第3章 数据结构基础

本章介绍基础的数据结构和一个比较简单的高级数据结构——并查集。它们是蓝桥杯软件赛的必考知识点。

很多计算机教材提到：程序＝数据结构＋算法[①]。数据结构是计算机存储、组织数据的方法。在常见的数据结构教材中一般包含数组(Array)、栈(Stack)、队列(Queue)、链表(Linked List)、树(Tree)、图(Graph)、堆(Heap)、散列表(Hash Table)等内容。数据结构分为线性表和非线性表两大类。数组、栈、队列、链表是线性表，其他是非线性表。

1. 线性数据结构概述

线性表有数组、链表、队列、栈，它们有一个共同的特征：把同类型的数据按顺序一个接一个地串在一起。与非线性数据结构相比，线性表的存储和使用显得很简单。由于简单，很多高级操作线性表无法完成。

下面对线性表做一个概述，并比较它们的优缺点。

(1) 数组。数组是最简单的数据结构，它的逻辑结构形式和数据在物理内存上的存储形式完全一样。例如在C语言中定义一个整型数组int a[5]，系统会分配一个20字节的存储空间，而且这20字节的存储地址是连续的。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[5];
    for(int i=0;i<5;i++)
        cout<<&a[i]<<" ";        //打印5个整数的存储地址
    return 0;
}
```

在作者的计算机上运行，输出5个整数的存储地址：

```
0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4 0x6dfee8
```

数组的优点如下：

① 简单，容易理解，容易编程。

② 访问快捷，如果要定位到某个数据，只需要使用下标即可。例如a[0]是第1个数据，a[i]是第i－1个数据。虽然a[0]在物理上是第1个数据，但是在编程时有时从a[1]开始更符合逻辑。

③ 与某些应用场景直接对应。例如数列是一维数组，可以在一维数组上进行排序操作；矩阵是二维数组，表示平面的坐标；二维数组还可以用来存储图。

数组的缺点：由于数组是连续存储的，中间不能中断，这导致删除和增加数据非常麻烦和耗时。例如要删除数组int a[1000]的第5个数据，只能使用覆盖的方法，从第6个数据开始，每个往前挪一个位置，需要挪动近1000次。增加数据也麻烦，例如要在第5个位置插入一个数据，只能把原来从第5个开始的数据逐个往后挪一位，空出第5个位置给新数据，也需要挪动近1000次。

(2) 链表。链表能克服数组的缺点，链表的插入和删除操作不需要挪动数据。简单地说，链表是“用指针串起来的数组”，链表的数据不是连续存放的，而是用指针串起来的。例如，删除链表的第3个数据，只要把原来连接第3个数据的指针断开，然后连接它前后的数据即可，不用挪动任何数据的存储位置，如图3.1所示。

链表的优点：增加和删除数据很便捷。这个优点弥补了数组的缺点。

① 本书作者曾写过一句赠言：“以数据结构为弓，以算法为箭”。

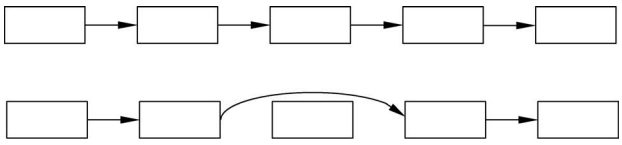


图 3.1 删除第 3 个数据

链表的缺点：定位某个数据比较麻烦。例如要输出第 500 个数据，需要从链表头开始，沿着指针一步一步走，直到第 500 个。

链表和数组的优缺点正好相反，它们的应用场合不同，数组适合静态数据，链表适合动态数据。

链表如何编程实现？在常见的数据结构教材中，链表的数据节点是动态分配的，各节点之间用指针来连接。但是在算法竞赛中，如果手写链表，一般不用动态分配，而是用静态数组来模拟[①]。当然，除非有必要，一般不手写链表，而是用系统提供的链表，例如 C++ STL 的 list。

（3）队列。队列是线性数据的一种使用方式，模拟现实世界的排队操作。例如排队购物，只能从队头离开队伍，新来的人只能排到队尾，不能插队。队列有一个出口和一个入口，出口是队头，入口是队尾。队列的编程实现可以用数组，也可以用链表。

队列这种数据结构无所谓优缺点，只有适合不适合。例如宽度优先搜索（BFS）就是基于队列的，用其他数据结构都不合适。

（4）栈。栈也是线性数据的一种使用方式，模拟现实世界的单出入口。例如一管泡腾片，先放进去的泡腾片位于最底层，最后才能拿出来。栈的编程比队列更简单，同样可以用数组或链表实现。

栈有它的使用场合，例如递归使用栈来处理函数的自我调用过程。

2. 非线性数据结构概述

（1）二叉树。二叉树是一种高度组织性、高效率的数据结构。例如在一棵有 n 个节点的满二叉树上定位某个数据，只需要走 $O(\log_2 n)$ 步就能找到这个数据；插入和删除某个数据也只需要 $O(\log_2 n)$ 次操作。不过，如果二叉树的形态没有组织好，可能退化为链表，所以维持二叉树的平衡是一个重要的问题。在二叉树的基础上发展出了很多高级数据结构和算法。大多数高级数据结构，例如树状数组、线段树、树链剖分、平衡树、动态树等，都是基于二叉树的[②]，可以说“高级数据结构≈基于二叉树的数据结构”。

（2）哈希表（Hash Table，又称为散列表）。哈希表是一种“以空间换时间”的数据结构，是一种重要的数据组织方法，用起来简单、方便，在算法竞赛中很常见。

在使用哈希表时，用一个哈希函数计算出它的哈希值，这个哈希值直接对应到空间的某个位置（在大多数情况下，这个哈希值就是存储地址），当后面需要访问这个数据时，只需要再次使用哈希函数计算出哈希值就能直接定位到它的存储位置，所以访问速度取决于哈希函数的计算量，差不多就是 O(1)。

哈希表的主要缺点：不同的数据，计算出的哈希值可能相同，从而导致冲突。所以在使用哈希表时一般需要使用一个修正方法，判断是否产生了冲突。当然，更关键的是设计一个

① 各种场景的手写链表参考：《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 3 页的“1.1.2 静态链表”。

② 本作者曾拟过一句赠言：“二叉树累并快乐着，她有一大堆孩子，都是高级数据结构”。

好的哈希函数，从根源上减少冲突的产生。设计哈希函数，一个重要的思想是“雪崩效应”，如果两个数据有一点不同，它们的哈希值就会差别很大，从而不容易冲突。

哈希表的空间效率和时间效率是矛盾的，使用的空间越大，越容易设计哈希函数。如果空间很小，再好的哈希函数也会产生冲突。在使用哈希表时，需要在空间和时间效率上取得平衡。

扫一扫

视频讲解

3.1 数组与高精度

数组是最简单的数据结构。数组的一个应用是高精度，高精度算法就是大数的计算方法。例如两个整数的计算，Java 和 Python 能直接计算任意大的数，而 C++的最大数据类型是 64 位的 long long，更大的数不能直接计算，需要用数组来模拟。例如两个 200 位的十进制数相加，定义 int a[205]和 int b[205]，a[i]和 b[i]代表整数的第 i 位，a[0]是个位，a[1]是十位，a[2]是百位，等等。

在竞赛中经常用到很大的数组。建议不要用 malloc()动态分配，因为动态分配需要多写代码而且容易出错。大数组应该定义为全局静态数组，而且不需要初始化为 0，因为全局变量在编译时会自动初始化为全 0。

```
#include<bits/stdc++.h>
using namespace std;
int a[10000000];                    //定义一个很大的全局数组，一千万个。自动初始化为 0
int main(){
    cout<<a[0];                     //输出 0
    return 0;
}
```

大数组不能定义在函数内部，可能会导致栈溢出错误。因为大多数编译器的局部变量是在用到时才分配，大小不能超过栈，而栈一般不会很大。下面这样做很可能会报错：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[10000000]={0};            //这样写大多数编译器会报错
    cout<<a[0];                     //出错
    return 0;
}
```

另外，注意全局变量和局部变量的初值。全局变量如果没有赋值，在编译时会被自动初始化为 0。在函数内部定义的局部变量，若需要初值为 0，一定要初始化为 0，否则初值不可预测。

```
#include<bits/stdc++.h>
using namespace std;
int a;                              //全局变量，如果不赋值，自动初始化为 0
int c = 999;                        //赋值为 999
int main(){
    int b;
    cout<<a<<endl;                  //输出 0
    cout<<c<<endl;                  //输出 999
    cout<<b<<endl;                  //由于 b 没有初始化，这里输出莫名奇妙的值
    return 0;
}
```

除了 C 语言的静态数组，还可以用 C++的 vector 定义大数组，见 3.2 节的介绍。

C++能表示的最大整数类型是 64 位的 long long，如果需要计算更大的数，需要使用“高精度”。在 C++ STL 中没有直接进行大数计算的库，遇到大数计算，需要自己写代码。

虽然在算法竞赛中高精度题目很罕见，但是用高精度来熟悉数组的应用是很好的练习。

基本的“加、减、乘、除”这 4 种高精度计算，做法是模拟每一位的计算，并处理进位或借位。

注意数字的读取和存储。设整数 a 有 1000 位，因为数值太大，无法直接赋值给变量，所以不能按数字读入，只能按字符读入。可以用字符串 string 读入大数 sa，然后转换为 int a[]，一个字符 sa[]存为一位数字 a[]。注意存储的顺序，在读入的时候，字符串 sa[0]是最高位，sa[n－1]是最低位；但是在计算时习惯用 a[0]表示最低位，a[n－1]表示最高位，所以需要把输入的字符串 sa 倒过来存到 a[]中。

1. 高精度加法

例 3.1　A＋B problem(高精)　https://www.luogu.com.cn/problem/P1601

问题描述：高精度加法，相当于 a＋b problem，不用考虑负数。

输入：分两行输入，a，b≤10^{500}。

输出：输出一行，代表 a＋b 的值。

输入样例：	输出样例：
22222222222222222222222222222 33333333333333333333333333333	35555555555555555555555555555

简单地模拟按位加即可，注意处理进位。

把输入的数字存到字符串中，然后用 add()求和。先把字符转换成数字，做完加法后再转换回字符。

```
#include<bits/stdc++.h>
using namespace std;
char a[1005],b[1005];                    //加数和被加数,或者定义为 char
string add(string sa,string sb){
    int lena=sa.size(),lenb=sb.size();
    for(int i=0;i<lena;i++)
        a[lena-1-i] = sa[i]-'0';         //把字符转换成数字然后翻转,使 a[0]是最低位
    for(int i=0;i<lenb;i++)
        b[lenb-1-i] = sb[i]-'0';
    int lmax = lena>lenb ? lena : lenb;
    for(int i=0;i<lmax;i++) {
        a[i] += b[i];
        a[i+1] += a[i]/10;               //处理进位
        a[i] %=10;
    }
    if(a[lmax]) lmax++;                  //若最高位相加后也有进位,数字长度加 1
    string ans;
    for(int i=lmax-1;i>=0;i--)           //把数字转换成字符,然后翻转
        ans += a[i]+'0';
    return ans;
}
int main(){
    string sa,sb;
```

```
    cin >> sa >> sb;
    cout << add(sa,sb);
    return 0;
}
```

2. 高精度减法

例 3.2　高精度减法　https://www.luogu.com.cn/problem/P2142

问题描述：给定两个整数 a、b（第二个数可能比第一个数大），求它们的差。

输入：分两行输入，$0<a,b\leqslant 10^{10086}$。

输出：输出一行，代表 a－b 的值。

输入样例：	输出样例：
5 3	2

简单地模拟按位减即可，注意处理借位。

下面是 C/C++代码。

```
#include <bits/stdc++.h>
using namespace std;
char a[11000],b[11000];                    //被减数和减数
string sub(string sa,string sb){
    if(sa == sb) return "0";               //特判：两数字相等
    bool neg = 0;                          //标记是否为负数
    if(sa.size() < sb.size() || sa.size() == sb.size() && sa < sb)
        swap(sa, sb), neg = 1;             //让 a 大于 b
    int lena = sa.size(),lenb = sb.size();
    for(int i = 0;i < lena;i++)            //把字符转换成数字然后翻转，使 a[0]是最低位
        a[lena - 1 - i] = sa[i] - '0';
    for(int i = 0;i < lenb;i++)
        b[lenb - 1 - i] = sb[i] - '0';
    int lmax = lena;
    for(int i = 0;i < lmax;i++){
        a[i] -= b[i];
        if(a[i]< 0){                       //处理借位
            a[i] += 10;
            a[i + 1] -- ;
        }
    }
    while(!a[ -- lmax] && lmax > 0) ;      //找到首位为 0 的位置，什么都不做
    lmax++;
    string ans;
    for(int i = lmax - 1;i > = 0;i -- )    //把数字转换成字符，然后翻转
        ans += a[i] + '0';
    if(neg) ans = "-" + ans;               //检查是否为负数
    return ans;
}
int main(){
    string sa,sb; cin >> sa >> sb;
    cout << sub(sa,sb);
    return 0;
}
```

3. 高精度乘法

例 3.3　A * B Problem　https://www.luogu.com.cn/problem/P1303

问题描述：给出两个非负整数，求它们的乘积。

输入：输入共两行，每行一个非负整数，不超过 10^{2000}。

输出：输出一个非负整数，表示乘积。

输入样例：	输出样例：
2 3	6

乘法模拟小学的竖式乘法操作，按位做乘法最后相加。例如 87×99，计算过程如下：

```
   87
×  99
-----
   63
  72_
  63_
+72_
-----
 8613
```

计算结果用 int c[]存储，首先计算出 c[0]＝7×9＝63，c[1]＝8×9＋7×9＝72＋63＝135，c[2]＝8×9＝72，然后处理进位，得到乘积 8613。

下面是 C/C++代码。注意第 3 行 int c[]不能定义为 char c[]，例如上面例子中 c[1]＝135，超出了 char 的范围。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int a[2005], b[2005], c[4005];          //c = a*b
4  string mul(string sa,string sb){
5      if(sa=="0"||sb=="0")  return "0";
6      int lena=sa.size(),lenb=sb.size();
7      for(int i=0;i<lena;i++)  a[lena-i]=sa[i]-'0';
8      for(int i=0;i<lenb;i++)  b[lenb-i]=sb[i]-'0';
9      for(int i=1;i<=lena;i++)
10         for(int j=1;j<=lenb;j++)
11             c[i+j-1] += a[i]*b[j];
12     for(int i=1;i<=lena+lenb;i++)
13         c[i+1]+=c[i]/10,c[i]%=10;
14     string ans;
15     if(c[lena+lenb])
16         ans += c[lena+lenb]+'0';
17     for(int i=lena+lenb-1;i>=1;i--)
18         ans += c[i]+'0';             //倒过来，把 c 的高位放在 ans 的前面
19     return ans;
20 }
21 int main(){
22     string sa,sb;
23     cin>>sa>>sb;
24     cout<<mul(sa,sb);
25     return 0;
26 }
```

4. 高精度除法

例 3.4　A/B Problem　https://www.luogu.com.cn/problem/P1480

问题描述：给出两个整数 a 和 b，求它们的商。

输入：输入共两行，第一行是被除数，第二行是除数。$0<a\leqslant 10^{5000}$，$0<b\leqslant 10^{9}$。

输出：商的整数部分。

输入样例：	输出样例：
5 3	1

本题的被除数 a 是大数，除数 b 很小，情况比较简单。下面的代码模拟了除法的计算。

另外还有一种做法，即用减法求除法。a 除以 b，转化为 a 连续减去 b，减了多少次就是商，最后不够减的是余数。

```
#include<bits/stdc++.h>
using namespace std;
long long a[10001],b,c[10001];                    //被除数、除数、商
int main(){
    string sa;
    cin>>sa>>b;                                   //读被除数、除数
    int len=sa.size();                            //被除数的位数
    for(int i=1;i<=len;i++)
        a[i]=sa[i-1]-'0';                         //将被除数一位一位放入 a 数组
    long long d=0;                                //余数
    for(int i=1;i<=len;i++){                      //被除数一位一位除以除数，若不够除，借位
        c[i]=(d*10+a[i])/b;
        d=(d*10+a[i])%b;
    }
    int lenc=1;                                   //商的位数
    while(c[lenc]==0&&lenc<len)lenc++;            //删除前导 0
    for(int i=lenc;i<=len;i++)cout<<c[i];         //输出结果
    return 0;
}
```

扫一扫

视频讲解

3.2　STL 概述

刚参加算法竞赛学习的新同学常听说 C++有一种特别神奇的工具——标准模板库（Standard Template Library，STL）。据说 STL 封装了算法竞赛用到的数据结构和算法，在做题时使用它就行了，不用管它具体如何实现，能大大降低编码的难度，在做竞赛题时无往不胜。

在很多场合下，这个说法是有道理的。例如在需要用队列时，虽然自己写代码实现队列也不难，但是如果直接用 STL queue，不用自己编码和管理队列，代码既简短又可靠，节省的时间可以去做其他题目，从而占据先机。在有些情况下必须使用 STL，例如优先队列，如果自己写代码非常麻烦，而直接使用 STL priority_queue 极为简单。所以参加 C/C++组的算法竞赛一定要学 STL，如果不用 STL，竞赛成绩会大受影响。

虽然 STL 在算法竞赛中必不可少，但是不要太夸大 STL 的作用，它并不是算法竞赛的

全部，只是一小部分。在使用 STL 之前，也需要掌握有关知识点的原理并自己手写代码实现，以了解原理、掌握思路，这些原理和思路可以帮助大家提高算法思维，并应用到算法题目中。

在介绍 C++ STL 之前，先说明 C++的版本问题。C++在 1985 年诞生，ISO(国际标准化组织)发布过多个正式标准，常见的有 C++ 98、C++ 11、C++ 14、C++ 20 等。这些标准除了规定 C++的语法、语言特性，还规定了 C++内置库的实现规范，即 C++标准库[①](C++ Standard Library)。标准库提供了可以在标准 C++中使用的各种功能，例如输入/输出、基本数据结构、内存管理、诊断、通用工具库等。掌握标准库是成为 C++程序员的基本功。

STL 是 C++标准库的一部分，包含了一些通用的数据结构和算法。STL 有很多组件[②]，在竞赛中常用的有 Strings library、迭代器(Iterators library)、容器(Containers library)、算法(Algorithms library)等。

STL 功能强大，庞大复杂，全部学习很困难，这常让算法竞赛的初学者感到困惑。其实不用担心，算法竞赛只用到 STL 的一小部分，掌握这部分即可。具体来说，数据结构有 vector、string、list、queue、stack、set、map、bitset 等，算法函数有 max、min、swap、sort、lower_bound、upper_bound、next_permutation、nth_element、unique、binary_search 等。本书将在有关章节介绍它们，读者也可以在阅读代码遇到这些概念的时候自己总结。

在代码中引用 C++的标准库时，用万能头文件即可，不用逐一列出每个头文件。

```
#include <bits/stdc++.h>      //万能头文件
```

这个万能头文件包含了每个标准库，几乎所有编译器都支持。蓝桥杯使用的编译器也支持它，所以大家不用担心。

3.2.1 String 库

在算法竞赛中，字符串处理是极为常见的考点。虽然 C 语言也有字符串操作，但是只能通过定义字符串数组，自己编码实现字符串运算，相当麻烦。用 STL 的 Strings library[③]提供的字符串处理函数，可以轻松、简便地处理字符串的计算。可以说，如果在竞赛时不用 string，成绩会大受影响。

表 3.1 列出了常用的 String 函数。

表 3.1 常用的 String 函数

函　数	功　能
length()	返回字符串的长度
size()	返回字符串的长度
push_back()	在字符串尾部添加一个字符
append()	在字符串尾部添加一个字符串
find(str,pos)	查找 str 在 pos(含)之后第一次出现的位置，若不写 pos，默认为 pos=0。如果找不到，返回 −1，注意需要强制转换为 int 型才能输出−1
substr(pos,len)	返回从 pos 位置开始截取最多 len 个字符组成的字符串，如果从 pos 开始的子串的长度不足 len，则截取这个子串

① https://zh.cppreference.com/w/cpp/standard_library

② 这个页面列出了 STL 的所有组件：https://en.cppreference.com/w/cpp

③ https://en.cppreference.com/w/cpp/string/basic_string

续表

函　数	功　能
insert(index,count,str)	在 index 处连续插入 count 次字符串 str
insert(index,str)	在 index 处插入字符串 str
erase(index,count)	将字符串从 index 位置开始(含)的 count 个字符删除,若不传参给 count,则表示删去 index 位置及以后的所有字符
replace(pos,count,str)	把从 pos 位置开始 count 个字符的子串替换为 str
replace(first,last,str)	把以 first 开始(含)、last 结束(不含)的子串替换为 str,其中 first 和 last 均为迭代器
empty()	判断字符串是否为空,如果为空返回 1,不空返回 0

下面给出例子。注意第 18 行和第 52～56 行,string 重载了符号+、==、<、>,可以直接用于计算。

特别是字符串的比较,请特别注意。两个 string 变量,按它们的字典序进行比较。例如"bcd">"abc"。容易让人误解的是两个字符串长度不一样时的情况,例如“123”和“99”,按字符串比较,"123"<"99",但按数字比较是 123>99,两种比较的结果不同。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int main(){
4  //定义、初始化、赋值
5      string s1;                                          //定义
6      string s2 = "bcd";                                  //定义并赋值
7      s2 = "efg";                                         //重新赋值
8      cout << s2 << endl;                                 //输出:efg
9      string s("abc");                                    //定义并初始化
10     string s3(s);                                       //复制
11 //长度
12     cout << s.length()<< endl;                          //或者:s.size()     输出:3
13 //遍历
14     for (int i = 0; i < s.size(); i++) cout << s[i]; cout << endl;    //输出:abc
15 //添加,合并字符串
16     s.push_back('d');cout << s << endl;          //在尾部添加字符。输出:abcd
17     s.append("efg"); cout << s << endl;          //在尾部添加字符串。输出:abcdefg
18     s = s + 'h'; s += 'i'; cout << s << endl;
19                  //用重载的 + 在尾部添加字符。输出:abcdefghi
20     s = s + "jk"; s += "lmnabc"; s = "xyz" + s; cout << s << endl;
21                  //输出:xyzabcdefghijklmnabc
22     string s4 = "uvw";
23     cout << s + s4 << endl;         //合并字符串。输出:xyzabcdefghijklmnabcuvw
24 //查找字符和字符串
25     cout <<"pos of b = " << s.find('b')<< endl;
26             //查字符第一次出现的位置。输出:pos of b = 4
27     cout <<"pos of ef = "<< s.find('ef')<< endl;
28             //查字符串第一次出现的位置。输出:pos of ef = 8
29     cout <<"pos of ab = "<< s.find('ab',5)<< endl;
30             //从 s[5]开始找字符串第一次出现的位置。输出:pos of ab = 18
31     cout <<"pos of hello = "<<(int)s.find('hello')<< endl;
32       //没找到的返回值是 - 1,需要强制转换为 int 型。输出:pos of hello = - 1
33 //截取字符串
34     cout << s.substr(3, 5)<< endl;
35             //从 s[3]开始截取 5 个字符构成的字符串。输出:abcde
36     cout << s.insert(4,"opq")<< endl;
```

```
37              //从 s[4]开始插入字符串。输出:xyzaopqbcdefghijklmnabc
38  //删除、替换
39      cout << s.erase(10,2)<< endl;
40              //从 s[10]开始删除两个字符。输出:xyzaopqbcdghijklmnabc
41      cout << s.erase(10)<< endl;
42              //从 s[10]开始删除后面的所有字符。输出:xyzaopqbcd
43      cout << s.replace(2,3,"1234")<< endl;
44              //把从 s[2]开始的 3 个字符替换为"1234"。输出:xy1234pqbcd
45      cout << s.replace(s.begin() + 7,s.begin() + 9,"5678")<< endl;
46              //把 s[7]~s[8]替换为"1234"。输出:xy1234p5678cd
47  //清理、判断
48      cout << s.empty()<< endl;           //判断是否为空,不空返回 0,空返回 1。输出:0
49      s.clear();                          //清空
50      cout << s.empty()<< endl;           //输出:1
51  //比较
52      string s5 = "abc";   string s6 = "abc"; string s7 = "bc";
53      if(s5  ==  s6) cout << " == "<< endl;
54      if(s5 < s7)   cout << "<" << endl;
55      if(s5 > s7)   cout << ">" << endl;
56      if(s5 !=  s7) cout << "!= " << endl;
57      return 0;
58  }
```

下面是一道简单字符串题。

例 3.5　烬寂海之谜　lanqiaoOJ 4050

问题描述：给定一个字符串 S,以及若干个模式串 P,统计每一个模式串在主串中出现的次数。

输入：第一行一个字符串 S,表示主串,只包含小写英文字母。第二行一个整数 n,表示有 n 个模式串。接下来 n 行,每行一个字符串,表示一个模式串 P,只包含小写英文字母。

输出：输出 n 行,每行一个整数,表示对应模式串在主串中出现的次数。

输入样例：	输出样例：
bluemooninthedarkmoon 3 moon blue dark	2 1 1

评测用例规模与约定：主串 S 的长度$\leqslant 10^5$,模式串的数量 n$\leqslant$100,模式串 P 的长度$\leqslant$1000。

由于测试数据小,可以直接暴力比较。对每个 P,逐一遍历 S 的字符,对比 P 是否出现,然后统计出现的次数。下面的代码用到 string 的 length()和 substr()函数。

```
1   #include <bits/stdc++.h>
2   using namespace std;
3   int main(){
4       string s;
5       int n;
6       cin >> s >> n;
7       while (n-- ){
```

```
8           string p;
9           cin >> p;
10          int cnt = 0;
11          for (int i = 0; i < s.length() - p.length() + 1; i++)  //遍历 S
12              if (p == s.substr(i, p.length()))                  //查询 P 是否在 S 中出现
13                  cnt++;
14          cout << cnt << endl;
15      }
16      return 0;
17  }
```

【练习题】

简单的字符串入门题[①]很多，大家请练习以下链接的题目。

洛谷的字符串入门题：https://www.luogu.com.cn/problem/list?tag=357

lanqiaoOJ 的字符串题：https://www.lanqiao.cn/problems/?first_category_id=1&tags=字符串

NewOJ 的字符串题：http://oj.ecustacm.cn/problemset.php?search=字符串

3.2.2 迭代器

在使用 STL 容器时，需要通过迭代器(iterator[②])来访问元素。迭代器变量可以看作一个指针，指向容器中某个元素所在的位置，然后做读取或写入数据的操作。

迭代器变量的定义，例如：

```
vector<int>::iterator it;        //vector 迭代器变量 it,vector 中的元素是 int 型
set<char>::iterator it;          //set 迭代器变量 it,vector 中的元素是 char 型
```

迭代器主要支持两个运算符，其用法和 C 语言的指针差不多：

(1) 自增，符号“++”，用来移动迭代器，有++it、it++两种写法，推荐使用++it，它的效率更高。注意不能把 it 加上一个偏移量，例如 it=it+5，这样是错的，因为 it 和 5 的类型不同。

(2) 解引用，单目运算符“*”，用来获取指向的元素。例如 *it 是 it 指向的元素的值。

在后面介绍各种容器的时候将举例说明迭代器如何使用。下面先用 vector 容器说明迭代器的简单应用。

第 5 行定义了迭代器 it，它指向 vector<int> a 的元素，第 6 行的 *it 是 it 当前指向元素的值。

由于迭代器的定义写起来有点麻烦，一般用 auto 关键字替代它。例如第 8 行，用 auto 替代 vector<int>::iterator，非常简洁。数组也可以用更简单的方式遍历，例如第 11、14 行。

```
1   #include<bits/stdc++.h>
2   using namespace std;
3   int main(){
```

① 字符串入门题大多逻辑简单，用杂题的思路和模拟法实现即可，适合初学者练习 string 和提高编码能力。不过，作为知识点出现的字符串算法很难。字符串算法有进制哈希、Manacher、KMP、字典树、回文树、AC 自动机、后缀树、后缀数组、后缀自动机等，都是中级和高级知识点。参考《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 549 页的“第 9 章 字符串”。

② https://en.cppreference.com/w/cpp/iterator

```
    vector<int> a{5,7,3,64,23};                    //定义和初始化一个 vector 数组
    for (vector<int>::iterator it = a.begin(); it != a.end(); ++it)
        cout << *it << " ";                        //通过迭代器访问,输出:5 7 3 64 23
    cout << endl;
    for (auto it = a.begin(); it != a.end(); ++it)  //用 auto 取代迭代器
        cout << *it << " ";                        //输出:5 7 3 64 23
    cout << endl;
    for (auto i : a)  cout << i << " ";            //直接遍历并输出:5 7 3 64 23
    cout << endl;
    int b[] = {5,7,3,64,23};                       //普通数组
    for (auto i : b)  cout << i << " ";            //直接遍历并输出:5 7 3 64 23
return 0;
}
```

3.2.3 容器概述

容器库(Containers library[①])是一些通用的类模板与算法的汇集,允许程序员简单地实现如队列、链表、栈这样的常见数据结构。

在 C++ 11 之前有两类容器:序列容器(Sequence containers)、关联容器(associative containers)。在 C++ 11 之后增加了无序关联容器(unordered associative containers)。

(1) 序列容器,实现能按顺序访问的数据结构。下面是一些序列容器。

array(C++ 11):静态数组。

vector:动态数组。蓝桥杯参赛必须掌握。

deque:双端队列。蓝桥杯参赛常用。

forward_list(C++ 11):单链表。

list:双链表。蓝桥杯参赛必须掌握。

使用序列容器,可以实现容器适配器[②]。容器适配器封装了序列容器,提供了更多的功能。常见的容器适配器如下。

stack:提供栈的功能,用 deque 实现。蓝桥杯参赛必须掌握。

queue:提供队列的功能,用 deque 实现。蓝桥杯参赛必须掌握。

priority_queue:提供优先级队列的功能,用 vector 容器存储数据,在 vector 上用堆进行计算。蓝桥杯参赛必须掌握。

(2) 关联容器,实现能快速查找的有序数据结构,计算复杂度为 $O(\log_2 n)$。下面是一些关联容器。

set:键的集合,按照键排序,键是唯一的。蓝桥杯参赛必须掌握。

map:键值对的集合,按照键排序,键是唯一的。蓝桥杯参赛必须掌握。

multiset:键的集合,按照键排序,键不是唯一的,允许多个键有等价的值。

multimap:键值对的集合,按照键排序,键不是唯一的,允许多个键有等价的值。

(3) 无序关联容器(从 C++ 11 起),提供能快速查找的无序(散列)数据结构,计算复杂度平均为 O(1),最坏情况为 O(n)。下面是一些无序关联容器。

---

① 本节内容引用自 https://en.cppreference.com/w/cpp/container

② 为帮助大家理解什么是适配器 adapter,可以用电源插头适配器打比方。如果计算机上的电源线插头是两头的,但是插座是 3 头的,那么可以用一个适配器把两头转换成 3 头。也就是说,适配器只是一个接口转换头。

unordered_set：键的集合，按照键生成散列，键是唯一的。

unordered_map：键值对的集合，按照键生成散列，键是唯一的。

unordered_multiset：键的集合，按照键生成散列，键不是唯一的，允许多个键有等价的值。

unordered_multimap：键值对的集合，按照键生成散列，键不是唯一的，允许多个键有等价的值。

本章后面几节将介绍 vector、list、queue、priority_queue、stack、set、map，这是算法竞赛常用的容器。大家一定要掌握和应用它们，如果不用它们而自己实现相关功能，编码量很大，甚至无法编码，导致参赛失败。

3.2.4 vector

在“3.1 数组与高精度”中提到，算法竞赛中使用全局静态数组既简单又方便，和静态数组差不多简单、方便的是 STL 的 vector 容器，而且它还有一些高级功能。

vector 是一个顺序容器(Sequence Container)，可以简单地认为 vector 是一个能够存放任意类型的动态数组。和普通数组一样，在 vector 的末尾添加和删除元素很快，但是在中间插入和删除元素很慢。

vector 的常见操作如表 3.2 所示。

表 3.2 vector 的常见操作

操　作	说　明
push_back()	在末尾添加一个新元素，容器的大小加 1
pop_back()	删除最后一个元素，容器的大小减 1
insert()	在指定位置的元素前插入新元素
erase()	删除一个元素或一段元素
clear()	清空 vector，容器的大小为 0
front()	访问第一个元素
back()	访问最后一个元素
begin()	返回指向第一个元素的迭代器
end()	返回指向最后一个元素的迭代器
size()	返回 vector 中元素的数量
resize()	调整容器的大小

vector 的迭代器可以使用 auto 关键字自动获取变量的类型。

下面是一些例子。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    vector<int> a;                                  //初始化一个 size 为 0 的 vector
    a.insert(a.begin(), 4, 2);                      //在 a 开始位置处插入 4 个 2
    //下面的 4 个 for 是 4 种遍历方法
    for (int i=0;i<a.size();i++)                    //通过下标访问，输出 2 2 2 2
        cout<<a[i]<<' '; cout<<"\n";
    for (vector<int>::iterator it = a.begin(); it != a.end(); ++it)
        cout<< *it<<" "; cout<<"\n";                //通过迭代器访问
    for (auto it = a.begin(); it != a.end(); ++it)  //用 auto 获得迭代器
        cout<< *it<<" "; cout<<"\n";
```

```
13      for (auto i : a)                          //直接访问
14          cout << i << " "; cout <<"\n";
15      a.insert(a.begin() + 1,9);                //在 a[1]处插入 9, a = {2 9 2 2 2}
16      a.insert(a.begin() + 2,2,5);              //在 a[2]处插入两个 5,a = {2 9 5 5 2 2 2}
17      a.resize(3);                              //改变 a 的大小。a = {2 9 5}
18      vector < int > c(a);                      //复制 a 到 c
19      vector < int > b;
20      b.insert(b.begin(), a.begin(), a.begin() + 3);              //将 a 的前 3 个数复制到 b
21      //下面演示二维 vector
22      vector < vector < char >> ch(2,vector < char >(3,'#'));     //2 行 3 列
23      for (int i = 0;i < 2;i++)                 //2 行
24          for (int j = 0;j < 3;j++)             //3 列
25              cout << ch[i][j];
26      //下面演示用 vector 存结构体
27      struct point{ int x,y;};
28      vector < point > PP;
29      for (int i = 0;i < 3;i++){                //赋值
30          point t; t.x = i; t.y = i;
31          PP.push_back(t);
32      }
33      for (int i = 0;i < PP.size();i++)         //输出
34          cout << PP[i].x << " " << PP[i].y << "\n";
35      auto it = PP.begin();                      //等同于:vector < point >::iterator it = PP.begin();
36      for (;it!= PP.end();++it) cout << it -> x <<" " << it -> y <<"\n";
37      return 0;
38  }
```

3.2.5 算法函数概述

STL 算法库[①]提供了大量的实用函数(例如查找、排序、计数等)。STL 把算法函数分为以下几类:

(1) 不修改序列的操作,有批量操作、搜索操作、折叠操作。例如搜索函数 find(first,last,value),用于搜索[first,last)区间内是否有 value。

(2) 修改序列的操作,有复制操作、交换操作、变换操作、生成操作、移除操作、顺序变更操作、采样操作。

(3) 排序和相关操作,有划分操作、排序操作、二分搜索操作(在已划分范围上)、集合操作(在已排序范围上)、归并操作(在已排序范围上)、堆操作、最大/最小操作、字典序比较操作、排列操作。

(4) 数值运算。

(5) 在未初始化内存上的操作。

下面简要介绍一些函数,它们在算法竞赛中很常用,如表 3.3 所示。

表 3.3 一些常用函数

函 数	功 能
find(a.begin(),a.end(),value)	顺序查找,value 为需要查找的值
reverse(a.begin(),a.end())	翻转数组、字符串
sort(a.begin(),a.end(),cmp)	排序,cmp 为自定义的比较函数

① https://zh.cppreference.com/w/cpp/algorithm

续表

函　数	功　能
unique(first,last)	去除容器中相邻的重复元素，所以一般需要先排序，然后用 unique 去重。返回值为指向去重后最后一个不同元素的迭代器。注意原容器大小不变，末尾的重复元素可以用 erase()删除
nth_element(a.begin(), a.begin()+mid,a.end(),cmp)	按指定范围进行分类，找出序列中第 n 大的元素，使其左边均为小于它的数，右边均为大于它的数。该函数一般用于求第 k 小的数
binary_search(a.begin(), a.end(),value)	在已经排好序的序列上做二分查找，需要先用 sort()排序。value 为需要查找的值，如果找到，返回 true，否则返回 false
lower_bound(a.begin(), a.end(),x)	在一个有序序列中进行二分查找，返回指向第一个大于或等于 x 的元素的位置。如果不存在这样的元素，则返回尾迭代器
upper_bound(a.begin(), a.end(),x)	在一个有序序列中进行二分查找，返回指向第一个大于 x 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器
next_permutation(a.begin(), a.end())	将当前排列更改为全排列中的下一个排列。如果当前排列已经是全排列中的最后一个排列，该函数返回 false，并将排列更改为全排列中的第一个排列
prev_permutation()	将当前排列更改为全排列中的上一个排列，用法与 next_permutation()相同
max()、min()	查找最大、最小值
swap()	交换两个元素的值

下面给出例子。为了方便读者理解，数列 a[]分别用 vector 和数组定义，然后用各种函数操作它。

（1）用 vector 定义数列 a，函数用迭代器访问。

```
#include<bits/stdc++.h>
using namespace std;
vector<int> a {23,5,36,4,8,7,5,23};
void out(){
    for(int i=0;i<a.size();i++)
                cout<<a[i]<<" ";
    cout<<endl;
}
int main(){
    auto it=find(a.begin(), a.end(), 5);
    if(it != a.end()) cout<<"yes"<<endl;                    //找到了.输出:yes
    else cout<<"no"<<endl;                                  //没找到
    nth_element(a.begin(),a.begin()+4,a.end());
        out();                                              //输出:5 4 5 7 8 23 36 23
    //下标为4,也就是第5个数放在正确的位置,即求的是第5小的数
    cout<<"第5小:"<<a[4]<<endl;                             //输出为第5小:8
    unique(a.begin(),a.end()); out();                       //输出:5 4 5 7 8 23 36 23
    sort(a.begin(),a.end()); out();                         //输出:4 5 5 7 8 23 23 36
    auto last = unique(a.begin(),a.end());
        out();                                              //输出:4 5 7 8 23 36 23 36
    a.erase(last,a.end());                                  //删除末尾的重复元素
    out();                                                  //输出:4 5 7 8 23 36
    it=lower_bound(a.begin(), a.end(),11);
    if(it != a.end()) cout<<*it<<endl;                      //输出:23
    it=upper_bound(a.begin(), a.end(),5);
    if(it != a.end()) cout<<*it<<endl;                      //输出:7
```

```
    if(binary_search(a.begin(), a.end(),23))
        cout <<"yes1"<< endl;                                   //输出:yes1
    else cout <<"no1"<< endl;
    reverse(a.begin(), a.end());
        out();                                                  //输出:36 23 8 7 5 4
    vector < int > b {2,5,7,4,5};
    do {
        for(int i = 1; i < 4; i++) cout << b[i] << " ";
        cout << endl;                                           //输出:5 7 4 ;  7 4 5 ;  7 5 4 ;
    } while (next_permutation(b.begin() + 1, b.begin() + 4));   //从{5,7,4}开始的全排列
    return 0;
}
```

(2) 用数组定义数列 a。

```
#include < bits/stdc++.h >
using namespace std;
int a[ ] = {23,5,36,4,8,7,5,23};
int n = 8;
void out(){
    for(int i = 0;i < n;i++)
                cout << a[i] << " ";
    cout << endl;
}
int main(){
    int k = find(a, a + n, 5) - a;
    if(k != n) cout <<"yes"<< endl;                             //找到了,输出:yes
    else cout <<"no"<< endl;                                    //没找到
    nth_element(a, a + 4, a + n); out();                        //输出:5 4 5 7 8 23 36 23
    //下标为 4,也就是第 5 个数放在正确的位置,即求的是第 5 小的数
    cout <<"第 5 小:"<< a[4] << endl;                           //输出为第 5 小:8
    sort(a, a + n); out();                                      //输出:4 5 5 7 8 23 23 36
    n = unique(a, a + n) - a;                                   //n:更新去重后 a 的元素数量
    out();                                                      //输出:4 5 7 8 23 36
    k = lower_bound(a, a + n, 11) - a;
    if(k != n) cout << a[k]<< endl;                             //输出:23
    k = upper_bound(a, a + n, 5) - a;
    if(k != n) cout << a[k]<< endl;                             //输出:7
    if(binary_search(a, a + n, 23)) cout <<"yes1"<< endl;       //输出:yes1
    else cout <<"no1"<< endl;
    reverse(a, a + n); out();                                   //输出:36 23 8 7 5 4
    int b[ ] = {2,5,7,4,5};
    do {
        for(int i = 1; i < 4; i++) cout << b[i] << " ";
        cout << ";  ";                                          //输出:5 7 4 ;  7 4 5 ;  7 5 4 ;
    } while(next_permutation(b + 1, b + 4));                    //从{5,7,4}开始的全排列
    return 0;
}
```

3.2.6 set 和 map

STL 的顺序容器有 vector、queue、list、string 等,它们能够快速地顺序访问元素。关联容器有 set、map 等。

顺序容器和关联容器的区别如下:

(1) 访问方式。关联容器中的元素按关键字来保存和访问;顺序容器中的元素按它们在容器中的位置来顺序保存和访问。

(2) 操作。关联容器不支持顺序容器的与位置相关的操作,因为关联容器中的元素是根据关键字存储的,这些操作对关联容器没有意义。关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

(3) 应用场景。关联容器支持高效的关键字查找和访问。map 中的元素是键-值(key-value)对,键起到索引的作用,值表示与索引相关联的数据。在 set 中每个元素只包含一个关键字,set 支持高效的关键字查询操作,检查一个关键字是否在 set 中。

1. set

set 容器有 4 种:①set,容器内只保存关键字;②multiset,关键字可以重复出现的 set;③unordered_set,用哈希函数组织的 set;④unordered_multiset,用哈希函数组织的 set,关键字可以重复出现。

下面是 set 的特点:

(1) 每个元素的键值都唯一,不允许两个元素有相同的键值。这使得 set 有去重的功能:在向 set 变量插入相同的数据时只会保留一个。

(2) 所有元素都会根据元素的键值自动排序,默认从小到大。如果要从大到小,需要加上 greater,例如"set < int,greater < int >> s1;"。

(3) set 中的元素不像 map 那样可以同时拥有实值(value)和键值(key),只能存储键,是单纯的键的集合。

(4) set 中元素的值不能直接被改变。

(5) set 支持大部分的 map 操作,但是 set 不支持下标操作。

set[①] 的常见操作如表 3.4 所示。

表 3.4 set 的常见操作

操 作	说 明
begin()	返回指向第一个元素的迭代器
end()	返回指向最后一个元素的后一个位置的迭代器
clear()	清除所有元素
count()	返回某个元素的个数,如果有这个元素,它的个数为 1,否则为 0
size()	返回集合中元素的数量
empty()	如果集合为空,返回 true,否则返回 false
insert()	在集合中插入元素
erase()	删除集合中的元素
find()	返回一个指向被查找到元素的迭代器,如果没有找到,返回 end()
upper_bound()	返回大于某个值元素的迭代器
lower_bound()	返回指向大于(或等于)某值的第一个元素的迭代器
max_size()	返回集合能容纳的元素的最大限值

set 在竞赛中常用于去重,把所有元素放进 set,重复的会被去掉,保留在 set 中的都是唯一的,且按从小到大排序。

下面是 set 操作的例子,用 count()判断 set 中是否有某个元素,见第 20 行和第 21 行。

① https://en.cppreference.com/w/cpp/container/set

```
#include <bits/stdc++.h>
using namespace std;
set<int> st{5,9,2,3};                    //定义 set,赋初值
void out(){                              //输出 set 的所有元素
    //set<int>::iterator it = st.begin();
    auto it = st.begin();                //与上一句的功能相同
    for( ; it != st.end(); ++it) cout << *it <<" ";
    cout << endl;
}
int main() {
    out();                               //set 的元素从小到大排序.输出:2 3 5 9
    st.insert(9);
    out();                               //重复元素被去重,输出:2 3 5 9
    auto it = st.lower_bound(6);         //返回大于或等于 6 的第一个元素的迭代器
    cout << *it << endl;                 //输出:9
    it = st.find(7);
    if(it == st.end()) cout <<"not find"<< endl;          //输出:not find
    else cout << *it << endl;                             //无输出
    st.erase(3);
    if(st.count(5)) cout <<"find 5"<< endl;               //输出:find 5
    if(st.count(7)) cout <<"find 7"<< endl;               //无输出
    st.clear();                                           //清空元素
    if(st.empty()) cout << "empty"<< endl;                //输出:empty
    return 0;
}
```

2. map

map[①] 是一种有序关联容器,它包含具有唯一键的键值对。键之间用比较函数 Compare 排序。map 的搜索、移除和插入操作的复杂度是 $O(\log_2 n)$,效率非常高。map 通常实现为红黑树。

键值对的例子,例如学生的姓名和学号,把姓名看成键,学号看成值,键值对是{姓名,学号}。当需要查某个学生的学号时,通过姓名可以查到。如果用 map 存{姓名,学号}键值对,只需要计算 O(1)次,就能通过姓名得到学号。map 的效率非常高。

表 3.5 列出 map 的常见操作,和 set 的操作差不多。

表 3.5 map 的常见操作

操　作	说　明
begin()	返回指向第一个元素的迭代器
end()	返回指向最后一个元素的后一个位置的迭代器
clear()	清除所有元素
count()	返回某个元素的个数,如果有这个元素,它的个数为 1,否则为 0
size()	返回元素的数量
empty()	如果为空,返回 true,否则返回 false
insert()	插入元素
erase()	删除的元素
find()	返回一个指向被查找到元素的迭代器,如果没有找到,返回 end()
upper_bound()	返回大于某个值元素的迭代器
lower_bound()	返回指向大于(或等于)某值的第一个元素的迭代器
max_size()	返回能容纳的元素的最大限值

① https://en.cppreference.com/w/cpp/container/map

下面是 map 操作的例子。往 map 中插入数据有两种方法，见第 15 行和第 16 行。map 有排序的功能，第 7 行遍历输出 map 的所有元素，是按键排序的。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  map<int, string> mp{{7,"tom"},{2,"Joy"},{3,"Rose"}};     //定义和赋初值
4  void out(){                                               //输出 map 的所有元素
5      //set<int>::iterator it = mp.begin();
6      auto it = mp.begin();                                 //与上一句的功能相同
7      for( ; it != mp.end(); ++it)
8          cout<<it->first<<" "<<it->second<<"; ";
9      cout<<endl;
10 }
11 int main() {
12     out();                                                //输出:2 Joy; 3 Rose; 7 tom;
13     cout<<"size = "<<mp.size()<<endl;                     //输出:size = 3
14     mp[3] = "Luo";                    //修改了 mp[3]的值。注意键是唯一的,不能改,值可以改
15     mp[5] = "Wang";
16     mp.insert(pair<int, string>(9, "Hu"));                //用 pair 定义键值对,然后插入
17     out();                                                //输出:2 Joy; 3 Luo; 5 Wang; 7 tom; 9 Hu;
18     mp.erase(5);
19     out();                                                //输出:2 Joy; 3 Luo; 7 tom; 9 Hu;
20     auto it = mp.find(3);                                 //查询
21     if(it != mp.end())
22         cout<<it->first<<" "<<it->second<<endl;           //输出:3 Luo
23     else cout<<"not find";                                //无输出
24     return 0;
25 }
```

下面给出一道例题，分别用 map 和 set 实现。

例 3.6　眼红的 Medusa　https://www.luogu.com.cn/problem/P1571

问题描述：Miss Medusa 到北京领了科技创新奖。她发现很多人都和她一样获了科技创新奖，某些人还获得了另一个奖项——特殊贡献奖。Miss Medusa 决定统计有哪些人获得了两个奖项。

输入：第一行两个整数 n、m，表示有 n 个人获得科技创新奖，m 个人获得特殊贡献奖。第二行 n 个正整数，表示获得科技创新奖的人的编号。第三行 m 个正整数，表示获得特殊贡献奖的人的编号。

输出：输出一行，为获得两个奖项的人的编号，按在科技创新奖获奖名单中的先后次序输出。

输入样例：	输出样例：
4 3 2 15 6 8 8 9 2	2 8

评测用例规模与约定：对于 60%的评测用例，$0\leqslant n,m\leqslant 1000$，获得奖项的人的编号 $<2\times10^9$；对于 100%的评测用例，$0\leqslant n,m\leqslant 10^5$，获得奖项的人的编号 $<2\times10^9$。输入数据保证第二行任意两个数不同，第三行任意两个数不同。

本题是查询 n 和 m 个数中哪些是重的。一种做法是检查 m 个数中的每一个数，如果

它在 n 个数中出现过，就说明获得了两个奖。下面分别用 map 和 set 实现。

(1) 用 map 实现。把 m 个数放进 map 中，然后遍历 map 的每个数，如果在 n 个数中有，就输出。代码的计算复杂度是多少？map 的每次操作是 $O(\log_2 m)$的，第 9～11 行的复杂度是 $O(m\log_2 m)$，第 13～15 行的复杂度是 $O(n\log_2 m)$，所以总计算复杂度是 $O(m\log_2 m+n\log_2 m)$。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  map<int,bool>mp;
4  int a[101000],b[101000];
5  int main(){
6      int n,m;
7      scanf("%d%d",&n,&m);
8            for(int i=1;i<=n;i++) scanf("%d",&a[i]);
9            for(int i=1;i<=m;i++) {
10         scanf("%d",&b[i]);
11         mp[b[i]]=true;
12     }
13     for(int i=1;i<=n;i++)
14         if(mp[a[i]])
15             printf("%d ",a[i]);   //如果出现过，直接输出
16     return 0;
17 }
```

(2) 用 set 实现。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  set<int>st;
4  int a[101000], b[101000];
5  int main() {
6      int n, m;
7          scanf("%d%d", &n, &m);
8      for(int i = 1; i<= n; i++) scanf("%d", &a[i]);
9      for(int i = 1; i<= m; i++) {
10         scanf("%d", &b[i]);
11         st.insert(b[i]);
12     }
13     for(int i = 1; i<= n; i++)
14         if(st.count(a[i]))
15             printf("%d ",a[i]);
16     return 0;
17 }
```

3.3 链　表

数组的特点是使用连续的存储空间，访问每个数组元素非常快捷、简单。但是在某些情况下，数组的这些特点变成了缺点：

(1) 需要占用连续的空间。若某个数组很大，可能没有这么大的连续空间给它用。不过这一般发生在较大的工程软件中，在竞赛中不用考虑占用的空间是否连续。例如一道题给定的存储空间是 256MB，那么定义 char a[100000000]，占用了连续的 100MB 空间，也是合法的。

（2）删除和插入的效率很低。例如删除数组中间的一个数据，需要把后面所有的数据往前挪动，以填补这个空位，产生大量的复制开销，计算量为 O(n)。中间插入数据，也同样需要挪动大量的数据。在算法竞赛中这是经常出现的考点，此时不能简单地使用数组。

数据结构“链表”能解决上述问题，它不需要把数据存储在连续的空间上，而且删除和增加数据都很方便。链表把数据元素用指针串起来，这些数据元素的存储位置可以是连续的，也可以不连续。

链表有单向链表和双向链表两种。单向链表如图 3.2 所示，指针是单向的，只能从左向右单向遍历数据。链表的头和尾比较特殊，为了方便从任何一个位置出发能遍历到整个链表，让首尾相接，尾巴 tail 的 next 指针指向头部 head 的 data。由于链表是循环的，所以任意位置都可以成为头或尾。有时应用场景比较简单，不需要循环，可以让第一个节点始终是头。

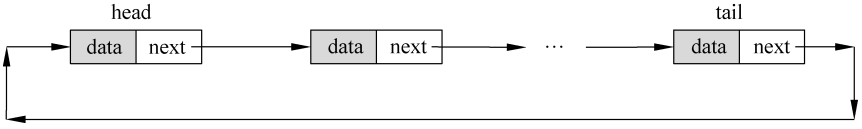


图 3.2　单向链表

双向链表是对单向链表的优化。每个数据节点有两个指针，pre 指针指向前一个节点，next 指针指向后一个节点。双向链表也可以是循环的，最后节点的 next 指针指向第一个节点，第一个节点的 pre 指针指向最后的节点，如图 3.3 所示。

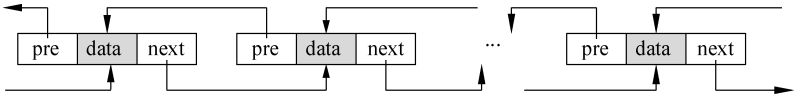


图 3.3　双向链表

在需要频繁访问前后几个节点的场合可以使用双向链表。例如删除一个节点 now 的操作，前一个节点是 pre，后一个节点是 next，那么让 pre 指向 next，now 被跳过，相当于被删除。此时需要找到 pre 和 next 节点，如果是双向链表，很容易得到 pre 和 next；如果是单向链表，不方便找到 pre。

链表的操作有初始化、遍历、插入、删除、查找、释放等。

和数组相比，链表的优点是删除和插入很快，例如删除功能，找到节点后，直接断开指向它的指针，再指向它后面的节点即可，不需要移动其他所有节点。

链表仍是一种简单的数据结构，和数组一样，它的缺点是查找慢。例如查找 data 等于某个值的节点时，需要遍历整个链表才能找到它，计算量是 O(n)。

3.3.1　手写链表

链表的编程实现有动态链表、静态链表、STL 链表等多种方法。动态链表是用 malloc()函数动态分配空间生成节点，静态链表是定义全局静态数组来模拟链表。在算法竞赛中为了加快编码速度，一般用静态链表或 STL list。

1. 静态链表

静态链表用数组来模拟链表，即“逻辑上是链表，物理上是数组”。

下面先给出单向链表的实现。

(1) 链表节点的定义。用 struct node 定义链表节点：id，表示这个节点的编号，一般用不着，因为可以用 nodes[i]的下标 i 来表示节点的编号；data，节点存储的数据；nextid，链表的指针，指向下一个节点的编号。

```
1  const int N = 10000;                                  //按需要定义静态链表的空间大小
2  struct node{                                          //单向链表
3      int id;                                           //这个节点的 id
4      int data;                                         //数据
5      int nextid;                                       //指向下一个节点的 id
6  }nodes[N];                                            //静态分配需要定义在全局
7  //为链表的 next 指针赋初值，例如：
8      nodes[0].nextid = 1;
9      for(int i = 1; i <= n; i++){
10         nodes[i].id = i;                              //把第 i 个节点的 id 赋值为 i
11         nodes[i].nextid = i + 1;                      //next 指针指向下一个节点
12     }
13 //定义为循环链表：尾指向头
14     nodes[n].nextid = 1;
15 //遍历链表，沿着 nextid 访问节点即可
16 //删除节点。设当前位于位置 now，删除这个节点。需要找到前一个节点，即 preid 指针
17     nodes[preid].nextid = nodes[now].nextid;          //跳过节点 now，即删除 now
18     now = nodes[preid].nextid;                        //更新 now
19 //插入节点，略
```

(2) 链表的初始化。因为直接用 i 来赋值 nodes[i]，第 10 行的 id 是不必要的，可以省去。重点是第 11 行的 nextid 和第 14 行的头尾相接，这样才能完成一个循环的链表结构。

(3) 遍历链表。沿着 nextid 指针访问所有节点。

(4) 删除节点。若当前节点是 now，删除它，需要先找到前一个节点 pre，然后 nodes[pre].nextid＝nodes[now].nextid。如何找到 pre，是单向链表的麻烦。

(5) 插入节点。由于是用数组来模拟链表的，当需要插入一个节点时，只能把数组末尾还没用到的新 nodes[]赋值给新节点。例如定义一个 nodes[10000]的静态链表，已经用 nodes[1]～nodes[100]构成了一个循环链表，现在需要在 nodes[1]和 nodes[2]之间插入一个新节点，那么新节点就使用 nodes[101]，然后赋值 nodes[1].nextid＝101，nodes[101].nextid＝2，这样就完成了插入。

用静态数组模拟链表的缺点是，若有频繁的删除和插入，会逐渐消耗数组空间。因为删除的节点不方便回收再使用。在插入节点时，不能使用已经被删除的节点，而是一直使用数组末尾的新 nodes[]。所以需要定义一个足够大的 nodes[N]，避免用完。

双向静态链表的实现和单向链表差不多，只是多了指向前一个节点的指针 preid。下面是部分代码。

```
1  const int N = 10000;
2  struct node{                                          //双向链表
3      int id;                                           //节点编号
4      int data;                                         //数据
5      int preid;                                        //前一个节点
6      int nextid;                                       //后一个节点
7  }nodes[N];
8  //为节点的指针赋初值
9      nodes[0].nextid = 1;
10     nodes[1].preid = 0;
```

```
    for(int i = 1; i <= n; i++){                    //建立链表
        nodes[i].id = i;
        nodes[i].preid = i - 1;                     //前节点
        nodes[i].nextid = i + 1;                    //后节点
    }
//定义为循环链表
    nodes[n].nextid = 1;                            //循环链表:尾指向头
    nodes[1].preid = n;                             //循环链表:头指向尾
```

用下面的简单题说明链表的应用。

例 3.7　小王子单链表　lanqiaoOJ 1110

问题描述：小王子有一天迷上了排队的游戏，桌子上有标号为 1～10 的 10 个玩具，现在小王子将它们排成一列，可小王子还是太小了，他不确定到底想把哪个玩具摆在哪里，直到最后才能排成一条直线，求玩具的编号。已知他排了 M 次，每次都是选取标号为 X 的放到最前面，求每次排完后玩具的编号序列。

输入：第一行是一个整数 M，表示小王子排玩具的次数。随后 M 行，每行包含一个整数 X，表示小王子要把编号为 X 的玩具放在最前面。

输出：共 M 行，第 i 行输出小王子第 i 次排完序后玩具的编号序列。

输入样例：	输出样例：
5	3 1 2 4 5 6 7 8 9 10
3	2 3 1 4 5 6 7 8 9 10
2	3 2 1 4 5 6 7 8 9 10
3	4 3 2 1 5 6 7 8 9 10
4	2 4 3 1 5 6 7 8 9 10
2	

本题是单链表的直接应用。

把 1～10 这 10 个数据存到 10 个节点上，即 toy[1]～toy[10]这 10 个节点。代码有一个小技巧：toy[0]始终是链表的头，但是不用它存数据，真正的数据在它的后面。当需要遍历链表的时候，直接从 toy[0]开始即可。

```
#include <bits/stdc++.h>
using namespace std;
struct node{
  //int id;                                   //没用到
    int data;
    int nextid;
}toy[20];
void init(){                                  //初始化链表
    toy[0].nextid = 1;                        //节点 0 是链表头,它指向下一个节点 1
    toy[10].nextid = -1;                      //最后的节点 10 指向 -1,表示没有后续节点
    for (int i = 1; i <= 10; i++) {
        toy[i].data = i;                      //节点的值
        toy[i].nextid = i + 1;                //指针,指向下一个节点
    }
}
void tohead(int x){                           //把 x 放到最前面
    int p = 0;                                //p 是 x 的前一个节点
```

```
18      while(toy[p].nextid != -1){              //遍历链表,查找 x 的前一个节点 p
19          if (toy[toy[p].nextid].data == x) break;
20          p = toy[p].nextid;
21      }
22      int now = toy[p].nextid;                 //now 是 x 所在的节点
23      toy[p].nextid = toy[now].nextid;         //删除 x 节点
24      toy[now].nextid = toy[0].nextid;
25      toy[0].nextid = now;                     //把 x 放到最前面
26  }
27  void out(){                                  //输出链表,就是题目的编号序列
28      int head = toy[0].nextid;                //toy[0]始终是链表头,并且不用来存编号
29      for (int i = 1; i <= 10; i++){
30          cout << toy[head].data << ' ';
31          head = toy[head].nextid;
32      }
33      cout << endl;
34  }
35  int main(){
36      init();
37      int m;
38      cin >> m;
39      while (m--) {
40          int x;
41          cin >> x;
42          tohead(x);
43          out();
44      }
45      return 0;
46  }
```

本节后面用 STL list 重新写这一题的代码,非常简短。

2. 极简链表

这里介绍一种极为简单的链表,虽然应用场景很少,不过也有它的优势。

定义一维数组 R[],R[i]的 i 是链表上节点 i 的值,而 R[i]的值是下一个节点,也就是说 R[i]是指向下一个节点的指针。

图 3.4 构造了一个单向链表,只要定义 R[1]=2,R[2]=3,R[3]=4,…,就完成了单向链表的构造。

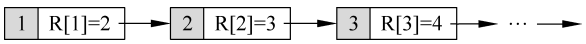


图 3.4 构造一个单向链表

添加或删除节点很简单。例如删除节点 2,只需要修改 R[1],从原来的 R[1]=2 改为 R[1]=3,这样就跳过了节点 2,如图 3.5 所示。

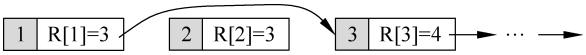


图 3.5 删除节点 2

极简链表的使用场景有限,因为一个节点 i 只能存一个数据,这个数据就等于 i。在特定场景下这种方法很有用,因为它有一个优点:可以通过 R[i]的下标 i 直接找到数据 i,而不需要像普通链表那样需要遍历整个链表才能找到 i。

下面的蓝桥杯真题巧妙地应用了极简链表的这个特点。

例 3.8　2023 年第十四届 C/C++大学 B 组试题 H：整数删除　lanqiaoOJ 3515

时间限制：1s　内存限制：256MB　本题总分：20 分

问题描述：给定一个长度为 N 的整数数列 $A_1,A_2,\cdots,A_N$，请重复以下操作 K 次。

每次选择数列中最小的整数（如果最小值不止一个，选择最靠前的），将其删除，并把与它相邻的整数加上被删除的数值。

输出 K 次操作后的序列。

输入：第一行包含两个整数 N 和 K。第二行包含 N 个整数 $A_1,A_2,A_3,\cdots,A_N$。

输出：输出 N−K 个整数，中间用一个空格隔开，代表 K 次操作后的序列。

输入样例：	输出样例：
5 3 1 4 2 8 7	17 7

评测用例规模与约定：对于 20%的评测用例，$1\leqslant K<N\leqslant 10000$；对于 100%的评测用例，$1\leqslant K<N\leqslant 5\times 10^5$，$0\leqslant A_i\leqslant 10^8$。

这道题在第 2 章用简单方法做过，能通过 20%的测试。下面的方法能通过 100%的测试。

本题的删除操作显然应该用链表。如果按第 2 章的简单方法，用数组来存数列，每次删除一个数，就需要把它后面所有的数往前挪一位，以填补它留下的空位，这样操作很耗时。如果用链表来处理数列，就不用挪动大量的数了。

另外，本题的每次操作要找到最小的数。如果遍历整个链表来找，每次查找是 O(N)的计算量，那么 K 次操作的总计算量仍然是 O(NK)，和第 2 章的简单方法的计算量差不多，仍不能通过 100%的测试。解决方案是用优先队列来找最小的数，在 N 个数中找最小值，计算量只有 $O(\log_2 N)$。优先队列的讲解见本章“3.5 优先队列”。

如何把链表和优先队列结合起来？数列分别由链表和优先队列来处理：

(1) 链表。把数列存到链表的节点上，在链表上删除最小值节点，并且更新它的邻居，也就是加上被删除的节点的值。最小值是通过优先队列找到的。

(2) 优先队列。把数列存到优先队列中，每次操作取出最小值。然后把更新后的数重新放进队列。

(3) 优先队列找到最小值后，如何在链表中定位？这是最关键的问题。如果是普通队列，那么就需要遍历链表，计算量是 O(N)，还是一样耗时，优先队列白用了，还不如直接在链表中找最小值。这里如果用极简链表，非常巧妙：用优先队列找最小值 t，t 对应的链表节点就是 R[t]，无须遍历链表，计算量是 O(1)。

总结以上思路，做一次操作的计算量是：用优先队列找最小值 t，计算量是 $O(\log_2 N)$；在链表上定位 t 的位置，计算量是 O(1)，并用链表处理删除和更新邻接，计算量是 O(1)。一次操作的计算量是 $O(\log_2 N)$，K 次操作是 $O(K\log_2 N)$，顺利通过 100%的测试。

第 5 行定义双向链表，L[]是左指针，R[]是右指针。

第 7 行删除节点 x。R[L[x]]=R[x]表示 x 的左节点指向 x 的右节点，L[R[x]]=L[x]表示 x 的右节点指向 x 的左节点，这样 x 的左、右节点就跳过了 x 节点，相当于删除了 x。

注意第 27 行优先队列的操作。优先队列的队头是最小值，但是这个最小值对应的节点

数据可能被第 31 行的 del()更新了，不再是最小值，所以需要重新放进队列，重新排序。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5e5 + 10;
long long v[N];                    //数列的值相加后可能超过 int,需要用 long long
int L[N], R[N];                    //双向链表
void del(int x) {                  //双向链表:删除 x 节点
    R[L[x]] = R[x], L[R[x]] = L[x];          //删除第 x 个节点
    v[L[x]] += v[x], v[R[x]] += v[x];        //更新左、右邻居
}
int main() {
    int n, k; cin >> n >> k;
    //优先队列, 优先队列的元素是{权值,节点下标}
    priority_queue<pair<long long, int>, vector<pair<long long, int>>,
                                    greater<pair<long long, int>>> Q;
    //输入并构造双向链表
    R[0] = 1;                                 //队头 0,右指针 R[0]指向节点 1
    L[n + 1] = n;                             //队尾 n+1,左指针 L[N+1]指向节点 n
    for (int i = 1; i <= n; i ++) {
        cin >> v[i];                          //读数列
        L[i] = i - 1, R[i] = i + 1;           //构造双向链表,第 i 个节点表示 V[i]
        Q.push({v[i], i});                    //把数列放进优先队列,求最小值
    }
    while (k--) {                             //k 次操作
        auto p = Q.top();
        //读优先队列的队头,队头是最小值.p.first 是值,p.second 是它的位置
        Q.pop();                    //弹走队头,优先队列会重新排序,新的队头仍是最小值
        if (p.first != v[p.second]){          //这个队头被 del()改过了,不一定最小
            Q.push({v[p.second], p.second}); //重新放进队列,重新排序
            k++;                              //撤销这次操作
        }
        else del(p.second);                   //删除节点并更新邻居
    }
    int t = R[0];                             //队头 0,R[0]指向第一个数
    while (t != n + 1) {                      //遍历链表
        cout << v[t] << " ";                  //输出链表元素
        t = R[t];
    }
    return 0;
}
```

3.3.2 STL list

在大多数情况下不需要手写链表，而是直接使用 C++的 STL list[①]，代码更简单。STL list 的常用操作如表 3.6 所示。

表 3.6 STL list 的常用操作

操　作	说　明
size()	链表的元素个数
empty()	链表是否为空
push_front()	在链表头添加元素

① https://zh.cppreference.com/w/cpp/container/list

续表

操　作	说　明
push_back()	在链表尾添加元素
unique()	将链表中相邻的重复元素去除,如果重复元素不相邻,则无效
sort()	升序排序
pop_back()	删除链表尾
pop_front()	删除链表头
front()	读队头
back()	读队尾
reverse()	反转链表
insert(it)	在 it 位置插入元素
erase(it)	在 it 位置删除元素

下面的代码演示了这些操作。

```
#include<bits/stdc++.h>
using namespace std;
list<int> lis = {1,2,3};                          //定义链表,初值为{1,2,3}
void out(){                                       //输出链表
    for (auto it = lis.begin(); it != lis.end(); ++it)   //遍历链表
        cout << *it << " ";
    cout << "\n";
}
int main(){
    list<int> list2(lis);                         //复制链表
    cout << lis.size()<<"\n";                     //链表的元素个数,输出:3
    if(!lis.empty()) cout <<"no empty"<<"\n";     //输出:no empty
    lis.push_front(9); out();                     //在链表头添加元素,输出:9 1 2 3
    lis.push_back(9); out();                      //末尾添加元素,输出:9 1 2 3 9
    lis.unique(); out();                          //相邻重复元素去重,输出:9 1 2 3 9
    lis.sort(); out();                            //升序排序,输出:1 2 3 9 9
    lis.unique(); out();                          //去重,此时有效,输出:1 2 3 9
    lis.pop_back(); out();                        //删除队尾,输出:1 2 3
    lis.pop_front(); out();                       //删除队头,输出:2 3
    cout << lis.front()<<"\n";                    //读队头,输出:2
    cout << lis.back() <<"\n";                    //读队尾,输出:3
    lis.reverse(); out();                         //反转,输出:3 2
    auto it = lis.begin();
    ++it;  //注意,像 it = it + 5 这样是错误的,因为 it 和 5 的类型不同
    lis.insert(it, 5); out();                     //在 it 位置插入,输出:3 5 2
    ++it;
    if(it == lis.end()) cout << "end" << "\n";    //输出:end
    --it;                                         //如果 it 是 end,不能 erase
    lis.erase(it); out();                         //删除 it 位置的元素,输出:3 5
    return 0;
}
```

用 STL list 写代码很简短。例如本节例题“小王子单链表 lanqiaoOJ 1110”,下面用 STL list 实现。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    list<int> toy{1,2,3,4,5,6,7,8,9,10};          //定义链表
```

```
    int m; cin >> m;
    while (m -- ) {
        int x; cin >> x;
        toy.remove(x);                          //删除链表中的 x
        toy.push_front(x);                      //把 x 插到链表头
        for(auto i:toy) cout << i <<" ";        //输出链表
        cout << endl;
    }
    return 0;
}
```

再看一道例题。

例 3.9　重新排队　lanqiaoOJ 3255

问题描述：给定按从小到大的顺序排列的数字 1 到 n，随后对它们进行 m 次操作，每次将一个数字 x 移动到数字 y 之前或之后。请输出完成这 m 次操作后它们的顺序。

输入：第一行为两个数字 n、m，表示初始状态为 1 到 n 从小到大排列，后续有 m 次操作。第二行到第 m+1 行，每行 3 个数 x、y、z。当 z=0 时，将 x 移动到 y 之后；当 z=1 时，将 x 移动到 y 之前。

输出：输出一行，包含 n 个数字，中间用空格隔开，表示 m 次操作完成后的排列顺序。

输入样例：	输出样例：
5 3 3 1 0 5 2 1 2 1 1	2 1 3 5 4

题目简单，下面是代码。

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, m;
    cin >> n >> m;
    list < int > lis;
    for (int i = 1; i <= n; i++)
        lis.push_back(i);                                   //链表赋值
    while(m -- ){
        int x, y, z;
        cin >> x >> y >> z;
        lis.remove(x);                                      //删除 x
        list < int >::iterator it = find(lis.begin(), lis.end(), y);   //找到 y
            //上一句可以把 list < int >::iterator it 改为 auto it
        if (z == 0) lis.insert(++it, x);                    //x 放在 y 后面
        if (z == 1) lis.insert(it, x);                      //x 放在 y 前面
    }
    for (int x : lis)
        cout << x << " ";
    cout << endl;
    return 0;
}
```

【练习题】

lanqiaoOJ：约瑟夫环 1111、小王子双链表 1112，以及种瓜得瓜，种豆得豆 3150。

洛谷：单向链表 B3631、队列安排 P1160。

扫一扫

视频讲解

3.4 队　列

队列(queue)也是一种简单的数据结构。普通队列的数据存取方式是“先进先出”，只能往队尾插入数据、从队头移出数据。队列在生活中的原型就是排队，例如在网红店排队买奶茶，排在队头的人先买到奶茶然后离开，后来的人排到队尾。

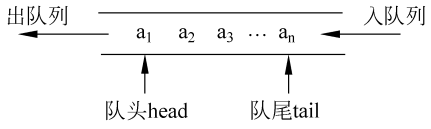


图 3.6　队列

图 3.6 是队列的原理，队头 head 指向队列中的第一个元素 a_1，队尾 tail 指向队列中的最后一个元素 a_n。元素只能从队头方向出去，并且只能从队尾进入队列。

3.4.1 手写队列

队列的代码很容易实现。如果使用环境简单，最简单的手写队列代码用数组实现。

```
const int N = 10005;                    //定义队列容量,确保够用
int que[N];                             //队列,用数组模拟
int head = 0;                           //head 始终指向队头.que[head]是队头
                                        //开始时队列为空,head = 0
int tail = -1;                          //tail 始终指向队尾.que[tail]是队尾
                                        //开始时队列为空,tail = -1
                                        //队列长度等于 tail - head + 1
head++;                                 //弹出队头元素,让 head 指向新队头
                                        //注意保持 head <= tail
que[head];                              //读队头
que[++tail] = data;                     //入队:先 tail 加 1,然后数据 data 入队
                                        //注意 tail 必须小于 N
```

把它封装到结构体里更整洁一些：

```
const int N = 10005;                    //定义队列容量,确保够用
struct myqueue{
    int a[N];                           //队列,用数组模拟
    int head = 0;
        //head 始终指向队头。que[head]是队头。开始时队列为空,head = 0
    int tail = -1;
        //tail 始终指向队尾。que[tail]是队尾。开始时队列为空,tail = -1
    int size(){return tail - head + 1;}     //队列长度等于 tail - head + 1
    void push(int data){a[++tail] = data;}
          //入队:先 tail 加 1,然后数据 data 入队
    int front(){return a[head];}            //读队头
    void pop(){head++;}
        //弹出队头元素,让 head 指向新队头。注意保持 head <= tail
};
```

这个手写代码有一个缺陷：如果进入队列的数据太多，使得 tail 超过了 N，数组 a[N]就会溢出，导致出错。

用下面的例子给出上述手写代码的应用。

例 3.10　约瑟夫问题　https://www.luogu.com.cn/problem/P1996

问题描述：n 个人，编号为 1～n，按顺序围成一圈，从第一个人开始报数，数到 m 的人出列，再由下一个人重新从 1 开始报数，数到 m 的人再出圈，以此类推，直到所有的人都出圈，请依次输出出圈人的编号。

输入：两个整数 n 和 m，1≤n，m≤100。

输出：n 个整数，按顺序输出每个出圈人的编号。

输入样例：	输出样例：
10 3	3 6 9 2 7 1 8 5 10 4

约瑟夫问题是一个经典问题，可以用队列、链表等数据结构实现。下面的代码用队列来模拟报数。方法是反复排队，从队头出去，然后重新排到队尾，每一轮数到 m 的人离开队伍。第 22 行把队头移到队尾，第 23 行让数到 m 的人离开。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  const int N = 10005;
4  struct myqueue{
5      int a[N];
6      int head;
7      int tail;
8      void init(){head=0, tail=-1;}
9      int size(){return tail-head+1;}
10     void push(int data){a[++tail]=data;}
11     int front(){return a[head];}
12     void pop(){head++;}
13 };
14 myqueue que;
15 int main(){
16     int n,m;
17     cin>>n>>m;
18     que.init();
19     for(int i=1;i<=n;i++) que.push(i);
20     while(que.size()!=0){
21         for(int i=1;i<m;i++){
22             que.push(que.front());
23             que.pop();
24         }
25         cout<<que.front()<<" ";
26         que.pop();
27     }
28     cout<<endl;
29     return 0;
30 }
```

代码第 3 行定义了队列的容量 N=10005。本题的 n 最大是 100，每人出圈一次，所以队列长度一定不超过 100×100。如果把 N 设置小了，例如 N=2000，提交到 OJ 会返回 RE，即 Runtime Error，说明溢出了。

如果要防止溢出，可以使用循环队列[①]。

队列是一种线性数据结构，线性数据结构的主要缺点是查找较慢。如果要在队列中查找某个元素，只能从头到尾一个一个查找。

3.4.2 STL queue

在竞赛时一般不自己手写队列，而是用 STL queue[②]，不用自己管理队列，也没有溢出的问题，大大加快了做题速度。STL queue 的主要操作如表 3.7 所示。

表 3.7 STL queue 的主要操作

操　作	说　明
queue＜Type＞q;	定义队列，Type 为数据类型，如 int、float、char 等
push()	在队列尾部插入一个元素
front()	返回队首元素，但不会删除
pop()	删除队首元素
back()	返回队尾元素
size()	返回元素的个数
empty()	检查队列是否为空

下面是“洛谷 P1996”的 STL queue 实现，代码很简短。

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin >> n >> m;
    queue < int > q;
    for(int i = 1;i < = n;i++) q.push(i);
    while(!q.empty()){
        for(int i = 1;i < m;i++){
            q.push(q.front());                    //读队头，重新排到队尾
            q.pop();                              //弹走队头
        }
        cout << q.front() << " ";
        q.pop();                                  //第 m 个人离开队伍
    }
    cout << endl;
    return 0;
}
```

下面用一道例题说明 queue 的应用。

例 3.11　机器翻译　lanqiaoOJ 511

问题描述：小晨的计算机上安装了一个机器翻译软件，他经常用这个软件来翻译英语文章。

① 手写循环队列的代码见《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 8 页的“1.2.2 手写循环队列”。

② C++ STL queue 的官方文档：https://cplusplus.com/reference/queue/queue/

这个翻译软件的原理很简单，它只是从头到尾，依次将每个英文单词用对应的中文含义来替换。对于每个英文单词，软件会先在内存中查找这个单词的中文含义，如果内存中有，软件就会用它进行翻译；如果内存中没有，软件就会在外存中的词典内查找，查出单词的中文含义然后翻译，并将这个单词和译义放入内存，以备后续的查找和翻译。

假设内存中有M个单元，每个单元能存放一个单词和译义。在软件将一个新单词存入内存前，如果当前内存中已存入的单词数不超过M－1，软件会将新单词存入一个未使用的内存单元；若内存中已存入M个单词，软件会清空最早进入内存的单词，腾出单元来存放新单词。

假设一篇英语文章的长度为N个单词。给定这篇待译文章，翻译软件需要去外存查找多少次词典？假设在翻译开始前内存中没有任何单词。

输入：输入共两行，每行中两个数之间用一个空格隔开。第一行为两个正整数M和N，代表内存容量和文章的长度。第二行为N个非负整数，按照文章的顺序，每个数(大小不超过1000)代表一个英文单词。文章中的两个单词是同一个单词，当且仅当它们对应的非负整数相同。其中，0＜M≤100，0＜N≤1000。

输出：输出一行，包含一个整数，为软件需要查词典的次数。

输入样例：	输出样例：
3 7 1 2 1 5 4 4 1	5

用一个哈希表hashtable[]模拟内存，若hashtable[x]＝true，表示x在内存中，否则表示不在内存中。用队列queue对输入的单词排队，当内存超过M时删除队头的单词。下面是代码。

```
#include<bits/stdc++.h>
using namespace std;
bool hashtable[1003];                          //全局数组，自动初始化为0
int main(){
    int m,n;
    cin>>m>>n;
    int ans=0;                                 //函数内部变量，一定要初始化为0
    queue<int> q;
    for(int i=0;i<n;i++) {
        int x; cin>>x;
        if(hashtable[x] == false) {
            hashtable[x] = true;
            if(q.size()<m) q.push(x);
            else {
                hashtable[q.front()] = false;
                q.pop();
                q.push(x);
            }
            ans++;                             //内存中没有这个单词，到外存中找，答案加1
        }
    }
    cout<<ans<<'\n';
    return 0;
}
```

【练习题】

lanqiaoOJ：餐厅排队 3745、小桥的神秘礼物盒 3746、CLZ 银行问题 1113、繁忙的精神疗养院 3747。

扫一扫

视频讲解

3.5 优先队列

前一节的普通队列，特征是只能从队头、队尾进出，不能在中间插队或出队。

本节的优先队列不是一种“正常”的队列。在优先队列中，所有元素有一个“优先级”，一般用元素的数值作为它的优先级，或者越小越优先，或者越大越优先。让队头始终是队列内所有元素的最值(最大值或最小值)。队头弹走之后，新的队头仍保持为队列中的最值。举个例子：一个房间，有很多人进来；规定每次出来一个人，要求这个人是房间中最高的那个；如果某人刚进去，发现自己是房间里面最高的，就不用等待，能立刻出去。

优先队列的一个简单应用是排序：以最大优先队列为例，先让所有元素进入队列，然后再一个一个弹出，弹出的顺序就是从大到小排序。优先队列更常见的应用是动态的，进出同时发生：一边进队列，一边出队列。

如何实现优先队列？先试一下最简单的方法。以最大优先队列为例，如果简单地用数组存放这些元素，设数组中有 n 个元素，那么其中的最大值是队头，要找到它，需要逐一在数组中找，计算量是 n 次比较。这样是很慢的，例如有 n＝100 万个元素，就得比较 100 万次。把这里的 n 次比较的计算量记为 O(n)。

优先队列有没有更好的实现方法？常见的高效方法是使用二叉堆这种数据结构[①]。它非常快，每次弹出最大值队头，只需要计算 $O(\log_2 n)$次。例如 n＝100 万的优先队列，取出最大值只需要计算 $\log_2(100\text{万})=20$ 次。

在竞赛中，一般不用自己写二叉堆来实现优先队列，而是直接使用 STL priority_queue，初学者只需要学会如何使用它即可。

STL 优先队列 priority_queue[②] 用堆来实现，堆是基于二叉树的一种数据结构。

定义：priority_queue＜Type，Container，Functional＞

Type 是数据类型，Container 是容器类型(默认是 vector)，Functional 是比较的方式，当需要用自定义的数据类型时才需要传入这 3 个参数，在使用基本数据类型时，只需要传入数据类型，默认是大顶堆。

priority_queue 的常见操作如表 3.8 所示。

表 3.8　priority_queue 的常见操作

操　作	说　明
priority_queue＜Type，Container，Functional＞	定义队列
pq. push()	入队
pop()	出队

① 《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 27 页的“1.5 堆”。

② https://en. cppreference. com/w/cpp/container/priority_queue

续表

操　　作	说　　明
size()	返回当前队列中元素的个数
top()	返回队首元素
empty()	判断是否为空(空返回 1,非空返回 0)

下面是例子。第 5 行定义了一个队首是最大值的优先队列,第 10 行的输出如下:

```
27 - wuhan   21 - shanghai   11 - beijing
```

第 13 行定义了一个队首是最小值的优先队列,第 19 行的输出如下:

```
11 - beijing   21 - shanghai   27 - wuhan
```

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  int main(){
4      priority_queue<pair<int, string>> pq;    //队首是最大值
5      pair<int,string> a(11,"beijing"), b(21,"shanghai"), c(27,"wuhan");
6      pq.push(a);
7      pq.push(b);
8      pq.push(c);
9          while (!pq.empty()){
10         cout << pq.top().first <<" - "<< pq.top().second <<"\n";
11         pq.pop();
12     }
13     priority_queue<pair<int,string>,vector<pair<int,string>>, \
14           greater<pair<int,string>>> pq2;      //队首是最小值
15     pq2.push(a);
16     pq2.push(b);
17     pq2.push(c);
18     while (!pq2.empty()) {
19         cout << pq2.top().first <<" - "<< pq2.top().second <<"\n";
20         pq2.pop();
21     }
22     return 0;
23 }
```

用下面的例题介绍优先队列的应用。

例 3.12　丑数　http://oj.ecustacm.cn/problem.php?id=1721

问题描述:给定素数集合 $S=\{p_1,p_2,\cdots,p_k\}$,丑数是指一个正整数满足所有质因数都出现在 S 中,1 默认是第 1 个丑数。例如 $S=\{2,3,5\}$时,此时前 20 个丑数为 1、2、3、4、5、6、8、9、10、12、15、16、18、20、24、25、27、30、32、36。现在 $S=\{3,7,17,29,53\}$,求第 20220 个丑数是多少?

这是一道填空题,下面直接给出代码。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  typedef long long ll;
4  int prime[] = {3,7,17,29,53};
5  int main(){
6      set<ll> s;                              //判重
7      s.insert(1);                            //第一个丑数是 1
```

```
    priority_queue<ll, vector<ll>, greater<ll>> q;    //队列中是新生成的丑数
    q.push(1);                                        //第一个丑数是1,进入队列
    int n = 20220;
    ll ans;
    for(int i = 1; i<=n; i++){                        //从队列中由小到大取出20220个丑数
        ll now = q.top();
        q.pop();                          //把队列中最小的取出来,它也是已经取出的最大的
        ans = now;
        for(int i = 0; i<5; i++){                     //5个素数
            ll tmp = now * prime[i];                  //从已取出的最大值开始乘以5个素数
            if(!s.count(tmp)) {                       //tmp这个数没有出现过
                s.insert(tmp);                        //放到set里面
                q.push(tmp);                          //把tmp放进队列
            }
        }
    }
    cout<<ans<<endl;
    return 0;
}
```

再看一道例题。

例 3.13　分牌　http://oj.ecustacm.cn/problem.php?id=1788

问题描述：有 n 张牌，每张牌上有一个数字 a[i]，现在需要将这 n 张牌尽可能地分给更多的人。每个人需要被分到 k 张牌，并且每个人被分到手的牌中不能有相同数字，同时需要保证可以尽可能地分给更多的人，输出任意的一种分法即可。

输入：输入第一行为正整数 n 和 k，1≤k≤n≤1000000。第二行包含 n 个整数 a[i]，1≤a[i]≤1000000。

输出：输出 m 行，m 为可以分给的人数，数据保证 m 大于或等于 1。第 i 行输出第 i 个人手中牌的数字。输出任意一个解即可。

输入样例：	输出样例：
样例 1：	样例 1：
6 3	1 2 4
1 2 1 2 3 4	1 2 3
样例 2：	样例 2：
14 3	6 1 3
3 4 1 1 1 2 3 1 2 1 1 5 6 7	2 4 1
	5 1 2
	1 3 7

题意是有 n 个数字，其中有重复数字，把 n 个数字分成多份，每份 k 个数字，问最多能分成多少份。

很显然这道题用“隔板法”。用板子隔成 m 个空间，每个空间有 k 个位置。把 n 个数按数量排序，先把数量最多的数，每个隔板内放一个；再把数量第 2 多的，每个隔板内放一个；依次类推，直到放完所有的数。由于每个数在每个空间内只放一个，所以每个空间内不会有重复的数。

例如 n=10,k=3,这 10 个数是{5,5,5,5,2,2,2,4,4,7},按数量从多到少排好了序。用隔板隔出 4 个空间[][][][]。

先放 5:[5][5][5][5]

再放 2:[5,2][5,2][5,2][5]

再放 4:[5,2,4][5,2,4][5,2][5]

再放 7:[5,2,4][5,2,4][5,2,7][5]

结束,答案是{5,2,4}、{5,2,4}、{5,2,7}。

如何编码?下面用优先队列编程。第 7 行用二元组 pair<int,int>表示每个数的数量和数字。优先队列 priority_queue 会把每个数按数量从多到少拿出来,相当于按数量多少排序。

代码的执行步骤:把所有数放进队列;每次拿出 k 个不同的数并输出,直到结束。

代码的计算复杂度:进出一次队列是 $O(\log_2 n)$,共 n 个数,总复杂度为 $O(n\log_2 n)$。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  const int N = 1e6 + 10;
4  int num[N];
5  int main(){
6      int n, k; cin >> n >> k;
7      priority_queue<pair<int, int>> q;
8      for (int i = 0; i < n; i++) {
9          int x; cin >> x;
10         num[x]++;                          //x 这个数有 num[x]个
11     }
12     for (int i = 0; i < N; ++)
13         if(num[i] > 0)
14             q.push({num[i], i});           //数 i 的个数,数 i
15     while(q.size() >= k) {                 //队列中数量大于 k,说明还够用
16         vector<pair<int, int>> tmp(k);
17         for (int i = 0; i < k; i++){       //拿 k 个数出来,且这 k 个数不同,这是一份
18             tmp[i] = q.top();              //先出来的是 num[]最大的
19             q.pop();
20         }
21         for (int i = 0; i < k; i++) {      //打印一份,共 k 个数
22             cout << tmp[i].second << " \n"[i == k - 1];
23             tmp[i].first -= 1;             //这个数用了一次,减 1
24             if(tmp[i].first > 0) q.push(tmp[i]);  //没用完,再次进队列
25         }
26     }
27     return 0;
28 }
```

【练习题】

lanqiaoOJ:小蓝的神奇复印机 3749、Windows 的消息队列 3886、小蓝的智慧拼图购物 3744、餐厅就餐 4348。

3.6 栈

扫一扫 视频讲解

栈(stack)是比队列更简单的数据结构,它的特点是“先进后出”。

队列有两个口,一个入口和一个出口。栈只有唯一的一个口,既从这个口进入,又从这

个口出来。栈像一个只有一个门的房子，而队列这个房子既有前门又有后门。所以如果自己写栈的代码，比写队列的代码更简单。

栈在编程中有基础的应用，例如常用的递归，在系统中是用栈来保存现场的。栈需要用空间存储，如果栈的深度太大，或者存进栈的数组太大，那么总数会超过系统为栈分配的空间，就会爆栈导致栈溢出。不过，算法竞赛一般不会出现这么大的栈。

栈的常见操作如下。

empty()：返回栈是否为空。

size()：返回栈的长度。

top()：查看栈顶元素。

push()：进栈，向栈顶添加元素。

pop()：出栈，删除栈顶元素。

栈的这些操作的计算量都是 O(1)，效率很高。

3.6.1 手写栈

如果使用环境简单，最简单的手写栈代码用数组实现。

```
1  const int N = 300008;                          //定义栈的大小
2  struct mystack{
3      int a[N];                                  //存放栈元素，从 a[0]开始
4      int t = -1;                                //栈顶位置，初始栈为空，置初值为 -1
5      void push(int data){a[++t] = data;}        //把元素 data 送入栈
6      int top() {return a[t];}                   //读栈顶元素，不弹出
7      void pop() {if(t>-1) t--;}                 //弹出栈顶
8      int size() {return t+1;}                   //栈内元素的数量
9      int empty() {return t== -1 ? 1:0;}         //若栈为空，返回 1
10 };
```

在使用栈时要注意不能超过栈的空间。上面第 1 行定义了栈的大小 N，进栈的元素数量不要超过它。

用下面的例子给出上述手写代码的应用。

例 3.14　表达式括号匹配　https://www.luogu.com.cn/problem/P1739

问题描述：假设一个表达式由英文字母（小写）、运算符（+、-、*、/）和左右小（圆）括号构成，以@作为表达式的结束符。请编写一个程序检查表达式中的左右圆括号是否匹配，若匹配，输出 YES，否则输出 NO。表达式的长度小于 255，左圆括号少于 20 个。

输入：一行，表达式。

输出：一行，YES 或 NO。

输入样例：	输出样例：
2*(x+y)/(1-x)@	YES

合法的括号串例如“(())”和“()()()”，像“)(()”这样是非法的。合法括号组合的特点是左括号先出现，右括号后出现；左括号和右括号一样多。

括号组合的合法检查是栈的经典应用。用一个栈存储所有的左括号。遍历字符串的每一个字符，处理流程如下。

（1）若字符是'('，进栈。

（2）若字符是')'，有两种情况：如果栈不空，说明有一个匹配的左括号，弹出这个左括号，然后继续读下一个字符；如果栈空了，说明没有与右括号匹配的左括号，字符串非法，输出 NO，程序退出。

（3）在读完所有字符后，如果栈为空，说明每个左括号有匹配的右括号，输出 YES，否则输出 NO。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 300008;                          //定义栈的大小
struct mystack{
    int a[N];                                  //存放栈元素，从 a[0]开始
    int t = -1;                                //栈顶位置，初始栈为空，置初值为 -1
    void push(int data){a[++t] = data;}        //把元素 data 送入栈
    int top() {return a[t];}                   //读栈顶元素，不弹出
    void pop() {if(t>-1) t--;}                 //弹出栈顶
    int size() {return t+1;}                   //栈内元素的数量
    int empty() {return t== -1 ? 1:0;}         //若栈为空，返回 1
}st;
int main(){
    char x;
    while(cin>>x){                             //循环输入
        if(x=='@') break;                      //输入为@停止
        if(x=='(') st.push(x);                 //左括号入栈
        if(x==')'){                            //遇到一个右括号
            if(st.empty()) {                   //栈空，没有左括号与右括号匹配
                cout<<"NO";
                return 0;
            }
            else st.pop();                     //匹配到一个左括号，出栈
        }
    }
    if(st.empty())                             //栈为空，所有左括号已经匹配到右括号，输出 YES
        cout<<"YES";
    else cout<<"NO";
    return 0;
}
```

3.6.2 STL stack

在竞赛时一般不自己手写栈，而是用 STL stack[①]。

STL stack 的主要操作如表 3.9 所示。

表 3.9 STL stack 的主要操作

操　作	说　明
stack<Type>s;	定义栈，Type 为数据类型，如 int、float、char 等
push(item)	把 item 放到栈顶
top()	返回栈顶的元素，但不会删除
pop()	删除栈顶的元素，但不会返回

① https://www.cplusplus.com/reference/stack/stack/

续表

操　作	说　明
size()	返回栈中元素的个数
empty()	检查栈是否为空，如果为空返回 true，否则返回 false

用下面的例题说明 STL stack 的应用。

例 3.15　排列　http://oj.ecustacm.cn/problem.php?id=1734

问题描述：给定一个 1～n 的排列，每个<i,j>对的价值是 j－i＋1，计算所有满足以下条件的<i,j>对的总价值。(1)1≤i<j≤n；(2)a[i]～a[j]的数字均小于 min(a[i],a[j])；(3)a[i]～a[j]不存在其他数字则直接满足。

输入：第一行包含正整数 N(N≤300000)，第二行包含 N 个正整数，表示一个 1～N 的排列 a。

输出：一个正整数，表示答案。

输入样例：	输出样例：
7 4 3 1 2 5 6 7	24

把符合条件的一对<i,j>称为一个“凹”。首先模拟检查“凹”，了解执行的过程。以“3 1 2 5”为例，其中的“凹”有“3-1-2”和“3-1-2-5”，以及相邻的“3-1”“1-2”“2-5”。一共有 5 个“凹”，总价值为 13。

像“3-1-2”和“3-1-2-5”这样的“凹”，需要检查连续 3 个以上的数字。

例如“3-1-2”，从“3”开始，下一个应该比“3”小，如“1”，再后面的数字比“1”大，才能形成“凹”。

再例如“3-1-2-5”，前面的“3-1-2”已经是“凹”了，最后的“5”会形成新的“凹”，条件是这个“5”必须比中间的“1-2”大才行。

总结上述过程是先检查“3”；再检查“1”符合“凹”；再检查“2”，比前面的“1”大，符合“凹”；再检查“5”，比前面的“2”大，符合“凹”。

以上是检查一个“凹”的两头，还有一种是“嵌套”。一旦遇到比前面小的数字，那么以这个数字为头，可能形成新的“凹”。例如“6 4 2 8”，其中的“6-4-2-8”是“凹”，内部的“4-2-8”也是“凹”。如果大家学过递归、栈，就会发现这是嵌套，所以本题用栈来做很合适。

以“6 4 2 8”为例，用栈模拟找“凹”。当新的数比栈顶的数小，就进栈；如果比栈顶的数大，就出栈，此时找到了一个“凹”并计算价值。图 3.7 中的圆圈数字是数在数组中的下标位置，用于计算题目要求的价值。

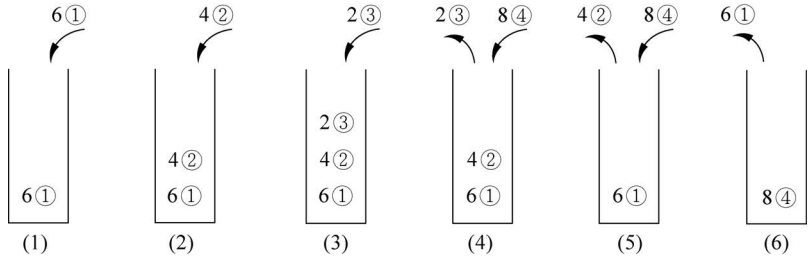


图 3.7　用栈统计凹

图(1)：6 进栈。

图(2)：4 准备进栈，发现比栈顶的 6 小，说明可以形成凹，4 进栈。

图(3)：2 准备进栈，发现比栈顶的 4 小，说明可以形成凹，2 进栈。

图(4)：8 准备进栈，发现比栈顶的 2 大，这是一个凹“4-2-8”，对应下标“②--④”，弹出 2，然后计算价值，j－i＋1＝④－②＋1＝3。

图(5)：8 准备进栈，发现比栈顶的 4 大，这是一个凹“6-4-8”，对应下标“①--④”，也就是原数列的“6-4-2-8”。弹出 4，然后计算价值，j－i＋1＝④－①＋1＝4。

图(6)：8 终于进栈了，数字也处理完了，结束。

在上述过程中，只计算了长度大于或等于 3 的凹，没有计算题目中“(3)a[i]～a[j]不存在其他数字”的长度为 2 的凹，所以最后统一加上这种情况的价值(n－1)×2＝6。

最后统计得“6 4 2 8”的总价值是 3＋4＋6＝13。

下面是代码。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 300008;
int a[N];                                          //这里 a[]是题目的数字排列
int main(){
    int n;cin>>n;
    for(int i = 1;i<=n;i++)cin>>a[i];              //输入数列
    stack<int> st;                                 //定义栈
    long long ans = 0;
    for(int i = 1;i<=n;i++){
        while(!st.empty() && a[st.top()]<a[i]){
            st.pop();
            if(!st.empty()){
                int last = st.top();
                ans += (i - last + 1);
            }
        }
        st.push(i);
    }
    ans += (n - 1) * 2;                            //(3)a[i]～a[j]不存在其他数字的情况
    cout<<ans;
}
```

【练习题】

lanqiaoOJ：妮妮的神秘宝箱 3743、直方图的最大建筑面积 4515、小蓝的括号串 2490、校邋遢的衣橱 1229。

洛谷：小鱼的数字游戏 P1427、后缀表达式 P1449、栈 P1044、栈 B3614、日志分析 P1165。

3.7 二 叉 树

在前几节介绍的数据结构中，数组、队列、栈和链表都是线性的，它们存储数据的方式是把相同类型的数据按顺序一个接一个地串在一起。线性表形态简单，难以实现高效率的操作。

二叉树是一种层次化的、高度组织性的数据结构。二叉树的形态使得它有天然的优势，在二叉树上做查询、插入、删除、修改、区间等操作极为高效，基于二叉树的算法也很容易实现高效率的计算。

3.7.1 二叉树的概念

二叉树的每个节点最多有两个子节点，分别称为左孩子、右孩子，以它们为根的子树称为左子树、右子树。二叉树的每一层以 2 的倍数递增，所以二叉树的第 k 层最多有 2^{k-1} 个节点。根据每一层的节点分布情况，有以下常见的二叉树。

(1) 满二叉树。其特征是每一层的节点数都是满的。第一层只有一个节点，编号为 1；第二层有两个节点，编号为 2、3；第三层有 4 个节点，编号为 4、5、6、7；……；第 k 层有 2^{k-1} 个节点，编号为 2^{k-1}，$2^{k-1}+1$，…，2^k-1。

一棵 n 层的满二叉树，一共有 $1+2+4+\cdots+2^{n-1}=2^n-1$ 个节点。

图 3.8 演示了一棵满二叉树和一棵完全二叉树。

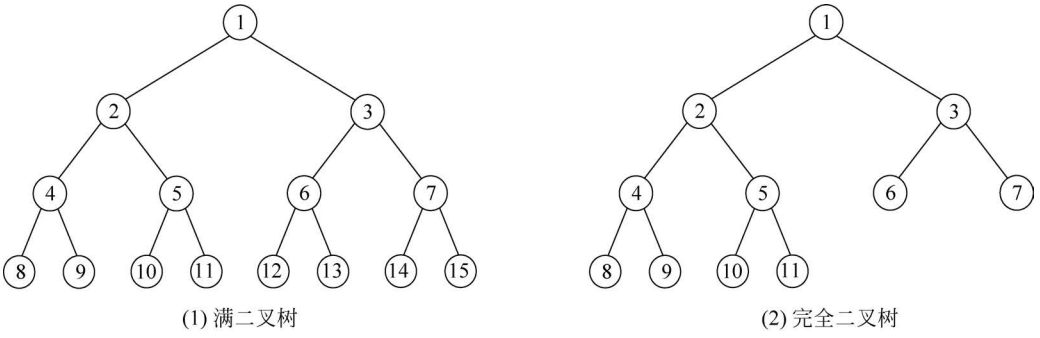


图 3.8　满二叉树和完全二叉树

(2) 完全二叉树。如果满二叉树只在最后一层有缺失，并且缺失的节点都在最后，称为完全二叉树。

(3) 平衡二叉树。任意左子树和右子树的高度差不大于 1，称为平衡二叉树。若只有少部分子树的高度差超过 1，则这是一棵接近平衡的二叉树。

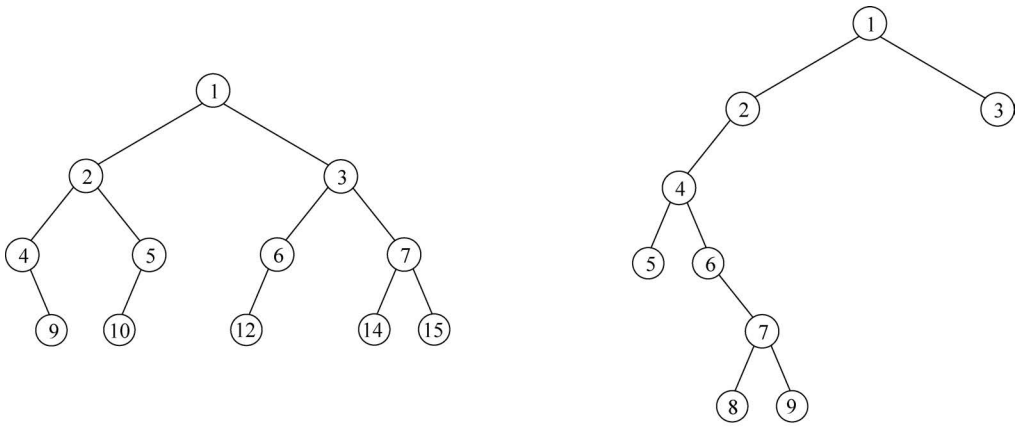
图 3.9 演示了一棵平衡二叉树和一棵退化二叉树。

(4) 退化二叉树[①]。如果树上的每个节点都只有一个孩子，称为退化二叉树。退化二叉树实际上已经变成了一根链表。如果绝大部分节点只有一个孩子，少数有两个孩子，也看成退化二叉树。

二叉树之所以应用广泛，得益于它的形态。高级数据结构大部分和二叉树有关，下面列举二叉树的一些优势。

(1) 在二叉树上能进行极高效率的访问。一棵平衡的二叉树，例如满二叉树或完全二叉树，每一层的节点数量大约是上一层数量的两倍，也就是说，一棵有 N 个节点的满二叉树，树的高度是 $O(\log_2 N)$。从根节点到叶子节点，只需要走 $\log_2 N$ 步，例如 $N=100$ 万，树的高度仅有 $\log_2 N=20$，只需要走 20 步就能到达 100 万个节点中的任意一个。但是，如果

① 本作者曾拟过一句赠言："二叉树对链表说，我也会有老的一天，那时就变成了你"。



(1) 平衡二叉树　　(2) 退化二叉树

图 3.9　平衡二叉树和退化二叉树

二叉树不是满的，而且很不平衡，甚至在极端情况下变成退化二叉树，访问效率会降低。维护二叉树的平衡是高级数据结构的主要任务之一。

(2) 二叉树很适合做从整体到局部、从局部到整体的操作。二叉树内的一棵子树可以看成整棵树的一个子区间，求区间最值、区间和，以及做区间翻转、区间合并、区间分裂等，用二叉树都很快捷。

(3) 基于二叉树的算法容易设计和实现。例如二叉树用 BFS 和 DFS 搜索处理都极为简便。二叉树可以一层一层地搜索，是 BFS 的典型应用场景。二叉树的任意一个子节点是以它为根的一棵二叉树，这是一种递归的结构，用 DFS 访问二叉树极容易编码。

3.7.2　二叉树的存储和编码

1. 二叉树的存储方法

如果要使用二叉树，首先得定义和存储它的节点。

二叉树的一个节点包括 3 个值：节点的值、指向左孩子的指针、指向右孩子的指针。需要用一个结构体来定义二叉树。

二叉树的节点有动态和静态两种存储方法，在竞赛中一般使用静态方法。

(1) 动态存储二叉树。例如写 C 代码，数据结构的教科书一般这样定义二叉树的节点：

```
1  struct Node{
2      int value;                          //节点的值,可以定义多个值
3      Node * lson, * rson;                //指针,分别指向左右子节点
4  };
```

其中，value 是这个节点的值，lson 和 rson 是指向两个孩子的指针。在动态创建一个 Node 时，用 new 运算符动态申请一个节点。在使用完毕后，需要用 delete 释放它，否则会内存泄漏。动态二叉树的优点是不浪费空间，缺点是需要管理，不小心会出错，在竞赛中一般不这样用。

(2) 用静态数组存储二叉树。在算法竞赛中，为了编码简单，加快速度，一般用静态数组来实现二叉树。下面定义一个大小为 N 的结构体数组。N 的值根据题目要求设定，有时节点多，例如 N=100 万，那么 tree[N]使用的内存是 12MB，不算大。

```
1  struct Node{                          //静态二叉树
2      int value;                        //可以把 value 简写为 v
3      int lson, rson;                   //左右孩子,可以把 lson、rson 简写为 ls、rs
4  }tree[N];                             //可以把 tree 简写为 t
```

tree[i]表示这个节点存储在结构体数组的第 i 个位置，lson 是它的左孩子在结构体数组的位置，rson 是它的右孩子在结构体数组的位置。lson 和 rson 指向孩子的位置，也可以称为指针。

图 3.10 演示了一棵二叉树的存储，圆圈内的字母是这个节点的 value 值。根节点存储在 tree[5]上，它的左孩子 lson＝7，表示左孩子存储在 tree[7]上；右孩子 rson＝3，表示右孩子存储在 tree[3]上。该图中把 tree 简写为 t，lson 简写为 l，rson 简写为 r。

在编码时一般不用 tree[0]，因为 0 常被用来表示空节点，例如叶子节点 tree[2]没有孩子，就把它的左右孩子赋值为 lson＝rson＝0。

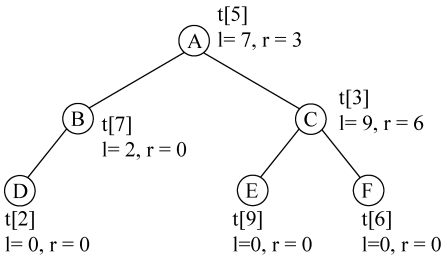


图 3.10　二叉树的静态存储

2. 二叉树存储的编码实现

下面写代码演示图 3.10 中二叉树的建立，并输出二叉树。

第 16～21 行建立二叉树，然后用 print_tree()输出二叉树。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  const int N = 100;                          //注意 const 不能少
4  struct Node{                                //定义静态二叉树结构体
5      char v;                                 //把 value 简写为 v
6      int ls, rs;                             //左右孩子,把 lson、rson 简写为 ls、rs
7  }t[N];                                      //把 tree 简写为 t
8  void print_tree(int u){                     //打印二叉树
9      if(u){
10         cout << t[u].v <<' ';               //打印节点 u 的值
11         print_tree(t[u].ls);                //继续搜左孩子
12         print_tree(t[u].rs);                //继续搜右孩子
13     }
14 }
15 int main(){
16     t[5].v = 'A'; t[5].ls = 7; t[5].rs = 3;
17     t[7].v = 'B'; t[7].ls = 2; t[7].rs = 0;
18     t[3].v = 'C'; t[3].ls = 9; t[3].rs = 6;
19     t[2].v = 'D';          //t[2].ls = 0;t[2].rs = 0;可以不写,t[]是全局变量,已初始化为 0
20     t[9].v = 'E';                           //t[9].ls = 0; t[9].rs = 0; 可以不写
21     t[6].v = 'F';                           //t[6].ls = 0; t[6].rs = 0; 可以不写
22     int root = 5;                           //根是 tree[5]
23     print_tree(5);                          //输出: A B D C E F
24     return 0;
25 }
```

初学者可能看不懂 print_tree()是怎么工作的。它是一个递归函数，先打印这个节点的值 t[u].v，然后继续搜它的左右孩子。图 3.10 的打印结果是“A B D C E F”，步骤如下：

(1) 打印根节点 A。

(2) 搜左孩子，是 B，打印出来。

(3) 继续搜 B 的左孩子,是 D,打印出来。

(4) D 没有孩子,回到 B,发现 B 也没有右孩子,继续回到 A。

(5) A 有右孩子 C,打印出来。

(6) 打印 C 的左右孩子 E、F。

这个递归函数执行的步骤称为“先序遍历”,先输出父节点,再搜左右孩子并输出。

另外还有“中序遍历”和“后序遍历”,将在后面讲解。

3. 二叉树的极简存储方法

如果是满二叉树或者完全二叉树,有更简单的编码方法,连 lson、rson 都不需要定义,因为可以用数组的下标定位左右孩子。

一棵节点总数量为 k 的完全二叉树,设 1 号点为根节点(如图 3.11 所示),有以下性质:

(1) p>1 的节点,其父节点是 p/2。例如 p=4,父亲是 4/2=2; p=5,父亲是 5/2=2。

(2) 如果 2×p>k,那么 p 没有孩子;如果 2×p+1>k,那么 p 没有右孩子。例如 k=11,p=6 的节点没有孩子; k=12,p=6 的节点没有右孩子。

(3) 如果节点 p 有孩子,那么它的左孩子是 2×p、右孩子是 2×p+1。

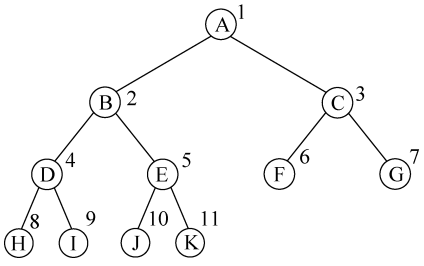


图 3.11 一棵完全二叉树

在该图中圆圈内是节点的值,圆圈外的数字是节点的存储位置。

下面是代码。用 ls(p)找 p 的左孩子,用 rs(p)找 p 的右孩子。在 ls(p)中把 p * 2 写成 p<<1,用了位运算。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100;                                  //注意 const 不能少
char t[N];                                          //简单地用一个数组定义二叉树
int ls(int p){return p << 1;}                       //定位左孩子,也可以写成 p * 2
int rs(int p){return p << 1 | 1;}                   //定位右孩子,也可以写成 p * 2 + 1
int main(){
    t[1] = 'A';  t[2] = 'B';  t[3] = 'C';
    t[4] = 'D';  t[5] = 'E';  t[6] = 'F';  t[7] = 'G';
    t[8] = 'H';  t[9] = 'I';  t[10] = 'J'; t[11] = 'K';
    cout << t[1]<<":lson = "<< t[ls(1)]<<" rson = "<< t[rs(1)];
                                                    //输出为 A:lson = B rson = C
    cout << endl;
    cout << t[5]<<":lson = "<< t[ls(5)]<<" rson = "<< t[rs(5)];
                                                    //输出为 E:lson = J rson = K
    return 0;
}
```

其实,即使二叉树不是完全二叉树,而是普通二叉树,也可以用这种简单方法来存储。如果某个节点没有值,那么就空着这个节点不用,方法是把它赋值为一个不该出现的值,例如赋值为 0 或无穷大 INF。这样虽然会浪费一些空间,但好处是编程非常简单。

3.7.3 例题

二叉树是很基本的数据结构,大量算法、高级数据结构都是基于二叉树的。二叉树有很

多操作，最基础的操作是搜索（遍历）二叉树的每个节点，有先序遍历、中序遍历、后序遍历。这 3 种遍历都用到了递归函数，二叉树的形态适合用递归来编程。图 3.12 所示为一个二叉树例子。

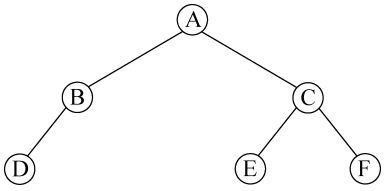


图 3.12　二叉树例子

（1）先（父）序遍历，父节点在最前面输出。先输出父节点，再访问左孩子，最后访问右孩子。该图的先序遍历结果是 ABDCEF。为什么？把结果分解为 A-BD-CEF。父亲是 A，然后是左孩子 B 和它带领的子树 BD，最后是右孩子 C 和它带领的子树 CEF。这是一个递归的过程，每个子树也满足先序遍历，例如 CEF，父亲是 C，然后是左孩子 E，最后是右孩子 F。

（2）中（父）序遍历，父节点在中间输出。先访问左孩子，然后输出父节点，最后访问右孩子。该图的中序遍历结果是 DBAECF。为什么？把结果分解为 DB-A-ECF。DB 是左子树，然后是父亲 A，最后是右子树 ECF。每个子树也满足中序遍历，例如 ECF，先是左孩子 E，然后是父亲 C，最后是右孩子 F。

（3）后（父）序遍历，父节点在最后输出。先访问左孩子，然后访问右孩子，最后输出父节点。该图的后序遍历结果是 DBEFCA。为什么？把结果分解为 DB-EFC-A。DB 是左子树，然后是右子树 EFC，最后是父亲 A。每个子树也满足后序遍历，例如 EFC，先是左孩子 E，然后是右孩子 F，最后是父亲 C。

这 3 种遍历，中序遍历是最有用的，它是二叉查找树的核心。

例 3.16　二叉树的遍历　https://www.luogu.com.cn/problem/B3642

问题描述：有一棵含 n（$n \leqslant 10^6$）个节点的二叉树，给出每个节点的两个子节点的编号（均不超过 n），建立一棵二叉树（根节点的编号为 1），如果是叶子节点，则输入 0 0。在建好这棵二叉树之后，依次求出它的先序、中序、后序遍历。

输入：第一行一个整数 n，表示节点数。之后 n 行，第 i 行两个整数 l、r，分别表示节点 i 的左、右子节点编号。若 l=0，表示无左子节点，r=0 同理。

输出：输出 3 行，每行 n 个数字，用空格隔开。其中，第一行是这个二叉树的先序遍历，第二行是这个二叉树的中序遍历，第三行是这个二叉树的后序遍历。

输入样例：	输出样例：
7 2 7 4 0 0 0 0 3 0 0 0 5 6 0	1 2 4 3 7 6 5 4 3 2 1 6 5 7 3 4 2 5 6 7 1

下面是代码。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100005;
struct Node{
    int v; int ls, rs;
}t[N];                                          //tree[0]不用,0 表示空节点
void preorder(int p){                           //求先序序列
    if(p != 0){
        cout << t[p].v <<" ";                   //先序输出
        preorder(t[p].ls);
        preorder(t[p].rs);
    }
}
void midorder(int p){                           //求中序序列
    if(p != 0){
        midorder(t[p].ls);
        cout << t[p].v <<" ";                   //中序输出
        midorder(t[p].rs);
    }
}
void postorder(int p){                          //求后序序列
    if(p != 0){
        postorder(t[p].ls);
        postorder(t[p].rs);
        cout << t[p].v <<" ";                   //后序输出
    }
}
int main() {
    int n; cin >> n;
    for(int i = 1; i <= n; i++) {
        int a, b; cin >> a >> b;
        t[i].v = i;
        t[i].ls = a;
        t[i].rs = b;
    }
    preorder(1); cout << endl;
    midorder(1); cout << endl;
    postorder(1); cout << endl;
}
```

再看一道例题。

例 3.17　2023 年第十四届蓝桥杯省赛 C/C++大学 C 组试题 J：子树的大小　lanqiaoOJ 3526

时间限制：2s　内存限制：256MB　本题总分：25 分

问题描述：给定一棵包含 n 个节点的完全 m 叉树，节点按从根到叶、从左到右的顺序依次编号。例如图 3.13 是一棵拥有 11 个节点的完全 3 叉树。

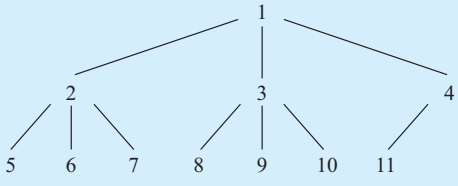


图 3.13　一棵包含 11 个节点的完全 3 叉树

请求出第 k 个节点对应的子树拥有的节点数量。

输入：输入包含多组询问。输入的第一行包含一个整数 t，表示询问次数。接下来 t 行，每行包含 3 个整数 n、m、k，表示一组询问。

输出：一个正整数，表示答案。

输入样例：	输出样例：
3 1 2 1 11 3 4 7 4 5 3	1 2 24

评测用例规模与约定：对于 40％的评测用例，$t\leqslant 50$，$n\leqslant 10^6$，$m\leqslant 16$；对于 100％的评测用例，$1\leqslant t\leqslant 10^5$，$1\leqslant k\leqslant n\leqslant 10^9$，$2\leqslant m\leqslant 10^9$。

这一题可以帮助读者理解树的结构。

第 u 个节点的最左孩子的编号是多少？第 u 号节点前面有 u－1 个点，每个节点各有 m 个孩子，再加上 1 号节点，可得第 u 个节点的左孩子下标为(u－1)×m＋2。例如该图中的 3 号节点，求它的最左孩子的编号。3 号节点前面有两个点，即 1 号节点和 2 号节点，每个节点都有 3 个孩子，1 号节点的孩子是 2、3、4，2 号节点的孩子是 5、6、7，共 6 个孩子。那么 3 号节点的最左孩子的编号是 1＋2×3＋1＝8。

同理，第 u 个节点的孩子如果是满的，它的最右孩子的编号为 u×m＋1。

分析第 u 个节点的情况：

① 节点 u 在最后一层。此时节点 u 的最左孩子的编号大于 n，即(u－1)×m＋2＞n，说明这个孩子不存在，也就是说节点 u 在最后一层，那么以节点 u 为根的子树的节点数量是 1，就是 u 自己。

② 节点 u 不是最后一层，且 u 的孩子是满的，即最右孩子的编号 u×m＋1≤n。此时可以继续分析 u 的孩子的情况。

③ 节点 u 不是最后一层，u 有左孩子，但是孩子不满，此时 u 在倒数第 2 层，它的最右孩子的编号就是 n。以 u 为根的子树的数量＝右孩子编号－(左孩子编号－1)＋u 自己，即 n－((u－1)×m＋1)＋1＝n－u×m＋m。

下面用两种方法求解。

(1) DFS，通过 40％的测试。DFS 在第 6 章讲解，请读者在学过第 6 章后回头看这个方法。

在情况 2)，用 DFS 继续搜 u 的所有孩子，下面的代码实现了上述思路。

代码的计算量是多少？每个点都要计算一次，共 t 组询问，所以总复杂度是 O(nt)，只能通过 40％的测试。

```
#include <iostream>
using namespace std;
typedef long long ll;                          //注意要用 long long,因为下面 m * u 可能超过 int
ll dfs(ll n, ll m, ll u) {
    ll ans = 1;                                //u 点自己算一个
    if(m * u - (m - 2) > n) return 1;          //情况 1),u 点在最后一层, ans = 1
    else if(m * u + 1 <= n) {                  //情况 2),u 在倒数第 2 层且孩子满了
        for(ll c = m * u - (m - 2); c <= m * u + 1; c++)  //深搜 u 的每个孩子
```

```
            ans += dfs(n, m, c);              //累加每个孩子的数量
        return ans;
    }
    else return n + m - m*u;                  //情况 3),u 在倒数第 2 层且孩子不满
}
int main() {
    int t; cin >> t;
    while(t--) {
        ll n, m, k;   cin >> n >> m >> k;
        cout << dfs(n,m,k) << endl;
    }
    return 0;
}
```

(2) 模拟。

上面的 DFS 方法,对于情况 2),把每个点的每个孩子都做了一次 DFS,计算量很大。

其实每一层算一次就行了,在情况 2)时每一层也只需要算一次。以题目的图为例,计算点 1 为根的节点数量。1 号节点这一层有 1 个;它的下一层是满的,有 3 个,左孩子是 2,右孩子是 4;再下一层,2 号节点的左孩子是 5,4 号节点的孩子是 11,那么这一层有 11−5+1=7 个。累加得 1+3+7=11。

计算量是多少?每一层只需要计算一次,共 $O(\log_2 n)$层,t 组询问,总计算复杂度是 $O(t\log_2 n)$,能通过 100%的测试。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;                         //注意要用 long long,因为下面 m*u 可能超过 int
int main(){
    int t; cin >> t;
    while(t--){
        ll n,m,k;
        cin >> n >> m >> k;
        ll ans = 1;                           //初值为 1,即 k 点自己
        ll ls = k, rs = k;                    //从 k 点开始,分析它的最左和最右孩子
        while(1) {                            //从第 k 点开始一层一层往下计算直到最后一层
            ls = (ls-1)*m + 2;                //这一层的最左孩子
            rs = rs*m + 1;                    //这一层的最右孩子
            if(ls > n) break;                 //情况 1),已经到最后一层,结束
            if(rs >= n){                      //情况 3),孩子不满
                ans += n-ls+1;                //加上孩子数量
                break;                        //结束
            }
            ans += rs-ls+1;                   //情况 2),这一层是满的,累加这一层的所有孩子
        }
        cout << ans << endl;
    }
    return 0;
}
```

再看一道例题。

例 3.18　FBI 树　lanqiaoOJ 571

问题描述:可以把由“0”和“1”组成的字符串分为 3 类,全“0”串称为 B 串,全“1”串称

为I串，既含“0”又含“1”的串称为F串。FBI树是一种二叉树，它的节点类型也包括F节点、B节点和I节点3种。由一个长度为2^N的“01”串S可以构造出一棵FBI树T，递归的构造方法如下：

（1）T的根节点为R，其类型与串S的类型相同。

（2）若串S的长度大于1，将串S从中间分开，分为等长的左右子串S_1和S_2；由左子串S_1构造R的左子树T_1，由右子串S_2构造R的右子树T_2。

现在给定一个长度为2^N的“01”串，请用上述构造方法构造出一棵FBI树，并输出它的后序遍历序列。

输入：第一行是一个整数N（$0 \leqslant N \leqslant 10$）。第二行是一个长度为$2^N$的“01”串。

输出：输出一个字符串，即FBI树的后序遍历序列。

输入样例：	输出样例：
3 10001011	IBFBBBFIBFIIIFF

评测用例规模与约定：对于40%的评测用例，$N \leqslant 2$；对于100%的评测用例，$N \leqslant 10$。

首先确定用满二叉树来存题目的FBI树，满二叉树用静态数组实现。当题目中N=10时，串的长度是$2^N=1024$，有1024个元素，需要建一棵大小为4096的二叉树tree[4096]。

题目要求建一棵满二叉树，从左到右的叶子节点就是给定的串S，并且把叶子节点按规则赋值为字符F、B、I，它们上一层的父节点上也按规则赋值为字符F、B、I。最后用后序遍历打印二叉树。

下面是代码。

```
#include<bits/stdc++.h>
using namespace std;
char s[1100],tree[4400];                          //tree[]存满二叉树
int ls(int p){return p<<1;}                       //定位左儿子:p*2
int rs(int p){return p<<1|1;}                     //定位右儿子:p*2+1
void build_FBI(int p,int left,int right){
        if(left==right){                          //到达叶子节点
        if(s[right]=='1') tree[p]='I';
        else tree[p]='B';
        return;
    }
    int mid=(left+right)/2;                       //分成两半
        build_FBI(ls(p),left,mid);                //递归左半
        build_FBI(rs(p),mid+1,right);             //递归右半
    if(tree[ls(p)]=='B'&&tree[rs(p)]=='B')        //左右儿子是B,自己也是B
        tree[p]='B';
    else if(tree[ls(p)]=='I'&&tree[rs(p)]=='I')   //左右儿子是I,自己也是I
        tree[p]='I';
         else tree[p]='F';
}
void postorder(int p){                            //后序遍历
    if(tree[ls(p)]) postorder(ls(p));
    if(tree[rs(p)]) postorder(rs(p));
        printf("%c",tree[p]);
}
int main(){
```

```
27      int n;cin >> n;
28      cin >> s + 1;
29          build_FBI(1,1,strlen(s + 1));
30      postorder(1);
31  }
```

【练习题】

lanqiaoOJ：完全二叉树的权值 183。

洛谷：American Heritage P1827，求先序排列 P1030。

扫一扫

视频讲解

3.8 并 查 集

并查集通常被认为是一种“高级数据结构”，可能是因为用到了集合这种“高级”方法。不过，并查集的编码很简单，数据存储方式也仅用到了最简单的一维数组，可以说并查集是“并不高级的高级数据结构”。

并查集，英文为 Disjoint Set，直译是“不相交集合”。意译为“并查集”非常好，因为它概括了并查集的 3 个要点：并、查、集。并查集是“不相交集合上的合并、查询”。

并查集精巧、实用，在算法竞赛中很常见，原因有三点：简单且高效、应用很直观、容易和其他数据结构与算法结合。并查集的经典应用有判断连通性、最小生成树 Kruskal 算法[①]、最近公共祖先(Least Common Ancestors，LCA)等。

通常用“帮派”的例子来说明并查集的应用背景。在一个城市中有 n 个人，他们分成不同的帮派；给出一些人的关系，例如 1 号、2 号是朋友，1 号、3 号也是朋友，那么他们都属于一个帮派；在分析完所有的朋友关系之后，问有多少个帮派，每个人属于哪个帮派。给出的 n 可能大于 10^6。如果用并查集实现，不仅代码很简单，而且计算复杂度几乎是 O(1)，效率极高。

并查集效率高，是因为用到了“路径压缩[②]”这一技术。

3.8.1 并查集的基本操作

用“帮派”的例子说明并查集的 3 个基本操作：初始化、合并、查找。

(1) 初始化。在开始的时候，帮派的每个人是独立的，相互之间没有关系。把每个人抽象成一个点，每个点有独立的集，n 个点就有 n 个集。也就是说，每个人的帮主就是自己，共有 n 个帮派。

如何表示集？非常简单，用一维数组 int s[]来表示，s[i]的值就是点 i 所属的并查集。初始化 s[i]=i，也就是说，点 i 的集就是 s[i]=i，例如点 1 的集 s[1]=1，点 2 的集 s[2]=2，等等。

用图 3.14 说明并查集的初始化。左边的图给出了点 i 与集 s[i]的值，下画线数字表示集。右边的图表示点和集的逻辑关系，用圆圈表示集，方块表示点。初始时，每个点属于独立的集，5 个点有 5 个集。

① 并查集是 Kruskal 算法的绝配，如果不用并查集，Kruskal 算法很难实现。本作者曾拟过一句赠言：“Kruskal 对并查集说，咱们一辈子是兄弟！”

② 本作者曾拟过一句赠言：“路径压缩担任总经理之后，并查集公司的管理效能实现了跨越式发展。”

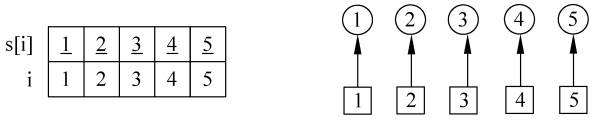


图 3.14　并查集的初始化

(2) 合并。把两个点合并到一个集,就是把两个人所属的帮派合并成一个帮派。

如何合并? 如果 s[i]=s[j],就说明 i 和 j 属于同一个集。操作很简单,把它们的集改成一样即可。下面举例说明。

例如点 1 和点 2 是朋友,把它们合并到一个集。具体操作是把点 1 的集 1 改成点 2 的集 2,s[1]=s[2]=2。当然,把点 2 的集改成点 1 的集也行。经过这个合并,1 和 2 合并成一个帮,帮主是 2。

图 3.15 演示了合并的结果,此时有 5 个点,有 4 个集,其中 s[2]包括两个点。

图 3.15　合并(1,2)

下面继续合并,合并点 1 和点 3。合并的结果是让 s[1]=s[3]。

首先查找点 1 的集,发现 s[1]=2。再继续查找点 2 的集,s[2]=2,点 2 的集是自己,无法继续,查找结束。这个操作是查找 1 的帮主。

再查找点 3 的集,s[3]=3。由于 s[2]不等于 s[3],说明 2 和 3 属于不同的帮派。下面把点 2 的集 2 合并到点 3 的集 3。具体操作是修改 s[2]=3,也就是让 2 的帮主成为 3。此时,点 1、2、3 都属于一个集:s[1]=2、s[2]=3、s[3]=3。1 的上级是 2,2 的上级是 3,这 3 个人的帮主是 3,形成了一个多级关系。

图 3.16 演示了合并的结果。为了简化图示,把点 2 和集 2 画在了一起。

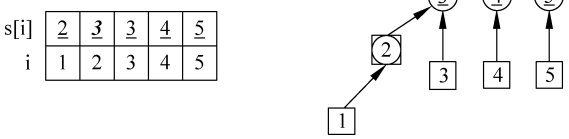


图 3.16　合并(1,3)

继续合并,合并点 2 和点 4。结果如图 3.17 所示,合并过程请读者自己分析。合并的结果是 s[1]=2、s[2]=3、s[3]=4、s[4]=4。4 是 1、2、3、4 的帮主。另外还有一个独立的集 s[5]=5。

(3) 查找某个点属于哪个集。从上面的图示可知,这是一个递归的过程,例如找点 1 的集,递归步骤是 s[1]=2、s[2]=3、s[3]=4、s[4]=4,最后点的值和它的集相等,递归停止,就找到了集。

(4) 统计有几个集。只要检查有多少个点的集等于自己(自己是自己的帮主),就有几个集。如果 s[i]=i,这是这个集的根节点,是它所在的集的代表(帮主);统计根节点的数量,就是集的数量。在上面的图示中,只有 s[4]=4、s[5]=5,有两个集。

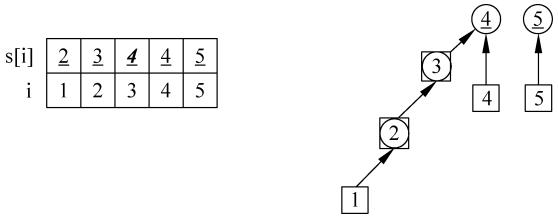


图 3.17　合并(2,4)

从上面的图中可以看到,并查集是“树的森林”,一个集是一棵树,有多少棵树就有多少个集。有些树的高度可能很大(帮会中每个人都只有一个下属),递归步骤是 O(n)的。此时这个集变成了一个链表,出现了并查集的“退化”现象,使得递归查询十分耗时。这个问题可以用“路径压缩”来彻底解决。经过路径压缩后的并查集,查询效率极高,是 O(1)的。

用下面的例题给出并查集的基本操作的编码。

例 3.19　亲戚　https://www.luogu.com.cn/problem/P1551

问题描述:若某个家族人员过多,要判断两个人是否为亲戚,确实很不容易,现在给出某个亲戚关系图,求任意给出的两个人是否具有亲戚关系。规定:x 和 y 是亲戚,y 和 z 是亲戚,那么 x 和 z 也是亲戚。如果 x 和 y 是亲戚,那么 x 的亲戚都是 y 的亲戚,y 的亲戚也都是 x 的亲戚。

输入:第一行有 3 个整数 n、m、p,n、m、p≤5000,分别表示有 n 个人,m 个亲戚关系,询问 p 对亲戚关系。以下 m 行,每行两个数 M_i、M_j、1≤M_i,M_j≤n,表示 M_i 和 M_j 有亲戚关系。接下来 p 行,每行两个数 P_i、P_j,询问 P_i,P_j 是否为一个亲戚关系。

输出:输出 p 行,每行一个 Yes 或 No,表示第 i 个询问的答案为“具有”或“不具有”亲戚关系。

输入样例:	输出样例:
6 5 3 1 2 1 5 3 4 5 2 1 3 1 4 2 3 5 6	Yes Yes No

下面是并查集的代码。

(1) 初始化 init_set()。

(2) 查找 find_set()。这是一个递归函数,若 x==s[x],则这是一个集的根节点,结束。若 x!=s[x],继续递归查找根节点。

(3) 合并 merge_set(x,y)。合并 x 和 y 的集,先递归找到 x 的集,再递归找到 y 的集,然后把 x 合并到 y 的集上。如图 3.18 所示,x 递归到根 b,y 递归到根 d,最后合并为 set[b]=d。合

并后,这棵树变长了,查询效率变低。

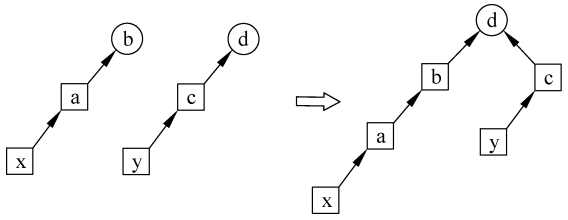


图 3.18　合并

```
#include<bits/stdc++.h>
using namespace std;
const int N = 5010;
int s[N];
void init_set(){                                    //初始化
    for(int i=1; i<=N; i++) s[i] = i;
}
int find_set(int x){                                //查找
   //if(x==s[x]) return x;
   //else return find_set(s[x]);                    //这两行合并为下面的一行
     return x==s[x]? x:find_set(s[x]);
}
void merge_set(int x, int y){                       //合并
    x = find_set(x);
    y = find_set(y);
    if(x != y) s[x] = s[y];                         //y成为x的上级,x的集改成是y的集
}
int main(){
    init_set();
    int n,m,p; cin>>n>>m>>p;
    while(m--){                                     //合并
        int x,y; cin>>x>>y;
        merge_set(x, y);
    }
    while(p--){                                     //查询
        int x,y; cin>>x>>y;
        if(find_set(x)==find_set(y)) cout<<"Yes"<<endl;
        else cout<<"No"<<endl;
    }
    return 0;
}
```

下面用路径压缩来优化并查集的退化问题。

3.8.2　路径压缩

在做并查集题目时,一定需要用到“路径压缩”这个优化技术。路径压缩是并查集真正的核心,不过它的原理和代码极为简单。

在上面的查询函数 find_set()中,查询元素 i 所属的集,需要递归搜索整个路径直到根节点,返回值是根节点。这条搜索路径可能很长,导致超时。

如何优化?如果在递归返回的时候,顺便把这条路径上所有的点所属的集改成根节点(所有人都只有帮主一个上级,不再有其他上级),那么下次再查这条路径上的点属于哪个集,就能在 O(1)的时间内得到结果。如图 3.19 的左图所示,在第一次查询点 1 的集时,需

要递归路径查 3 次，在递归返回时，把路径上的 1、2、3 所属的集都改成 4，使得所有点的集都是 4，如右图所示。下次再查询 1、2、3、4 所属的集，只需递归一次，就查到了根。

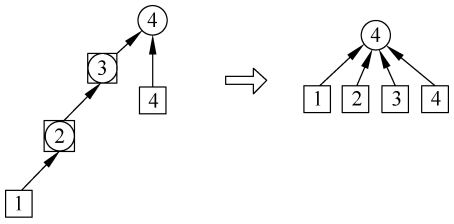


图 3.19　路径压缩

路径压缩的代码非常简单。把上面代码中的 find_set()改成以下路径压缩的代码。

```
int find_set(int x){
    if(x != s[x]) s[x] = find_set(s[x]);   //路径压缩
    return s[x];
}
```

以上介绍了查询时的路径压缩，那么合并时也需要做路径压缩吗？一般不需要，因为合并需要先查询，查询用到了路径压缩，间接地优化了合并。

在路径压缩之前，查询和合并都是 O(n)的。经过路径压缩之后，查询和合并都是 O(1)的，并查集显示出了巨大的威力。

3.8.3　例题

例 3.20　修复公路　https://www.luogu.com.cn/problem/P1111

问题描述：A 地区在地震过后，连接所有村庄的公路都造成了损坏而无法通车。政府派人修复这些公路，给出 A 地区的村庄数 N 和公路数 M，公路是双向的，告知每条公路连着哪两个村庄，并告知什么时候能修完这条公路。问最早什么时候任意两个村庄能够通车，即最早什么时候任意两个村庄都至少存在一条修复完成的道路(可以由多条公路连成一条道路)。

输入：第一行两个正整数 N、M。下面 M 行，每行 3 个正整数 x、y、t，告知这条公路连着 x 和 y 两个村庄，在时间 t 时能修复完成这条公路。

输出：如果全部公路修复完毕仍然存在两个村庄无法通车，输出－1，否则输出最早什么时候任意两个村庄能够通车。

输入样例：	输出样例：
4 4 1 2 6 1 3 4 1 4 5 4 2 3	5

评测用例规模与约定：$1 \leqslant x,y \leqslant N \leqslant 10^3$，$1 \leqslant M,t \leqslant 10^5$。

题目看起来像图论的最小生成树，不过用并查集可以简单地解决。

本题实际上是连通性问题，连通性也是并查集的一个应用场景。

先按时间 t 把所有道路排序，然后按时间 t 从小到大逐个加入道路，合并村庄。如果在某个时间所有村庄已经通车，这就是最小通车时间，输出并结束。如果所有道路都已经加入，但是还有村庄没有合并，就输出－1。

用并查集处理村庄合并，在合并时统计通车村庄的数量。

下面的代码没有写合并函数 merge_set()，而是把合并功能写在第 19～21 行，做了灵活处理。第 20 行，如果村庄 x、y 已经连通，那么连通的村庄数量不用增加；第 21 行，如果 x、y 没有连通，则合并并查集。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  struct node{int x,y,t;} e[100010];
4  bool cmp(node a, node b){return a.t<b.t;}            //sort 函数的比较
5  int s[100010];
6  int find_set(int x){                                 //用"路径压缩"优化的查询
7      if(x != s[x]) s[x] = find_set(s[x]);             //路径压缩
8      return s[x];
9  }
10 int main(){
11     int n,m; scanf("%d%d",&n,&m);
12     for(int i=1;i<=m;i++) s[i]=i;                    //并查集的初始化
13     for(int i=1;i<=m;i++)
14         scanf("%d%d%d",&e[i].x,&e[i].y,&e[i].t);
15     sort(e+1,e+1+m,cmp);                             //按时间 t 排序
16     int ans=0;                                       //答案，最早通车时间
17     int num=0;                                       //已经连通的村庄数量
18     for(int i=1;i<=m;i++){
19         int x=find_set(e[i].x), y=find_set(e[i].y);
20         if(x==y) continue;                           //x、y 已经连通，num 不用增加
21         s[x]=y;                                      //合并并查集，即把村庄 x 合并到 y 上
22         num++;                                       //连通的村庄加 1
23         ans = max(ans,e[i].t);                       //当前最大通车时间
24     }
25     if(num!=n-1) printf("-1\n");
26     else printf("%d\n",ans);
27     return 0;
28 }
```

再看一道比较难的例题。

例 3.21　2019 年第十届蓝桥杯省赛 C/C++大学 A 组试题 H：修改数组 lanqiaoOJ 185

时间限制：1s　内存限制：256MB　本题总分：20 分

问题描述：给定一个长度为 N 的数组 $A=[A_1,A_2,\cdots,A_N]$，数组中可能有重复出现的整数。现在小明要按以下方法将其修改为没有重复整数的数组。小明会依次修改 A_2、A_3、…、A_N。当修改 A_i 时，小明会检查 A_i 是否在 A_1～A_{i-1} 中出现过。如果出现过，则小明会给 A_i 加上 1；如果新的 A_i 仍在之前出现过，小明会持续给 A_i 加 1，直到 A_i 没有在 A_1～A_{i-1} 中出现过。当 A_N 也经过上述修改之后，显然 A 数组中就没有重复的整数

了。给定初始的 A 数组，请计算出最终的 A 数组。

输入：第一行包含一个整数 N，第二行包含 N 个整数 $A_1, A_2, \cdots, A_N$。

输出：输出 N 个整数，依次是最终的 $A_1, A_2, \cdots, A_N$。

输入样例：	输出样例：
5 2 1 1 3 4	2 1 3 4 5

评测用例规模与约定：对于 80% 的评测用例，$1 \leqslant N \leqslant 10000$；对于所有评测用例，$1 \leqslant N \leqslant 100000$，$1 \leqslant A_i \leqslant 1000000$。

这是一道好题，很难想到可以用并查集来做。

先尝试暴力的方法：每读入一个新的数，就检查前面是否出现过，每一次需要检查前面所有的数。共有 n 个数，每个数检查 O(n)次，所以总复杂度是 $O(n^2)$，写代码提交可能通过 30% 的测试。

大家容易想到一个改进的方法：用 hash。定义 vis[]数组，vis[i]表示数字 i 是否已经出现过。这样就不用检查前面所有的数了，基本上可以在 O(1)的时间内定位到。

然而，本题有一个特殊的要求："如果新的 A_i 仍在之前出现过，小明会持续给 A_i 加 1，直到 A_i 没有在 $A_1 \sim A_{i-1}$ 中出现过。"这导致在某些情况下仍然需要大量的检查。以 5 个 6 为例：A[]={6,6,6,6,6}。

第一次读 A[1]=6，设置 vis[6]=1。

第二次读 A[2]=6，先查到 vis[6]=1，则把 A[2]加 1，变为 A[2]=7；再查 vis[7]=0，设置 vis[7]=1。检查了两次。

第三次读 A[3]=6，先查到 vis[6]=1，则把 A[3]加 1 得 A[3]=7；再查到 vis[7]=1，再把 A[3]加 1 得 A[3]=8，设置 vis[8]=1；最后查 vis[8]=0，设置 vis[8]=1。检查了 3 次。

……

每次读一个数，仍需检查 O(n)次，总复杂度仍然是 $O(n^2)$。

下面给出 hash 代码，提交后能通过 80% 的测试。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1000002;                  //A 的 hash,1≤A_i≤1000000
int vis[N];                             //hash: vis[i] = 1 表示数字 i 已经存在
int main(){
    int n;
    scanf("%d",&n);
     for(int i = 0;i < n;i++){
        int a;
        scanf("%d",&a);                 //读一个数字
        while(vis[a] == 1)              //若 a 已经出现过,加 1。若 a 加 1 后再出现,则继续加 1
            a++;
        vis[a] = 1;                     //标记该数字
        printf("%d ",a);                //打印
    }
}
```

这道题用并查集非常巧妙。

前面提到，本题用 hash 方法，在特殊情况下仍然需要大量的检查。问题出在“持续给 A_i 加 1，直到 A_i 没有在 $A_1 \sim A_{i-1}$ 中出现过”。也就是说，问题出在那些相同的数字上。当处理一个新的 A[i]时，需要检查所有与它相同的数字。

如果把这些相同的数字看成一个集合，就能用并查集处理。

用并查集 s[i]表示访问到 i 这个数时应该将它换成的数字。以 A[]={6,6,6,6,6}为例。初始化时 set[i]=i，处理过程如图 3.20 所示。

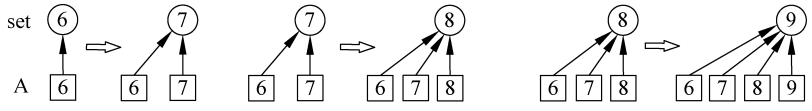


图 3.20　用并查集处理数组 A

图 3.20(a)读第一个数 A[0]=6。6 的集 set[6]=6。紧接着更新 set[6]=set[7]=7，作用是后面再读到某个 A[k]=6 时，可以直接赋值 A[k]=set[6]=7。

图 3.20(b)读第二个数 A[1]=6。6 的集 set[6]=7，更新 A[1]=7。紧接着更新 set[7]=set[8]=8。如果后面再读到 A[k]=6 或 7 时，可以直接赋值 A[k]=set[6]=8 或者 A[k]=set[7]=8。

图 3.20(c)读第三个数 A[2]=6。请读者自己分析。

下面是 C++代码，只用到并查集的查询，没用到合并，必须用“路径压缩”优化才能加快查询速度，通过 100%的测试。没有路径压缩，仍然超时。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1000002;
int A[N];
int s[N];                                          //并查集
int find_set(int x){                               //用“路径压缩”优化的查询
    if(x != s[x]) s[x] = find_set(s[x]);           //路径压缩
    return s[x];
}
int main(){
    for(int i=1;i<N;i++) s[i]=i;                   //并查集的初始化
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d",&A[i]);
        int root = find_set(A[i]);                 //查询到并查集的根
        A[i] = root;
        s[root] = find_set(root+1);                //加 1
    }
    for(int i=1;i<=n;i++) printf("%d ",A[i]);
    return 0;
}
```

【练习题】

lanqiaoOJ：蓝桥幼儿园 1135、简单的集合合并 3959、合根植物 110。

洛谷：一中校运会之百米跑 P2256、村村通 P1536、家谱 P2814、选择题 P6691。

3.9 扩展学习

数据结构是算法大厦的砖石，它们渗透在所有问题的代码实现中。数据结构和算法密不可分。

本章介绍了一些基础数据结构：数组、链表、队列、栈、二叉树。在竞赛中这些数据结构可以用 STL 实现，也可以手写代码实现。STL 应该重点掌握，大多数题目能直接用 STL 实现，编码简单、快捷，不容易出错。如果需要手写数据结构，一般都使用静态数组来模拟，这样做编码快且不容易出错。

对于基础数据结构，程序员应该不假思索地、条件反射般地写出来，使得它们成为大脑的“思想钢印”。

在学习基础数据结构的基础上可以继续学习高级数据结构。大部分高级数据结构很难，是算法竞赛中的难题。在蓝桥杯这种短时个人赛中，高级数据结构并不多见，近年来考过并查集、线段树。读者可以多练习线段树，线段树是标志性的中级知识点，是从初级水平进入中级水平的里程碑。

学习计划可以按以下知识点展开。

中级：树上问题、替罪羊树、树状数组、线段树、分块、莫队算法、块状链表、LCA、树上分治、Treap 树、笛卡儿树、K-D 树。

高级：Splay 树、可持久化线段树、树链剖分、FHQ Treap 树、动态树、LCT。

在计算机科学中，各种数据结构的设计、实现、应用非常精彩，它们对数据的访问方式、访问的效率、空间利用各有侧重。通过合理选择和设计数据结构，可以优化算法的执行时间和空间复杂度，从而提高程序的性能。