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。

```
1  public class Main {
2      public static void main(String[] args) {
3          int[] a = new int[5];
4          int size = a.length * Integer.BYTES;        //计算 int 类型数组的字节数
5          System.out.println("Size of int a[5]: " + size + " bytes");
6      }
7  }
```

在作者的计算机上运行，输出 5 个整数的字节数：

```
Size of int a[5]: 20 bytes
```

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，需要挪接近 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，也需要挪动接近 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。

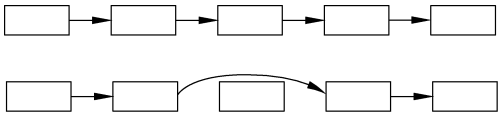


图 3.1　删除第 3 个数据

① 本书作者曾写过一句赠言：“以数据结构为弓，以算法为箭”。

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。

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，除非必要，一般不手写链表，而是用系统提供的链表，例如 LinkedList。

（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)。

哈希表的主要缺点是，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。

---

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 本书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”。

扫一扫

视频讲解

3.1 Java 常用功能

Java 是一种面向对象的编程语言，以简单、可移植、安全、高性能和可靠而闻名，被广泛应用于各种应用程序的开发。Java 有以下特点：

（1）简单易学。Java 语法相对简单，和 C/C++差不多，而且去除了一些复杂的特性，使得初学者更容易上手。

（2）面向对象。Java 是一种纯粹的面向对象编程语言，一切皆对象。它支持封装、继承和多态等面向对象的特性。

（3）平台无关性。Java 程序可以在不同的操作系统上运行，只需要在目标平台上安装 Java 虚拟机（JVM）即可。

（4）垃圾回收机制。Java 拥有自动垃圾回收机制，开发者无须手动管理内存，减少了内存泄漏和野指针的问题。

（5）强大的类库。Java 提供了丰富的类库，涵盖了各种功能，例如网络、数据库、图形用户界面等，开发者可以直接使用这些类库快速构建应用程序。

Java 的常用类库有很多，以下是一些常见的类库。

（1）java.lang：提供 Java 的核心类，例如 String、Math、Object 等。

（2）java.util：提供一系列实用的工具类，例如 ArrayList、LinkedList、HashMap 等。

（3）java.io：提供对输入/输出流的支持，例如 File、InputStream、OutputStream 等。

（4）java.net：提供与网络编程相关的类，例如 Socket、URL 等。

（5）java.sql：提供对数据库的访问支持，例如 Connection、Statement 等。

本书不会详细介绍 Java 语法，因为本书是一本算法竞赛教材，而不是 Java 语言教材，对于 Java 语言的学习，请读者通过阅读 Java 教材来掌握。不过，有一些竞赛中常用的 Java 数据结构、函数[①]等，一般的 Java 教材通常不涉及，本章将做详细介绍。

3.1.1 String

在算法竞赛中，字符串处理是极为常见的考点。用 java.lang.String 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，如果竞赛时不用 String，成绩会大受影响。

String 类提供了许多方法来操作字符串，包括字符串的拼接、截取、查找、替换、大小写转换等。

注意，String 对象一旦创建，其值不可以被修改。任何对字符串的操作都会返回一个新的 String 对象，而不会修改原有的 String 对象。

表 3.1 中列出一些常用的 String 方法。

表 3.1　一些常用的 String 方法

方　法	说　明
length()	返回字符串的长度
contains()	检查字符串是否包含指定的字符序列，并返回布尔值

① Java 的官方帮助文档：https://docs.oracle.com/en/

续表

方　法	说　明
isEmpty()	检查字符串是否为空(长度为 0),并返回布尔值
compareTo()	按字典顺序比较两个字符串,返回一个整数值
valueOf()	将其他类型的数据转换为字符串。例如,valueOf(int i)可以将整数转换为字符串
toCharArray()	将字符串转换为字符数组,并返回该数组
charAt()	返回指定索引位置的字符
trim()	去除字符串首尾的空格,并返回新的字符串
concat()	将指定的字符串连接到原字符串的末尾,或者用'+'运算符
equals()	字符串比较
substring()	返回从指定索引开始到指定索引结束的子字符串
indexOf()	返回指定字符串在原字符串中第一次出现的索引位置
replace()	将字符串中的指定字符替换为新的字符
toUpperCase()	将字符串转换为大写
toLowerCase()	将字符串转换为小写
split()	将字符串按照指定的正则表达式分割成字符串数组

下面给出例子,请特别注意从第 59 行开始的字符串的比较。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  import java.util.*;
2  public class Main {
3      public static void main(String[] args) {
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          System.out.println(s2);             //输出:efg
9          String s = "abc";                   //定义并初始化
10         String s3 = s;                      //复制
11     //长度
12         System.out.println(s.length());     //输出:3
13     //遍历
14         for (int i = 0; i < s.length(); i++)
15             System.out.print(s.charAt(i)); //输出:abc
16         System.out.println();
17     //添加,合并字符串
18         s = s + 'd';
19         System.out.println(s);              //在尾部添加字符。输出:abcd
20         s = s.concat("efg");
21         System.out.println(s);              //在尾部添加字符串。输出:abcdefg
22         s = s + 'h';
23         s += 'i';
24         System.out.println(s);              //用'+'在尾部添加字符。输出:abcdefghi
25         s = s + "jk";
26         s += "lmnabc";
27         s = "xyz" + s;
28         System.out.println(s);              //输出:xyzabcdefghijklmnabc
29         String s4 = "uvw";
30         System.out.println(s + s4);
31     //合并字符串。输出:xyzabcdefghijklmnabcuvw
32     //查找字符和字符串
33         System.out.println("pos of b=" + s.indexOf('b'));
34     //找字符第一次出现的位置。输出:pos of b=4
```

```
35          System.out.println("pos of ef = " + s.indexOf("ef"));
36      //找字符串第一次出现的位置。输出:pos of ef = 7
37          System.out.println("pos of ab = " + s.indexOf("ab", 5));
38      //从 s[5]开始找字符串第一次出现的位置。输出:pos of ab = 17
39          System.out.println("pos of hello = " + (int) s.indexOf("hello"));
40      //没找到的返回值是 - 1。输出:pos of hello = - 1
41      //截取字符串
42          System.out.println(s.substring(3, 8));
43      //从 s[3]开始截取 5 个字符构成的字符串。输出:abcde
44          System.out.println(s.substring(0, 4) + "opq" + s.substring(4));
45      //输出:xyzaopqbcdefghijklmnabc
46      //删除、替换
47          System.out.println(s.substring(0, 10) + s.substring(12));
48      //从 s[10]开始删除后面所有字符。输出:xyzabcdefgjklmnabc
49          System.out.println(s.substring(0, 2) + "1234" + s.substring(5));
50      //把从 s[2]开始的 3 个字符替换为"1234"。输出:xy1234cdefghijklmnabc
51          System.out.println(s.substring(0, 7) + "5678" + s.substring(9));
52      //把 s[7]～s[8]替换为"5678"。输出:xyzabcd5678ghijklmnabc
53      //清理、判断
54          System.out.println(s.isEmpty());
55      //判断是否为空,不空返回 false,空返回 true。输出:false
56          s = "";                              //清空
57          System.out.println(s.isEmpty());     //输出:true
58      //比较
59          String s5 = "abc";
60          String s6 = "abc";
61          String s7 = "bc";
62          if (s5.equals(s6)) System.out.println(" == ");              //输出: ==
63          if (s5.compareTo(s7) < 0) System.out.println("<");          //输出:<
64          if (s5.compareTo(s7) > 0) System.out.println(">");
65          if (!s5.equals(s7)) System.out.println("!= ");              //输出:!=
66      }
67  }
```

下面是一道简单字符串题。

例 3.1　烬寂海之谜 lanqiaoOJ 4050

问题描述：给定一个字符串 S,以及若干模式串 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。例如 S="aaaa"，P="aa"，答案是 3，不是 2。

下面的代码用到 String 的 length()和 substring()。

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.next();
        int n = sc.nextInt();
        while (n-- > 0) {
            String p = sc.next();
            int cnt = 0;
            for (int i = 0; i < s.length() - p.length() + 1; i++)
                if (p.equals(s.substring(i, i + p.length())))
                    cnt++;
            System.out.println(cnt);
        }
    }
}
```

【练习题】

简单字符串入门题[①]很多，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

3.1.2 BigInteger

在算法竞赛中经常需要计算极大的数。long 型整数只有 64 位，如果需要计算更大的数，需要使用 BigInteger 类。

BigInteger 类是用于处理任意大小整数的类。它可以表示和执行大整数的算术运算，包括加法、减法、乘法和除法，而不会受到 Java 原生整数类型的范围限制。

BigInteger 类提供了一系列方法，用于执行各种操作，例如比较、求幂、取模等。它还支持位操作、位移和按位逻辑运算。注意，BigInteger 类的实例是不可变的，一旦创建，就不能被修改。每次执行算术操作时，都会创建一个新的 BigInteger 对象来保存结果。

BigInteger 的常用方法如表 3.2 所示。

表 3.2 BigInteger 的常用方法

方　法	说　明
abs()	绝对值
negate()	相反值

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”。

续表

方　法	说　明
add(BigInteger val)	加
subtract(BigInteger val)	减
multiply(BigInteger val)	乘
divide(BigInteger val)	整除
remainder(BigInteger val)	整数的余数
mod(BigInteger val)	求余
pow(int e)	幂
shiftLeft(int n)	左移 n 位
shiftRight(int n)	右移 n 位
and(BigInteger val)	与
or(BigInteger val)	或
not()	非
xor(BigInteger val)	异或
max(BigInteger val)	较大值
min(BigInteger val)	较小值
bitCount()	二进制中不包括符号位的 1 的个数
bitLength()	二进制中不包括符号位的长度
getLowestSetBit()	二进制中最右边的位置
toString()	十进制字符串表示形式
toString(int radix)	radix 进制字符串表示形式
gcd(BigInteger val)	绝对值的最大公约数
isProbablePrime(int val)	是否为素数
nextProbablePrime()	第一个大于 this 的素数
modPow(BigInteger b,BigInteger p)	a^b mod p
modInverse(BigInteger p)	a mod p 的乘法逆元

下面举例说明 BigInteger 的用法，包括加、减、乘、除和计算阶乘。表 3.2 中的其他功能请读者自己熟悉。

```
1  import java.math.BigInteger;
2  public class Main {
3      public static void main(String[] args) {
4          BigInteger a = new BigInteger("123456789012345645343433223443");
5          BigInteger b = new BigInteger("98765432109876554545453443210322");
6          BigInteger sum = a.add(b);                    //加
7          BigInteger d = a.subtract(b);                 //减
8          BigInteger p = a.multiply(b);                 //乘
9          BigInteger q = a.divide(b);                   //商
10         BigInteger r = a.remainder(b);                //余数
11         System.out.println("Sum: " + sum);
12         System.out.println("Difference: " + d);
13         System.out.println("Product: " + p);
14         System.out.println("Quotient: " + q);
15         System.out.println("Remainder: " + r);
16         BigInteger fac = BigInteger.valueOf(1);       //计算 100 的阶乘
17         for (int i = 1; i <= 100; i++)
18             fac = fac.multiply(BigInteger.valueOf(i));
19         System.out.println("100! =" + fac);
20     }
21 }
```

下面是一道简单的例题。

例 3.2　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

直接计算。

```
1  import java.math.BigInteger;
2  import java.util.*;
3  public class Main {
4      public static void main(String[]arges) {
5          Scanner sc = new Scanner(System.in);
6          BigInteger a = sc.nextBigInteger();
7          BigInteger b = sc.nextBigInteger();
8          System.out.println(a.add(b));
9      }
10 }
```

【练习题】

洛谷：高精度减法 P2142、A * B Problem P1303、A/B Problem P1480。

3.1.3　日期类

日期问题是蓝桥杯的常见题型。在《蓝桥杯算法入门(Python)》第 2 章的“2.4 填空题例题”中用“工作时长”这道例题说明了 Python 的 datetime 库在日期问题中的应用。其实 Java 也有日期包 java.time，虽然用起来没有 Python 简洁，但是功能差不多。

time 包主要包含以下类。

- LocalDate：表示日期，年、月、日。
- LocalTime：表示时间，时、分、秒、纳秒。
- LocalDateTime：表示日期和时间，年、月、日、时、分、秒、纳秒。
- Duration：表示时间段，用于计算两个时间之间的差异。
- Period：表示日期段，用于计算两个日期之间的差异。
- DateTimeFormatter：用于将日期和时间格式化为字符串，或将字符串解析为日期和时间对象。

下面详细说明。

1. LocalDate 类

LocalDate 是 Java 用于表示日期的类，它提供了处理日期的各种方法和操作。以下是 LocalDate 类的一些重要特性和用法。

(1) 创建 LocalDate 对象。

用 now()方法获取当前日期,例如 LocalDate.now()。

用 of()方法创建指定日期的 LocalDate 对象,例如 LocalDate.of(2023,3,1)表示 2023 年 3 月 1 日。

(2) 获取日期。

用 getYear()、getMonthValue()和 getDayOfMonth()方法获取年、月、日的值,例如 LocalDate.now().getYear()获取当前年份。

(3) 日期加减和修改。

用 plusXXX()和 minusXXX()方法在日期上进行加减操作,例如 LocalDate.now().plusDays(7)表示当前日期加 7 天。

用 withXXX()方法修改日期的某个部分,例如 LocalDate.now().withMonth(2)将当前日期的月份修改为 2。

(4) 比较日期。

用 isEqual()、isBefore()、isAfter()方法比较两个日期的关系,例如 d1.isBefore(d2)判断 d1 是否在 d2 之前。

(5) 日期格式。

用 format()方法将日期格式化为字符串,例如 LocalDate.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd"))。

(6) 其他常用方法。

- isLeapYear():判断该年份是否为闰年。
- lengthOfMonth():获取该月份的天数。
- getDayOfWeek():获取该日期是星期几。

下面举例说明它们的功能。

```
1  import java.time.LocalDate;
2  import java.time.format.DateTimeFormatter;
3  import java.time.temporal.ChronoUnit;
4  import java.time.Period;
5  public class Main {
6      public static void main(String[] args) {
7          LocalDate today = LocalDate.now();          //当前日期
8          System.out.println(today);                  //打印:2024-03-22
9          LocalDate minDate = LocalDate.MIN;          //最小日期
10         System.out.println(minDate);                //打印:-999999999-01-01
11         LocalDate maxDate = LocalDate.MAX;          //最大日期
12         System.out.println(maxDate);                //打印:+999999999-12-31
13         LocalDate a = LocalDate.of(2024, 3, 14);
14         LocalDate b = LocalDate.of(2022, 2, 15);
15         System.out.println(a);                      //打印:2024-03-14
16         System.out.println(a.toString());           //打印:2024-03-14
17     System.out.println(a.format(DateTimeFormatter.ofPattern("yyyyMMdd")));
18 //按格式打印:20240314
19     System.out.println(a.format(DateTimeFormatter.ofPattern("yyMMdd")));
20 //按格式打印:240314
21     System.out.println(a.format(DateTimeFormatter.ofPattern("yyyy-MM-dd")));
22 //按格式打印:2024-03-14
23         System.out.println(a.getYear());            //打印:2024
24         System.out.println(a.getMonthValue());      //打印:3
25         System.out.println(a.getDayOfMonth());      //打印:14
```

```
26          System.out.println(a.getDayOfWeek().getValue());
27  //星期一是1,星期天是7,打印:4
28          System.out.println(a.isAfter(b));          //日期比较,打印:true
29          System.out.println(Period.between(b, a));  //日期之差,打印:P2Y28D
30          System.out.println(ChronoUnit.DAYS.between(b, a));
31                          //日期之差,打印:758
32      }
33  }
```

第17行的ofPattern方法接收一个字符串参数,该参数定义了日期时间格式的模式。模式由一系列的字母和符号组成,用于表示日期时间的不同部分,例如年份、月份、日、小时、分钟和秒等。以下是一些常用的模式字母和符号。

- yyyy:四位数的年份。
- MM:两位数的月份。
- dd:两位数的日期。
- HH:两位数的小时(24小时制)。
- hh:两位数的小时(12小时制)。
- mm:两位数的分钟。
- ss:两位数的秒钟。

这些格式也在下面的LocalTime和LocalDateTime类中使用。

2. LocalTime类

LocalTime是Java用于表示时间的类,它提供了处理时间的各种方法和操作。以下是LocalTime类的一些重要特性和用法。

(1) 创建LocalTime对象。

用now()方法获取当前时间,例如LocalTime.now()。

用of()方法创建指定时间的LocalTime对象,例如LocalTime.of(12,0)表示12:00。

(2) 获取时间。

用getHour()、getMinute()、getSecond()等方法获取小时、分钟、秒等时间部分的值。

(3) 日期加减和修改。

用plusXXX()和minusXXX()方法在时间上进行加减操作,例如LocalTime.now().plusHours(2)表示当前时间加两小时。

用withXXX()方法修改时间的某个部分,例如LocalTime.now().withMinute(30)将当前时间的分钟修改为30。

(4) 比较时间。

用isBefore()和isAfter()方法比较两个时间的关系,例如t1.isBefore(t2)判断t1是否在t2之前。

(5) 时间格式。

用format()方法将时间格式化为字符串,例如LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm:ss"))。

(6) 其他常用方法。

- toSecondOfDay():获取该时间从当天开始的秒数。

- truncatedTo()：截断时间到指定精度，例如 LocalTime.now().truncatedTo(ChronoUnit.MINUTES)将当前时间截断到分钟级别。

下面是一个例子。

```
1  import java.time.LocalTime;
2  public class Main {
3      public static void main(String[] args) {
4          LocalTime minTime = LocalTime.MIN;
5          System.out.println(minTime);              //打印:00:00
6          LocalTime maxTime = LocalTime.MAX;
7          System.out.println(maxTime);              //打印: 23:59:59.999999999
8          LocalTime a = LocalTime.of(23, 59, 34, 333);
9          LocalTime b = LocalTime.of(22, 9, 4, 3);
10         System.out.println(a);                    //打印:23:59:34.000000333
11         System.out.println(a.getHour());          //打印:23
12         System.out.println(a.getMinute());        //打印:59
13         System.out.println(a.getSecond());        //打印:34
14         System.out.println(a.getNano());          //打印:333
15         System.out.println(a.isAfter(b));         //比较。打印:true
16     }
17 }
```

3. LocalDateTime 类

LocalDateTime 可以看作 LocalDate 类和 LocalTime 类的合体。

LocalDateTime 是 Java 中用于表示日期和时间的类，它提供了处理日期和时间的各种方法和操作。以下是 LocalDateTime 类的一些重要特性和用法。

(1) 创建 LocalDateTime 对象。

用 now()方法获取当前日期和时间，例如 LocalDateTime.now()。

用 of()方法创建指定日期和时间的 LocalDateTime 对象，例如 LocalDateTime.of(2023,3,1,12,0)表示 2023 年 3 月 1 日 12：00。

(2) 获取日期和时间。

用 getYear()、getMonthValue()、getDayOfMonth()获取年、月、日等日期部分的值。

用 getHour()、getMinute()、getSecond()获取小时、分钟、秒等时间部分的值。

(3) 日期加减和修改。

用 plusXXX()和 minusXXX()方法在日期和时间上进行加减操作，例如 LocalDateTime.now().plusDays(7)表示当前日期加 7 天。

用 withXXX()方法修改日期和时间的某个部分，例如 LocalDateTime.now().withHour(10)将当前时间的小时修改为 10。

(4) 比较日期。

用 isEqual()、isBefore()、isAfter()方法比较两个日期和时间的关系，例如 d1.isBefore(d2)判断 d1 是否在 d2 之前。

(5) 日期格式。

用 format()方法将日期和时间格式化为字符串，例如 LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"))。

(6) 其他常用方法。

- toLocalDate()：获取日期部分，返回 LocalDate 对象。
- toLocalTime()：获取时间部分，返回 LocalTime 对象。

下面是一个例子。

```
import java.time.LocalDateTime;
import java.time.Duration;
public class Main {
    public static void main(String[] args) {
        LocalDateTime start = LocalDateTime.now();          //当前时间
        System.out.println(LocalDateTime.now());
// 打印:2024-03-22T08:55:23.175
        LocalDateTime a = LocalDateTime.of(2026, 5, 14, 23, 56, 9);
        System.out.println(a.getMonthValue());              //打印:5
        System.out.println(a.getSecond());                  //打印:9
        LocalDateTime b = a.plusWeeks(7).plusDays(7);
        b = b.plusHours(8).plusMinutes(23).plusSeconds(47);
        System.out.println(Duration.between(a, b));         //打印:PT1352H23M47S
        System.out.println(a.isAfter(b));                   //比较时间。打印:false
    }
}
```

下面看一道例题。

例 3.3　2021 年第十二届 Python 大学 A 组试题 F：时间显示 lanqiaoOJ 1452

时间限制：1s　**内存限制**：512MB　**本题总分**：15 分

问题描述：小蓝要和朋友合作开发一个显示时间的网站。在服务器上，朋友已经获取了当前时间，用一个整数表示，值为从 1970 年 1 月 1 日 00:00:00 到当前时刻经过的毫秒数。

现在小蓝要在客户端显示出这个时间。小蓝不用显示出年、月、日，只需要显示出时、分、秒即可，毫秒也不用显示，直接舍去。给定一个用整数表示的时间，请将这个时间对应的时、分、秒输出。

输入：输入一个正整数表示时间，时间不超过 10^{18}。

输出：输出用时、分、秒表示的当前时间，格式形如 HH:MM:SS，其中 HH 表示时，值为 0～23；MM 表示分，值为 0～59；SS 表示秒，值为 0～59。时、分、秒不足两位时补前导 0。

输入样例：	输出样例：
46800999	13:00:00

这道题是 Java 日期功能的简单应用。代码如下：

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        long n = sc.nextLong();          //需要 long,int 不够
        LocalDateTime a = LocalDateTime.of(1970, 1, 1, 0, 0);
        Duration d = Duration.ofMillis(1);
        LocalDateTime b = a.plus(d.multipliedBy(n));
        DateTimeFormatter f = DateTimeFormatter.ofPattern("HH:mm:ss");
        String formattedResult = b.format(f);
        System.out.println(formattedResult);
```

```
    }
}
```

再看一道例题。

例 3.4　2012 年第三届国赛 星期几 lanqiaoOJ 729

问题描述：本题为填空题。1949 年的国庆节是星期六，2012 年的国庆节是星期一，那么从中华人民共和国成立到 2012 年有几次国庆节正好是星期日？

这种简单的日期问题，用 Java 可以直接求解。

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        int cnt = 0;
        for (int i = 1949; i <= 2013; i++) {
            LocalDate a = LocalDate.of(i, 10, 1);
            if (a.getDayOfWeek() == DayOfWeek.SUNDAY)
                cnt++;
        }
        System.out.println(cnt);
    }
}
```

【练习题】

lanqiaoOJ：高斯日记 711、星系炸弹 670、日期问题 103、第几天 614、回文日期 498、跑步锻炼 597、航班时间 175、特殊时间 2119、日期统计 3492。

3.1.4　Set 和 Map

当题目需要对数据去重时，可以用 Set 和 Map。

1. Set

Java 中的 Set 是一种集合，它用于存储不重复的元素。Java 提供了多个 Set 的实现类，有 HashSet、TreeSet、LinkedHashSet。不同的实现类可能具有不同的性能特点和迭代顺序，具体选择哪个实现类取决于需求和场景。

HashSet 基于哈希表，元素无序且唯一，它提供了 O(1)时间复杂度的常数时间查找、插入和删除操作。

TreeSet 基于红黑树，元素按照自然排序或指定的 Comparator 进行排序。它提供了 $O(\log_2 n)$时间复杂度的有序操作。

LinkedHashSet 基于哈希表和链表，按照元素的插入顺序来遍历元素，即元素的遍历顺序与插入顺序一致。通过使用一个双向链表来实现，链表中的元素按照插入的先后顺序连接在一起。它提供了 O(1)时间复杂度的插入和删除操作，O(n)时间复杂度的查找操作。

Set 的特点如下：

(1) 无序性。Set 中的元素是无序的，即元素没有固定的位置或顺序。

(2) 唯一性。Set 中不允许有重复的元素,每个元素在 Set 中只能出现一次。当向 Set 中添加重复的元素时,添加操作将被忽略。

(3) 不支持索引访问。Set 不提供通过索引访问元素的方法,因为元素在 Set 中没有固定的位置。

(4) 高效的查找和插入操作。Set 的实现类通常通过哈希表或红黑树实现,这使得查找和插入操作非常高效。查找和插入操作的时间复杂度通常是 O(1)或 $O(\log_2 n)$。

(5) 可用于去重。Set 不允许重复元素存在,因此常被用来进行去重操作。通过将元素添加到 Set 中,可以快速地判断某个元素是否已经存在。

Set 的常用方法如表 3.3 所示。

表 3.3 Set 的常用方法

方 法	说 明
boolean add(E element)	向 Set 中添加指定元素,如果元素已经存在,则不添加,返回 false
boolean remove(Object element)	从 Set 中移除指定元素,如果元素存在并成功移除,则返回 true,否则返回 false
boolean contains(Object element)	判断 Set 中是否包含指定元素,如果包含,则返回 true,否则返回 false
int size()	返回 Set 中元素的数量
boolean isEmpty()	判断 Set 是否为空,如果为空,则返回 true,否则返回 false
void clear()	清空 Set 中的所有元素
Iterator<E> iterator()	返回一个用于遍历 Set 中元素的迭代器
boolean containsAll(Collection<?> collection)	判断 Set 是否包含指定集合中的所有元素,如果是,则返回 true,否则返回 false
boolean addAll(Collection<? extends E> collection)	将指定集合中的所有元素添加到 Set 中,如果 Set 发生了改变,则返回 true,否则返回 false

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。注意,Java 中的 Set 没有排序功能,遍历 Set 输出的结果不一定有序,这和 C++ STL 中的 set 不同。

下面是 Set 应用的例子,用 contains()判断 Set 中是否有某个元素,见第 17、18 行代码。

```
1  import java.util.*;
2  public class Main {
3      static Set<Integer> st = new HashSet<>(Arrays.asList(5, 9, 2, 3));
4  //定义 set,赋初值
5      static void out() {                    //输出 set 的所有元素
6          for (int num : st) System.out.print(num + " ");
7          System.out.println();
8      }
9      public static void main(String[] args) {
10         out();                             //打印 set 的元素,不一定是有序的。输出:2 3 5 9
11         st.add(9);                         //插入重复数字 9
12         System.out.println(st.size());         //set 元素的数量.输出:4
13         out();                             //重复元素 9 被去重,输出:2 3 5 9
14         if (st.contains(7)) System.out.println(st.contains(7));     //无输出
15         else System.out.println("not find");                        //输出:not find
16         st.remove(3);                                               //删除
17         if (st.contains(5)) System.out.println("find 5");           //输出:find 5
18         if (st.contains(7)) System.out.println("find 7");           //无输出
```

```
19          st.clear();                                          //清空元素
20          if (st.isEmpty()) System.out.println("empty");       //输出:empty
21      }
22  }
```

2. Map

Java 中的 Map 是一种用键值对存储的数据结构,它提供了一种快速查找和访问数据的方式,每个键对应一个唯一的值。

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。

Map 常用的实现类有 HashMap、TreeMap 和 LinkedHashMap。

HashMap 基于哈希表,提供了快速的插入和查找操作。它不保证元素的顺序,允许使用 null 键和 null 值。

TreeMap 基于红黑树,提供了有序的键值对集合。它根据键的自然顺序或者自定义比较器进行排序。

LinkedHashMap 基于哈希表和双向链表,在 HashMap 的基础上维护了元素的插入顺序。它允许使用 null 键和 null 值,并且可以按照插入顺序或者访问顺序进行迭代。

Map 接口提供了丰富的方法来操作键值对,例如 put(key,value)添加键值对、get(key)获取键对应的值、containsKey(key)判断是否包含指定的键等。

需要注意的是,Map 中的键是唯一的,如果插入相同的键,则会覆盖之前的键值对。值可以重复。此外,在使用 Map 时需要注意选择适合自己需求的实现类,以及根据具体场景选择合适的方法来操作数据。

表 3.4 中列出 Map 的常用方法。

表 3.4　Map 的常用方法

方　法	功　能
put(key,value)	将指定的键值对添加到 Map 中,如果键已经存在,则覆盖之前的值
get(key)	返回指定键对应的值,如果键不存在,则返回 null
containsKey(key)	判断 Map 中是否包含指定的键
containsValue(value)	判断 Map 中是否包含指定的值
keySet()	返回所有键的集合
values()	返回所有值的集合
remove(key)	从 Map 中删除指定的键及其对应的值
clear()	清空 Map 中的所有键值对
size()	返回 Map 中键值对的数量
isEmpty()	判断 Map 是否为空

下面是 Map 应用的例子。

```
1   import java.util.HashMap;
2   import java.util.Map;
3   public class Main {
4       public static void out(Map<Integer, String> mp) {
5           for (Map.Entry<Integer, String> entry : mp.entrySet())
```

```
            System.out.print(entry.getKey() + " " + entry.getValue() + "; ");
        System.out.println();
    }
    public static void main(String[] args) {
        Map<Integer, String> mp = new HashMap<>();
        mp.put(7, "tom"); mp.put(2, "Joy"); mp.put(3, "Rose");
        out(mp);                    //输出:2 Joy; 3 Rose; 7 tom;
        System.out.println("size = " + mp.size());     //输出:size = 3
        mp.put(3, "Luo");           //修改了 mp[3]的值。键是唯一的,不能改,值可以改
        mp.put(5, "Wang");
        mp.put(9, "Hu");
        out(mp);                    //输出:2 Joy; 3 Luo; 5 Wang; 7 tom; 9 Hu;
        mp.remove(5);
        out(mp);                    //输出:2 Joy; 3 Luo; 7 tom; 9 Hu;
        String value = mp.get(3);//查询
        if (value != null) System.out.println("3 " + value);     //输出:3 Luo
        else System.out.println("not find");                     //无输出
    }
}
```

下面给出一道例题,分别用 Map 和 Set 实现。

例 3.5 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得了科技创新奖,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%的数据,$0 \leqslant n, m \leqslant 1000$,获得奖项的人的编号$<2\times10^9$;对于 100%的数据,$0 \leqslant n, m \leqslant 10^5$,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。

本题查询 n 和 m 个数中哪些是重的,检查 m 个数中的每个数,如果它在 n 个数中出现过,则说明获得了两个奖项。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 个数中出现过,则输出。那么代码的计算复杂度是多少? Map 的每次操作是 $O(\log_2 m)$,第 14~16 行的复杂度是 $O(m\log_2 m)$,第 18~20 行的复杂度是 $O(n\log_2 m)$,所以总计算复杂度是 $O(m\log_2 m+n\log_2 m)$。

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Map<Integer, Boolean> mp = new HashMap<>();
        int[] a = new int[101000];
        int[] b = new int[101000];
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        for (int i = 1; i <= n; i++)
            a[i] = sc.nextInt();
        for (int i = 1; i <= m; i++) {
            b[i] = sc.nextInt();
            mp.put(b[i], true);
        }
        for (int i = 1; i <= n; i++)
            if (mp.containsKey(a[i]))
                System.out.print(a[i] + " ");
    }
}
```

（2）用 Set 实现。

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Set<Integer> set = new HashSet<>();
        int n = sc.nextInt();
        int m = sc.nextInt();
        int[] a = new int[n + 1];
        int[] b = new int[m + 1];
        for (int i = 1; i <= n; i++)
            a[i] = sc.nextInt();
        for (int i = 1; i <= m; i++) {
            b[i] = sc.nextInt();
            set.add(b[i]);
        }
        for (int i = 1; i <= n; i++)
            if (set.contains(a[i]))
                System.out.print(a[i] + " ");
    }
}
```

扫一扫

视频讲解

3.2 数　组

数组是最简单的数据结构，下面举例说明数组的应用。

1. 一维数组

定义一个数组 a[]，第一个元素是从 a[0]开始的，不是从 a[1]开始。不过，有些题目从 a[1]开始更符合逻辑。

用下面的例题说明一维数组的应用。

例 3.6　2022 年第十三届蓝桥杯省赛 Java 大学 C 组试题 F：选数异或 lanqiaoOJ 2081

时间限制：1s　**内存限制**：512MB　**本题总分**：15 分

问题描述：给定一个长度为 n 的数列 $A_1,A_2,\cdots,A_n$ 和一个非负整数 x，给定 m 次查询，每次询问能否从某个区间[l,r]中选择两个数使得它们的异或等于 x。

输入：输入的第一行包含 3 个整数 n、m、x；第二行包含 n 个整数 A_1、A_2、…、A_n；接下来 m 行，每行包含两个整数 l_i、r_i，表示询问区间$[l_i,r_i]$。

输出：对于每个询问，如果该区间内存在两个数的异或为 x，则输出 yes，否则输出 no。

输入样例：	输出样例：
4 4 1 1 2 3 4 1 4 1 2 2 3 3 3	yes no yes no

评测用例规模与约定：对于 20%的评测用例，$1\leqslant n,m\leqslant 100$；对于 40%的评测用例，$1\leqslant n,m\leqslant 1000$；对于所有评测用例，$1\leqslant n,m\leqslant 100000$，$0\leqslant x<220$，$1\leqslant l_i\leqslant r_i\leqslant n$，$0\leqslant A_i<220$。

这里用暴力法做：对每个查询，验算区间内两个数的异或，计算复杂度为 $O(n^2)$，共 m 个查询，总复杂度为 $O(mn^2)$，只能通过 40%的测试。

```
1  import java.util.Scanner;
2  public class Main {
3      public static void main(String[] args) {
4          Scanner sc = new Scanner(System.in);
5          int n = sc.nextInt();
6          int m = sc.nextInt();
7          int x = sc.nextInt();
8          int[] a = new int[n + 1];
9          for (int i = 1; i <= n; i++)
10             a[i] = sc.nextInt();
11         for (int i = 0; i < m; i++) {
12             int flag = 0;
13             int L = sc.nextInt();
14             int R = sc.nextInt();
15             for (int j = L; j < R; j++)
16                 for (int k = j + 1; k <= R; k++)
17                     if ((a[j] ^ a[k]) == x)
18                         flag = 1;
19             if (flag == 1) System.out.println("yes");
20             else System.out.println("no");
21         }
22     }
23 }
```

2. 二维数组

用下面的例题说明二维数组的定义和应用。

例 3.7　2023 年第十四届蓝桥杯省赛 Java 大学 C 组试题 G：子矩阵 lanqiaoOJ 3521

时间限制：5s　**内存限制**：512MB　**本题总分**：20 分

问题描述：给定一个 n×m(n 行 m 列)的矩阵。设一个矩阵的价值为其所有数中最大值和最小值的乘积。求给定矩阵中所有大小为 a×b(a 行 b 列)的子矩阵的价值的和。答案可能很大，只需要输出答案对 998244353 取模后的结果。

输入：输入的第一行包含 4 个整数，分别表示 n、m、a、b，相邻整数之间使用一个空格分隔；接下来 n 行，每行包含 m 个整数，相邻整数之间使用一个空格分隔，表示矩阵中的每个数 $A_{i,j}$。

输出：输出一个整数，代表答案。

输入样例：	输出样例：
2 3 1 2 1 2 3 4 5 6	58

评测用例规模与约定：对于 40%的评测用例，$1 \leqslant n, m \leqslant 100$；对于 70%的评测用例，$1 \leqslant n, m \leqslant 500$；对于所有评测用例，$1 \leqslant a \leqslant n \leqslant 1000$，$1 \leqslant b \leqslant m \leqslant 1000$，$1 \leqslant A_{i,j} \leqslant 10^9$。

本题 70%和 100%的测试需要高级算法。这里只给出能通过 40%测试的简单代码，该代码直接模拟了题目的要求。

```
1  import java.util.Scanner;
2  public class Main {
3      public static void main(String[] args) {
4          Scanner sc = new Scanner(System.in);
5          int n = sc.nextInt();
6          int m = sc.nextInt();
7          int a = sc.nextInt();
8          int b = sc.nextInt();
9          int[][] A = new int[n][m];
10         for (int i = 0; i < n; i++)
11             for (int j = 0; j < m; j++)
12                 A[i][j] = sc.nextInt();
13         int ans = 0;
14         for (int i = 0; i < n - a + 1; i++) {
15             for (int j = 0; j < m - b + 1; j++) {
16                 int Zmax = A[i][j];
17                 int Zmin = A[i][j];
18                 for (int u = 0; u < a; u++) {
19                     for (int v = 0; v < b; v++) {
20                         Zmax = Math.max(Zmax, A[i + u][j + v]);
21                         Zmin = Math.min(Zmin, A[i + u][j + v]);
22                     }
```

```
                }
                ans = (ans + Zmax * Zmin) % 998244353;
            }
        }
        System.out.println(ans);
    }
}
```

3.3 链　表

扫一扫 视频讲解

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使用。这一般发生在较大的工程软件中，在竞赛中不必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填补这个空位，产生了大量的复制开销，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，也可以是不连续的。

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整个链表，让首尾相接，尾部 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。

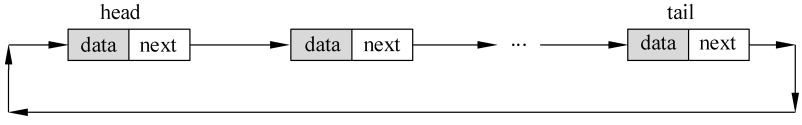


图 3.2　单向链表

双向链表是对单向链表的优化，如图 3.3 所示。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。

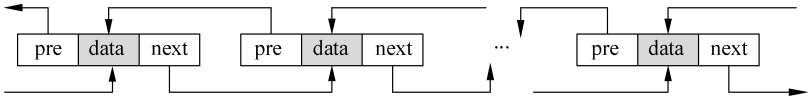


图 3.3　双向链表

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、查找和释放等。

与数组相比，链表的优点是删除和插入很快，例如删除，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。

链表仍然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，计算量是 O(n)。

在参加算法竞赛时，参赛者虽然可以自己手写链表，但是为了加快编码速度，一般直接使用 LinkedList 来实现链表的功能。

Java 的 LinkedList 是一种双向链表数据结构，实现了 List 和 Deque 接口。它可以用来存储任意类型的对象，并提供了多种操作方法对链表进行增、删、改、查操作。

LinkedList 的特点如下：

(1) 链表结构。LinkedList 通过节点之间的链接(引用)来组织数据，每个节点都包含一个存储元素以及指向前一个和后一个节点的引用。

(2) 双向遍历。LinkedList 支持双向遍历，可以通过 getFirst()和 getLast()方法分别获取链表的第一个和最后一个元素，也可以使用 get(index)方法按索引访问元素。

(3) 插入和删除。LinkedList 提供了多种插入和删除元素的方法，例如 addFirst()、addLast()、add(index,element)、removeFirst()、removeLast()、remove(index)等。

(4) 队列和栈操作。LinkedList 实现了 Deque 接口，可以用作队列和栈。用户可以使用 addLast()和 removeFirst()方法实现队列的先进先出(FIFO)操作，使用 addLast()和 removeFirst()方法实现栈的后进先出(LIFO)操作。

(5) 迭代器支持。LinkedList 提供了 ListIterator 接口的实现，可以使用迭代器遍历链表，并在遍历过程中进行插入、删除和修改操作。

LinkedList 的使用场景包括需要频繁插入和删除元素、需要双向遍历、需要实现队列或栈等。

LinkedList 的常用方法如表 3.5 所示。

表 3.5　LinkedList 的常用方法

方　法	说　明
add()	将元素添加到链表的末尾
addFirst()	将元素添加到链表的开头
addLast()	将元素添加到链表的末尾
add(int index,E element)	将元素插到指定索引位置
getFirst()	返回链表的第一个元素
getLast()	返回链表的最后一个元素
get(int index)	返回指定索引位置的元素
remove()	删除并返回链表的指定位置元素，如果无参数，则删除第一个
removeFirst()	删除并返回链表的第一个元素
removeLast()	删除并返回链表的最后一个元素
remove(int index)	删除指定索引位置的元素
size()	返回链表中元素的数量

续表

方　法	说　明
isEmpty()	判断链表是否为空
clear()	清空链表,将所有元素移除

下面的代码演示了部分操作。

```
1  import java.util.LinkedList;
2  public class Main {
3      public static void main(String[] args) {
4          LinkedList<String> lst = new LinkedList<>();
5          //添加元素
6          lst.add("tom"); lst.addFirst("rose"); lst.addLast("joy");
7          //遍历链表
8          for (String i : lst)
9              System.out.println(i);                    //分3行打印:rose tom joy
10         //获取元素
11         String first = lst.getFirst();
12         String last = lst.getLast();
13         String second = lst.get(1);
14         System.out.println("First: " + first);        //打印:First: rose
15         System.out.println("Last: " + last);          //打印:Last: joy
16         System.out.println("Second: " + second);      //打印:Second: tom
17         //删除元素
18         lst.removeFirst(); lst.removeLast(); lst.remove(0);
19         //判断链表是否为空
20         boolean isEmpty = lst.isEmpty();
21         System.out.println("Is empty: " + isEmpty);   //打印 Is empty: true
22     }
23 }
```

用 LinkedList 写代码很简短。下面用一个简单题说明链表的应用。

例 3.8　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,但小王子还是太小了,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 的玩具放到最前面,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;接下来 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：输出共 M 行,第 i 行输出小王子第 i 次排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[0]～toy[9]这10个节点。toy[0]始终是链表的头。下面给出代码。

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Integer> toy = new LinkedList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
        10));                                                    //定义链表
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        while (m > 0) {
            int x = sc.nextInt();
            toy.remove((Integer) x);                             //删除链表中的x
            toy.add(0, x);                                       //把x插到链表的头
            for (int i : toy) System.out.print(i + " ");         //输出链表
            System.out.println();
            m--;
        }
    }
}
```

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，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，后续有m次操作；第二行到第m+1行，每行3个数字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，下面直接给出代码。

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        List<Integer> lis = new ArrayList<>();
        for (int i = 1; i <= n; i++) lis.add(i);                 //lis={1,2,3,…,n}
        while (m-- > 0) {
            int x = sc.nextInt();
```

```
11              int y = sc.nextInt();
12              int z = sc.nextInt();
13              lis.remove(Integer.valueOf(x));              //删除 x
14              int idx = lis.indexOf(y);                    //找到 y
15              if (z == 0) lis.add(idx + 1, x);             //x 放在 y 的后面
16              if (z == 1) lis.add(idx, x);                 //x 放在 y 的前面
17          }
18          for (int x : lis) System.out.print(x + " ");
19          System.out.println();
20      }
21  }
```

【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。队列在人们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。

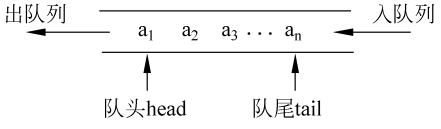


图 3.4　队列

图 3.4 所示为队列的原理，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，只能从队尾进入队列。

Java 用 LinkedList 实现基本队列 Queue[①]，常用方法如表 3.6 所示。

表 3.6　Java 用 LinkedList 实现基本队列的常用方法

方　法	说　明
add()	将指定元素插到队列的尾部
offer()	将指定元素插到队列的尾部
remove()	移除并返回队列的头部，如果队列为空，抛出 NoSuchElementException 异常
poll()	移除并返回队列的头部，如果队列为空，返回 null
element()	返回队列的头部但不移除，如果队列为空，抛出 NoSuchElementException 异常
peek()	返回队列的头部但不移除，如果队列为空，返回 null
size()	返回元素的个数
isEmpty()	检查队列是否为空

下面用一个例子说明 Queue 的应用。

① Queue 的文档：https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Queue.html

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

问题描述：有 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 13 行把队头移到队尾，第 17 行让数到 m 的人离开队伍。

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3  import java.util.Scanner;
4  public class Main {
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7          int n = sc.nextInt();
8          int m = sc.nextInt();
9          Queue<Integer> q = new LinkedList<>();
10         for (int i = 1; i <= n; i++) q.offer(i);
11         while (!q.isEmpty()) {
12             for (int i = 1; i < m; i++) {
13                 q.offer(q.peek());                    //读队头，重新排到队尾
14                 q.poll();
15             }
16             System.out.print(q.peek() + " ");
17             q.poll();                                 //第 m 个人离开队伍
18         }
19     }
20 }
```

再看一道例题。

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，他经常用这个软件翻译英语文章。

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。当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那个单词，腾出单元，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。

```
import java.util.*;
public class Main {
    static boolean[] hashtable = new boolean[1003];
    static Queue<Integer> q = new LinkedList<>(); //队列
    public static void main(String[] args) {
        int m, n;
        Scanner sc = new Scanner(System.in);
        m = sc.nextInt();
        n = sc.nextInt();
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int x = sc.nextInt();
            if (hashtable[x] == false) {
                hashtable[x] = true;
                if (q.size() < m)
                    q.add(x);              //用 add 方法添加元素到队列中
                else {
                    hashtable[q.poll()] = false;
                    //用 poll 方法取出队列头部的元素并移除
                    q.add(x);
                }
                ans++;
            }
        }
        System.out.println(ans);
    }
}
```

【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。

3.5 优先队列

扫一扫 视频讲解

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）。在队头弹出之后，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一个；如果有人刚进去，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，要想找到它，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 PriorityQueue，初学者只需要学会如何使用它即可。

PriorityQueue 是一种特殊的队列，其中的元素按照优先级进行排序。在优先队列中，每个元素都有一个与之相关联的优先级。具有较高优先级的元素在队列中排在前面，而较低优先级的元素排在后面。

PriorityQueue 的常用方法如表 3.7 所示。

表 3.7　PriorityQueue 的常用方法

方　法	说　明
add()、offer()	将元素添加到队列中
remove()、poll()	移除并返回队列中的第一个元素
peek()	返回队列中的第一个元素，但不移除它
isEmpty()	判断队列是否为空
size()	返回队列中的元素个数

下面用一个例题说明优先队列的应用。

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，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，代码的解析见注释。

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

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<Long> set = new HashSet<>();  //判重
        set.add(1L);                      //第1个丑数是1
        PriorityQueue<Long> pq = new PriorityQueue<>();
//队列中是新生成的丑数
        pq.offer(1L);                     //第1个丑数是1,进入队列
        int n = 20220;
        long ans = 0;
        int[] prime = {3,7,17,29,53};
        for(int i = 1; i <= n; i++) {     //从队列中按从小到大取出20220个丑数
            long now = pq.poll();
            ans = now;          //把队列中最小的值取出来,它也是已经取出的最大的值
            for(int j = 0; j < 5; j++) {  //5个素数
                long tmp = now * prime[j]; //从已取出的最大值开始乘以5个素数
                if(!set.contains(tmp)) {   //tmp这个数没有出现过
                    set.add(tmp);          //放到set里面
                    pq.offer(tmp);         //把tmp放进队列
                }
            }
        }
        System.out.println(ans);
    }
}
```

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不能有相同数字。输出任意一种分法即可。

输入：输入的第一行为正整数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为可以分牌的人数，要保证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”。用隔板隔出 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；再把数量第二多的数，每个隔板放一个；类似操作，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}。

那么如何编码？下面用优先队列编程。第 16 行用二元组{num[i],i}表示每个数的数量和数字。优先队列会把每个数按数量从多到少弹出，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 poll 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(logn)，共 n 个数，总复杂度为 O(nlogn)。

```
1  import java.util.*;
2  public class Main {
3      static final int N = 1000010;
4      static int[] num = new int[N];
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7          int n = sc.nextInt();
8          int k = sc.nextInt();
9          PriorityQueue<int[]> q = new PriorityQueue<>((a, b) -> b[0] - a[0]);
10         for (int i = 0; i < n; i++) {
11             int x = sc.nextInt();
12             num[x]++;                          //x 这个数有 num[x]个
13         }
14         for (int i = 0; i < N; i++)
15             if (num[i] > 0)
16                 q.offer(new int[] {num[i], i});          //数 i 的个数以及数 i
17         while (q.size() >= k) {                //队列中数量大于 k,说明还够用
18             List<int[]> tmp = new ArrayList<>();
19             for (int i = 0; i < k; i++)        //拿 k 个数出来,且 k 个数不同,这是一份
20                 tmp.add(q.poll());             //先出来的是 num[]最大的
21             for (int i = 0; i < k; i++) {      //打印一份,共 k 个数
22                 System.out.print(tmp.get(i)[1] + " ");
23                 tmp.get(i)[0]--;               //这个数用了一次,减去 1
24                 if (tmp.get(i)[0] > 0)
25                 q.offer(tmp.get(i));           //没用完,再次进队列
26             }
27             System.out.println();
28         }
29     }
30 }
```

【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。

Stack[①] 的常用方法如表 3.8 所示。

表 3.8 Stack 的常用方法

方 法	说 明
push(item)	把 item 放到栈顶
pop()	把栈顶元素弹出,并返回该元素
peek()	返回栈顶元素,但不弹出
empty()	检查栈是否为空,如果为空,返回 true,否则返回 false

下面用一个例子给出栈的应用。

例 3.14 表达式括号的匹配 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(+、-、*、/)和左右圆(小)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“)(()”这样的是非法的。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,遍历字符串中的每一个字符,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。

① https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Stack.html

(3) 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。

下面是代码。

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Stack<Character> st = new Stack<Character>();
        String s = sc.next();
        for (int i = 0; i < s.length(); i++) {
            char x = s.charAt(i);
            if (x == '@') break;
            if (x == '(') st.push(x);
            if (x == ')') {
                if(st.empty()) {
                    System.out.println("NO");
                    return;
                }
                else st.pop();
            }
        }
        if (st.empty()) System.out.println("YES");
        else System.out.println("NO");
    }
}
```

再看一道例题。

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: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”也会形成新的“凹”,条件是这个“5”必须比中间的“1-2”大才可以。

总结上述过程：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”。

以上方法是检查一个“凹”的两头，还有一种方法是“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适合。

以“6 4 2 8”为例，用栈模拟找“凹”，如图 3.5 所示。当新的数比栈顶的数小时就进栈；如果比栈顶的数大就出栈，此时找到了一个“凹”，并计算价值。该图中圆圈内的数字是数在数组中的下标位置，用于计算题目要求的价值。

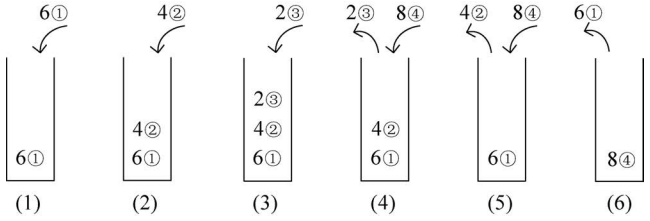


图 3.5 用栈统计“凹”

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，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。

最后统计出“6 4 2 8”的总价值是 3＋4＋6＝13。

下面是代码。

```
1  import java.util.Scanner;
2  import java.util.Stack;
3  public class Main {
4      static final int N = 300008;
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7          int n = sc.nextInt();
8          int[] a = new int[N];
9          for(int i = 1; i <= n; i++) a[i] = sc.nextInt();
10         Stack<Integer> st = new Stack<>();
11         long ans = 0;
12         for(int i = 1; i <= n; i++) {
13             while(!st.empty() && a[st.peek()] < a[i]) {
14                 st.pop();
15                 if(!st.empty()) {
16                     int last = st.peek();
17                     ans += (long)(i - last + 1);
```

```
                    }
                }
                st.push(i);
            }
            ans += (n - 1) * 2;
            System.out.println(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。

扫一扫

视频讲解

3.7 二 叉 树

前几节介绍的数据结构数组、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。二叉树的每一层的节点数以 2 的倍数递增，所以二叉树的第 k 层最多有 2^{k-1} 个节点。根据每一层节点的分布情况，二叉树分为以下常见类型。

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，节点一共有 $1+2+4+\cdots+2^{n-1}=2^n-1$ 个。

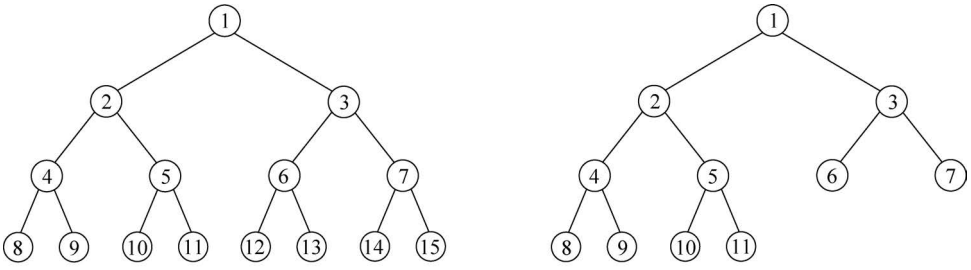
2. 完全二叉树

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，称之为完全二叉树。

图 3.6 演示了一棵满二叉树和一棵完全二叉树。

3. 平衡二叉树

任意左子树和右子树的高度差不大于 1，该树称为平衡二叉树。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。



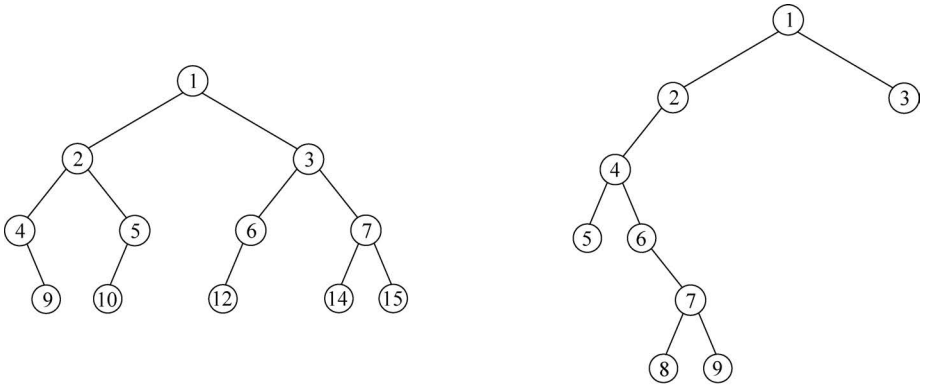
(a) 满二叉树　　(b) 完全二叉树

图 3.6　满二叉树和完全二叉树

4. 退化二叉树[①]

如果树上的每个节点都只有一个孩子，称之为退化二叉树。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，也将该树看成退化二叉树。

图 3.7 演示了一棵平衡二叉树和一棵退化二叉树。



(a) 平衡二叉树　　(b) 退化二叉树

图 3.7　平衡二叉树和退化二叉树

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，下面列出二叉树的一些优势。

（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、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，这是 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。

① 本书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。"

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，首先要定义和存储它的节点。

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

在算法竞赛中一般用类来定义二叉树。下面定义一个大小为 N 的类。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，不算多。

```
1  class Node {                    //静态二叉树
2      int value;                  //可以把 value 简写为 v
3      int lson, rson;             //左孩子和右孩子，可以把 lson、rson 简写为 ls、rs
4  }
5  Node[ ] tree = new Node[N];     //可以把 tree 简写为 t
```

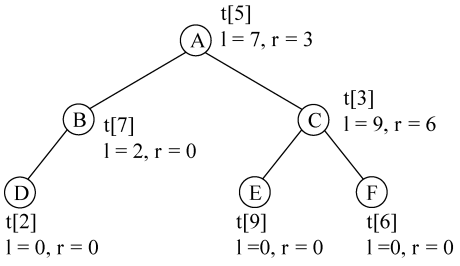


图 3.8 二叉树的静态存储

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

图 3.8 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。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，就把它的左孩子和右孩子均赋值为 0。

2. 二叉树存储的编码实现

下面用代码演示图 3.8 中二叉树的建立，并输出二叉树。

第 17～22 行建立二叉树，然后用 print_tree()输出二叉树。

```
1  import java.util.*;
2  class Main {
3      static class Node {
4          char v;
5          int ls, rs;
6      }
7      static final int N = 100;
8      static Node[ ] t = new Node[N];
9      static void print_tree(int u) {
10         if (u != 0) {
11             System.out.print(t[u].v + " ");
12             print_tree(t[u].ls);
13             print_tree(t[u].rs);
14         }
15     }
16     public static void main(String[ ] args) {
17         t[5] = new Node(); t[5].v = 'A'; t[5].ls = 7; t[5].rs = 3;
18         t[7] = new Node(); t[7].v = 'B'; t[7].ls = 2; t[7].rs = 0;
```

```
        t[3] = new Node(); t[3].v = 'C'; t[3].ls = 9; t[3].rs = 6;
        t[2] = new Node(); t[2].v = 'D';
        t[9] = new Node(); t[9].v = 'E';
        t[6] = new Node(); t[6].v = 'F';
        int root = 5;
        print_tree(5);          //输出: A B D C E F
    }
}
```

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 3.8 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 3.7.3 节讲解。

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,甚至 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,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 2p>k,那么 p 没有孩子;如果 2p+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 2p+1。

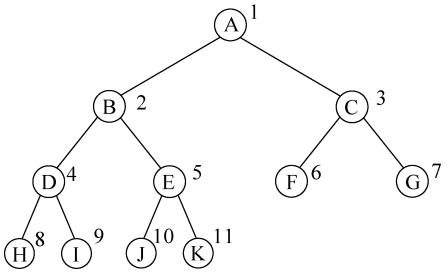


图 3.9 一棵完全二叉树

如图 3.9 所示,图中圆圈内的内容是节点的值,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。

```
import java.util.Arrays;
public class Main {
    static int ls(int p){ return p << 1;}
    static int rs(int p){ return p << 1 | 1;}
    public static void main(String[] args) {
        final int N = 100;
        char[] t = new char[N];
        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';
        System.out.print(t[1] + ":lson = " + t[ls(1)] + " rson = " + t[rs(1)]);
```

```
12  //输出: A:lson = B rson = C
13          System.out.println();
14          System.out.print(t[5] + ":lson = " + t[ls(5)] + " rson = " + t[rs(5)]);
15  //输出: E:lson = J rson = K
16      }
17  }
```

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,最基础的操作是遍历(搜索)二叉树的每个节点,有先序遍历、中序遍历和后序遍历。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。

如图 3.10 所示为一个二叉树例子。

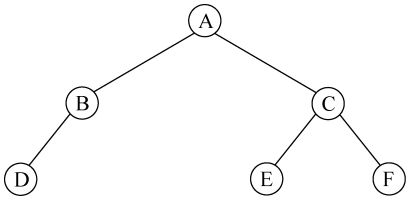


图 3.10 二叉树的例子

(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。图 3.10 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。图 3.10 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。图 3.10 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 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,依次求出它的前序、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 和 r,分别表示节点 i 的左右子节点的编号。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。第一行是这棵二叉树的前序遍历，第二行是这棵二叉树的中序遍历，第三行是这棵二叉树的后序遍历。

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，包括 3 种遍历。

```
import java.util.*;
class Main {
    static class Node {
        int v, ls, rs;
        Node(int v, int ls, int rs) {
            this.v = v; this.ls = ls; this.rs = rs;
        }
    }
    static final int N = 100005;
    static Node[] t = new Node[N];
    static void preorder(int p, StringJoiner joiner) {
        if (p != 0) {
            joiner.add(t[p].v + "");
            preorder(t[p].ls,joiner);
            preorder(t[p].rs,joiner);
        }
    }
    static void midorder(int p, StringJoiner joiner) {
        if (p != 0) {
            midorder(t[p].ls,joiner);
            joiner.add(t[p].v + "");
            midorder(t[p].rs,joiner);
        }
    }
    static void postorder(int p, StringJoiner joiner) {
        if (p != 0) {
            postorder(t[p].ls,joiner);
            postorder(t[p].rs,joiner);
            joiner.add(t[p].v + "");
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            int a = sc.nextInt(), b = sc.nextInt();
            t[i] = new Node(i, a, b);
        }
```

```
39          StringJoiner joiner = new StringJoiner(" ");
40          preorder(1, joiner); System.out.println(joiner);
41          joiner = new StringJoiner(" ");
42          midorder(1, joiner); System.out.println(joiner);
43          joiner = new StringJoiner(" ");
44          postorder(1, joiner); System.out.println(joiner);
45      }
46  }
```

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 3.11 是一棵拥有 11 个节点的完全三叉树。

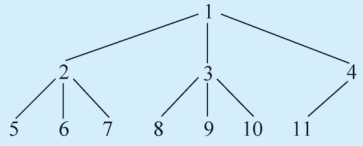


图 3.11　一棵拥有 11 个节点的完全三叉树

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：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 74 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 个节点前面有 u−1 个节点，每个节点各有 m 个孩子，再加上 1 号节点，可得第 u 个节点的左孩子的下标为(u−1)×m+2。例如图 3.11 中的 3 号节点，求它的最左孩子的编号。3 号节点前面有两个节点，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，则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 不在最后一层，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 不在最后一层，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 将在第 6 章讲解，请读者在学过第 6 章以后再看这个方法。

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

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

```
import java.util.Scanner;
public class Main {
    public static long dfs(long n, long m, long u) {
        long ans = 1;          //u 自己算一个,需要用 long,下面的 m * u 可能超过 int
        if (m * u - (m - 2) > n) return 1;     //情况(1),u 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 (long 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
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        while (t-- > 0) {
            long n = sc.nextLong();
            long m = sc.nextLong();
            long k = sc.nextLong();
            System.out.println(dfs(n, m, k));
        }
    }
}
```

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

其实每一层计算一次即可，在情况(2)时每一层也只需要计算一次。以图 3.11 为例，计算以节点 1 为根的树的节点数量。1 号节点这一层有一个节点；其下一层是满的，有 3 个节点，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 个节点。累加得 1＋3＋7＝11。

那么计算量是多少？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。

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        while (t-- > 0) {
            long n = sc.nextLong();
            long m = sc.nextLong();
            long k = sc.nextLong();
            long ans = 1;
//k 节点自己算一个,注意用 long
            long ls = k, rs = k;     //从 k 节点开始,分析它的最左和最右孩子
            while (true) {           //从 k 节点开始,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),该层是满的,累加该层的所有孩子
            }
            System.out.println(ans);
        }
    }
}
```

再看一道例题。

例 3.18　FBI 树 lanqiaoOJ 571

问题描述：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，它的节点包括 F 节点、B 节点和 I 节点 3 种类型。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：输入的第一行是一个整数 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储题目的 FBI 树，满二叉树用静态数组实现。当 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。

```
import java.util.Scanner;
public class Main {
    static char[] s = new char[1100];
    static char[] tree = new char[4400];                    //tree[]存满二叉树
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        String input = sc.next();
        s = input.toCharArray();
        buildFBI(1, 0, s.length-1);
        postorder(1);
    }
    public static int ls(int p) {return p << 1;}            //定位左儿子:p*2
    public static int rs(int p) {return p << 1 | 1;}        //定位右儿子:p*2 + 1
    public static void buildFBI(int p, int left, int right) {
        if (left == right) {                                //到达叶节点
            if (s[left] == '1') tree[p] = 'I';
            else tree[p] = 'B';
            return;
        }
        int mid = (left + right) / 2;                       //分成两半
        buildFBI(ls(p), left, mid);                         //递归左半
        buildFBI(rs(p), mid + 1, right);                    //递归右半
        if (tree[ls(p)] == 'B' && tree[rs(p)] == 'B')
            tree[p] = 'B';                                  //左右儿子是B,自己也是B
        else if (tree[ls(p)] == 'I' && tree[rs(p)] == 'I')
            tree[p] = 'I';                                  //左右儿子是I,自己也是I
        else tree[p] = 'F';
    }
    public static void postorder(int p) {                   //后序遍历
        if (tree[ls(p)] != 0) postorder(ls(p));
        if (tree[rs(p)] != 0) postorder(rs(p));
        System.out.print(tree[p]);
    }
}
```

【练习题】

lanqiaoOJ：完全二叉树的权值 183。

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

3.8 并 查 集

扫一扫

视频讲解

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”。其实意译为“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、容易与其他数据结构和算法结合。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，不仅代码简单，而且计算复杂度几乎是 O(1)，效率极高。

并查集的效率高，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基本操作，包括初始化、合并和查找。

1. 初始化

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 3.12 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。

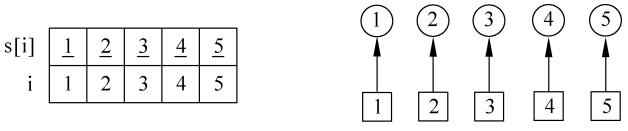


图 3.12 并查集的初始化

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]，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 改成点 1 的集也可以。经过这次合并，1 和 2 合并成一个帮派，帮主是 2。

图 3.13 演示了合并的结果，此时有 5 个点，4 个集，其中 s[2]包括两个点。

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。这个操作是查找点 1 的帮主。

---

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。本书作者拟过一句赠言：“Kruskal 对并查集说，咱们一辈子是兄弟！”

② 本书作者拟过一句赠言：“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 3.13　合并(1,2)

再查找点 3 的集是 s[3]=3。由于 s[2]不等于 s[3],说明点 2 和点 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,也就是让点 2 的帮主成为点 3。此时,点 1、2、3 都属于一个集:s[1]=2、s[2]=3、s[3]=3。点 1 的上级是点 2,点 2 的上级是点 3,这 3 个人的帮主是点 3,形成了一个多级关系。

图 3.14 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。

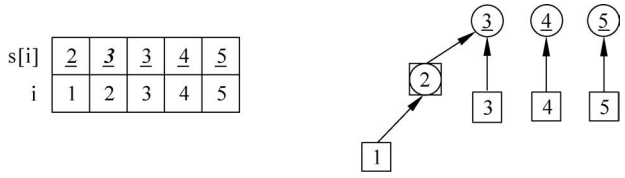


图 3.14　合并(1,3)

继续合并,合并点 2 和点 4。结果如图 3.15 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。点 4 是点 1、2、3、4 的帮主。另外,还有一个独立的集 s[5]=5。

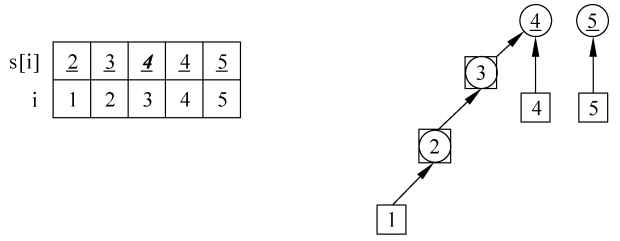


图 3.15　合并(2,4)

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. 统计有多少个集

只要检查有多少个点的集等于自己(自己是自己的帮主),就有多少个集。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。

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(帮派中每个人都只有一个下属),递归步骤是复杂度 O(n)。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,复杂度是 O(1)。

下面用一个例题给出并查集的基本操作。

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

问题描述：若某个家族的人数过多，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、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 和 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

在该例中并查集的基本操作如下。

（1）初始化：init_set()。

（2）查找：find_set()是递归函数，若 x == s[x]，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 3.16 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。

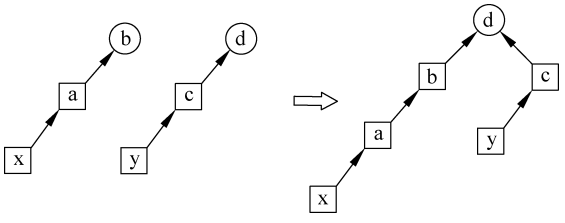


图 3.16　合并

下面是代码。

```
1  import java.util.*;
2  public class Main {
```

```
3       static int[] s;
4       static int N = 5010;
5       public static void init_set() {                 //初始化
6           s = new int[N + 1];
7           for (int i = 1; i <= N; i++) s[i] = i;
8       }
9       public static int find_set(int x) {             //查找
10          return x == s[x] ? x : find_set(s[x]);
11      }
12      public static void merge_set(int x, int y) {    //合并
13          x = find_set(x);
14          y = find_set(y);
15          if (x != y) s[x] = s[y];                    //y 成为 x 的上级,x 的集改成 y 的集
16      }
17      public static void main(String[] args) {
18          Scanner sc = new Scanner(System.in);
19          int n = sc.nextInt();
20          int m = sc.nextInt();
21          int p = sc.nextInt();
22          init_set();
23          while (m-- > 0) {                           //合并
24              int x = sc.nextInt();
25              int y = sc.nextInt();
26              merge_set(x, y);
27          }
28          while (p-- > 0) {                           //查询
29              int x = sc.nextInt();
30              int y = sc.nextInt();
31              if (find_set(x) == find_set(y))
32                  System.out.println("Yes");
33              else System.out.println("No");
34          }
35      }
36  }
```

3.8.2 节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。

在前面的查询函数 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,从而导致超时。

如何优化? 如果在递归返回的时候,顺便把这条路径上所有点所属的集改成根节点(所有人都只有帮主一个上级,而不再有其他上级),那么下次再查询这条路径上的点属于哪个集时,就能在 O(1)的时间内得到结果。如图 3.17 所示,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。下次再查询 1、2、3、4 所属的集,只需要递归一次就查到了根节点。

路径压缩的代码非常简单。把 3.8.1 节代码中的 find_set()改成以下路径压缩的代码即可。

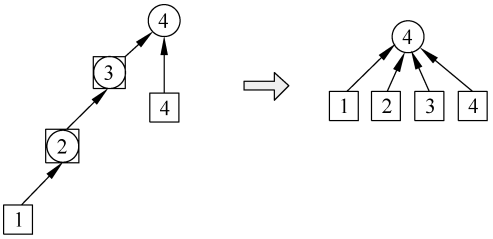


图 3.17　路径压缩

```
1  public static int find_set(int x){
2      if(x != s[x]) s[x] = find_set(s[x]);          //路径压缩
3      return s[x];
4  }
```

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，并告知每条公路连着哪两个村庄，以及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；接下来 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，并在时间 t 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，则输出－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，则输出－1。

用并查集处理村庄的合并，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 18～20 行，做了灵活处理。第 19 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 20 行，如果 x、y 没有连通，则合并并查集。

```
1  import java.util.*;
2  class Main {
3      static class Node {
4          int x, y, t;
5          public Node(int x, int y, int t) {
6              this.x = x;
7              this.y = y;
8              this.t = t;
9          }
10     }
11     static int[] s;
12     public static int find_set(int x) {          //用"路径压缩"优化的查询
13         if (x != s[x]) s[x] = find_set(s[x]); //路径压缩
14         return s[x];
15     }
16     public static void main(String[] args) {
17         Scanner sc = new Scanner(System.in);
18         int n = sc.nextInt();
19         int m = sc.nextInt();
20         s = new int[m + 1];
21         for (int i = 1; i <= m; i++) s[i] = i;      //并查集的初始化
22         Node[] e = new Node[m + 1];
23         for (int i = 1; i <= m; i++) {
24             int x = sc.nextInt();
25             int y = sc.nextInt();
26             int t = sc.nextInt();
27             e[i] = new Node(x, y, t);
28         }
29         Arrays.sort(e, 1, m + 1, new Comparator<Node>() {
30             public int compare(Node a, Node b) {
31                 return a.t - b.t;
32             }
33         });                                      //按时间 t 排序
34         int ans = 0;                             //答案，最早通车时间
35         int num = 0;                             //已经连通的村庄数量
36         for (int i = 1; i <= m; i++) {
37             int x = find_set(e[i].x);
38             int y = find_set(e[i].y);
39             if (x == y) continue;                //x、y 已经连通，num 不用增加
40             s[x] = y;                            //合并并查集，即把村庄 x 合并到 y 上
41             num++;                               //连通的村庄数量加 1
42             ans = Math.max(ans, e[i].t);         //当前最大通车时间
43         }
44         if (num != n - 1) System.out.println("-1");
45         else System.out.println(ans);
46     }
47 }
```

再看一道比较难的例题。

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

时间限制：1s　**内存限制**：512MB　**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,…,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，则小明会将 A_i 加 1；如果新的 A_i 仍在之前出现过，小明会持续将 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 数组中没有重复的整数了。现在给定初始的 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、…、A_n。

输出：输出 n 个整数，依次是最终的 A_1、A_2、…、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≤n≤10000；对于所有评测用例，1≤n≤100000，1≤A_i≤1000000。

这道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。

容易想到一个改进的方法——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，小明会持续将 A_i 加 1，直到 A_i 没有在 A_1～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，则将 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，则将 A[3]加 1 得 A[3]=7；再查到 vis[7]=1，再将 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，仍然需要检查 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。

```
1  import java.util.*;
2  public class Main {
3      public static void main(String[] args) {
4          Scanner sc = new Scanner(System.in);
```

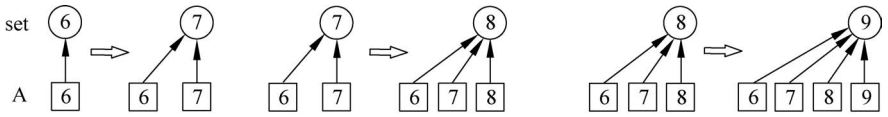
```
5           int n = sc.nextInt();
6           int[] vis = new int[1000002];  //hash: vis[i] = 1 表示数字 i 已经存在
7           for (int i = 0; i < n; i++) {
8               int a = sc.nextInt();      //读一个数字
9               while (vis[a] == 1)
10                  a++;                  //若 a 已经出现过,加 1。若加 1 后再出现,继续加
11              vis[a] = 1;                //标记该数字
12              System.out.print(a + " "); //打印
13          }
14      }
15  }
```

这道题使用并查集非常巧妙。

前面提到,本题用 Hash 方法,在特殊情况下仍然需要做大量的检查。问题出在“持续将 A_i 加 1,直到 A_i 没有在 $A_1 \sim A_{i-1}$ 中出现过”上。也就是说,问题出在相同的数字上。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,如图 3.18 所示。初始化时 set[i]=i。



(1) 读第一个6　(2) 读第二个6　(3) 读第三个6

图 3.18　用并查集处理数组 A

图(1)读第一个数 A[0]=6。6 的集 set[6]=6。紧接着更新 set[6]=set[7]=7,作用是在后面再读到某个 A[k]=6 时,可以直接赋值 A[k]=set[6]=7。

图(2)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。如果在后面再读到 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)读第三个数 A[2]=6。请读者自己分析。

下面是代码,只用到并查集的查询,没有用到合并;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。

```
1   import java.util.*;
2   public class Main {
3       static int[] s;
4       public static int find_set(int x) {              //用“路径压缩”优化的查询
5           if (x != s[x]) s[x] = find_set(s[x]);        //路径压缩
6           return s[x];
7       }
8       public static void main(String[] args) {
9           Scanner sc = new Scanner(System.in);
10          int N = 1000002;
11          s = new int[N];
12          for (int i = 1; i < N; i++) s[i] = i;        //并查集的初始化
13          int n = sc.nextInt();
14          int[] A = new int[n + 1];
15          for (int i = 1; i <= n; i++) {
```

```
16              A[i] = sc.nextInt();
17              int root = find_set(A[i]);              //查询到并查集的根
18              A[i] = root;
19              s[root] = find_set(root + 1);           //加 1
20          }
21          for (int i = 1; i <= n; i++)
22              System.out.print(A[i] + " ");
23      }
24  }
```

【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。

扫一扫

视频讲解

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，包括数组、链表、队列、栈和二叉树。在竞赛中题目可以用库函数实现，也可以手写代码实现。库函数应该重点掌握，大多数题目能直接用库函数实现，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。通过合理地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。