减一进行与运算，将该整数的最右边的1变为0，其余位保持不变。直到该整数变为0，进行的与运算的次数即为整数中1的个数。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{          int n = 0,num;
         unsigned int flag = 1;
         cin>>num;
          while(num)
         {         num = num & (num - 1);
                 n++;
         }
         cout<<n;
}
```

（2）按位或运算符（|）

参加运算的两个对象，按二进制位进行“或”运算。

运算规则：0|0=0； 0|1=1； 1|0=1； 1|1=1；

即：参加运算的两个对象只要有一个为1，其值为1。

例如:3|5 即 00000011 | 0000 0101 = 00000111 因此，3|5的值得7。

另，负数按补码形式参加按位或运算。

按位或 （|） 比较实用的例子

可以用一个unsigned int 来存储多个布尔值。比如一个文件有读权限，写权限，执行权限。看起来要记录3个布尔值。我们可以用一个unsigned int也可以完成任务。

一个数r来表示读权限，它只更改个位来记录读权限的布尔值

00000001（表示有读权限） 00000000（表示没有读权限）

一个数w表示写权限，它只用二进制的倒数第二位来记录布尔值

00000010 (表示有写权限) 00000000 (表示没有写权限)

一个数x表示执行权限，它只用倒数第三位来记录布尔值

00000100 (表示有执行权限)　00000000 (表示没有执行权限)

那么一个文件同时没有3种权限就是　　~r | ~ w | ~ x 即为 00000000，就是0

只有读的权限就是　　r | ~w | ~x 即为 00000001，就是1

只有写的权限就是　　~r | w | ~x 即为 00000010，就是2

一个文件同时有3种权限就是　　r | w | x 即为 00000111，就是7

（3）按位异或运算符（^）**

参加运算的两个数据，按二进制位进行“异或”运算。

运算规则：0 ^ 0=0；　0 ^ 1=1；　1^ 0=1；　1^1=0；

　即：参加运算的两个对象，如果两个相应位为“异”（值不同），则该位结果为1，否则为0。

下面重点说一下按位异或,异或 其实就是不进位加法,如1+1=0，,0+0=0,1+0=1。

异或的几条性质:

1、交换律:a ^ b=b ^ a

2、结合律:(a ^ b) ^ c == a^ (b ^ c)

“异或运算”的特殊作用：

（1）使特定位翻转:　例：X=10101110，使X低4位翻转，用X ^ 0000 1111 = 1010 0001即可得到。

（2）与0相异或，保留原值，10101110^ 00000000 = 1010 1110。

（3）对于任何数x都有——自反性:x^ x=0，x^ 0=x　例如：A^B ^ B = A

（4）交换二个数：a　=a ^ b;　b = b ^ a;　a = a ^ b;

按位异或应用举例1:**

给出 n 个整数，n 为奇数，其中有且仅有一个数出现了奇数次，其余的数都出现了偶数次。用线性时间复杂度、常数空间复杂度找出出现了奇数次的那个数。

【输入样例】

9

3 3 7 2 4 2 5 5 4

【输出样例】

7

```
#include<bits/stdc++.h>
using namespace std;
int main()
{   int i,n,m,a;
    cin>>n;
    cin>>a;
    for(int i = 2; i <= n; i++)
    {cin>>m; a^=m;   }
    cout<<a<<endl;
}
```

按位异或 应用举例2:

1-1000放在含有1001个元素的数组中，只有唯一的一个元素值重复，其它均只出现

一次。每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空

间，能否设计一个算法实现?

解法一、显然已经有人提出了一个比较精彩的解法，将所有数加起来，减去1+2+...+1000的和。

这个算法已经足够完美了，相信出题者的标准答案也就是这个算法，唯一的问题是，如果数列过大，则可能会导致溢出。

解法二、异或就没有这个问题，并且性能更好。

将所有的数全部异或，得到的结果与1^2^3^...^1000的结果进行异或，得到的结果就是重复数。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{  int i,n,a[11]={1,2,5,3,4,5,6,7,8,9,10};
   n=a[0]^a[1];
  for(i=2;i<=10;i++)n=n^a[i];
  for(i=1;i<=10;i++)n=n^i;
  cout<<n;
}
```

按位异或** 应用举例3:**

一系列数中,除两个数外其他数字都出现过两次,求这两个数字,并且按照从小到大的顺序输出.例如 2 2 1 1 3 4.最后输出的就是3 和4

```
#include<bits/stdc++.h>
using namespace std;
int a[1000];
int main()
{       int n;
        scanf("%d", &n);
        int x = 0;
        for(int i = 1; i <= n; i++) {       scanf("%d", &a[i]); x ^= a[i];    }
        int num1 = 0, num2 = 0;
        int tmp = 1;
        while(!(tmp & x)) tmp <<= 1;
        cout<<tmp<<endl;
        for(int i = 1; i <= n; i++) {
            if(tmp & a[i]) num1 ^= a[i];
            else num2 ^= a[i];
        }
        printf("%d %d\n", min(num1, num2), max(num1, num2));
    return 0;
}
```

（4）按位取反运算符（~）

按位取反运算符（~）是指将整数的各个二进制位都取反，即1变为0，0变为1。

例如，~9 = - 10，因为9（00001001）所有位取反即为（11110110），这个数最高位是1，所以是补码。补码还原成反码（反码等于补码减1）得到（11110101），再还原为原码（反码到原码最高位不变，其它各位取反）等于（10001010），　十进制为 - 10。

（5）按位左移运算符（<<）

左移运算符是用来将一个数的各二进制位左移若干位，移动的位数由右操作数指定（右操作数必须是非负值），其右边空出的位用0填补，高位左移溢出则舍弃该高位。

在高位没有1的情况下，左移1位相当于该数乘以2，左移2位相当于该数乘以2*2 = 4,15 < < 2=60，即乘了 4 。

但此结论只适用于该数左移时被溢出舍弃的高位中不包含1的情况。

例如：143<<2 结果为60　因为143转换为进制为10001111，左移2得00111100 ，结果为60。

（6）按位右移运算符（>>）

右移运算符是用来将一个数的各二进制位右移若干位，移动的位数由右操作数指定（右操作数必须是非负值），移到右端的低位被舍弃，对于无符号数，高位补0。对于有符号数，某些机器将对左边空出的部分用符号位填补（即“算术移位”），而另一些机器则对左边空出的部分用0填补（即“逻辑移位”）。

注意：对无符号数,右移时左边高位移入0；对于有符号的值,如果原来符号位为0(该数为正),则左边也是移入0。

如果符号位原来为1(即负数),则左边移入0还是1,要取决于所用的计算机系统。有的系统移入0,有的

系统移入1。移入0的称为“逻辑移位”,即简单移位；移入1的称为“算术移位”。

例： a的值是八进制数113755：

a:1001011111101101 （用二进制形式表示）

a>>1: 0100101111110110 (逻辑右移时)

a>>1: 1100101111110110 (算术右移时)

在有些系统中,a>>1得八进制数045766,而在另一些系统上可能得到的是145766。Turbo C和其他一些C

编译采用的是算术右移,即对有符号数右移时,如果符号位原来为1，左面移入高位的是1。

源代码：

```
#include <stdio.h>
main()
{ int a=0113755; printf("%d",a>>1);}
```

（7）位运算优先级**

总的来说比较低，逻辑运算符和数学运算符出现在同一个表达式中，那么需要用括号来表达运算次序。

（8）复合赋值运算符**

位运算符与赋值运算符结合，组成新的复合赋值运算符，它们是：

1、&= 例：a &=b　　相当于a=a& b

2、|= 例：a |=b　　相当于a=a |b

3、>>= 例：a >>=b　相当于a=a>> b

4、<<= 例：a<<=b　　相当于a=a<< b

5、^= 例：a ^= b　　相当 a=a ^b

运算规则：和前面讲的复合赋值运算符的运算规则相似。

（9）不同长度的数据进行位运算****

如果两个不同长度的数据进行位运算时，系统会将二者按右端对齐，然后进行位运算。

以“与”运算为例说明如下：如果一个4个字节的数据与一个2个字节数据进行“与”运算，右端对齐后，左边不足的位依下面三种情况补足：

（1）如果整型数据为正数，左边补16个0。

（2）如果整型数据为负数，左边补16个1。

（3）如果整形数据为无符号数，左边也补16个0。

（10）下面列举一些常见的二进制位的变换操作****

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。