

基本代码：

序列

切片：

```
list[a:b:c]  
#a为起始下标, b为结束下标, c为步长, 左开右闭  
  
#按照元素的长度进行排序  
tele = sorted(tele, key=lambda x: len(x), reverse=True)
```

tips：通过切片可以获得原列表的一个副本，这样后续更改一个列表的内容时不会影响其他副本的值

排序：

```
list=list.sort()#从小到大（按照字典序, 如果元素是序列, 则从第一个往后依次比较）  
list=list.sort(reverse=True)#从大到小
```

插入：

```
list.insert(i,x)#表示在列表的第i索引处插入x, 插入后x的索引就是i
```

查找某元素数量：

```
num1=list.count(a)
```

查找某一元素的下标：

```
str.find(a,b,c)  
  
list.index(a,b,c)
```

其中a表示要找的元素, b表示起始索引, c表示终止索引, 左开右闭.

区别：find只适用于字符串, index适用于列表和字符串。

find找不到会返回-1, index找不到会报ValueError

```
#字典  
a=dict()  
a={}  
c=sorted(a)#将字典的键进行排序, 返回一个有序的键列表  
#a={1:1,1:2}  
#c=sorted(a)  
#c=[1]  
  
#get() 方法用于获取字典中指定键的值。其语法为 dictionary.get(key, default=None)。如果键存在于字典中, 则返回对应的值; 如果键不存在, 则返回默认值 (如果提供了默认值), 否则返回 None。这对于避免 KeyError 非常有用  
ver.get(vert, None)  
  
#items() 方法用于返回字典的键值对视图。该视图以元组的形式返回字典中的键值对, 允许迭代遍历字典中的键值对。语法为 dictionary.items()。例如:
```

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key, value in my_dict.items():
    print(key, value)
# 输出:
# a 1
# b 2
# c 3
```

去除一个列表中重复的元素:

```
numbers = list(map(int, input().split()))
numbers = list(dict.fromkeys(numbers)) # remove duplicates
```

```
for y in range(m):
    print(long1[x][y], end=" ")
print()

#假设 row 是 [1, 2, 3]，那么 print(*row) 就等价于 print(1, 2, 3)，它会打印出每个元素之间用空格分隔的内容。这种语法对于打印列表、元组等可迭代对象的内容非常方便。

#对于可迭代对象，可以通过*解决
#media=[1,2,3,4]
#print(*media)
#print(*media)
#输出: 1 2 3 4
```

输出列表内所有元素:

```
print(*lst) ##把列表中元素顺序输出
```

保留小数的多种方式:

```
print("{:.2f}".format(3.146)) # 3.15
print(round(3.123456789,5)) # 3.12346四舍六入五成双
#保留n位小数:
print(f'{x:.nf}')
```

不定行输入: 套用try-except循环

Try except 无法使用break来跳出循环, 如果已知结束的特定输入 (比如说输入0代表结束),

那么可以用以下代码:

```
while True:
    try:
        XXXXX
        if input()=='0':
            break
    except EOFError:
        break
```

动态规划：

02945: 拦截导弹

<http://cs101.openjudge.cn/practice/02945/>

某国为了防御敌国的导弹袭击，开发出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭，并观测到导弹依次飞来的高度，请计算这套系统最多能拦截多少导弹。拦截来袭导弹时，必须按来袭导弹袭击的时间顺序，不允许先拦截后面的导弹，再拦截前面的导弹。

输入

输入有两行，第一行，输入雷达捕捉到的敌国导弹的数量k ($k \leq 25$)，第二行，输入k个正整数，表示k枚导弹的高度，按来袭导弹的袭击时间顺序给出，以空格分隔。

输出

输出只有一行，包含一个整数，表示最多能拦截多少枚导弹。

样例输入

```
8  
300 287 155 300 299 178 158 65
```

```
k=int(input())  
l=list(map(int,input().split()))  
dp=[0]*k  
for i in range(k-1,-1,-1):  
    maxn=1  
    for j in range(k-1,i,-1):  
        if l[i]>=l[j] and dp[j]+1>maxn:  
            maxn=dp[j]+1  
    dp[i]=maxn  
print(max(dp))
```

递归：

04147: 汉诺塔问题(Tower of Hanoi)

recursion, <http://cs101.openjudge.cn/practice/04147/>

一、汉诺塔问题

有三根杆子A, B, C, A杆上有N个($N > 1$)穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至C杆：每次只能移动一个圆盘；大盘不能叠在小盘上面。提示：可将圆盘临时置于B杆，也可将从A杆移出的圆盘重新移回A杆，但都必须遵循上述两条规则。

问：如何移？最少要移动多少次？

```
# 将编号为numDisk的盘子从init杆移至desti杆  
def moveOne(numDisk : int, init : str, desti : str):  
    print("{}:{}->{}".format(numDisk, init, desti))  
  
# 将numDisk个盘子从init杆借助temp杆移至desti杆  
def move(numDisks : int, init : str, temp : str, desti : str):  
    if numDisks == 1:  
        moveOne(1, init, desti)  
    else:  
        # 首先将上面的 (numDisk-1) 个盘子从init杆借助desti杆移至temp杆  
        move(numDisks-1, init, desti, temp)  
  
        # 然后将编号为numDisk的盘子从init杆移至desti杆  
        moveOne(numDisks, init, desti)  
  
        # 最后将上面的 (numDisk-1) 个盘子从temp杆借助init杆移至desti杆  
        move(numDisks-1, temp, init, desti)  
  
n, a, b, c = input().split()  
move(int(n), a, b, c)
```

欧拉筛：

```
n = int(input())  
x = [int(i) for i in input().split()]  
s = [True] * (10 ** 6 + 1)  
euler(10 ** 6, s)  
for i in x:  
    if i < 4:  
        print('NO')  
        continue  
    elif int(i ** 0.5) ** 2 != i:  
        print('NO')  
        continue  
    if s[int(i ** 0.5)]:  
        print('YES')  
    else:  
        print('NO')
```

栈

02694: 波兰表达式

recursion/strings, <http://cs101.openjudge.cn/practice/02694>

波兰表达式是一种把运算符前置的算术表达式，例如普通的表达式 $2 + 3$ 的波兰表示法为 $+ 2 3$ 。波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序，例如 $(2 + 3) * 4$ 的波兰表示法为 $* + 2 3 4$ 。本题求解波兰表达式的值，其中运算符包括 $+, -, *$ 四个。

输入

输入为一行，其中运算符和运算数之间都用空格分隔，运算数是浮点数。

输出

输出为一行，表达式的值。可直接用`printf("%f\n", v)`输出表达式的值`v`。

思路：最一开始接触到函数递归调用就是这道题，当时惊赞题解里的做法真的是妙哉。现在学了栈这种数据结构，我发现可以写一个不用函数递归调用的版本。基本想法如下：从后往前读取表达式，碰见数字则压入栈，碰见运算符号则从栈顶弹出两个数据进行计算，将计算结果再压入栈即可。

```
# http://cs101.openjudge.cn/practice/02694/
expression = input().split()
stack = []
while expression:
    a = expression.pop(-1)
    if a in ['+', '-', '*', '/']:
        c = stack.pop(-1)
        d = stack.pop(-1)
        if a == '+':
            stack.append(c + d)
        elif a == '-':
            stack.append(c - d)
        elif a == '*':
            stack.append(c * d)
        else:
            stack.append(c / d)
    else:
        stack.append(float(a))
print("{:.6f}".format(stack[0]))
```

24591: 中序表达式转后序表达式

<http://cs101.openjudge.cn/dsapre/24591/>

中序表达式是运算符放在两个数中间的表达式。乘、除运算优先级高于加减。可以用“()”来提升优先级 --- 就是小学生写的四则算术运算表达式。中序表达式可用如下方式递归定义：

1) 一个数是一个中序表达式。该表达式的值就是数的值。

2. 若`a`是中序表达式，则“`(a)`”也是中序表达式(引号不算)，值为`a`的值。

3. 若`a, b`是中序表达式，`c`是运算符，则“`acb`”是中序表达式。“`acb`”的值是对`a`和`b`做`c`运算的结果，且`a`是左操作数，`b`是右操作数。

输入一个中序表达式，要求转换成一个后序表达式输出。

输入

第一行是整数`n`($n < 100$)。接下来`n`行，每行一个中序表达式，数和运算符之间没有空格，长度不超过700。

输出

对每个中序表达式，输出转成后序表达式后的结果。后序表达式的数之间、数和运算符之间用一个空格分开。

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*':
                while stack and stack[-1] in '+-*' and precedence[char] <= precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
    print(" ".join(postfix))
```

```

if number:
    num = float(number)
    postfix.append(int(num)) if num.is_integer() else num

while stack:
    postfix.append(stack.pop())

return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))

```

22068: 合法出栈序列

<http://cs101.openjudge.cn/dsapre/22068/>

给定一个由大小写字母和数字构成的，没有重复字符的长度不超过62的字符串x，现在要将该字符串的字符依次压入栈中，然后再全部弹出。

要求左边的字符一定比右边的字符先入栈，出栈顺序无要求。

再给定若干字符串，对每个字符串，判断其是否是可能的x中的字符的出栈序列。

输入

第一行是原始字符串x 后面有若干行(不超过50行)，每行一个字符串，所有字符串长度不超过100

输出

对除第一行以外的每个字符串，判断其是否是可能的出栈序列。如果是，输出“YES”，否则，输出“NO”

```

def is_valid_pop_sequence(origin, output):
    if len(origin) != len(output):
        return False # 长度不同，直接返回False

    stack = []
    bank = list(origin)

    for char in output:
        # 如果当前字符不在栈顶，且bank中还有字符，则继续入栈
        while (not stack or stack[-1] != char) and bank:
            stack.append(bank.pop(0))

        # 如果栈为空，或栈顶字符不匹配，则不是合法的出栈序列
        if not stack or stack[-1] != char:
            return False

    stack.pop() # 匹配成功，弹出栈顶元素

    return True # 所有字符都匹配成功

# 读取原始字符串
origin = input().strip()

# 循环读取每一行输出序列并判断
while True:
    try:
        output = input().strip()
        if is_valid_pop_sequence(origin, output):
            print('YES')
        else:
            print('NO')
    except EOFError:
        break

```

字典：

27925: 小组队列

<http://cs101.openjudge.cn/practice/27925/>

有 n 个小组要排成一个队列，每个小组中有若干人。当一个人来到队列时，如果队列中已经有了自己小组的成员，他就直接插队排在自己小组成员的后面，否则就站在队伍的最后面。请你编写一个程序，模拟这种小组队列。

注意：每个人的编号不重复，另外可能有散客。

输入

第一行：小组数量 t ($t < 100$)。接下来 t 行，每行输入一个小组描述，表示这个小组的人的编号。编号是 0 到 999999 范围内的整数，一个小组最多可包含 1000 个人。最后，命令列表如下。有三种不同的命令：1、ENQUEUE x - 将编号是 x 的人插入队列；2、DEQUEUE - 让整个队列的第一个人出队；3、STOP - 测试用例结束 每个命令占一行，不超过 50000 行。

输出

对于每个 DEQUEUE 命令，输出出队的人的编号，每个编号占一行。

```
from collections import deque # 时间: 105ms

# Initialize groups and mapping of members to their groups
t = int(input())
groups = {}
member_to_group = {}

for _ in range(t):
    members = list(map(int, input().split()))
    group_id = members[0] # Assuming the first member's ID represents the group ID
    groups[group_id] = deque()
    for member in members:
        member_to_group[member] = group_id

# Initialize the main queue to keep track of the group order
queue = deque()
# A set to quickly check if a group is already in the queue
queue_set = set()

while True:
    command = input().split()
    if command[0] == 'STOP':
        break
    elif command[0] == 'ENQUEUE':
        x = int(command[1])
        group = member_to_group.get(x, None)
        # Create a new group if it's a new member not in the initial list
        if group is None:
            group = x
            groups[group] = deque([x])
            member_to_group[x] = group
        else:
            groups[group].append(x)
        if group not in queue_set:
            queue.append(group)
            queue_set.add(group)
    elif command[0] == 'DEQUEUE':
        if queue:
            group = queue[0]
            x = groups[group].popleft()
            print(x)
            if not groups[group]: # If the group's queue is empty, remove it from the main queue
                queue.popleft()
                queue_set.remove(group)
        else:
            print('')

# n = int(input())
dic = {}
time = list(map(int, (input().split())))
# 记录学生编号及实验时间
for i in range(n):
    dic[i+1] = time[i]
a = sorted(dic.items(), key=lambda x: x[1]) # 按实验时间从小至大排序
student_sort = []
time_sort = []
for i in range(n):
    student_sort.append(a[i][0])
    time_sort.append(a[i][1])
time_wait = 0
for j in range(n-1):
    time_wait += time_sort[j] * (n-j-1) / n # 计算平均等待时间
print(''.join(str(k) for k in student_sort))
print('.2f' % time_wait)
```

21554: 排队做实验

greedy, <http://cs101.openjudge.cn/practice/21554/>

某学院的学生都需要在某月的1号到2号期间完成课程实验，他们每个人的实验需要持续不同的时长。而实验室管理员要安排这些学生，按照一定顺序，在1号到2号期间全部完成实验。假设该学院的实验室同时只能容纳一名学生实验，而这些学生都非常积极，都希望被排在1号的第一个尽快完成实验。

假设该学院有n个学生，实验室管理员收到了n个学生每位需要占用实验室的时长T1,T2,...,Tn，由于学生发送预约邮件的时间比较接近，没办法完全按照先到先得的办法给学生分配实验时间（实验时间相同的话，先到先得）。管理员很犯愁，他希望有一种能让所有学生平均等待时间尽可能小的顺序，来安排这n位同学的实验时间，请问你能帮帮他吗？



27300: 模型整理

<http://cs101.openjudge.cn/practice/27300/>

深度学习模型（尤其是大模型）是近两年计算机学术和业界热门的研究方向。每个模型可以用“模型名称-参数量”命名，其中参数量的单位会使用两种：M，即百万；B，即十亿。同一个模型通常有多个不同参数的版本。例如，Bert-110M，Bert-340M 分别代表参数量为 1.1 亿和 3.4 亿的 Bert 模型，GPT-350M，GPT-1.3B 和 GPT-175B 分别代表参数量为 3.5 亿，13 亿和 1750 亿的 GPT 模型。参数量的数字部分取值在 [1, 1000] 区间（一个 8 亿参数的模型表示为 800M 而非 0.8B，10 亿参数的模型表示为 1B 而非 1000M）。计算机专业的学生小 A 从网上收集了一份模型列表，他需要将它们按照名称归类排序，并且同一个模型的参数量从小到大排序，生成“模型名称: 参数量1, 参数量2, ...”的列表。请你帮他写一个程序实现。

输入

第一行为一个正整数 n ($n \leq 1000$)，表示有 n 个待整理的模型。

接下来 n 行，每行一个“模型名称-参数量”的字符串。模型名称是字母和数字的混合。

输出

每行一个“模型名称: 参数量1, 参数量2, ...”的字符串，符号均为英文符号，模型名称按字典序排列，参数量按从小到大排序。

样例输入

```
5
GPT-1.3B
Bert-340M
GPT-350M
Bert-110M
GPT-175B
```



样例输出

```
Bert: 110M, 340M
GPT: 350M, 1.3B, 175B
```



```
# from collections import defaultdict

n = int(input())
typ_dict = defaultdict(list)
for i in range(n):
    typ, size = map(str, input().split('-'))
    if size[-1] == 'M':
        typ_dict[typ].append((size, float(size[:-1])/1000)) # value为二位列表，便于后续排序
    else:
        typ_dict[typ].append((size, float(size[:-1])))
typ_sort = sorted(typ_dict) # 按key值排序
for j in typ_sort:
    size_sort = sorted(typ_dict[j], key=lambda x: x[1]) # 按value值排序
    value = ', '.join(k[0] for k in size_sort)
    print('{}: {}'.format(j, value))
```

树

27638: 求二叉树的高度和叶子数目

<http://cs101.openjudge.cn/dsapre/27638/>

给定一棵二叉树，求该二叉树的高度和叶子数目二叉树高度定义：从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的结点数减1为树的高度。只有一个结点的二叉树，高度是0。

输入

第一行是一个整数n，表示二叉树的结点个数。二叉树结点编号从0到n-1，根结点n <= 100 接下来有n行，依次对应二叉树的编号为0,1,2,...,n-1的节点。每行有两个整数，分别表示该节点的左儿子和右儿子的编号。如果第一个（第二个）数为-1则表示没有左（右）儿子。

输出

在一行中输出2个整数，分别表示二叉树的高度和叶子结点个数

由于输入无法分辨谁为根节点，所以写寻找根节点语句。

```
class TreeNode:  
    def __init__(self):  
        self.left = None  
        self.right = None  
  
def tree_height(node):  
    if node is None:  
        return -1 # 根据定义，空树高度为-1  
    return max(tree_height(node.left), tree_height(node.right)) + 1  
  
def count_leaves(node):  
    if node is None:  
        return 0  
    if node.left is None and node.right is None:  
        return 1  
    return count_leaves(node.left) + count_leaves(node.right)  
  
n = int(input()) # 读取节点数量  
nodes = [TreeNode() for _ in range(n)]  
has_parent = [False] * n # 用来标记节点是否有父节点  
  
for i in range(n):  
    left_index, right_index = map(int, input().split())  
    if left_index != -1:  
        nodes[i].left = nodes[left_index]  
        has_parent[left_index] = True  
    if right_index != -1:  
        nodes[i].right = nodes[right_index]  
        has_parent[right_index] = True  
  
# 寻找根节点，也就是没有父节点的节点  
root_index = has_parent.index(False)  
root = nodes[root_index]  
  
# 计算高度和叶子节点数  
height = tree_height(root)  
leaves = count_leaves(root)  
  
print(f'{height} {leaves}')
```

24729: 括号嵌套树

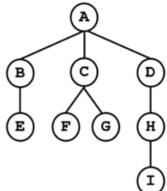
<http://cs101.openjudge.cn/practice/24729/>

可以用括号嵌套的方式来表示一棵树。表示方法如下：

1. 如果一棵树只有一个结点，则该树就用一个大写字母表示，代表其根结点。
2. 如果一棵树有子树，则用“树根(子树1,子树2,...,子树n)”的形式表示。树根是一个大写字母，子树之间用逗号隔开，没有空格。子树都是用括号嵌套法表示的树。

给出一棵不超过26个结点的树的括号嵌套表示形式，请输出其前序遍历序列和后序遍历序列。

输入样例代表的树如下图：



输入

一行，一棵树的括号嵌套表示形式

输出

两行。第一行是树的前序遍历序列，第二行是树的后序遍历序列

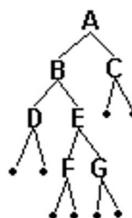
样例输入

```
A(B(E),C(F,G),D(H(I)))
```

```
class TreeNode:  
    def __init__(self, value): #类似字典  
        self.value = value  
        self.children = []  
  
def parse_tree(s):  
    stack = []  
    node = None  
    for char in s:  
        if char.isalpha(): # 如果是字母，创建新节点  
            node = TreeNode(char)  
        if stack: # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中  
            stack[-1].children.append(node)  
        elif char == '(': # 遇到左括号，当前节点可能会有子节点  
            if node:  
                stack.append(node) # 把当前节点推入栈中  
                node = None  
        elif char == ')': # 遇到右括号，子节点列表结束  
            if stack:  
                node = stack.pop() # 弹出当前节点  
    return node # 根节点  
  
  
def preorder(node):  
    output = [node.value]  
    for child in node.children:  
        output.extend(preorder(child))  
    return ''.join(output)  
  
def postorder(node):  
    output = []  
    for child in node.children:  
        output.extend(postorder(child))  
    output.append(node.value)  
    return ''.join(output)  
  
# 主程序  
def main():  
    s = input().strip()  
    s = ''.join(s.split()) # 去掉所有空白字符  
    root = parse_tree(s) # 解析整棵树  
    if root:  
        print(preorder(root)) # 输出前序遍历序列  
        print(postorder(root)) # 输出后序遍历序列  
    else:  
        print("input tree string error!")  
  
if __name__ == "__main__":  
    main()
```

<http://cs101.openjudge.cn/dsapre/08581/>

由于先序、中序和后序序列中的任一个都不能唯一确定一棵二叉树，所以对二叉树做如下处理，将二叉树的结点用补齐，如图所示。我们把这样处理后的二叉树称为原二叉树的扩展二叉树，扩展二叉树的先序和后序序列能唯一确定其二叉树。现给出扩展二叉树的先序序列，要求输出其中序和后序序列。



输入

扩展二叉树的先序序列（全部都由大写字母或者.组成）

输出

第一行：中序序列 第二行：后序序列

样例输入

ABD..EF..G...C..



嵌套括号表示法 Nested parentheses representation。直接用元组 (root, left, right) 来代表一棵树。

ABD..EF..G..C.. ('A', ('B', ('D', None, None), ('E', ('F', None, None), ('G', None, None))), ('C', None, None))

```

def build_tree(preorder):
    if not preorder or preorder[0] == '.':
        return None, preorder[1:]
    root = preorder[0]
    left, preorder = build_tree(preorder[1:])
    right, preorder = build_tree(preorder)
    return (root, left, right), preorder

def inorder(tree):
    if tree is None:
        return ''
    root, left, right = tree
    return inorder(left) + root + inorder(right)

def postorder(tree):
    if tree is None:
        return ''
    root, left, right = tree
    return postorder(left) + postorder(right) + root

# 输入处理
preorder = input().strip()

# 构建扩展二叉树
tree, _ = build_tree(preorder)

# 输出结果
print(inorder(tree))
print(postorder(tree))
  
```

25140: 根据后序表达式建立队列表达式

<http://cs101.openjudge.cn/practice/25140/>

后序算术表达式可以通过栈来计算其值，做法就是从左到右扫描表达式，碰到操作数就入栈，碰到运算符，就取出栈顶的2个操作数做运算(先出栈的是第二个操作数，后出栈的是第一个)，并将运算结果压入栈中。最后栈里只剩下1个元素，就是表达式的值。

有一种算术表达式不妨叫做“队列表达式”，它的求值过程和后序表达式很像，只是将栈换成了队列：从左到右扫描表达式，碰到操作数就入队列，碰到运算符，就取出队头2个操作数做运算(先出队的是第2个操作数，后出队的是第1个)，并将运算结果加入队列。最后队列里只剩下一个元素，就是表达式的值。

给定一个后序表达式，请转换成等价的队列表达式。例如， $3 \ 4 + 6 \ 5 * -$ 的等价队列表达式就是 $5 \ 6 \ 4 \ 3 * + -$ 。

输入

第一行是正整数n(n<100)。接下来是n行，每行一个由字母构成的字符串，长度不超过100。表示一个后序表达式，其中小写字母是操作数，大写字母是运算符。运算符都是需要2个操作数的。

输出

对每个后序表达式，输出其等价的队列表达式。

```

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def build_tree(postfix):
    stack = []
    for char in postfix:
        node = TreeNode(char)
        if char.isupper():
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]

def level_order_traversal(root):
    queue = [root]
    traversal = []
    while queue:
        node = queue.pop(0)
        traversal.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return traversal

n = int(input().strip())
for _ in range(n):
    postfix = input().strip()
    root = build_tree(postfix)
    queue_expression = level_order_traversal(root)[::-1]
    print(''.join(queue_expression))

```

22158: 根据二叉树前中序序列建树

<http://cs101.openjudge.cn/practice/22158/>

假设二叉树的节点里包含一个大写字母，每个节点的字母都不同。

给定二叉树的前序遍历序列和中序遍历序列(长度均不超过26)，请输入该二叉树的后序遍历序列

输入

多组数据 每组数据2行，第一行是前序遍历序列，第二行是中序遍历序列

输出

对每组序列建树，输出该树的后序遍历序列

```

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_value = preorder[0]
    root = TreeNode(root_value)
    root_index_inorder = inorder.index(root_value)
    root.left = build_tree(preorder[1:root_index_inorder], inorder[:root_index_inorder])
    root.right = build_tree(preorder[root_index_inorder+1:], inorder[root_index_inorder+1:])
    return root

def postorder_traversal(root):
    if root is None:
        return ''
    return postorder_traversal(root.left) + postorder_traversal(root.right) + root.value

while True:
    try:
        preorder = input().strip()
        inorder = input().strip()
        root = build_tree(preorder, inorder)
        print(postorder_traversal(root))
    except EOFError:
        break

```

22275: 二叉搜索树的遍历

<http://cs101.openjudge.cn/practice/22275/>

给出一棵二叉搜索树的前序遍历，求它的后序遍历

输入

第一行一个正整数n ($n \leq 2000$) 表示这棵二叉搜索树的结点个数 第二行n个正整数，表示这棵二叉搜索树的前序遍历 保证第二行的n个正整数中，1~n的每个值刚好出现一次

输出

一行n个正整数，表示这棵二叉搜索树的后序遍历

建树思路：数组第一个元素是根节点，紧跟着是小于根节点值的节点，在根节点左侧，直至遇到大于根节点值的节点，后续节点都在根节点右侧，按照这个思路递归即可

```
class Node():
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def buildTree(preorder):
    if len(preorder) == 0:
        return None

    node = Node(preorder[0])

    idx = len(preorder)
    for i in range(1, len(preorder)):
        if preorder[i] > preorder[0]:
            idx = i
            break
    node.left = buildTree(preorder[1:idx])
    node.right = buildTree(preorder[idx:])

    return node

def postorder(node):
    if node is None:
        return []
    output = []
    output.extend(postorder(node.left))
    output.extend(postorder(node.right))
    output.append(str(node.val))

    return output

n = int(input())
preorder = list(map(int, input().split()))
print(' '.join(postorder(buildTree(preorder))))
```

并查集：

02524: 宗教信仰

disjoint set, <http://cs101.openjudge.cn/dsapre/02524/>

世界上有许多宗教，你感兴趣的是你学校里的同学信仰多少种宗教。

你的学校有n名学生 ($0 < n \leq 50000$)，你不太可能询问每个人的宗教信仰，因为他们不太愿意透露。但是当你同时找到2名学生，他们却愿意告诉你他们是否信仰同一宗教，你可以通过很多这样的询问估算学校里的宗教数目的上限。你可以认为每名学生只会信仰最多一种宗教。

输入

输入包括多组数据。每组数据的第一行包括n和m， $0 \leq m \leq n(n-1)/2$ ，其后m行每行包括两个数字i和j，表示学生i和学生j信仰同一宗教，学生被标号为1至n。输入以一行 $n = m = 0$ 作为结束。

输出

对于每组数据，先输出它的编号（从1开始），接着输出学生信仰的不同宗教的数目上限。

```
def init_set(n):
    return list(range(n))

def get_father(x, father):
    if father[x] != x:
        father[x] = get_father(father[x], father)
    return father[x]

def join(x, y, father):
    fx = get_father(x, father)
    fy = get_father(y, father)
    if fx == fy:
        return
    father[fx] = fy

def is_same(x, y, father):
    return get_father(x, father) == get_father(y, father)

def main():
    case_num = 0
    while True:
        n, m = map(int, input().split())
        if n == 0 and m == 0:
            break
        count = 0
        father = init_set(n)
        for _ in range(m):
            s1, s2 = map(int, input().split())
            join(s1 - 1, s2 - 1, father)
        for i in range(n):
            if father[i] == i:
                count += 1
        case_num += 1
        print(f"Case {case_num}: {count}")

if __name__ == "__main__":
    main()
```

dfs

18160: 最大连通域面积

matrix/dfs similar, <http://cs101.openjudge.cn/practice/18160>

一个棋盘上有棋子的地方用('W')表示,没有的地方用点来表示,现在要找出其中的最大连通区域,一个格子被视作和它周围八个格子都相邻。

现在需要找出最大的连通区域的面积是多少,一个格子代表面积为1。

输入

输入的第一行是一个整数,表示一共有T组数据。每组第一行包含两个整数N和M。接下来的N行,每行有M个字符('W'或者'.'),表示格子的当前状态。字符之间没有空格。

输出

每组数据对应一行,输出最大的连通域的面积,不包含任何空格。

```
dire = [[-1,-1],[-1,0],[-1,1],[0,-1],[0,1],[1,-1],[1,0],[1,1]]  
  
area = 0  
def dfs(x,y):  
    global area  
    if matrix[x][y] == '.':return  
    matrix[x][y] = '.'  
    area += 1  
    for i in range(len(dire)):  
        dfs(x+dire[i][0], y+dire[i][1])  
  
  
for _ in range(int(input())):  
    n,m = map(int,input().split())  
  
    matrix =[['.' for _ in range(m+2)] for _ in range(n+2)]  
    for i in range(1,n+1):  
        matrix[i][1:-1] = input()  
  
    sur = 0  
    for i in range(1, n+1):  
        for j in range(1, m+1):  
            if matrix[i][j] == 'W':  
                area = 0  
                dfs(i, j)  
                sur = max(sur, area)  
print(sur)
```



01426: Find The Multiple

<http://cs101.openjudge.cn/dsapre/01426/>

Given a positive integer n, write a program to find out a nonzero multiple m of n whose decimal representation contains only the digits 0 and 1. You may assume that n is not greater than 200 and there is a corresponding m containing no more than 100 decimal digits.

输入

The input file may contain multiple test cases. Each line contains a value of n ($1 \leq n \leq 200$). A line containing a zero terminates the input.

输出

For each value of n in the input print a line containing the corresponding value of m. The decimal representation of m must not contain more than 100 digits. If there are multiple solutions for a given value of n, any one of them is acceptable.

```

from collections import deque

def find_multiple(n):
    # 使用队列实现BFS
    q = deque()
    # 初始化队列，存储的是(模n值, 对应的数字字符串)
    q.append((1 % n, "1"))
    visited = set([1 % n])  # 用于记录访问过的模n值，避免重复搜索

    while q:
        mod, num_str = q.popleft()

        # 检查当前模n值是否为0，是则找到答案
        if mod == 0:
            return num_str

        # 尝试在当前数字后加0或加1，生成新的数字，并计算模n值
        for digit in ["0", "1"]:
            new_num_str = num_str + digit
            new_mod = (mod * 10 + int(digit)) % n

            # 如果新模n值未访问过，则加入队列继续搜索
            if new_mod not in visited:
                q.append((new_mod, new_num_str))
                visited.add(new_mod)

def main():
    while True:
        n = int(input())
        if n == 0:
            break
        print(find_multiple(n))

if __name__ == "__main__":
    main()

```

bfs :

04115: 鸣人和佐助

bfs, <http://cs101.openjudge.cn/practice/04115/>

佐助被大蛇丸诱骗走了，鸣人在多少时间内能追上他呢？

已知一张地图（以二维矩阵的形式表示）以及佐助和鸣人的位置。地图上的每个位置都可以走到，只不过有些位置上有大蛇丸的手下，需要先打败大蛇丸的手下才能到这些位置。鸣人有一定数量的查克拉，每一个单位的查克拉可以打败一个大蛇丸的手下。假设鸣人可以往上下左右四个方向移动，每移动一个距离需要花费1个单位时间，打败大蛇丸的手下不需要时间。如果鸣人查克拉消耗完了，则只可以走到没有大蛇丸手下的位置，不能再移动到有大蛇丸手下的位置。佐助在此期间不移动，大蛇丸的手下也不移动。请问，鸣人要追上佐助最少需要花费多少时间？

输入

输入的第一行包含三个整数：M, N, T。代表M行N列的地图和鸣人初始的查克拉数量T。 $0 < M, N < 200$, $0 \leq T < 10$ 后面是M行N列的地图，其中@代表鸣人，+代表佐助。*代表通路，#代表大蛇丸的手下。

输出

输出包含一个整数R，代表鸣人追上佐助最少需要花费的时间。如果鸣人无法追上佐助，则输出-1。

样例输入

```

样例输入1
4 4 1
#@##
*###
*###
###+
****
```

```

样例输入2
4 4 2
#@##
*###
*###
###+
****
```

样例输出

```

样例输出1
6
```

```

样例输出2
4
```

思路：稍复杂的bfs问题。visited需要维护经过时的最大查克拉数，只有大于T值时候才能通过，然后就是常见bfs。

夏天明 元培学院

1

```

from collections import deque

M, N, T = map(int, input().split())
graph = [list(input()) for i in range(M)]
direc = [(0,1), (1,0), (-1,0), (0,-1)]
start, end = None, None
for i in range(M):
    for j in range(N):
        if graph[i][j] == '@':
            start = (i, j)

def bfs():
    q = deque([start + (T, 0)])
    visited = [[False]*N for i in range(M)]
    visited[start[0]][start[1]] = True
    while q:
        x, y, t, time = q.popleft()
        time += 1
        for dx, dy in direc:
            if 0 <= x+dx < M and 0 <= y+dy < N:
                if (elem := graph[x+dx][y+dy]) == '*' and t > visited[x+dx][y+dy]:
                    visited[x+dx][y+dy] = t
                    q.append((x+dx, y+dy, t, time))
                elif elem == '#' and t > 0 and t-1 > visited[x+dx][y+dy]:
                    visited[x+dx][y+dy] = t-1
                    q.append((x+dx, y+dy, t-1, time))
                elif elem == '+':
                    return time
    return -1

print(bfs())

```