第3章 Python语言和数据结构基础

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。在 Python 中，用列表 list 定义数组，例如定义一个数组 a[]：

```
1  a = [0,1,2,3,4,5]
2  for i in range(5): print(hex(id(a[i])), end = " ")        # 用十六进制表示地址
```

在作者的计算机上运行，输出 5 个数的存储地址：

0x279edb300d0 0x279edb300f0 0x279edb30110 0x279edb30130 0x279edb30150

系统为每个 a[i]分配了一个 32 字节的存储空间，而且这 5 个数的存储地址是连续的。

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 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，每个数据往前挪一个位置，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。

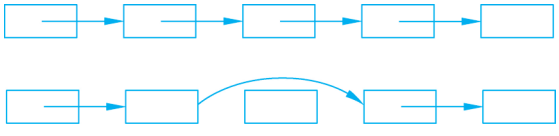


图 3.1　删除第 3 个数据

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。当然，除非有必要，一般不手写链表，而是用系统提供的链表。Python 可以用列表 list 来实现链表的功能，不过列表是基于数组的，它的插入和删除会导致元素的移动，效率不高。

（3）队列。队列是线性数据的一种使用方式，用于模拟现实世界中的排队操作。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。Python 编程一般用 deque()实现队列。

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，用于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。Python 编程一般用 deque()实现栈。

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[①]，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。

① 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。”

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。

3.1 Python 常用功能

首先概述 Python 语言的特点。

Python 是一门解释型语言，它不需要编译和链接，在运行时由解释器逐行将源代码转换成机器码并执行，所以效率较低。C++语言是编译型语言，需要经过编译把源代码转换成机器码，生成可执行文件，才能在机器上运行，所以执行效率很高。解释型语言的一个优点是兼容性强，Python 代码可以直接在各种操作系统上运行。

Python 是一门交互式语言：Python 解释器实现了交互式操作，可以直接在终端输入并执行指令。

Python 程序简洁、易读，易学易用。做同一道题，Python 代码通常比 C++代码少 1/3。

Python 实用性强，它实现了大量数据结构和算法。在算法竞赛中，可以使用 Python 提供的数据结构和函数轻松地完成一些功能。在这一点上和 C++ STL 差不多，可能比 STL 更强大。

本书不会详细介绍 Python 语法，因为本书是一本算法竞赛教材，而不是 Python 语言教材，对 Python 语言的学习请大家通过阅读 Python 教材来掌握。不过，有一些竞赛中常用的 Python 数据结构、函数[①]等，一般的 Python 教材不太会涉及，本章将做详细的介绍。

3.1.1 输入和输出

大家在做竞赛题时，首先要处理题目的输入和输出。输入和输出有各种各样的格式要求，初学者经常会感到困惑，下面做详细的介绍[②]。

1. 在多行中每行输入多个整数

例 1： 输入共两行，第一行包含一个整数 n，第二行包含 n 个整数，用空格隔开。

这是最常见的输入格式，下面是输入方法。

```
n = int(input())                # 读入整数 n,注意用 int 转换成整数
a = input().split(" ")          # 读第二行的所有整数,用 split 分开
int(a[i])                       # 在使用时用 int 转换为整数
# 读第二行的整数也可以用下面一句代码完成,这样更加简洁
a = [int(i) for i in input().split()]
# 或者这样,用 map 转换
a = list(map(int, input().split()))
# 有时候不想用 a[0],从 a[1]开始,可以这样
a = [0] + [int(i) for i in input().split()]
```

① Python 3 的官方帮助文档：https://docs.python.org/3/library/stdtypes.html。

② 内容主要引用自《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 684 页“附录 A Python 在竞赛中的应用”。这篇附录介绍了 Python 的 3 个应用：大数计算、构造测试数据和对拍、输入/输出。

例 2：输入共 n+1 行，第一行包含一个整数 n，后面 n 行，每行包含一个整数 A_i。

```
n = int(input())
a = []                              # 定义 a 为列表，存数组的 n 个整数
for i in range(n):                  # 读 n 行
    a.append(int(input()))          # 每行读一个整数并添加到数组 a 的末尾
```

2. 用 map 转换格式

map 也是 Python 常见的输入方式，用于把输入转换成需要的类型。

例 1：输入共两行，第一行包括 4 个正整数 A、B、C、m；第二行包含 A×B×C 个整数。

从下面第 2 行代码可以看出，Python 不用关心一行中有多少个数，统一读到列表 a 中即可。

```
A,B,C,m = map(int,input().split())      # 读第一行的 4 个整数
a = list(map(int,input().split()))
          # 读第二行的多个整数，读取后用 a[i]访问第 i 个数
```

例 2：第一行包含两个整数 n 和 k，后面 n 行，每行包含两个整数：h 和 w。

```
n,k = map(int,input().split())
w = []
h = []
for i in range(n):
    a,b = map(int,input().split())
    w.append(a)
    h.append(b)
```

3. 二维数组的输入

例：第一行包含 3 个整数 n、m、T，后面 m 行，每行包含两个整数。

```
first = input()
n, m, T = [int(i) for i in first.split()]
a = []                                          # a 是二维数组，这样使用它：a[i][0], a[i][1]
for i in range(m):                              # 读 m 行
    a.append([int(i) for i in input().split()])   # 每行读多个整数
```

4. 输入用非空格字符隔开的数字

例：输入的第一行为一个正整数 T，表示输入数据的组数。每组数据包含两行，每一行表示时间，有两种格式。

```
h1:m1:s1 h2:m2:s2
h1:m1:s1 h2:m2:s2(+1)
```

例如：

```
11:05:18 15:14:23
17:21:07 00:31:46(+1)
```

下面用两种方法处理这两种格式的输入。

第一种方法是用切片提取字符串中的数字。

```
line = input().split()          # 一行字符串，以空格分开，分别读取
h1 = int(line[0][0:2])          # 切片，提取字符串中的数字
m1 = int(line[0][3:5])
s1 = int(line[0][6:8])
h2 = int(line[1][0:2])
m2 = int(line[1][3:5])
s2 = int(line[1][6:8])
```

```
day = 0
if(len(line) == 3):                # line 中有 3 个元素,最后一个是(+1)的数字 1,赋值给 day
    day = int(line[2][2])
```

第二种方法更简单,用 split()去掉':'。

```
line = input().split()                    # 一行字符串,以空格分开,分别读取
h1,m1,s1 = map(int,line[0].split(':'))    # 再按":"分开
h2,m2,s2 = map(int,line[1].split(':'))
day = 0
if(len(line) == 3):                # line 中有 3 个元素,最后一个是(+1)的数字 1,赋值给 day
    day = int(line[2][2])
```

5. 输入字符

读入一个字符串,处理其中的每个字符。

例:输入一个由"x()|"组成的字符串。

```
s = input()                        # 读字符串
if s[i] == '(':                    # 若第 i 个字符是'(',做相应处理
```

6. 输入时未明确说明哪一行是终止

有时题目没有明确说明什么时候输入终止,例如"存在多组测试数据,每组测试数据一行,包含一个正整数 n。"

解决方法 1:for n in sys.stdin。这个语句的作用和 C++的"while(cin >> n)"语句类似。

```
import sys
for n in sys.stdin:                # 读入 n
    n = int(n)
# 下面处理 n
```

解决方法 2:读入出错就停止。

```
while True:                        # 多组数据
    try:
        n, m = map(int,input().split())
        # 然后写代码处理 n、m
    except EOFError:               # 输入出错,说明输入终止了
        break
```

7. 带格式输出

例 1:输出四舍五入保留 4 位小数。下面给出 3 种写法的代码。

```
n = 1.23438234
print('{:.4f}'.format(n))          # 输出:1.2344
print("%.4f" % n)                  # 输出:1.2344
print(round(n, 4))                 # 输出:1.2344
```

例 2:输出 hh:mm:ss,表示时间为 hh 小时 mm 分 ss 秒。当时间为一位数时,要补齐前导零,例如三小时二四分五秒写为 03:24:05。下面给出两种写法,第 2 种写法显得更简洁。

```
hh,mm,ss = 3,24,5
print("{:0>2d}:{:0>2d}:{:0>2d}".format(hh,mm,ss))      #输出:03:24:05
print("%02d:%02d:%02d" %(hh,mm,ss))                    #输出:03:24:05
```

3.1.2 字符串

在算法竞赛中,字符串处理是极为常见的考点。Python 的字符串处理是有名的简洁、

易写。

下面举例说明字符串的操作。

（1）字符串的输入。直接用 input()输入，如果有空格，也会被看成字符串的一部分。如果在一行中输入多个字符串，需要用 split()分开。

str.split(sep=None, maxsplit=-1)[①]：按照指定的“分隔字符串 sep”对输入进行分隔，返回一个列表。如果不写 sep，默认按空格分隔。maxsplit 是分隔的个数，如果不写，默认全部按 sep 分隔。

在下面的例子中，输入同样的字符串给 s1、s2、s3，结果不同。s1 是一个字符串，字符串中包括空格。s2 是列表，输入用空格分开，输入了两个字符串。s3 是列表，输入用字符串“:#”分开，输入了两个字符串。

```
s1 = input()                          # 输入样例：aaa:#bbb ccc
print(s1)                             # 打印：aaa:#bbb ccc
s2 = input().split()                  # 输入样例：aaa:#bbb ccc
print(s2)                             # 打印:['aaa:#bbb', 'ccc']
s3 = input().split(':#')              # 输入样例：aaa:#bbb ccc
print(s3)                             # 打印：['aaa', 'bbb ccc']
s4 = 'ab#ceef#ghk'
print(s4.split('#'))                  # 打印：['ab', 'ceef', 'ghk']
print(s4.split('#', 1))               # 打印:['ab', 'ceef#ghk']
```

（2）format()[②]格式化输出。有时需要灵活地输出，此时可以用 format()指定格式。请读者自己熟悉 format()的用法。

下面是几个比较特殊的例子。第 2 行代码用“{}”占位符；注意第 3 行和第 2 行的区别；第 4 行用转义符“f”格式化输出。后面几行打印了不同的进制。

```
s,num = 'tom', 19
print("name:{}, age:{}".format(s, num))          # 打印：name:tom, age:19
print("name:{1}, age:{0}".format(s, num))        # 打印：name:19, age:tom
print(f'name:{s}, age:{num}')                    # 打印：name:tom, age:19
print("{:b}".format(17))                         # 打印 17 的二进制：10001
print("{:o}".format(17))                         # 打印 17 的八进制：21
print("{:d}".format(17))                         # 打印 17 的十进制：17
print("{:x}".format(17))                         # 打印 17 的十六进制：11
print("{:#b}".format(17))                        # 以 0b 开头打印 17 的二进制：0b10001
```

（3）字符串切片(slice[③])。切片是 Python 极为灵活的应用，是对操作的对象截取一部分的操作，字符串、列表、元组都支持切片操作。切片通常使用索引来完成，下面是例子。在切片时可以有 3 个参数：开始、结束、步长。从当前索引开始，一直到索引结束，但是不包含最后的索引值，步长就是隔多少个数取值。

```
s = "abcdefghijk"
print(s[2:5])          # 从 2 开始，到 5 结束(不包括 5)。打印：cde
print(s[: 5])          # 从 0 开始，到 5 结束(不包括 5)。打印：abcde
print(s[1:])           # 从 1 开始，一直到结束。打印：bcdefghijk
print(s[:])            # 打印全部：abcdefghijk
print(s[:-2])          # 从头到倒数第 2 个(不包括)。打印：abcdefghi
```

① https://docs.python.org/3/library/stdtypes.html#str.split。

② https://docs.python.org/3/library/stdtypes.html#Formatting

③ https://docs.python.org/3/library/functions.html#slice

```
print(s[-1:])                    # 打印最后一个。打印：k
print(s[2:7:2])                  # 从2开始到7结束(不包括7),步长为2。打印：ceg

s2 = ['tom','joy','rose','jack']  # 列表的例子
print(s2[1:2])                   # 打印：['joy']
```

(4) 字符串的查找。字符串的查找有多种方法，下面是几种常用的方法。

in 和 not in：判断某个字符串是否为待判断字符串的子串，返回 True 或 False。

str.find(sub[, start[, end]])：查找字符串 sub，如果存在，返回第一个匹配位置，否则返回－1。

str.index(sub[, start[, end]])：查找字符串 sub，如果存在，返回第一个匹配位置，如果不存在，系统会报错，所以大家在使用 index()时需要注意。

另外还有 str.rfind(sub[, start[, end]])和 str.rindex(sub[, start[, end]])，从字符串右边开始查找。

```
s = 'apple_bananna'
print('a' in s)                  # 打印：True
print('b' not in s)              # 打印：False
print(s.find('e'))               # 找到e停止，返回下标。打印：4
print(s.find('v'))               # 没找到。打印：-1
print(s.find('ba',2,6))          # 从索引2开始找，直到索引6(不包含6)。打印：-1
s = "fine thank you and you"
print(s.index('you'))            # 找到you停止，返回下标。打印：11
print(s.index('and',2,25))       # 从2找到25(不包含25)，返回下标。打印：15
print(s.rfind("you"))            # 查找从右侧开始。打印：19
print(s.rindex("you"))           # 查找从右侧开始。打印：19
print(s.index('tom'))            # 没找到，报错
```

(5) 字符串的个数。

str.count(sub[, start[, end]])：查找子串 sub 的个数，start 和 end 是查找范围，返回字符串中子串出现的次数。

```
s = "fine thank you and you"
print(s.count("you"))            # 打印：2
print(s.count("you",12))         # 从索引12开始查找。打印：1
print(s.count("you",1,10))       # 从索引1找到10(不包括)。打印：0
```

(6) 字符串的替换。

str.replace(old, new[, count])：把旧子串 old 替换为新子串 new，count 是替换次数。如果替换次数超过子串出现的次数，则替换次数为该子串出现的次数。

```
s = 'fine thank you and you'
print(s.replace("you","tom"))      # 打印：fine thank tom and tom
print(s.replace("you","tom",1))    # 打印：fine thank tom and you
```

(7) 字符串的合并。

join()：合并多个子串。注意下列代码中第 6 行集合的合并结果。

```
s1 = ["fine", "thank", "you", "and", "you"]      # 列表合并，用“,”分隔
s2 = ("fine", "thank", "you", "and", "you")      # 元组合并
s3 = {"fine", "thank", "you", "and", "you"}      # 集合合并，用空格分隔
print(','.join(s1))                              # 打印：fine,thank,you,and,you
print(''.join(s2))                               # 打印：finethankyouandyou
print(' '.join(s3))                              # 打印：thank and fine you
```

（8）字母的大小写转换。

- capitalize()：将字符串的第一个字符转换为大写。
- title()：将字符串中每个单词的首字母转换为大写。
- istitle()：检查是否为小写。
- isupper()：检查是否为大写。
- upper()：将字符串中的小写转换为大写。
- lower()：将字符串中的小写转换为大写(英语)。
- casefold()：将所有字母都转换为小写(所有语言)。
- swapcase()：将所有字母转换大小写，原来大写转换成小写，原来小写转换成大写。
- lstrip()：删除字符串的左侧子串，无参数默认为空格。
- rstrip()：删除字符串的右侧子串，无参数默认为空格。
- strip()：删除字符串的两侧子串，无参数默认为空格。

```
s = "hello WORLD"
print(s.capitalize())          # 打印：Hello world
print(s.title())               # 打印：Hello World
print(s.istitle())             # 打印：False
print(s.isupper())             # 打印：False
print(s.upper())               # 打印：HELLO WORLD
print(s.lower())               # 打印：hello world
print(s.casefold())            # 打印：hello world
print(s.swapcase())            # 打印：HELLO world
s = " hello WORLD "
print(s.lstrip())              # 删除左侧空格。打印：hello WORLD
print(s.lstrip('he'))          # 删除左侧子串。打印：llo WORLD
print(s.rstrip())              # 删除右侧空格。打印：  hello WORLD
print(s.strip())               # 删除两侧空格。打印：hello WORLD
```

（9）字母和数字检查。

- isalpha()：如果字符串中的所有字符都是字母，返回 True，否则返回 False。
- isdigit()：如果字符串只包含数字，返回 True，否则返回 False。
- isalnum()：如果字符串中的所有字符都是字母和数字，返回 True，否则返回 False。
- isspace()：如果字符串中只包含空白，返回 True，否则返回 False。

```
s = 'hello123'
print(s.isalpha())             # 打印：False
print(s.isdigit())             # 打印：False
print(s.isalnum())             # 打印：True
print(s.isspace())             # 打印：False
```

（10）其他。请大家特别注意下面代码中从第 8 行开始的字符串的比较。两个字符串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。

```
a,b = 'hello','world'
print(a + b)                   # 字符串连接。打印：helloworld
print(3 * a)                   # 字符串复制。打印：hellohellohello
print(a.center(15))            # 居中对齐。打印：hello
print(a.ljust(15))             # 左对齐。打印：hello
print(a.rjust(15))             # 右对齐。打印：      hello
```

```
print(a.zfill(15))              # 右对齐,左边填 0。打印:0000000000hello
s1,s2 = "abc", "Abc"
print(s1 == s2)                 # 字符串比较。打印:False
print(s1 != s2)                 # 字符串比较。打印:True
print(s1 < s2)                  # 字符串比较。打印:False
print(s1 > s2)                  # 字符串比较。打印:True
print(s1 <= s2)                 # 字符串比较。打印:False
print(s1 >= s2)                 # 字符串比较。打印:True
```

下面是一道简单的字符串题。

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	2
3	1
moon	1
blue	
dark	

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。对于每个 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。

读者可能会觉得这个题目很简单。下面的代码用 count()统计 P 在 S 中出现的次数。这些代码对吗?

```
s = input()
n = int(input())
while n > 0:
    p = input()
    print(s.count(p))
    n -= 1
```

很可惜,代码是错的。例如 S="aaaa",P="aa",上面代码的答案是 2,但实际答案是 3。请读者思考原因。下面是正确的代码。

```
s = input()
n = int(input())
while n > 0:
    p = input()
    cnt = 0
    for i in range(len(s) - len(p) + 1):          # 遍历 S[i]
        if p == s[i:i+len(p)]:                    # 从 s[i]开始匹配 P
            cnt += 1
```

```
    print(cnt)
    n -= 1
```

【练习题】

简单的字符串入门题[①]很多，请大家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

3.1.3 日期库

用 Python 求解日期问题既简单又快。在第 2 章“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，不过 date(year, month, day)中 year 的范围是 1～9999，如果超过这个范围就不能用 datetime 了。例如“2.4 填空题例题”中的“特殊日期”，year=2000000，不能用 datetime。

datetime[②] 库主要包含以下类：date、time、datetime、timedelta。

1. date 类

date 类提供了一种方便的方式来处理和操作日期。以下是 date 类的常用方法和属性。

- date(year, month, day)：创建一个 date 对象，参数分别为年份、月份和日期。
- today()：以 date 对象的形式返回当前的日期。
- strftime(format)：将 date 对象格式化为指定的字符串形式，使用 strftime()函数格式化字符串。
- isoformat()：将 date 对象转换为 ISO 格式的字符串，即"YYYY-MM-DD"的形式。
- weekday()：返回星期几的数字，星期一为 0，星期日为 6。
- isoweekday()：返回星期几的数字，星期一为 1，星期日为 7。
- replace(year, month, day)：创建一个新的 date 对象，替换指定的年份、月份和日期。
- toordinal()：返回 date 对象自公元 1 年 1 月 1 日起的天数。

下面举例说明它们的功能。

```
from datetime import *
print(date.today())                  # 当前日期。打印：2024-03-14
print(date.min, date.max)            # 最小和最大日期。打印：0001-01-01 9999-12-31
a = date(2034,3,14)
b = date(2022,2,15)
print(a)                             # 打印：2034-03-14
print(a.ctime())                     # 打印：Tue Mar 14 00:00:00 2034
```

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，适合初学者提高编码能力。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，它们都是中级和高级知识点。请读者参考《算法竞赛》，清华大学出版社，罗勇军、郭卫斌著，第 549 页“第 9 章　字符串”。

② https://docs.python.org/3/library/datetime.html#module-datetime

```
print(a.strftime("%Y%m%d"))          # 按格式打印：20340314
print(a.strftime("%y%m%d"))          # 按格式打印：340314
print(a.strftime("%Y-%m-%d"))        # 按格式打印：2034-03-14
print(a.year,a.month,a.day)          # 打印：2034 3 14
print(a.weekday())                   # 星期一是0,星期天是6。打印：1
print(a.isoweekday())                # 星期一是1,星期天是7。打印：2
print(a>=b)                          # 日期比较有a>=b、a>b、a<=b、a<b、a!=b。打印：True
print(a-b)                           # 日期之差,返回timedelta。打印：4410 days, 0:00:00
print((a-b).days)                    # 日期之差,返回整数。打印：4410
b = a + timedelta(weeks=7)
b += timedelta(days=7)
b += timedelta(hours=8)              # 还有minutes、seconds
print(b)                             # 打印：2034-05-09
print(b.toordinal())                 # 打印：742667
```

对于 strftime()中时间、日期的格式，常用格式如表 3.1 所示。

表 3.1 strftime()中时间、日期的常用格式

格　式	含　义	格　式	含　义
%y	两位数年份表示(00～99)	%Y	四位数年份表示(0000～9999)
%m	月份(01～12)	%d	月内的一天(0～31)
%H	24 小时制的小时数(0～23)	%I	12 小时制的小时数(01～12)
%M	分钟数(00～59)	%S	秒(00～59)
%a	本地简化的星期名称	%A	本地完整的星期名称
%b	本地简化的月份名称	%B	本地完整的月份名称

这些格式也在下面的 time 和 datetime 类中使用。

2. time 类

time 类表示时间，包括小时、分钟、秒和微秒。以下是 time 类常用的方法和属性。

(1) time(hour,minute, second, microsecond)：创建一个时间对象，参数有小时、分钟、秒、微秒。

- hour：获取时间对象的小时部分。
- minute：获取时间对象的分钟部分。
- second：获取时间对象的秒部分。
- microsecond：获取时间对象的微秒部分。

(2) strftime(format)：将时间对象格式化为指定的字符串格式。

下面是一些例子。

```
from datetime import *
print(time.min, time.max)                #最小、最大时刻。打印：00:00:00 23:59:59.999999
a = time(23,59,34,333)
b = time(22,9,4,3)
print(a)                                 # 打印：23:59:34.000333
print(a.hour,a.minute,a.second,a.microsecond)    # 打印：23 59 34 333
print(a>b)                               # 比较。打印：True
print(a.strftime('%H:%M:%S'))            # 按格式打印：23:59:34
print(a.strftime('%H-%M-%S'))            # 按格式打印：23-59-34
#print(a-b)                              # 这一句是错的,time不能相减,datetime可以减
```

3. datetime 类

datetime 类可以看作 date 类和 time 类的合体，由 year、month、day、hour、minute、

second、microsecond 等组成，功能差不多。下面是例子。

```
from datetime import *
start = datetime.now()
print(datetime.now())                    # 当前时间。打印：2024 - 03 - 14 17:18:52.743116
a = datetime(2026,5,14,23,56,9)
print(a.month, a.second)                 # 打印：5 9
b = a + timedelta(weeks = 7)
b += timedelta(days = 7)
b += timedelta(hours = 8)                # 还有 minutes、seconds
print(b - a)                             # 打印：56 days, 8:00:00
print(a > b)                             # 打印：False
end = datetime.now()                     # 当前时间
print((end - start).microseconds)        # 打印时间差：459741
```

下面看一道例题。

例 3.2　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的时间，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，毫秒也不用显示，直接舍去即可。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：输入一行，包含一个正整数，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 到 23；MM 表示分，值为 0 到 59；SS 表示秒，值为 0 到 59。当时、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 Python 日期功能的简单应用，代码如下：

```
from datetime import *
s = datetime(1970, 1, 1)
d = timedelta(milliseconds = 1)
n = int(input())
n = s + n * d
print(n.strftime("%H:%M:%S"))
```

再看一道例题。

例 3.3　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，那么从建国到 2012 年有几次国庆节正好是星期日？

对于这种简单的日期问题，用 Python 可以直接求解。

```
from datetime import *
```

```
cnt = 0
for i in range(1949,2013):
    a = date(i,10,1)
    if a.weekday() == 6: cnt += 1
print(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 和字典去重

算法竞赛经常需要去除重复的数据，在 Python 中这个需求可以用 set 或字典实现。

1. set()

在 Python 中，集合 set 是一种无序、可变的数据类型，用于存储多个不重复的元素。set() 基于哈希表实现，查找和插入速度快。

set()的常用操作如下：

(1) 创建一个空集合。用 set()函数创建一个空集合，例如 st=set()。

(2) 创建一个非空集合。用{}创建一个非空集合，例如 t={1, 2, 3}。

(3) 添加元素。用 add()向集合中添加元素，例如 st.add(4)。

(4) 删除元素。用 remove()从集合中移除指定的元素，如果元素不存在，则抛出 KeyError 异常，例如 st.remove(3)。

(5) 集合运算。集合支持多种常见的集合运算，包括并集、交集和差集等。用 union()计算两个集合的并集，用 intersection()计算两个集合的交集，用 difference()计算两个集合的差集。

(6) 成员关系判断。用 in 关键字判断一个元素是否属于集合，例如 if 1 in st:。

(7) 长度计算。用 len()函数获取集合中元素的个数，例如 length=len(st)。

(8) 遍历集合。用 for 循环遍历集合中的元素。

2. 字典

在 C++中有 map 映射，在 Python 中类似的功能是字典。

字典中存储键值对。每个键都是唯一的，一个键与一个值相关联，值可以是任意数据类型，例如数、字符串、列表甚至是字典。

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。如果用字典存{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。字典的效率非常高。

字典的常用操作如下：

(1) 字典的定义和初始化。用{}定义字典，可以存任意多的键值对。键和值用冒号分开，键值对之间用逗号分开。例如 dict={'tom': 'boy', 'joy': 30, 'jack':'beijing'}。

(2) 访问字典的值。例如 print(dict['tom'])，打印了键'tom'的值。

(3) 添加键值对。可以随时往字典中添加键值对。例如 dict['rose']=35，添加了一个键值对。

（4）修改字典中的值。例如 dict['tom']='girl'。

（5）删除键值对。例如 del dict['tom']。

3. 例题

下面用两道例题说明如何用 set()和字典实现去重。

例 3.4　Pow Set　http://oj.ecustacm.cn/problem.php?id=1797

问题描述：给定 n、m，考虑集合 S={a^b|2≤a≤n，2≤b≤m}，其中 a^b 表示 a 的 b 次方，求集合 S 去重后有多少个元素。

输入：输入两个正整数 n 和 m，2≤n，m≤500。

输出：输出一个数字，表示答案。

输入样例：	输出样例：
5 5	15

本题很直白，首先算幂 a^b，然后判重。集合 S 内的元素最多有 n×m 个，由于 n 和 m 都不大，可以直接算。

Python 可以直接算大数 a^b，然后用 set 或者字典判重，代码非常简单。

（1）用 set 判重。

```
n,m = map(int,input().split())
s = set()
for i in range(2,n+1):
    for j in range(2,m+1):
        s.add(i**j)
print(len(s))
```

（2）用字典判重。

```
n, m = map(int, input().split())
d = {}
for i in range(2, n+1):
    for j in range(2, m+1):
        d[i**j] = 1     # 将指数的结果作为键，初始值为 1
print(len(d))           # 输出字典中键的数量，即不重复的指数的个数
```

再看一道例题。

例 3.5　眼红的 Medusa　https://www.luogu.com.cn/problem/P1571

问题描述：Miss Medusa 到北京领科技创新奖。她发现很多人和她一样获得了科技创新奖，有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：输入的第一行包含两个整数 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。第三行有 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： 4 3 2 15 6 8 8 9 2	输出样例： 2 8
评测用例规模与约定：对于60%的评测用例，0≤n，m≤1000，获得奖项的人的编号小于2×10^9；对于100%的评测用例，0≤n，m≤10^5，获得奖项的人的编号小于2×10^9。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。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项。下面分别用字典和set实现。

(1) 用字典实现。把m个数放进字典mp中，然后遍历mp中的每个数，如果在n个数中有，就输出。下面代码的计算复杂度是多少？字典的每次操作是$O(\log_2 m)$的，第6行的复杂度是$O(m\log_2 m)$，第7～9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  n, m = map(int, input().split())
2  if n == 0 and m == 0: exit()            # 特判
3  mp = {}
4  a = list(map(int, input().split()))
5  b = list(map(int, input().split()))
6  for i in range(m): mp[b[i]] = True
7  for i in range(n):
8      if a[i] in mp:
9          print(a[i], end = " ")          # 如果出现过,则直接输出
```

(2) 用set实现。

```
1  n, m = map(int, input().split())
2  if n == 0 and m == 0: exit()            # 特判
3  st = set()
4  a = list(map(int, input().split()))
5  b = list(map(int, input().split()))
6  for i in range(m): st.add(b[i])
7  for i in range(n):
8      if a[i] in st:
9          print(a[i], end = " ")          # 如果出现过,则直接输出
```

3.2 列表与数组

扫一扫

视频讲解

列表是Python最常用、最强大的序列类型。列表由一系列按特定顺序排列的元素组成，在列表中可以存放任意类型的元素，可以把数字、字符串、混合数据、嵌套列表等放进列表，甚至同一个列表中的元素之间可以没有任何关系。

数组是最简单的数据结构，有一维数组、二维数组、三维数组等。Python如何实现数组？用列表(list)。由于数组在Python中用列表list实现，本书后文提到数组时常用列表来替代。

3.2.1 列表的常用功能

列表用于处理线性数据。所谓线性数据，是指数据按顺序一个接一个地串在一起。

```
a = []                          # 创建空列表，注意列表使用方括号
a = [0, 1, 2, 3, 5, 8, 13]      # 初始化列表，注意整个列表可以直接打印
print(a)
print(*a)                       # 加上“*”，在打印时不出现括号
a[0] = 1                        # 数组的索引和修改
a.append(a[-2]+a[-1])           # append()
a.pop()                 # 弹出并返回末尾元素，可以当栈使用；其实还可以指定位置，默认是末尾
a.insert(0, 1)                  # 同 vector 的 insert(position, val)
a.remove(1)                     # 按值移除元素(只删第一个出现的)，若不存在，则抛出错误
print(len(a))                   # 求列表的长度
a.reverse()                     # 原地逆置
print(a)
sorted(a)                       # 获得排序后的列表，但是 a 不变
print(a)
a.sort()                        # 原地排序，可以指定参数 key 作为排序标准
print(a)
a.count(1)                      # 类似 std::count()
a.index(1)                      # 返回值首次出现项的索引号，若不存在，则抛出错误
a.clear()                       # 同 vector 的 clear()
```

3.2.2 用列表实现数组

用列表实现一维、二维、三维数组。

1. 一维数组

```
a = []                                  # 定义一个空的一维数组
for i in range(1,10): a.append(i)
print(a)                                # 打印：[1, 2, 3, 4, 5, 6, 7, 8, 9]
b = [1, 2, 3, 4, 5]                     # 定义一个包含多个元素的一维数组
print(b)                                # 打印：[1, 2, 3, 4, 5]
c = [1, "hello", 3.14, True]            # 定义一个包含不同数据类型的一维数组
print(c)                                # 打印：[1, 'hello', 3.14, True]
d = [0]*10
d[3] = 5
d.append(9)
print(d)                                # 打印：[0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 9]
e = [i for i in range(1,10)]            # 定义并赋值
print(e)                                # 打印：[1, 2, 3, 4, 5, 6, 7, 8, 9]
print(min(e))                           # 打印：1
print(max(e))                           # 打印：9
print(max(e[2:5]))                      # 打印：5
print(max(e[2:-1]))                     # 打印：8
print(sum(e[2:]))                       # 打印：42
print(sum(e))                           # 打印：45
```

数组 a[]是从 a[0]开始的，不是从 a[1]开始的。不过，有些题目从 a[1]开始更符合逻辑。

例如第一行输入一个整数 n，第二行输入 n 个整数。把 n 个整数存入数组 a[1]～a[n]，如下写代码：

```
n = int(input())                                # 例如输入：5
```

```
a = [0] + list(map(int,input().split()))          # 例如输入：1 2 3 4 5
#a = [0] + [int(i) for i in input().split()]      # 这样写也行
print(a)                                          # 打印：[0, 1, 2, 3, 4, 5]
```

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 C/C++大学 A 组试题 D：选数异或 lanqiaoOJ 2081

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，有 m 次查询，问每次查询能否从某个区间[l, r]中选择两个数使它们的异或等于 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 $A_1, A_2, \cdots, A_n$。接下来的 m 行，每行包含两个整数 l_i、r_i，表示查询区间$[l_i, r_i]$。

输出：对于每个查询，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：对于每个查询，验算区间内两个数的异或，复杂度为 $O(n^2)$。共 m 个查询，总复杂度为 $O(mn^2)$，只能通过 20%的测试。

```
n, m, x = map(int,input().split())
a = [0] + list(map(int,input().split()))          # 从 a[1]开始
for i in range(m):
    flag = 0                                      # =0,表示不存在异或为 x
    L, R = map(int,input().split())
    for j in range(L,R):
        for k in range(j+1,R+1):
            if a[j]^a[k] == x: flag = 1
    if flag == 1: print('yes')
    else: print('no')
```

2. 二维数组

```
# 定义一个全 0 的二维数组,4 行 3 列
a = [[0 for _ in range(3)] for _ in range(4)]
a[1][2] = 9                          # 数组元素的访问和赋值
print(a)                             # 打印：[[0, 0, 0], [0, 0, 9], [0, 0, 0], [0, 0, 0]]
b = [[]]                             # 定义空二维数组
c = [[1, 2], [4, 5], [7, 8]]         # 定义 3 行 2 列的二维数组
# 遍历二维数组
for row in c:                        # 遍历行
    for e in row:                    # 遍历一行中的所有元素
```

```
        print(e,end = ' ')              # 打印：1 2 4 5 7 8
print()
# 也可以这样按二维数组的方式遍历
for x in range(3):                      # 遍历 x
    for y in range(2):                  # 遍历 y
        print(c[x][y],end = ' ')        # 打印：1 2 4 5 7 8
```

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 Python 大学 A 组试题 F：子矩阵 lanqiaoOJ 3521

时间限制：20s 内存限制：512MB 本题总分：15 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。求给定矩阵的所有大小为 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，$1 \leqslant a \leqslant n \leqslant 1000$，$1 \leqslant b \leqslant m \leqslant 1000$，$1 \leqslant A_{i,j} \leqslant 10^9$。

本题通过 70%和 100%的测试需要用到高级算法。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。

第 5、6、8、9 行共有 4 重 for 循环，计算复杂度是 $O(n^2m^2)$，只能通过 40%的测试。

```
n,m,a,b = map(int,input().split())
A = [[]for i in range(n)]                                   # 定义二维矩阵
for i in range(n): A[i] = list(map(int,input().split()))     # 读二维矩阵
ans = 0
for i in range(n-a+1):
    for j in range(m-b+1):
        Zmax,Zmin = A[i][j], A[i][j]                        # 最大、最小的初值就设为子矩阵第一个
        for u in range(a):
            for v in range(b):
                Zmax = max(Zmax,A[i+u][j+v])
                Zmin = min(Zmin,A[i+u][j+v])
        ans = (ans + Zmax*Zmin) % 998244353
print(ans)
```

3. 三维数组

```
# 定义一个全 0 的三维数组
a = [[[0 for _ in range(2)] for _ in range(3)] for _ in range(4)]
a[3][2][1] = 9              # 数组元素的访问和赋值
print(a)                    # 打印：[[[0, 0], [0, 0], [0, 0]], [[0, 0], [0, 0], [0, 0]],
                            [[0, 0], [0, 0], [0, 0]], [[0, 0], [0, 0], [0, 9]]]
```

```
b = [[[]]]                                                    # 定义空三维数组
c = [[[1, 2],[3, 4]],[[5, 6],[7, 8]],[[9, 10],[11, 12]]] # 定义三维数组
# 遍历三维数组
for x in c:
    for y in x:
        for z in y:
            print(z,end = ' ')                                # 打印：1 2 3 4 5 6 7 8 9 10 11 12
print()
# 或者直接用三维数组的方式遍历
for x in range(3):                                            # 遍历 x
    for y in range(2):                                        # 遍历 y
        for z in range(2):                                    # 遍历 z
            print(c[x][y][z],end = ' ')                       # 打印：1 2 3 4 5 6 7 8 9 10 11 12
```

扫一扫

视频讲解

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。

(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。有时其应用场景比较简单，不需要循环，可以让第一个节点始终是头。

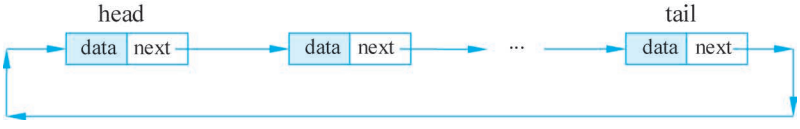


图 3.2　单向链表

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。

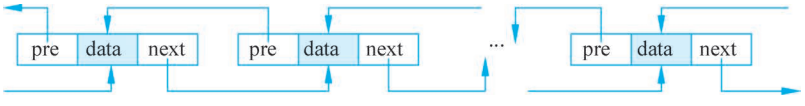


图 3.3　双向链表

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，在找到节点后，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。例如在查找 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)。

在参加算法竞赛时，参赛者虽然可以自己手写链表，不过为了加快编码速度，一般用列表 list 来实现链表。

3.3.1 用列表 list 实现链表

列表 list 可以当成数组用，也可以实现链表、队列、栈的功能。

下面的代码演示了用列表 list 实现链表的各种操作。

```
# 链表的初始化
li = [1,2,3,4,5,6,5];
# 在链表的末尾添加一个节点，数字 99
li.append(99)
print(li)                              # 打印：[1, 2, 3, 4, 5, 6, 5, 99]
# 统计链表中数字 5 的个数
print(li.count(5))                     # 打印：2
# 链表的插入：在链表头插入一个节点，数字 88
li.insert(0,88)
print(li)                              # 打印：[88, 1, 2, 3, 4, 5, 6, 5, 99]
# 链表的插入：在数字 5 的前面插入 33
index = li.index(5)                    # 先找到 5 的位置，然后再插入
li.insert(index,33)
print(li)                              # 打印：[88, 1, 2, 3, 4, 33, 5, 6, 5, 99]
# 链表的插入：在 5 的后面插入 56
index = li.index(5)
li.insert(index + 1,56)
print(li)                              #打印：[88, 1, 2, 3, 4, 33, 5, 56, 6, 5, 99]
# 链表节点的删除：找到 4，删除 4
index = li.index(4)
li.pop(index)
print(li)                              # 打印：[88, 1, 2, 3, 33, 5, 56, 6, 5, 99]
# 链表节点的删除：删除第一个 5
li.remove(5)
print(li)                              # 打印：[88, 1, 2, 3, 33, 56, 6, 5, 99]
# 链表节点的删除：删除第一个节点
del li[0]
# 链表节点的删除：删除最后一个节点
li.pop()                               # 第一种删除方法
print(li)                              # 打印：[1, 2, 3, 33, 56, 6, 5]
del li[ - 1]                           # 第二种删除方法
print(li)                              # 打印：[1, 2, 3, 33, 56, 6]
```

在上面的例子中，链表内的元素是数字，其实其他类型也可以，例如字符串，而且在同一个链表中可以混用，这是列表的功能。

用下面的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，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。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：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[0]～toy[9]这 10 个节点。toy[0]始终是链表的头。注意第 7 行、第 8 行的功能相同，都是打印整个链表，第 8 行的写法更简单。

```
1  toy = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]          # 定义链表
2  m = int(input())
3  while m > 0:
4      x = int(input())
5      toy.remove(x)                               # 删除链表中的 x
6      toy.insert(0, x)                            # 在链表的头部插入 x
7      #for i in toy: print(i, end = " ")           # 输出链表
8      print( * toy)                               # 输出链表
9      m -= 1
```

3.3.2　手写链表

用列表实现链表在逻辑上没有问题，但实际上列表并不能达到真正链表的效率。因为列表实际上是一个数组，插入和删除数据需要挪动很多节点。

下面手写一个链表，并对比它和用列表实现的链表的性能。

(1) 手写链表，并测试性能。

```
1  from time import *
2  class Node():
3      def __init__(self,data):                    # 初始化一个链表节点
4          self.data = data
5          self.next = None
6  class SingleLinkList():
7      def __init__(self, node = None):            # 给链表的节点赋值
8          self.__head = node
9      def right_insert(self,x, y):                # 插入一个节点
```

```
        pre = self.__head
        while pre.data != x: pre = pre.next
        node = Node(y)
        node.next = pre.next
        pre.next = node
    def print_list(self):                    # 打印链表
        cur = self.__head
        while cur != None:
            print(cur.data, end = ' ')
            cur = cur.next
n = 100000
start = time()
a = SingleLinkList(Node(0))
for i in range(1,n): a.right_insert(0,i)     # 连续插入 n 个节点
end = time()
print("time = ", end - start)                # 统计时间
#a.print_list()
```

（2）测试用列表模拟的链表的性能。

```
from time import *
n = 100000
start = time()
a = [0]
for i in range(1,n): a.insert(a.index(0) + 1,i)        # 连续插入 n 个节点
end = time()
print("time = ", end - start)                          # 统计时间
```

这两份代码的功能一样，都是往链表的第 2 个节点插入 n＝100000 个数，用时分别为 0.117 秒和 2.34 秒。这说明列表模拟的链表性能很低。如果题目对链表的性能要求很高，只能手写链表。

最后用一道例题说明链表的应用。

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

下面的代码用列表实现链表的功能。

```
n, m = map(int, input().split())
lis = list(range(1, n + 1))                    # 初始化 list = {1,2,3,…,n}
for _ in range(m):
```

```
    x, y, z = map(int, input().split())
    lis.remove(x)                          # 删除 x
    idx = lis.index(y)                     # 找到 y
    if z == 0: lis.insert(idx+1, x)        # 将 x 放在 y 的后面
    if z == 1: lis.insert(idx, x)          # 将 x 放在 y 的前面
print(*lis)                                # 打印完整链表
```

【练习题】

lanqiaoOJ：约瑟夫环 1111、小王子双链表 1112、种瓜得瓜种豆得豆 3150、自行车停放 1518。

洛谷：单向链表 B3631、队列安排 P1160。

3.4 队　列

扫一扫 视频讲解

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，例如人们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 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。

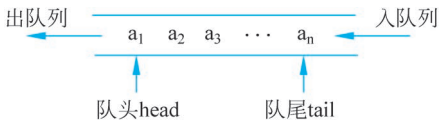


图 3.4　队列

队列的主要操作如表 3.2 所示。

表 3.2　队列的主要操作

操　作	说　明
push()	在队列的尾部插入一个元素
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

先给出手写队列代码。这个手写队列是用列表 list 实现的，进队尾用 append()实现，队列自动扩展，不会有溢出问题。

```
n, m = map(int, input().split())
que = [i for i in range(1, n+1)]
head, tail = 0, n-1                    # 队头和队尾
while tail - head + 1 != 0:
    for i in range(1, m):
        que.append(que[head])
        head += 1
        tail += 1
    print(que[head], end=' ')
    head += 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。

3.4.1 Python 队列

Python[①] 的队列可以用 list、Queue、deque 实现。list 极慢，Queue 较慢，deque 最快。后面对它们进行测试，结论是 deque 比 Queue 快 10 倍以上。

1. list

list 当作队列使用时有如表 3.3 所示的操作。

表 3.3 list 当作队列使用时的操作

操　作	说　明
q.append()	从队尾插入
del q[0]	从队头删除，并返回
len(q)	队列的大小
if not q	判断队列是否为空

下面是样例：

```
n = 10
q = []                              # 定义队列
for i in range(n): q.append(i)      # 加入队列
print(q)                            # 打印：[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(len(q))                       # 打印：10
for i in range(n):
    print(q[0],end = ' ')           # 打印队头。打印：0 1 2 3 4 5 6 7 8 9
    del q[0]                        # 删除队头
print()
print(len(q))                       # 打印：0
if not q: print('empty')            # 打印：empty
```

2. Queue

Queue 当作队列使用时有如表 3.4 所示的操作。

表 3.4 Queue 当作队列使用时的操作

操　作	说　明
q.put()	从队尾插入
q.get()	从队头删除，并返回
q.qsize()	队列的大小
q.empty()	队列是否为空

下面是样例：

```
from queue import *
n = 10
q = Queue()
for i in range(n): q.put(i)         # 加入队列
print(q.qsize())                    # 打印：10
print(q.empty())                    # 打印：False
```

① deque 文档：https://docs.python.org/3/library/collections.html#collections.deque。

```
for i in range(n):
    a = q.get()                    # 读队头,并删除
    print(a,end = ' ')             # 打印:0 1 2 3 4 5 6 7 8 9
print()
print(q.qsize())                   # 打印:0
print(q.empty())                   # 打印:True
```

3. deque

建议大家在参加算法竞赛时只使用 deque,原因是算法竞赛的代码都是单线程的,在这种场景下 deque 比 list、Queue 快很多。

deque 是双向队列,队头和队尾都能插入和弹出。在当成普通队列使用时,只用它的队头弹出、队尾插入功能即可。deque 的常用操作如表 3.5 所示。

表 3.5 deque 的常用操作

操 作	说 明
append(x)	在 deque 的右侧插入元素 x
appendleft(x)	在 deque 的左侧插入元素 x
pop()	从 deque 的右侧删除并返回一个元素
popleft()	从 deque 的左侧删除并返回一个元素
clear()	清空 deque 中的所有元素
count(x)	返回 deque 中元素 x 的个数
remove(value)	从 deque 中删除第一个值为 value 的元素,如果没有找到,则抛出 ValueError 异常
rotate(n)	将 deque 中的所有元素向右移动 n 步,如果 n 为负数,则向左移动

下面是 deque 当作队列使用时的样例:

```
from collections import *
n = 10
q = deque()                        # 定义队列
for i in range(n): q.append(i)     # 加入队列
print(len(q))                      # 打印:10
if q: print('not empty')           # 打印:not empty
for i in range(n):
    a = q.popleft()
    print(a,end = ' ')             # 打印:0 1 2 3 4 5 6 7 8 9
print()
print(len(q))                      # 打印:0
if not q: print('empty')           # 打印:empty
```

4. 对比 list、Queue 和 deque 的性能

(1) 测试 list 实现的队列。

```
from time import *
start = time()
n = 100000
q = []
for i in range(n): q.append(i)     # 加入队列
for i in range(n): del q[0]        # 删除队头
end = time()
print("time = ", end - start)      # 统计时间
```

代码的功能是往队列中加入 100000 个数,并逐一删除,用时 1.2352 秒。如果加入 100 万个数,则运行时间超过一分钟。

（2）测试 Queue 实现的队列。

```
1  from queue import *
2  from time import *
3  start = time()
4  n = 1000000
5  q = Queue()
6  for i in range(n): q.put(i)                # 加入队列
7  for i in range(n): q.get()                 # 删除队头
8  end = time()
9  print("time = ", end - start)              # 统计时间
```

代码的功能是往队列中加入 100 万个数，并逐一删除，用时 4.0948 秒，比 list 快很多。

（3）测试 deque 实现的队列。

```
1  from collections import *
2  from time import *
3  start = time()
4  n = 1000000
5  q = deque()
6  for i in range(n): q.append(i)             # 加入队列
7  for i in range(n): q.popleft()             # 删除队头
8  end = time()
9  print("time = ", end - start)              # 统计时间
```

代码的功能是往队列中加入 100 万个数，并逐一删除，用时 0.2187 秒，比 Queue 快 19 倍。

3.4.2 例题

用下面的例题介绍队列的应用。

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 的人出圈，再由下一个人重新从 1 开始报数，数到 m 的人再出圈，依次类推，直到所有的人都出圈，请依次输出出圈人的编号。

输入：输入两个整数 n 和 m，1≤n，m≤100。

输出：按顺序输出 n 个整数，表示每个出圈人的编号。

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。

先用 Queue 实现，第 6 行把队头移到队尾，第 7 行让数到 m 的人离开。

```
1  from queue import Queue
2  n, m = map(int, input().split())
3  q = Queue()
4  for i in range(1, n + 1): q.put(i)
5  while not q.empty():
6      for i in range(1, m): q.put(q.get())      # 读队头，重新排到队尾
7      print(q.get(), end = ' ')                 # 第 m 个人离开队伍，并输出
```

再用 deque 实现，第 5 行用了简单的写法。

```
1  from collections import deque
2  n, m = map(int, input().split())
3  dq = deque(range(1, n+1))
4  while dq:
5      dq.rotate(-(m-1))                    # 把前 m-1 个数挪到队列的尾部
6      print(dq.popleft(), end=' ')         # 队头是第 m 个数，删除并打印它
```

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。每行中的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时。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。用队列对输入的单词排队，当内存超过 M 时，删除队头的单词。

```
1  from collections import deque
2  hashtable = [False] * 1003                  # 哈希表的初始化，默认为 False
3  m, n = map(int, input().split())            # 输入 m 和 n
4  ans = 0                                     # 初始化答案为 0
5  q = deque()                                 # 初始化队列
6  line = list(map(int, input().split()))      # 读第 2 行
7  for x in line:                              # 处理每个数
8      if hashtable[x] is False:               # 如果 x 不在哈希表中
9          hashtable[x] = True                 # 将 x 加入哈希表
10         if len(q) < m:                      # 如果队列未满
11             q.append(x)                     # 将 x 加入队列
12         else:                               # 如果队列已满
13             hashtable[q.popleft()] = False  # 队首元素出队并从哈希表中删除
```

```
            q.append(x)                    # 将 x 加入队列
        ans += 1                           # 答案加 1
print(ans)                                 # 输出答案
```

【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”，一般就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，则不用等待，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？先试一试最简单的方法。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，参赛人员一般不用自己写二叉堆来实现优先队列，而是直接使用系统提供的 PriorityQueue。初学者只需要学会如何使用它即可。

Python 优先队列 PriorityQueue 的基本操作如下：

```
pq = PriorityQueue()              # 定义队列
pq.put([priority, value])         # 进队列
pq.get()                          # 取出队首
```

put()函数进入队列的可以是列表，第一个参数 priority 表示数据的优先级。如果只有一个参数，表示优先级和值，值越小优先级越高，队首总是最小值。下面是例子。

（1）一个参数。

```
from queue import *
pq = PriorityQueue()
pq.put(1)
```

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

```
pq.put(73)
pq.put(6)
while not pq.empty():                    # 逐个从优先队列取出最小值
    print(pq.get(),end = ' ')            # 打印
```

这个参数也可以是字符串，注意这个参数是可以比较大小的，类型需要一致。如果两个元素分别是数字和字符，则无法比较，是非法的。

```
from queue import *
pq = PriorityQueue()
pq.put('ab')
pq.put('bc')
pq.put('dd')
while not pq.empty():                    # 逐个从优先队列取出最小值
    print(pq.get(),end = ' ')            # 打印：ab bc dd
```

（2）用列表做参数。列表的第一个元素可以是字符串。

```
from queue import *
pq = PriorityQueue()
pq.put([1, 'tom', 33])
pq.put([73, 453])
pq.put([6, True])
while not pq.empty():                    # 逐个从优先队列取出最小值
    print(pq.get(),end = ' ')            # 打印：[1, 'tom', 33] [6, True] [73, 453]
```

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 $S=\{p_1,p_2,\cdots,p_k\}$，丑数指一个正整数满足所有质因数都出现在 S 中，1 默认是第一个丑数。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。

```
from queue import *
s = set()                                # 判重
s.add(1)                                 # 第一个丑数是 1
q = PriorityQueue()                      # 队列中是新生成的丑数
q.put(1)                                 # 第一个丑数是 1，进入队列
n = 20220
ans = 0
prime = [3, 7, 17, 29, 53]
for i in range(1, n + 1):                # 从队列中由小到大取出 20220 个丑数
    now = q.get()
    ans = now                            # 把队列中最小的数取出来，它也是已经取出的最大的数
    for j in range(5):                   # 5 个素数
        tmp = now * prime[j]             # 从已取出的最大值开始乘以 5 个素数
        if tmp not in s:                 # tmp 这个数没有出现过
            s.add(tmp)                   # 放到 set 的里面
            q.put(tmp)                   # 把 tmp 放进队列
print(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，每个人需要分到 k 张牌，并且每个人分到手的牌中不能有相同数字。请保证可以将牌尽可能地分给更多的人，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；再把数量第二多的，每个隔板放一个；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 10 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 $O(n\log_2 n)$。

Python 的优先队列实现是最小堆，因此在将元素插入堆时需要将其取相反数，以达到最大堆的效果。

```
1 from heapq import *
```

```
N = 1000010
num = [0] * N
n, k = map(int, input().split())
nums = list(map(int, input().split()))
for x in nums: num[x] += 1                    # x这个数有num[x]个
q = []
for i in range(N):
    if num[i] > 0:
        heappush(q, (-num[i], i))             # 将元素插入堆时将其取相反数
while len(q) >= k:                            # 队列中数量大于k,说明还够用
    tmp = []
    for i in range(k):                        # 取k个数,且这k个数不同,这是一份
        tmp.append(heappop(q))
    for i in range(k):                        # 打印一份,共k个数
        print(tmp[i][1], end=' ')
        if i == k - 1: print()
        tmp[i] = (tmp[i][0] + 1, tmp[i][1])
        if tmp[i][0] != 0:                    # 没用完,再次进队列
            heappush(q, tmp[i])
```

【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

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，其中一个入口、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，在算法竞赛中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。

Python 的栈可以用 list、deque 和 LifoQueue 这 3 种方法实现。比较它们的运行速度，可见 list 和 deque 一样快，而 LifoQueue 慢得多，建议大家使用 list。

用 list 模拟栈有一个好处——不用担心栈空间不够大，因为 list 自动扩展空间。list 的栈操作非常快，因为栈顶是 list 的末尾元素，栈只有一个出入口，只在 list 的末尾进行进栈

和出栈操作,操作极为快捷。用 list 实现的栈功能如表 3.6 所示。

表 3.6　用 list 实现的栈功能

操　作	说　明
st=[]	定义栈 st
append()	把 item 放到栈顶
st[-1]	返回栈顶的元素,但不会删除
pop()	删除栈顶的元素
len()	返回栈中元素的个数

用下面的例子给出用列表 list 模拟栈的应用。

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:输入一行,为表达式。

输出:输出一行,为 YES 或 NO。

输入样例:	输出样例:
2*(x+y)/(1-x)@	YES

合法的括号组合例如"(())"和"()()()",像")(()"这样是非法的。括号组合合法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。

下面是例 3.14 的 Python 代码,栈用 list 模拟。

```
1  st = []                              # 定义栈,用 list 实现
2  flag = True                          # 判断左括号和右括号的数量是否一样多
3  s = input().strip()
4  for x in s:
5      if x == '(': st.append("(")      # 进栈
6      if x == ")":
7          if len(st)!= 0:              # len(): 栈的长度
8              st.pop()                 # 出栈,也可以写成 del st[-1],st[-1]是栈顶
9          else:                        # 栈已空,没有匹配的左括号
10             flag = False
11             break
12 if len(st) == 0 and flag: print('YES')
13 else: print('NO')
```

再用一道例题说明栈的应用。

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：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 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”大才行。

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合适。

以“6 4 2 8”为例，用栈模拟找“凹”。若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。在图 3.5 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。

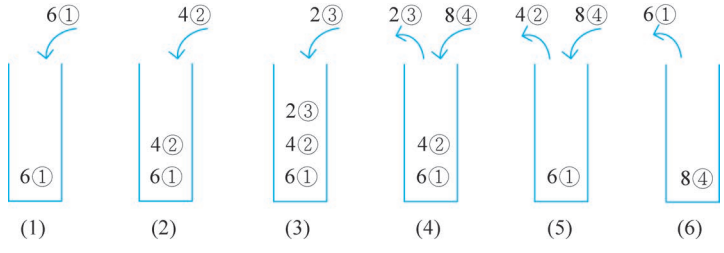


图 3.5　用栈统计“凹”

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“6 4 2 8”的总价值是 3＋4＋6＝13。

下面是例题的代码。

```
n = int(input())
a = [int(x) for x in input().split()]
st = []                                          # 定义栈，用 list 实现
ans = 0
for i in range(n):
    while len(st) != 0 and a[st[-1]] < a[i]:     # st[-1]是栈顶
        st.pop()                                 # 弹出栈顶
        if len(st) != 0:
            last = st[-1]                        # 读栈顶
            ans += (i - last + 1)
    st.append(i)                                 # 进栈
ans += (n - 1) * 2
print(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

前几节介绍的数组、队列、栈、链表等数据结构都是线性的，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。

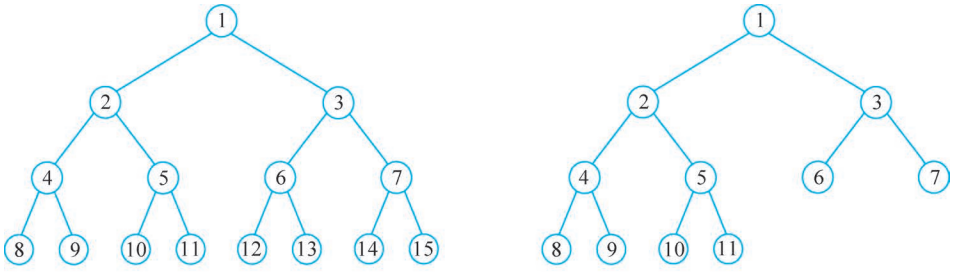
(1) 满二叉树。其特征是每一层的节点数都是满的。第一层只有一个节点，编号为 1；

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。

(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。



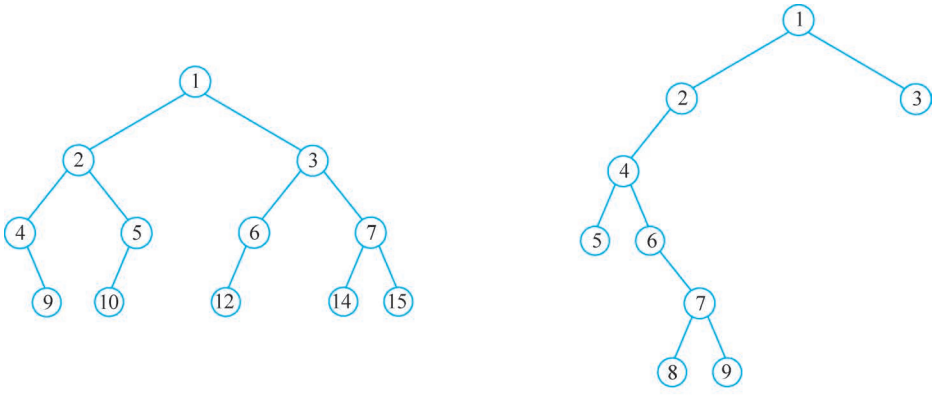
(1) 满二叉树　　(2) 完全二叉树

图3.6　满二叉树和完全二叉树

(3) 平衡二叉树。如果一棵二叉树的任意左子树和右子树的高度差不大于1，称之为平衡二叉树。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。

(4) 退化二叉树[①]。如果二叉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。



(1) 平衡二叉树　　(2) 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

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。"

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。二叉树中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，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，首先要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。

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。

```
1  class Node:                          # 定义二叉树结构体
2      def __init__(self):
3          self.v = ''                  # 把 value 简写为 v
4          self.ls = 0                  # 左右孩子，把 lson、rson 简写为 ls、rs
5          self.rs = 0
6  t = [Node() for i in range(N)]       # 把 tree 简写为 t
```

这段代码定义了一个二叉树的类。首先定义了一个名为 Node 的类，表示二叉树的节点。每个节点有 3 个属性：v 表示节点的值，ls 表示左孩子节点的索引，rs 表示右孩子节点的索引。然后创建一个名为 t 的列表，创建了 N 个 Node 对象，用于构建二叉树。代码中缩写的命名是为了简洁和方便理解，v 代表 value，ls 和 rs 代表 lson 和 rson。

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。

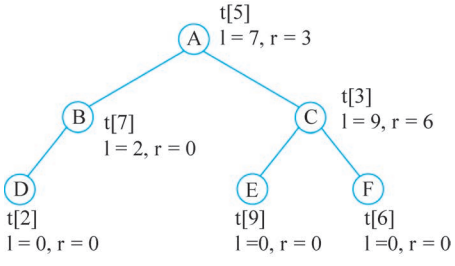


图 3.8 二叉树的静态存储

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。在该图中把 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。

2. 二叉树存储的编码实现

下面编写代码演示图 3.8 中二叉树的建立，并输出二叉树。

第 13～18 行建立二叉树，然后用 print_tree()输出二叉树。

```
N = 100
class Node:                          # 定义静态二叉树结构体
    def __init__(self):
        self.v = ''                  # 把 value 简写为 v
        self.ls = 0                  # 左右孩子，把 lson、rson 简写为 ls、rs
        self.rs = 0
t = [Node() for i in range(N)]       # 把 tree 简写为 t
def print_tree(u):                   # 打印二叉树
    if u:
        print(t[u].v, end=' ')       # 打印节点 u 的值
        print_tree(t[u].ls)          # 继续搜左孩子
        print_tree(t[u].rs)          # 继续搜右孩子
t[5].v, t[5].ls, t[5].rs = 'A', 7, 3
t[7].v, t[7].ls, t[7].rs = 'B', 2, 0
t[3].v, t[3].ls, t[3].rs = 'C', 9, 6
t[2].v = 'D'                         # t[2].ls=0; t[2].rs=0; 可以不写，因为 t[]已初始化为 0
t[9].v = 'E'                         # t[9].ls=0; t[9].rs=0; 可以不写
t[6].v = 'F'                         # t[6].ls=0; t[6].rs=0; 可以不写
root = 5                             # 根是 tree[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，然后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，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。

在图 3.9 中，圆圈内是节点的值，圆圈外的数字是节点的存储位置。

下面的代码，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。

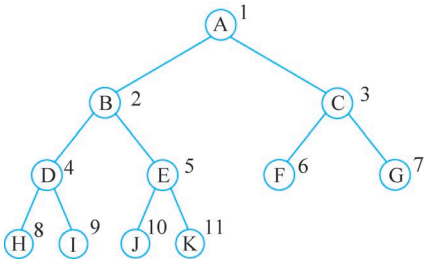


图 3.9 一棵完全二叉树

```
N = 100
t = [''] * N                                                    # 简单地用一个数组定义二叉树
def ls(p): return p << 1                                        # 定位左孩子，也可以写成 p * 2
def rs(p): return (p << 1) | 1                                  # 定位右孩子，也可以写成 p * 2 + 1
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'
print(t[1] + ':lson = ' + t[ls(1)] + ' rson = ' + t[rs(1)])     # 输出 A:lson = B rson = C
print(t[5] + ':lson = ' + t[ls(5)] + ' rson = ' + t[rs(5)])     # 输出 E:lson = J rson = K
```

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，二叉树的形态天然适合用递归来编程。图 3.10 所示为一个二叉树例子。

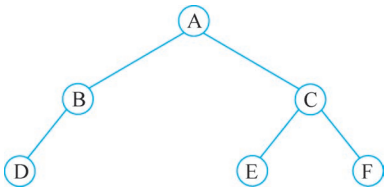


图 3.10 二叉树例子

（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。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，每棵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。每棵子树也满足中序遍历，例如 ECF，先左孩子 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。每棵子树也满足后序遍历，例如 EFC，先左孩子 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：给定一棵有 n（$n \leqslant 10^6$）个节点的二叉树，给出每个节点的两个子节点编号（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。第一行是这棵二叉树的先序遍历，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。

```
N = 100005
t = [0] * N                          # tree[0]不用,0 表示空节点
class Node:
    def __init__(self, v, ls, rs):
        self.v = v
        self.ls = ls
        self.rs = rs
def preorder(p):                     # 求先序序列
    if p != 0:
        print(t[p].v, end = ' ')     # 先序输出
        preorder(t[p].ls)
        preorder(t[p].rs)
def midorder(p):                     # 求中序序列
    if p != 0:
        midorder(t[p].ls)
        print(t[p].v, end = ' ')     # 中序输出
        midorder(t[p].rs)
def postorder(p):                    # 求后序序列
    if p != 0:
        postorder(t[p].ls)
        postorder(t[p].rs)
        print(t[p].v, end = ' ')     # 后序输出
n = int(input())
for i in range(1, n + 1):
    a, b = map(int, input().split())
    t[i] = Node(i, a, b)
preorder(1);print()
midorder(1);print()
postorder(1); print()
```

再看一道例题。

例 3.17　2023 年第十四届蓝桥杯省赛 Python 大学 A 组试题 I：子树的大小 lanqiaoOJ 3526

时间限制：15s 内存限制：512MB 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。

请求出第 k 个节点对应的子树拥有的节点数量。

输入：输入包含多组查询。输入的第一行包含一个整数 T，表示查询次数。接下来 T 行，每行包含 3 个整数 n、m、k，表示一组查询。

输出：输出一个正整数，表示答案。

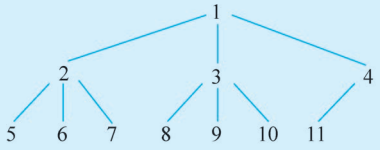


图 3.11　一棵完全三叉树

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，$1 \leqslant T \leqslant 10^5$，$1 \leqslant k \leqslant n \leqslant 10^9$，$2 \leqslant m \leqslant 10^9$。

这道题可以帮助大家理解树的结构。

第 u 个节点的最左孩子的编号是多少？第 u 号点前面有 u－1 个点，每个点各有 m 个孩子，再加上 1 号点，可得第 u 个点的左孩子下标为(u－1)×m＋2。例如图 3.11 中的 3 号点，求它的最左孩子的编号。3 号点前面有两个点，即 1 号点和 2 号点，每个点都有 3 个孩子，1 号点的孩子是{2，3，4}，2 号点的孩子是{5，6，7}，共 6 个孩子。那么 3 号点的最左孩子的编号是 1＋2×3＋1＝8。

同理，如果第 u 个节点的孩子是满的，它的最右孩子的编号为 u×m＋1。

分析第 u 个节点的情况：

① 点 u 在最后一层。此时点 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，也就是说点 u 在最后一层，那么以点 u 为根的子树的节点数量是 1，就是 u 自己。

② 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 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 将在第 6 章讲解，请大家在学过第 6 章以后回头看这个方法。

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 组查询，所以总复杂度是 O(nt)，只能通过 40％的测试。

```
1  def dfs(n,m,u):
2      ans = 1                                  #u点自己算一个
3      if m*u-(m-2)>n: return ans               # 情况1),u点在最后一层, ans=1
4      elif m*u+1<=n:                           # 情况2),u在倒数第2层且孩子满了
5          for c in range(m*u-(m-2),m*u+2):     # 深搜u的每个孩子
6              ans += dfs(n,m,c)                # 累加每个孩子的数量
7          return ans
8      else: return n + m - m*u                 # 情况3),u在倒数第2层且孩子不满
```

```
t = int(input())
for _ in range(t):
    n,m,k = map(int,input().split())
    print(dfs(n,m,k))
```

（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。

其实每一层算一次即可，在情况②时每一层也只需要算一次。以题目中的图为例，计算以点 1 为根的节点数量。1 号点这一层有一个节点；它的下一层是满的，有 3 个节点，左孩子是 2，右孩子是 4；再下一层，2 号点的左孩子是 5，4 号点的孩子是 11，那么这一层有 11－5＋1＝7 个节点。累加得 1＋3＋7＝11。

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

```
t = int(input())
for _ in range(t):
    n, m, k = map(int, input().split())
    ans = 1                              # k 点自己算一个
    ls,rs = k,k                          # 从 k 点开始，分析它的最左和最右孩子
    while True:
        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            # 加上孩子的数量
            break
        ans += rs - ls + 1               # 情况 2)，这一层是满的，累加这一层的所有孩子
    print(ans)
```

再看一道例题。

例 3.18　FBI 树　https://www.luogu.com.cn/problem/P1087

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”的串则称为 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 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，把它们上一层的父节点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。

```
def ls(p): return p<<1                      # 定位左孩子：p*2
def rs(p): return p<<1|1                    # 定位右孩子：p*2+1
def build_FBI(p,left,right):
    if left==right:                         # 到达叶子节点
        if s[right]=='1': tree[p]='I'
        else: tree[p]='B'
        return
    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' and tree[rs(p)]=='B':
        tree[p]='B'                         # 左右孩子都是B,自己也是B
    elif tree[ls(p)]=='I' and tree[rs(p)]=='I':
        tree[p]='I'                         # 左右孩子都是I,自己也是I
    else: tree[p]='F'
def postorder(p):                           # 后序遍历
    if tree[ls(p)]!='': postorder(ls(p))
    if tree[rs(p)]!='': postorder(rs(p))
    print(tree[p],end='')
n=int(input())
s=input().strip()
tree=['']*4400                              # tree[]存满二叉树
build_FBI(1,0,len(s)-1)
postorder(1)
```

【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”。

并查集的英文为 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

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！”

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①”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。

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。

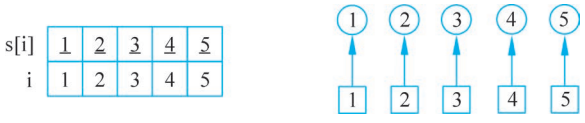


图 3.12 并查集的初始化

(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],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。

图 3.13 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 2 的集,s[2]=2,2 的集是自己,无法继

① 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。”

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 3.14 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。

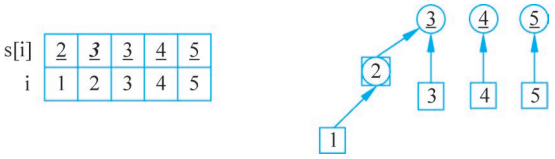


图 3.14　合并(1，3)

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。4 是 1、2、3、4 的帮主。另外还有一个独立的集 s[5]=5。

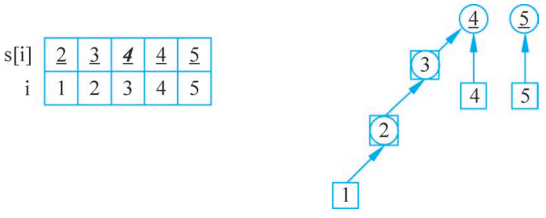


图 3.15　合并(2，4)

(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 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：若某个家族过于庞大，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：第一行包含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，查询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，查询P_i和P_j是否具有亲戚关系。

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

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

下面是并查集的代码。

```
N = 5010
s = []                          # 定义并查集
def init_set():                 # 并查集的初始化
    global s
    #s = [i for i in range(N)]   # 与下一行的功能一样
    s = list(range(N))
def find_set(x):                # 查找
    global s
    #if x == s[x]: return x
    #else: return find_set(s[x])  # 将这两行合并为下面一行
    return x if x == s[x] else find_set(s[x])
def merge_set(x, y):            # 合并
    global s
    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y的集
def main():
    init_set()
    n, m, p = map(int, input().split())
    for _ in range(m):          # 合并
        x, y = map(int, input().split())
        merge_set(x, y)
    for _ in range(p):          # 查询
        x, y = map(int, input().split())
        if find_set(x) == find_set(y): print("Yes")
        else: print("No")
if __name__ == "__main__": main()
```

（1）初始化init_set()。

（2）查找find_set()。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。

下一节用路径压缩来优化并查集的退化问题。

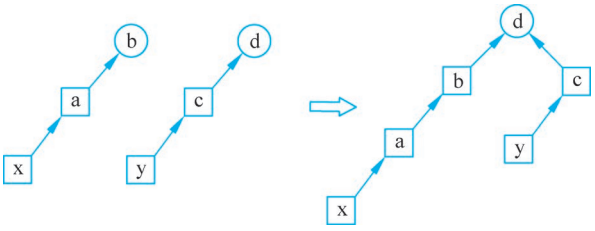


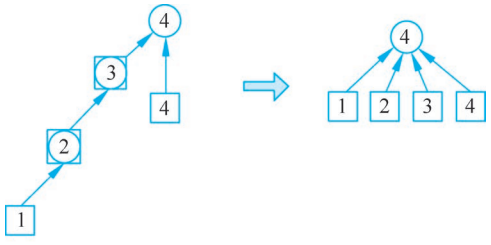
图 3.16　合并

3.8.2　路径压缩

在做并查集题目时，大家一定会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。

在上一节的查询函数 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），那么下次再查询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 3.17(a)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，如图 3.17(b)所示。这样下次再查询 1、2、3、4 所属的集，只需要递归一次就查到了根。



(a)　　(b)

图 3.17　路径压缩

路径压缩的代码非常简单。把上一节代码中的 find_set()改成以下路径压缩的代码。

```
1  def find_set(x):
2      if(x != s[x])s[x] = find_set(s[x])        # 路径压缩
3      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，查询和合并平均都是 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被损坏而无法通车。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存在至少一条修复完成的道路（可以由多条公路连成一条道路）。

输入：第一行包含两个正整数 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，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 16～19 行，做了灵活处理。第 18 行，如果村庄 x 和 y 已经连通，那么连通的村庄数量不用增加；第 19 行，如果村庄 x 和 y 没有连通，则合并并查集。

```
1  def find_set(x, s):                          # 用“路径压缩”优化的查询
2      if x != s[x]: s[x] = find_set(s[x], s)   # 路径压缩
3      return s[x]
4  def main():
5      n, m = map(int, input().split())
6      #s = [i for i in range(m+1)]              # 并查集的初始化
7      s = list(range(m+1))                     # 和上一行的功能一样
8      e = []
9      for _ in range(m):
10         x, y, t = map(int, input().split())
11         e.append((x, y, t))
12     e.sort(key=lambda x: x[2])               # 按时间 t 排序
13     ans = 0                                  # 答案，最早通车时间
14     num = 0                                  # 已经连通的村庄数量
15     for i in range(m):
16         x = find_set(e[i][0], s)             # 第 18～20 行
17         y = find_set(e[i][1], s)
18         if x == y: continue                  # x 和 y 已经连通，num 不用增加
19         s[x] = y                             # 合并并查集，即把村庄 x 合并到 y 上
20         num += 1                             # 连通的村庄加 1
21         ans = max(ans, e[i][2])              # 当前最大通车时间
22     if num != n-1:print("-1")
```

```
23      else: print(ans)
24  if __name__ == "__main__":main()
```

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

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。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，小明会给 A_i 加 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)，编写代码提交可能通过 30％的测试。

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～A_{i-1} 中出现过。"这导致在某些情况下仍然需要做大量的检查。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  N = 1000002                          # A 的 hash,1≤Ai≤1000000
2  vis = [0] * N                        # hash: vis[i] = 1 表示数字 i 已经存在
3  n = int(input())
```

```
A = [int(i) for i in input().split()] # 读一行 n 个整数
for a in A:
    while vis[a] == 1:                # 若 a 已经出现过,加 1。若加 1 后再出现,则继续加 1
        a += 1
    vis[a] = 1                        # 标记该数字
    print(a, end = " ")               # 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做大量的检查,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,如图 3.18 所示。初始化时 set[i]=i。

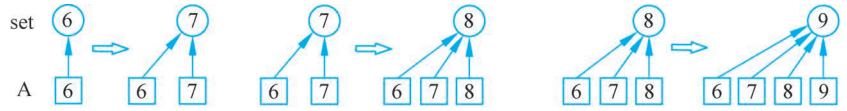


图 3.18 用并查集处理数组 A

图 3.18(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 3.18(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,作用是后面再读到 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 3.18(c)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,没用到合并。这里必须用“路径压缩”优化才能加快查询速度,通过 100%的测试,如果没有用路径压缩,仍然会超时。

```
def find_set(x, s):                          # 用“路径压缩”优化的查询
    if x != s[x]: s[x] = find_set(s[x], s)   # 路径压缩
    return s[x]
def main():
    N = 1000002
    A = [0] * N
    s = list(range(N))                       # 定义并查集,并完成初始化
    n = int(input())
    A = [0] + [int(i) for i in input().split()]   # 读一行 n 个整数,存到 A[1]~A[N]
    for i in range(1, n + 1):
        root = find_set(A[i], s)             # 查询到并查集的根
        A[i] = root
        s[root] = find_set(root + 1, s)      # 加 1
    #for i in range(1, n + 1): print(A[i], end = " ")
    print( * A[1:])                          # 功能和上一行一样,打印 A[1]~A[N]
if __name__ == "__main__": main()
```

【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：数组、链表、队列、栈、二叉树。在竞赛中既可以用库函数来实现题目，也可以手写代码实现。对于库函数，大家应该重点掌握，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，程序员应该能不假思索地、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。