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
```

stack

单调栈：

单调栈（Monotonic Stack）是一种常用的数据结构，通常用于解决一些数组或字符串相关的问题，特别是在需要寻找某个元素左右第一个比它大或小的位置时非常有用。在 Python 中，可以使用列表来实现单调栈的功能

```
def monotonic_stack(nums):
    stack = []  # 用列表模拟栈
    result = [-1] * len(nums)  # 初始化结果列表，默认值为-1

    for i in range(len(nums)):
        # 当栈非空且当前元素比栈顶元素大时，出栈并更新结果
        while stack and nums[i] > nums[stack[-1]]:
            result[stack.pop()] = i

        # 将当前元素的索引入栈
        stack.append(i)

    return result

# 示例
nums = [3, 1, 5, 7, 2, 6]
print("原始数组:", nums)
print("单调递增栈结果:", monotonic_stack(nums))
#这段代码实现了一个单调递增栈，它能够找到数组中每个元素右边第一个比它大的元素的位置。

#要实现单调递减栈，只需在 while 循环中将 nums[i] > nums[stack[-1]] 改为 nums[i] <
nums[stack[-1]] 即可。
```

输出/输入

同一行输出：

使用end=''，引号中间填写希望中间隔开的东西，比如说一个空格

在下一行print()从而达到换行的目的

```
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)
#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 EOFFError:
        break
```

矩阵：

1. 基本方法：列表套列表，用i，j两个指针
2. 将n*n的矩阵用某一元素进行填充的格式：

e.g：a是列数，b是行数

matrix = [[1]*a for _ in range(b)]

3. 双重循环防止index out of range：下界使用max（0，a），上界使用min（n，b），n为矩阵长。
   注意：range是左开右闭，列表的指针是0-n-1

4. 可以套一层保护圈，如最外圈包一层0，减轻考虑循环的时间

其他

```
float(“inf”)#表示正无穷，可以用于赋值

output = ','.join(my_list)
print(output)#把列表中的所有元素一串输出，条件是列表中的元素都是str类型

mylist=[1,2,3,4,5]
print(mylist,sep=',')#把列表中所有元素之间用sep内部的东西分开，输出形式仍然是一个列表[1, 2, 3, 4, 5]

ord() #返回对应的ASCII表值
chr() #返回ASCII值对应的字符
bin(),oct(),hex()#分别表示二进制，八进制，十六进制的转换

# 二进制转十进制
binary_str = "1010"
decimal_num = int(binary_str, 2) # 第一个参数是字符串类型的某进制数，第二个参数是他的进制，最终转化为整数
print(decimal_num)  # 输出 10


#Dp写不出来用递归，为了防止爆栈（re）：使用缓存
from functools import lru_cache
@lru_cache(maxsize=128)
#注意：使用的时候函数的参数需要是不可修改的，即不可以为列表字典等，可以改用元组（）或者set

#判断完全平方数
import math
def isPerfectSquare(num):
    if num < 0:
        return False
    sqrt_num = math.isqrt(num)
    return sqrt_num * sqrt_num == num
print(isPerfectSquare(97)) # False

#年份calendar
import calendar
print(calendar.isleap(2020)) # True，判断是否是闰年

#字符串的连接
str=str1+str2

#用同一个数填充整个列表
dp=[[0]*(i+1) for i in range(5)]
#输出[[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]

#str相关
```

str.lstrip() / str.rstrip(): 移除字符串左侧/右侧的空白字符。

str.find(sub): 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。

str.replace(old, new): 将字符串中的`old`子字符串替换为`new`。

str.isalpha() / str.isdigit() / str.isalnum(): 检查字符串是否全部由字母/数字/字母和数字组成。

str.title()：每个单词首字母大写。

```
#保留小数
#1.round函数
number = 3.14159
rounded_number = round(number, 1)

print(rounded_number)
#输出3.1，保留一位小数
#2.format格式化
number = 3.14159
formatted_number = "{:.1f}".format(number)

print(formatted_number)
#保留一位小数，输出的类型为str

#字符串的连接
''.join()
#eg：','.join([2,3,4,5])
#output:2,3,4,5

#eval函数
#eval() 是 python 中功能非常强大的一个函数，将字符串当成有效的表达式来求值，并返回计算结果，就是实现 list、dict、tuple、与str 之间的转化
result = eval("1 + 1")
print(result)  # 2

result = eval("'+' * 5")
print(result)  # +++++

# 3. 将字符串转换成列表
a = "[1, 2, 3, 4]"
result = type(eval(a))
print(result)  # <class 'list'>

input_number = input("请输入一个加减乘除运算公式：")
print(eval(input_number))
## 1*2 +3
## 5



for key,value in dict.items()   #遍历字典的键值对。

for index,value in enumerate(list)  #枚举列表，提供元素及其索引。
```

```
dic.setdefault(key,[]).append(value) #常用在字典中加入元素的方式（如果没有值就建空表，有值就直接添加）

dict.get(key,default)   #从字典中获取键对应的值，如果键不存在，则返回默认值`default`。

list(zip(a,b))  #将两个列表元素一一配对，生成元组的列表。
```

类的格式和使用

```
#创建一个类
class person():
    #类变量的创建
    name='aa'
    #类方法的创建
    def who(self):
        print(name)

#类变量的访问：类名.变量名
p=person.name
print(p)


#实例化类（类函数的使用）：类名.函数名
c=person()
c.who()
#输出：aa

#类变量的修改:实例化后只修改自己内部而不影响原始的类；若直接用类名修改则会影响所有的实例化
c=person()
c.name=0
print(c.name)
#0
a=person()
print(a.name)
#aa
person.name=1
#则后续所有都会改变

#构造器：
class person():
    #self 表示的就是类的实例
    def __init__(self,a)
    #实例变量：依靠输入
    self.name=a
    #实例变量：默认值
    self.age=10



print(person(a).name)
#a

#用类实现双端队列：
class deque:
    def __init__(self):
```

```
        self.queue=[]

    def push(self,a):#进队
        self.queue.append(a)

    def post_out(self):
        self.queue.pop()

    def pre_out(self):
        self.queue.pop(0)

    def empty(self):
        if self.queue==[]:
            return False
        else:
            return True


t=int(input())
for i in range(t):
    p=deque()
    n=int(input())
    for j in range(n):
            t,x=map(int,input().split())
            if t==1:
                p.push(x)
            elif t==2:
                if x==0:
                    p.pre_out()
                elif x==1:
                    p.post_out()
    if p.empty():
        ans=[]
        for z in p.queue:
            ans.append(str(z))
        print(' '.join(ans))
    else:
            print('NULL')
```

算法相关

1.质数筛

```
#欧拉筛 输出素数列表
def Euler_sieve(n):
    primes = [True for _ in range(n+1)]
    p = 2
    while p*p <= n:
        if primes[p]:
            for i in range(p*p, n+1, p):
                primes[i] = False
        p += 1
    primes[0]=primes[1]=False
```

```
    return primes
print(Euler_sieve(20))
# [False, False, True, True, False, True, False, True, False, False, False, True,
False, True, False, False, False, True, False, True, False]

#埃氏筛 输出素数列表
N=20
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(1,N):
    if is_prime[i]:
        primes.append(i)
        for k in range(2*i,N,i): #用素数去筛掉它的倍数
            is_prime[k] = False
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]

#欧拉筛 直接判断某个数是否是质数
prime = []
n = c
is_prime=[True] * (n+1)  # 初始化为全是素数
is_prime[0]=is_prime[1]=False#把0和1标记为非素数

        def euler_sieve():

            for i in range(2, n // 2 + 1):
                if is_prime[i]:
                    prime.append(i)
                for j in prime:  # 将所有质数的倍数标记为非素数

                    if i * j > n:
                        break
                    is_prime[j * i] = False
                    if i % j == 0:
                        break


        # 测试
        euler_sieve()

        if is_prime[n]:
            return True
        else:
            return False

#6k+1质数判断法即Miller-Rabin素性测试算法
import math

def is_prime(num):
    if num <= 1:
        return False
    elif num <= 3:
        return True
    elif num % 2 == 0 or num % 3 == 0:
        return False
```

```
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True
#其原理基于以下观察：除了2和3之外，所有的质数都可以表示为 6k ± 1 的形式，其中 k 是一个整数
```

2.二分查找:

```
import bisect
sorted_list = [1,3,5,7,9] #[(0)1, (1)3, (2)5, (3)7, (4)9]
position = bisect.bisect_left(sorted_list, 6)#查找某一元素插入后的索引
print(position)  # 输出：3，因为6应该插入到位置3，才能保持列表的升序顺序

bisect.insort_left(sorted_list, 6)#将某个元素插入原列表并不改变升序/降序
print(sorted_list)  # 输出：[1, 3, 5, 6, 7, 9]，6被插入到适当的位置以保持升序顺序

sorted_list=(1,3,5,7,7,7,9)
print(bisect.bisect_left(sorted_list,7))
print(bisect.bisect_right(sorted_list,7))
# 输出：3 6
#右侧插入，如果有相同元素，就输出最大的索引，左侧输入则相反
```

3.heap 优先堆:

特别地，使用了heap结构之后该列表就只能用heap包里的函数，不能再用remove，append等

python中堆默认是最小堆，如果需要最大堆可以把所有的元素都加负号即可

```
import heapq # 优先队列可以实现以log复杂度拿出最小（大）元素
lst=[1,2,3]
heapq.heapify(lst) # 将lst优先队列化，获得最小堆，从小到大排序
heapq.heappop(lst) # 从队列中弹出树顶元素，实际上是从前往后弹（默认最小，相反数reverse一下lst）
heapq.heappush(lst,element) # 把元素压入堆中
top_value = heap[0]#查看最小值但不弹出
```

3.

```
import heapq

def dynamic_median(nums):
    # 维护小根和大根堆（对顶），保持中位数在大根堆的顶部
    min_heap = []  # 存储较大的一半元素，使用最小堆
    max_heap = []  # 存储较小的一半元素，使用最大堆

    median = []
    for i, num in enumerate(nums):
        # 根据当前元素的大小将其插入到对应的堆中
        if not max_heap or num <= -max_heap[0]:
            heapq.heappush(max_heap, -num)
```

```
            else:
                heapq.heappush(min_heap, num)

            # 调整两个堆的大小差，使其不超过 1
            if len(max_heap) - len(min_heap) > 1:
                heapq.heappush(min_heap, -heapq.heappop(max_heap))
            elif len(min_heap) > len(max_heap):
                heapq.heappush(max_heap, -heapq.heappop(min_heap))

            if i % 2 == 0:
                median.append(-max_heap[0])

        return median

T = int(input())
for _ in range(T):
    #M = int(input())
    nums = list(map(int, input().split()))
    median = dynamic_median(nums)
    print(len(median))
    print(*median)
```

4.default_dict：

defaultdict是Python中collections模块中的一种数据结构，它是一种特殊的字典，可以为字典的值提供默认值。当你使用一个不存在的键访问字典时，defaultdict会自动为该键创建一个默认值，而不会引发KeyError异常。

defaultdict的优势在于它能够简化代码逻辑，特别是在处理字典中的值为可迭代对象的情况下。通过设置一个默认的数据类型，它使得我们不需要在访问字典中不存在的键时手动创建默认值，从而减少了代码的复杂性。

使用defaultdict时，首先需要导入collections模块，然后通过指定一个默认工厂函数来创建一个defaultdict对象。一般来说，这个工厂函数可以是int、list、set等Python的内置数据类型或者自定义函数。

```
from collections import defaultdict
# 创建一个defaultdict，值的默认工厂函数为int，表示默认值为0
char_count = defaultdict(int)
# 统计字符出现次数
input_string = "hello"
for char in input_string:
    char_count[char] += 1
print(char_count)  # 输出 defaultdict(<class 'int'>, {'h': 1, 'e': 1, 'l': 2, 'o': 1})）
print(char_count['h'])
#输出：1

dict.get(key,default=None)
# 其中，my_dict是要操作的字典，key是要查找的键，default是可选参数，表示当指定的键不存在时要返回的默认值
```

5.栈

单调栈：可以找到第一个比当前节点高的节点的位置

```
#
n=int(input())
a=list(map(int,input().split()))


def monotonic_stack(nums):
    stack = []
    result = [str(0)] * len(nums)

    for i in range(len(nums)):
        # 当栈非空且当前元素比栈顶元素大时，出栈并更新结果
        while stack and nums[i] > nums[stack[-1]]:
            result[stack.pop()] = str(i+1)

        # 将当前元素的索引入栈
        stack.append(i)

    return result
print(' '.join(monotonic_stack(a)))
```

波兰表达式（前缀表达式）

```
# 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。
#原理：从后往前遍历，依次把数字放入栈中，遇到一个运算符就从栈顶弹出两个数字进行运算然后再将结果压入栈中。
a=input().split()
s=[]#存储运算符
n=[]#存储数字
for i in range(len(a)):
    x=a.pop()
    if x=='+' or x=='-' or x=='*' or x=='/':
        s.append(x)
        y = n.pop()
        z = n.pop()
        k = s.pop()
        if k == '+':
            n.append(y + z)
        elif k == '-':
            n.append(y-z)
        elif k == '*':
            n.append(y * z)
        elif k == '/':
            n.append(y/z)

    else:
```

```
        n.append(float(x))

ans=n[0]
print("{:.6f}".format(ans))
```

中序表达式转后序表达式：shunting yard调度场算法

原理：从前往后遍历中缀表达式，如果遇到数字就直接放入output，遇到左括号就入栈，遇到右括号就弹出到output直到遇到左括号，遇到运算符时：如果栈顶是运算符且优先级大于等于该运算符，则一直弹出直到优先级小于该运算符，然后再将当前运算符入栈。

遍历完后把栈中剩余元素加到output里

```
#
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]]:#判断优先级
                    postfix.append(stack.pop())#优先级大于等于当前运算符的全部出栈
                stack.append(char)#最后再入栈
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    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))
```

快速堆猪：辅助栈

根本思路是辅助栈中存储当前栈中的最小值。每次执行 push 操作时，如果新元素比辅助栈中的栈顶元素更小，则将新元素也压入辅助栈；否则，将辅助栈栈顶元素再次压入辅助栈。这样，辅助栈的栈顶元素始终是当前栈中的最小值。

如果pop出的是最小值，说明最小值刚被压入栈中，此时辅助栈的栈顶就是最小值，第二个元素是第二小的值，因此即使pop出来最小值之后通过代码逻辑也可以保证辅助栈栈顶还是此时栈中的最小元素，非常巧妙，原因是在每次push的时候都相应push一个元素进辅助栈，使得辅助栈与标准栈始终保持一一对应的关系。

描述

小明有很多猪，他喜欢玩叠猪游戏，就是将猪一头头叠起来。猪叠上去后，还可以把顶上的猪拿下来。小知道每头猪的重量，而且他还随时想知道叠在那里的猪最轻的是多少斤。

输入

有三种输入
1)push n
n是整数(0<=0 <=20000)，表示叠上一头重量是n斤的新猪
2)pop
表示将猪堆顶的猪赶走。如果猪堆没猪，就啥也不干
3)min
表示问现在猪堆里最轻的猪多重。如果猪堆没猪，就啥也不干

输入总数不超过100000条

输出

对每个min输入，输出答案。如果猪堆没猪，就啥也不干

```
#
pig1=[]
pig2=[]
while True:
    try:
        a=input().split()
        if a[0]=='pop':
            if pig1:
                pig1.pop()
                if pig2:
                    pig2.pop()

        elif a[0]=='min':
            if pig2:
                print(pig2[-1])

        else:
            b=int(a[1])
            pig1.append(b)
            if not pig2:
```

```
                pig2.append(b)
            else:
                minn=min(pig2[-1],b)
                pig2.append(minn)
    except EOFError:
        break
```

6.并查集：

```
class disj_set:
    def __init__(self,n):#n表示并查集的初始大小
        self.rank = [1 for i in range(n)]
        #用于记录每个元素所在树的深度
        self.parent = [i for i in range(n)]
        #用于记录每个元素的父节点，初始时父节点都是自己

    def find(self,x):#用于查找x所在的集合，即数根节点（根节点条件：parent=自己本身）
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])#路径压缩：把父节点不停往根节点找
        return self.parent[x]#返回最终的根节点


    def union(self,x,y):#用于合并x，y所在的两个集合
        x_root = self.find(x)#分别找到两者的根节点
        y_root = self.find(y)

        if x_root == y_root:
            return
#哪颗树更深就把哪颗树设为主树，把另一棵树连到根节点
        if self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        elif self.rank[y_root] < self.rank[x_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1


##计算根节点个数
count = 0
for x in range(1,n+1):
    if D.parent[x-1] == x - 1:#如果parent就是本身，说明是根节点
        count += 1
```

经典例题：冰阔落

易错点：在更新根节点的时候没有同时更新所有其他在同一个cups里的可乐，导致出现逻辑错误。

老王喜欢喝冰阔落。

初始时刻，桌面上有n杯阔落，编号为1到n。老王总想把其中一杯阔落倒到另一杯中，这样他一次性就能喝很多很多阔落，假设杯子的容量是足够大的。

有m 次操作，每次操作包含两个整数x与y。

若原始编号为x 的阔落与原始编号为y的阔落已经在同一杯，请输出"Yes"；否则，我们将原始编号为y 所在杯子的所有阔落，倒往原始编号为x 所在的杯子，并输出"No"。

最后，老王想知道哪些杯子有冰阔落。

输入

有多组测试数据，少于 5 组。
每组测试数据，第一行两个整数 n, m (n, m<=50000)。接下来 m 行，每行两个整数 x, y (1<=x, y<=n)。

输出

每组测试数据，前 m 行输出 "Yes" 或者 "No"。
第 m+1 行输出一个整数，表示有阔落的杯子数量。
第 m+2 行有若干个整数，从小到大输出这些杯子的编号。

```
#
class Node:
    def __init__(self, v):
        self.ancestor = self
        self.value = v
        self.children = []

def find_root(node):
    if node.ancestor != node:
        node.ancestor = find_root(node.ancestor)
    return node.ancestor

while True:
    try:
        n, m = map(int, input().split())
        cups = [Node(i) for i in range(1, n + 1)]
        roots = set(range(1, n + 1))  # 初始时所有杯子都是根节点

        for i in range(m):
            x, y = map(int, input().split())
            root_x = find_root(cups[x - 1])
            root_y = find_root(cups[y - 1])
            if root_x == root_y:
                print('Yes')
            else:
                print('No')
                root_y.ancestor = root_x  # 合并
                roots.remove(root_y.value)

        print(len(roots))  # 输出数量
        print(" ".join(map(str, sorted(roots))))  # 编号
```

```
    except EOFError:
        break
```

图：

判断无向图是否联通/有回路：

```
class Node:
    def __init__(self, v):
        self.value = v
        self.joint = set()

def connected(x, visited, num):#判断是否联通
    visited.add(x)
    al = 1
    q = [x]
    while al != num and q:
        x = q.pop(0)
        for y in x.joint:
            if y not in visited:
                visited.add(y)
                al += 1
                q.append(y)
    return al == num

def loop(x, visited, parent):#判断是否有环
    visited.add(x)
    if x.joint:
        for a in x.joint:
            if a in visited and a != parent:
                return True
            elif a != parent and loop(a, visited, x):
                return True
    return False

n, m = map(int, input().split())

vertex = [Node(i) for i in range(n)]
for i in range(m):
    a, b = map(int, input().split())
    vertex[a].joint.add(vertex[b])
    vertex[b].joint.add(vertex[a])

if connected(vertex[0], set(), n):
    print('connected:yes')
else:
    print('connected:no')
x=0
for i in range(n):
    if loop(vertex[i],set(),None):
        print('loop:yes')
        x=1
```

```
        break
if x==0:
    print('loop:no')
```

判断有向图是否有环：拓扑排序

原理：每次选入度为0的点，将该点放入output，并删掉该点的出边，同时更新其他点的入度。

如果最后output的长度为n则说明无环

```
class Node:
    def __init__(self, v):
        self.val = v
        self.to = []

from collections import deque

t = int(input())
for _ in range(t):
    n, m = map(int, input().split())
    node = [Node(i) for i in range(1, n + 1)]
    into = [0 for _ in range(n)]  # 记录入度
    for _ in range(m):  # 构建图
        x, y = map(int, input().split())
        node[x - 1].to.append(node[y - 1])
        into[y - 1] += 1#更新入度

    queue = deque([node[i] for i in range(n) if into[i] == 0])
    output = []

    while queue:
        a = queue.popleft()
        output.append(a)#把a增加进输出列表中
        for x in a.to:#对于a的指出去的边
            num = x.val
            into[num - 1] -= 1#删除a指出去的边，同时更新有向边终点的入度
            if into[num - 1] == 0:#如果更新后入度为0就入队
                queue.append(x)

    if len(output) == n:#如果output的长度是n说明没有环
        print('No')
    else:#否则说明有环
        print('Yes')
```

判断有向图是否强连通： Kosaraju's 算法函数

核心：两次dfs

作用：用于查找有向图中强连通分量的算法。强连通分量是指在有向图中，任意两个节点都可以相互到达的一组节点。

```
def dfs1(graph, node, visited, stack):
    # 第一个深度优先搜索函数，用于遍历图并将节点按完成时间压入栈中
    visited[node] = True  # 标记当前节点为已访问
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
```

```
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs1(graph, neighbor, visited, stack)  # 递归调用深度优先搜索函数
    stack.append(node)  # 将当前节点压入栈中，记录完成时间

def dfs2(graph, node, visited, component):
    # 第二个深度优先搜索函数，用于在转置后的图上查找强连通分量
    visited[node] = True  # 标记当前节点为已访问
    component.append(node)  # 将当前节点添加到当前强连通分量中
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs2(graph, neighbor, visited, component)  # 递归调用深度优先搜索函数

def kosaraju(graph):
    # Kosaraju's 算法函数
    # Step 1: 执行第一次深度优先搜索以获取完成时间
    stack = []  # 用于存储节点的栈
    visited = [False] * len(graph)  # 记录节点是否被访问过的列表
    for node in range(len(graph)):  # 遍历所有节点
        if not visited[node]:  # 如果节点未被访问过
            dfs1(graph, node, visited, stack)  # 调用第一个深度优先搜索函数

    # Step 2: 转置图
    transposed_graph = [[] for _ in range(len(graph))]  # 创建一个转置后的图
    for node in range(len(graph)):  # 遍历原图中的所有节点
        for neighbor in graph[node]:  # 遍历每个节点的邻居节点
            transposed_graph[neighbor].append(node)  # 将原图中的边反向添加到转置图中

    # Step 3: 在转置后的图上执行第二次深度优先搜索以找到强连通分量
    visited = [False] * len(graph)  # 重新初始化节点是否被访问过的列表
    sccs = []  # 存储强连通分量的列表
    while stack:  # 当栈不为空时循环
        node = stack.pop()  # 从栈中弹出一个节点
        if not visited[node]:  # 如果节点未被访问过
            scc = []  # 创建一个新的强连通分量列表
            dfs2(transposed_graph, node, visited, scc)  # 在转置图上执行深度优先搜索
            sccs.append(scc)  # 将找到的强连通分量添加到结果列表中
    return sccs  # 返回所有强连通分量的列表

# 例子
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]  # 给定的有向图，第i个列表表示第i个元素与列表中的这些元素相连，比如节点0的相邻节点为节点1，节点1的相邻节点为节点2和节点4
sccs = kosaraju(graph)  # 使用Kosaraju's 算法查找强连通分量
print("Strongly Connected Components:")
for scc in sccs:  # 遍历并打印所有强连通分量
    print(scc)

"""
输出结果：
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]
其中每个列表都表示一个有向图中的强连通分量。强连通分量是指其中任意两个顶点之间都存在双向路径的顶点集合。
"""
```

DFS深度搜索

```
#例题：迷宫的所有路径
#骑士周游也是经典dfs，按照马走日的规则更改一下每次可移动的dx，dy即可
n,m=map(int,input().split())
ma=[]

for _ in range(n):
    ma.append(list(map(int,input().split())))
ans=0
def dfs(ma,i,j):
    global ans#要在函数内部设置全局变量，方便更新ans的值
    if i==n-1 and j==m-1:
        ans+=1
        #走到终点一次就增加一个ans

    elif ma[i][j]==0:
        dx=[0,1,-1,0]
        dy=[1,0,0,-1]
        for k in range(4):
            if 0<=i+dx[k]<n and 0<=j+dy[k]<m:
                ma[i][j]=1
                #修改为墙，防止往回走
                dfs(ma,i+dx[k],j+dy[k])
                #往四面八方走
        ma[i][j]=0
        #还原为初始值
dfs(ma,0,0)
print(ans)


#八皇后
board=[[0]*8 for i in range(8)]
solutions=[]
visited_y=set()#存储有皇后的列
visited_sum=set()#存储x+y，用于判断是否在左上→右下的斜线
visited_dif=set()#存储x-y，用于判断是否在右上→左下的斜线
def dfs(x,y,board,visited_y,visited_sum):
    global solutions
    if y not in visited_y and x+y not in visited_sum and x-y not in visited_dif:
        board[x][y]=1
        visited_y.add(y)
        visited_sum.add(x+y)
        visited_dif.add(x-y)

        if x == 7:
            output = []
            for i in range(8):
                for j in range(8):
                    if board[i][j] == 1:#把每一行的皇后对应的列加入output里
                        output.append(str(j + 1))
```

```
            ans = int(''.join(output))
            solutions.append(ans)

        if x<7:
            for i in range(8):
                dfs(x+1,i,board,visited_y,visited_sum)

        board[x][y] = 0
        visited_y.discard(y)
        visited_sum.discard(x + y)
        visited_dif.discard(x - y)#在每一个皇后位置以后的所有皇后都访问完后，还原初始状态，以便访问下一个皇后的位置

for i in range(8):
    dfs(0,i,board,visited_y,visited_sum)

solutions.sort()
n=int(input())

for i in range(n):
    a=int(input())
    print(solutions[a-1])
```

BFS广度优先搜索

核心：使用deque队列

```
# 最大连通域面积
#使用了bfs的思想遍历整个棋盘，在数过W后将其标记为‘.’，确保后续不会再重复数这个元素。同时，在周围没有W的情况下，因为队列无法弹出会自动终止循环，不需要额外添加一个参数来计算，这样子会使代码更加简洁。
t = int(input())
result = []
for _ in range(t):
    re = []
    ans = 0
    N, M = map(int, input().split())
    ma = []
    for _ in range(N):
        list1 = list(input())
        ma.append(list1)

    for i in range(N):
        for j in range(M):
            if ma[i][j] == 'W':
                ans += 1
                ma[i][j] = '.'
                queue = [(i, j)]
                while queue:
                    x, y = queue.pop()
                    for dx in [-1, 0, 1]:
                        for dy in [-1, 0, 1]:
                            nx, ny = x + dx, y + dy
```

```
                            if 0 <= nx < N and 0 <= ny < M and ma[nx][ny] == 'W':
                                ma[nx][ny] = '.'
                                queue.append((nx, ny))
                                ans += 1

            if ans > 0:
                re.append(ans)
                ans = 0

    re.append(ans)
    result.append(max(re))

for z in result:
    print(z)

#数字操作：给定一个整数n，计算从1开始每次可以*2或者+1，最少经历多少次才可以得到n
from collections import deque

def minOperations(target):
    queue = deque([(1, 0)])  # (当前数字，操作次数)

    while queue:
        num, operations = queue.popleft()

        if num == target:  # 如果当前数字等于目标数字，返回操作次数
            return operations

        # 尝试加1和乘2两种操作
        next_add = num + 1
        next_mul = num * 2

        # 将加1和乘2得到的数字加入队列
        queue.append((next_add, operations + 1))
        queue.append((next_mul, operations + 1))

    return -1  # 如果无法得到目标数字，返回-1

# 读取输入并调用函数输出结果
target = int(input())
print(minOperations(target))


#最短迷宫路径
from collections import deque
n,m=map(int,input().split())
ma=[]
result=float('inf')
for i in range(n):
    ma.append(list(map(int,input().split())))
def bfs(ma):
    global result
    queue = deque([(0, 0, 0)])
    visited = set()  # 用集合记录已经访问过的位置
    while queue:
        i, j, k = queue.popleft()  # 从左侧弹出队列中的第一个元素
        if i == n - 1 and j == m - 1:
```

```
            result = min(result, k)  # 更新最短路径长度
        else:
            dx = [1, 0, 0, -1]
            dy = [0, 1, -1, 0]
            if ma[i][j] == 0:
                for _ in range(4):
                    x, y = i + dx[_], j + dy[_]
                    if 0 <= x < n and 0 <= y < m and (x, y) not in visited:
                        visited.add((x, y))  # 标记位置(x, y)为已访问，防止重复访问导致
RE，TLE
                        queue.append((x, y, k + 1))  # 将符合条件的位置加入队列
bfs(ma)
if result!=float('inf'):
    print(result)
else:
    print(-1)
```

bfs常用数据结构：deque，类似列表，但是更高效

deque相关操作和作用：

```
from collections import deque
d = deque()  # 创建一个空的deque对象

d.append(x)  # 在deque的右端添加元素x

d.appendleft(x)  # 在deque的左端添加元素x

d.pop()  # 从deque的右端删除并返回最右边的元素

d.popleft()  # 从deque的左端删除并返回最左边的元素

len(d)  # 返回deque中元素的数量
d.clear()  # 清空deque中的所有元素

d = deque(maxlen=5)  # 创建一个最大长度为5的deque，当超出长度时，自动弹出最左边元素

list(d)  # 将deque转换为列表
```

BFS中的Dijkstra算法：带权重的最小路径

tips：超时的时候可以尝试把bfs写成函数，调用时会更快防止TLE

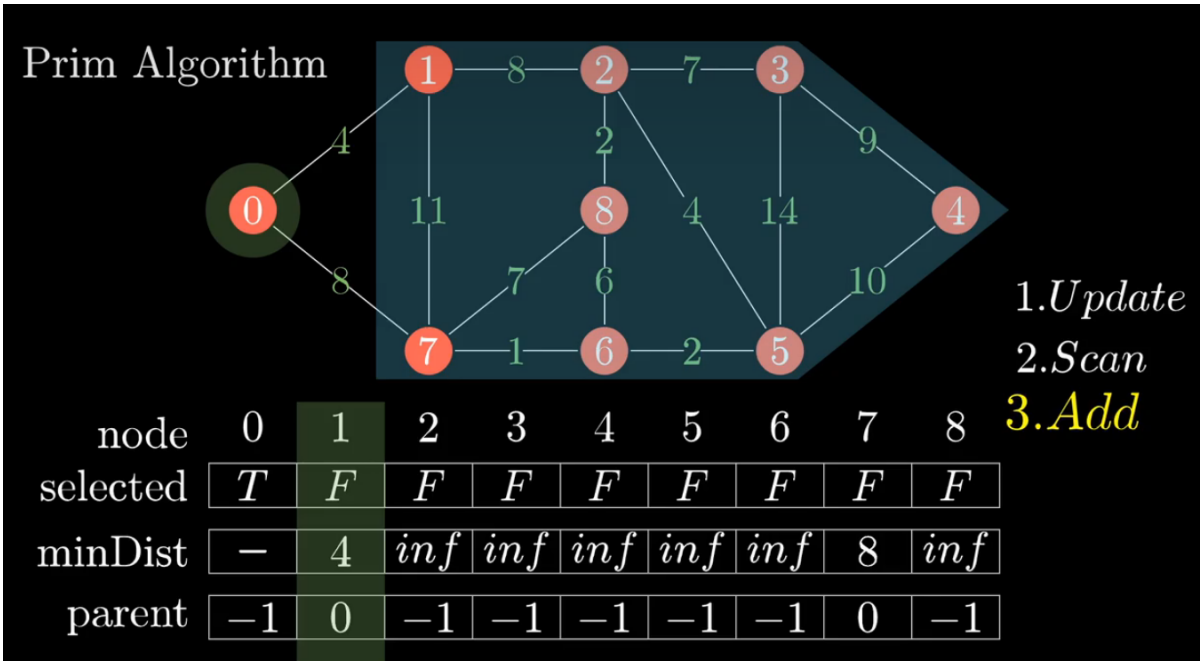
```
#经典bfs:
def bfs(x, y):
    # 定义方向的偏移量
    directions = [(0, -1), (0, 1), (1, 0), (-1, 0)]
    # 初始化队列，将起点加入队列
```

```
    queue = [(x, y)]
    # 初始化距离字典，将起点的距离设为0，其他节点设为无穷大
    distances = {(x, y): 0}

    while queue:
        # 弹出队列中的节点
        current_x, current_y = queue.pop(0)

        # 遍历四个方向上的相邻节点
        for dx, dy in directions:
            new_x, new_y = current_x + dx, current_y + dy

            # 判断新节点是否在合法范围内
            if 0 <= new_x < m and 0 <= new_y < n:
                # 判断新节点是否为墙
                if d[new_x][new_y] != '#':
                    # 计算新节点的距离
                    new_distance = distances[(current_x, current_y)] +
abs(int(d[new_x][new_y]) - int(d[current_x][current_y]))

                    # 如果新节点的距离小于已记录的距离，则更新距离字典和队列
                    if (new_x, new_y) not in distances or new_distance <
distances[(new_x, new_y)]:
                        distances[(new_x, new_y)] = new_distance
                        queue.append((new_x, new_y))

    return distances

# 读取输入
m, n, p = map(int, input().split())
d = []
for _ in range(m):
    row = input().split()
    d.append(row)

for _ in range(p):
    # 读取起点和终点坐标
    x1, y1, x2, y2 = map(int, input().split())

    # 判断起点和终点是否为墙
    if d[x1][y1] == '#' or d[x2][y2] == '#':
        print('NO')
        continue

    # 使用BFS计算最短距离
    distances = bfs(x1, y1)

    # 输出结果，如果终点在距离字典中，则输出对应的最短距离，否则输出'NO'
    if (x2, y2) in distances:
        print(distances[(x2, y2)])
    else:
        print('NO')



 #用堆实现：
```

```
#graph 应该是一个字典，其中键表示图中的节点，而值表示从每个节点到其相邻节点的距离或权重。
def djs(start,end,graph):#初始位置、结束位置、邻接表
    heap=[(0,start,[start])]
    heapq.heapify(heap)
    visited=set()#初始化已访问列表
    while heap:
        (length,start,path)=heapq.heappop(heap)#每次pop出来的一定是该点到其他相邻点距离
最小的路径
        if start in visited:
            continue
        visited.add(start)
        if start==end:#意味着已经到达重点，返回路径
            return path
        for i in graph[start]:#在start的邻居节点中，如果尚未被访问
            if i not in visited:#则更新到i的新距离，方法是将从起始节点到start的距离
（length）和从start到i的距离（graph[start][i]）相加
                heapq.heappush(heap,(length+graph[start][i],i,path+[i]))
```

prim算法：本质是贪心

作用：求解最小生成树，即把所有节点都相连且权值最小的树




```
#返回最小生成树的所有权重之和
class Node:
    def __init__(self,v):
        self.value=v
        self.joint=set()#存储相邻节点
        self.way=dict()#存储到相邻节点的边的权重

def prim(x,num,al,ans,farm):#num是总节点数量，al是已经遍历的节点数，x是初始节点，ans是权重
之和，farm是包含所有节点的列表
    visited=set()
    visited.add(x)
    min_way=[100000]*num#设置inf的初始值
    while al!=num:#当还没有遍历完每个节点的时候
        for a in range(numm):
```

```
            if min_way[a]!=200000:
                for b in visited:
                    if farm[a] in b.joint:

                        min_way[a]=min(min_way[a],b.way[farm[a]])#更新最小距离
        shortest=min(min_way)
        y=min_way.index(shortest)
        ans+=shortest
        visited.add(farm[y])
        min_way[y]=200000#表示该节点已经访问过
        al+=1
    if al==num:
        return ans
while True:
    try:
        numm=int(input())
        farm=[Node(i) for i in range(numm)]
        maps=[]
        ans=[]
        for i in range(numm):
            maps.append(list(map(int,input().split())))
        for x in range(numm):#构图
            for y in range(numm):
                farm[x].joint.add(farm[y])
                farm[x].way[farm[y]]=maps[x][y]
                farm[y].joint.add(farm[x])
                farm[y].way[farm[x]] = maps[x][y]

        ans=prim(farm[0],numm,0,0,farm)
        print(ans)

    except EOFError:
        break



#返回最小生成树的所有边的集合
from heapq import heappop, heappush, heapify
#graph是一个字典套字典，如graph = {'A': {'B': 3, 'C': 2}},items()会返回所有的字典里的键值对
def prim(graph, start_node):#graph 表示无向图的邻接表表示，start_node 表示起始节点。
    mst = set()#初始化空集合，用来存放mst的边
    visited = set([start_node])#记录已经访问过的节点
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
    ]
    #初始化一个列表 edges，用于存储从起始节点出发的边，每个元素是一个三元组 (cost, from_node, to_node)，表示边的权重、起始节点和目标节点。
    heapify(edges)
    #将 edges 列表转换为堆，以便在之后的操作中能够高效地找到最小权重的边

    while edges:
        cost, frm, to = heappop(edges)
        #每次弹出权重最小的边，cost是边的权重、frm是起始节点、to是目标节点
        if to not in visited:
```

```
            #若to没被访问过则进行操作
            visited.add(to)
            mst.add((frm, to, cost))
            for to_next, cost2 in graph[to].items():#对于to的邻居节点，若未访问过则把
to到邻居的边入堆
                if to_next not in visited:
                    heappush(edges, (cost2, to, to_next))

    return mst
#最终返回的 mst 是一个集合，其中每个元素表示最小生成树中的一条边。每条边表示为一个三元组
(from_node, to_node, cost)，其中 from_node 是边的起始节点，to_node 是边的目标节点，cost
是边的权重。
```

DP动态规划

```
#斐波那契数列 一维数组 up-down类型 递归写法
dp=[-1 for i in range(200)]#初始化一个数组
dp[0]=0#设置好初始值
dp[2]=1
dp[1]=1
n=int(input())
def f(x):

    if dp[x-1]!=-1 and dp[x-2]!=-1:
        dp[x]=dp[x-1]+dp[x-2]#如果数组里面已经有了，就直接加
    else:
        f(x-1)#如果没有，就需要递归
        dp[x] = dp[x - 1] + dp[x - 2]
for i in range(n):
    a=int(input())
    if a>2:#注意边界情况，避免指针出现负值
        f(a)
    print(dp[a])

#数字三角形 bottom-up类型
n=int(input())
tri=[]
dp=[]
for i in range(n):
    tri.append(list(map(int,input().split())))
    dp.append([0]*(i+1))
for i in range(n):#先把最后一行更新为tri里面的
    dp[n-1][i]=tri[n-1][i]
for z in range(n-2,-1,-1):
    for k in range(z+1):#运用贪心策略，每次加底下两个中比较大的数，存储在上一行的相应位置
        dp[z][k]=max(dp[z+1][k],dp[z+1][k+1])+tri[z][k]
print(dp[0][0])#最后输出顶端的就是所求的最大数

#数字三角形 top-down类型
from functools import lru_cache
```

```
@lru_cache(maxsize = 128)
def f(i, j):
    if i == N-1:
        return tri[i][j]
    return max(f(i+1, j), f(i+1, j+1)) + tri[i][j]#和斐波那契数列思路差不多

N = int(input())
tri = []
for _ in range(N):
    tri.append([int(i) for i in input().split()])
print(f(0, 0))


#最大连续子序列和 在dp库里找最大值即可
n = int(input())
a = list(map(int, input().split()))

dp = [0]*n
dp[0] = a[0]

for i in range(1, n):
    dp[i] = max(dp[i-1]+a[i], a[i])#状态转移方程：只有两种可能，取较大的

print(max(dp))

#最长上升子序列的长度
n = int(input())
b = list(map(int, input().split()))
dp = [1]*n

for i in range(1, n):
    for j in range(i):
        if b[j] < b[i]:
            dp[i] = max(dp[i], dp[j]+1)#dp对应索引存储的是以第i个数为结尾的上升子序列长
度，对于i之前的数，如果小于这个，就可以更新dp[i]的值

print(max(dp))


#小偷背包

n,b=map(int, input().split())
price=[0]+[int(i) for i in input().split()]
weight=[0]+[int(i) for i in input().split()]
bag=[[0]*(b+1) for _ in range(n+1)]
#bag[i][j] 表示前 i 件物品放入容量为 j 的背包可以获得的最大价值。
#price[i] 表示第 i 件物品的价值。
#weight[i] 表示第 i 件物品的重量。
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
#状态转移方程的意义是在考虑第 i 件物品时，背包的容量为 j 时的最大价值取决于两种情况的较大值
```

```
#1，第 i 件物品放入背包，其价值为 price[i]，并且需要腾出 weight[i] 的空间，此时背包容量变为 j-weight[i]，因此最大价值为 price[i] + bag[i-1][j-weight[i]]。
#2，不放入第 i 件物品，背包容量为 j 时的最大价值为 bag[i-1][j]，即前 i-1 件物品放入容量为 j 的背包时的最大价值。
#通过状态转移方程，逐步计算出放入不同数量和重量的物品时，背包在不同容量下可以获得的最大价值，最终得到的结果就是整个问题的解。
print(bag[-1][-1])


#采药 类似小偷背包
# 首先将草药按顺序存入列表，然后构造一个二维dp表，i表示第i个草药，j表示此时剩余的时间数，如果下一个草药的时间比j小，那么就分两种情况，放入或者不放入该草药。如果下一个草药的时间比t大，那么就只能不放入这个草药。对i和j进行双循环遍历，最后到第M组的第T次即为所求
T, M = map(int, input().split())  # T是总时间，M是草药数目
herbal = []
for _ in range(M):
    t, v = map(int, input().split())
    herbal.append([t, v])

dp = [[0] * (T + 1) for _ in range(M + 1)]

for a in range(1, M + 1):
    for b in range(1, T + 1):
        if herbal[a - 1][0] <= b:  # 如果草药采摘时间小于等于剩余时间
            dp[a][b] = max(dp[a-1][b], dp[a-1][b-herbal[a-1][0]] + herbal[a-1]
[1])
        else:
            dp[a][b] = dp[a-1][b]

print(dp[M][T])
```

约瑟夫问题：

```
while True:
    n, p, m = map(int, input().split())
    if {n,p,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    for _ in range(p-1):
        tmp = monkey.pop(0)
        monkey.append(tmp)
    # print(monkey)

    index = 0
    ans = []
    while len(monkey) != 1:
        temp = monkey.pop(0)
        index += 1
        if index == m:
            index = 0
            ans.append(temp)
```

```
            continue
        monkey.append(temp)

    ans.extend(monkey)

    print(','.join(map(str, ans)))
```

math库

```
#开头要写
import math
math.sqrt() #开方
math.ceil()#取浮点数上整数
math.floor()#取浮点数的下整数
math.gcd(a,b)#取两个数的最大公约数
math.pow(2,3) # 8.0 幂运算
math.inf#表示正无穷
math.log(100,10) # 2.0
#math.log(x,base) 以base为底，x的对数
math.comb(5,3) # 组合数，C53
math.factorial(5) # 5! 阶乘
```

树相关知识点：

```
#用类建树：n为树的节点个数
class tree:
    def __init__(self):
        self.left=None#左节点
        self.right=None#右节点
n=int(input())
Tree=[tree() for i in range(n)]

#树的高度（可能需要+1）
def tree_height(node):
    if node is None:
        return -1
    left_height=tree_height(node.left)
    right_height=tree_height(node.right)
    return max(left_height,right_height)+1

#如何寻找根节点
parent=[False]*n#初始所有节点都没有parent
#一边接收子节点一边把子节点的parent改成True，最后一个为False的节点就是root
for i in range(n):
    l,r=map(int,input().split())
    if l!=-1:
        Tree[i].left=Tree[l]
        parent[l]=True
    if r!=-1:
        Tree[i].right=Tree[r]
        parent[r]=True

#括号嵌套树的解析方法：
```

```
class tree:
    def __init__(self,v):
        self.children=[]
        self.value=v

tr=input()
stack=[]
node=None#node指向的是当前的节点
for i in tr:
    if i.isalpha():
        node=tree(i)
        if stack:#如果栈不为空，则该节点是栈顶节点的children
            stack[-1].children.append(node)


    elif i=='(':#出现左括号表示当前节点可能有children
        if node:#如果当前节点存在
            stack.append(node)#将其压入栈中
            node=None#把新节点初始化为none，准备读取新节点
    elif i==')':#说明结束了一个子树
        if stack:
            node=stack.pop()#从栈中弹出父节点 便于初始化新树或者更换父节点
root=node#最后弹出的是根节点
```

树：

基础知识/名词：

层级：从根节点开始到达一个节点的路径，所包含的==边的数量==，称为这个节点的层级。
如图 D 的层级为 2，根节点的层级为 0。

高度=深度-1（空树深度为0，高度为-1）

计算深度：

```
def depth(root):
    if root is None: # 先判断是否为空树
        return 0 # 若计算高度，则return -1
    else:
        left_depth = depth(root.left) # 递归
        right_depth = depth(root.right)
        return max(left_depth, right_depth)+1
```

计算叶子节点数目

```
def count_leaves(root):
    if root is None: # 先判断是否为空树
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left)+count_leaves(root.right)
```

按形态分类（主要是二叉树）

（1）完全二叉树——第n-1层全满，最后一层按顺序排列

（2）满二叉树——二叉树的最下面一层元素全部满就是满二叉树

（3）avl树——平衡因子，左右子树高度差不超过1

（4）二叉查找树(二叉排序\搜索树)

==特点：没有相同键值的节点。

==若左子树不空，那么其所有子孙都比根节点小。

==若右子树不空，那么其所有子孙都比根节点大。

==左右子树也分别为二叉排序树。

树的遍历方法：

```
#前序遍历
def preorder(node):
    print(self.value)
    if self.left:
        preorder(self.left)
    if self.right:
        preorder(self.right)

#后续遍历：
def postorder(node):
    if self.left:
        postorder(self.left)
    if self.right:
        postorder(self.right)
    print(self.value)

#按层次遍历：（队列表达式例题）
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
#这段代码通过队列实现了层序遍历的逻辑，确保了按照每一层从左到右的顺序访问树的所有节点。

#中序遍历：
def inorder(root):
    result=[]
    if root is not None:
        result+=postorder(root.left)
```

```
        result+=[root.val]
        result+=postorder(root.right)
    return result
```

树的三种遍历方式的原理：

一：后序遍历

先递归遍历左子树（从下往上）

再递归遍历右子树（从下往上）

最后访问根节点

特点：每一个子树中最后一位都是根节点；后序遍历的reverse就是前序遍历先递归右子树再递归左子树的情况

二：前序遍历

1. 先访问根节点
2. 递归遍历左子树
3. 最后递归地遍历右子树.

   特点：每一个子树中第一位都是根节点

三：中序遍历

1. 先递归地遍历左子树。
2. 访问根结点
3. 最后递归地遍历右子树.

   特点：每一个子树的根节点都在中间，左边部分为左子树，右边部分为右子树

二叉树不同遍历方式的互相转换：

```
#前序+中序=后序
#后序遍历=前序遍历先遍历右子树再进行一次reverse
class Node:
    def __init__(self,val):
        self.left=None
        self.right=None
        self.value=val

def build_tree(midtree,pretree,output):#递归函数
        root=pretree[0]
        output.append(root)
        root_index=midtree.index(root)#将mid和post都切分为左子树和右子树
        mid_left_tree=midtree[0:root_index]
        mid_right_tree=midtree[root_index+1:len(midtree)]
        pre_left_tree=pretree[1:1+len(mid_left_tree)]
        pre_right_tree=pretree[1+len(mid_left_tree):len(pretree)]
        if pre_right_tree:
            build_tree(mid_right_tree,pre_right_tree,output)
```

```
        if pre_left_tree:#递归调用
            build_tree(mid_left_tree,pre_left_tree,output)
        return output

    while True:
    try:
        a,b=input().split()
        pre,mid=[],[]
        for i in range(len(a)):
            pre.append(a[i])
            mid.append(b[i])

        ans=build_tree(mid,pre,[])
        ans.reverse()

        print(''.join(ans))
    except EOFError:
        break

#中序+后序=前序
# 通过后序遍历最后一位是根节点的结论，不断把树分割为左子树和右子树，然后递归调用建树的函数进行操作。
class Tree:
    def __init__(self,value):
        self.left=None
        self.right=None
        self.value=value

def build_tree(midtree,posttree,output):#递归函数
    root=posttree[-1]
    output.append(root)
    root_index=midtree.index(root)#将mid和post都切分为左子树和右子树
    mid_left_tree=midtree[0:root_index]
    mid_right_tree=midtree[root_index+1:len(midtree)]
    post_left_tree=posttree[0:len(mid_left_tree)]
    post_right_tree=posttree[len(mid_left_tree):len(posttree)-1]
    if post_left_tree:#递归调用
        build_tree(mid_left_tree,post_left_tree,output)
    if post_right_tree:
        build_tree(mid_right_tree,post_right_tree,output)
    return output


a=input()
mid=[]
for i in a:
    mid.append(i)
b=input()
post=[]

for i in b:
    post.append(i)


ans=build_tree(mid,post,[])
root=Tree(post[-1])
```

```
print(''.join(ans))
```

二叉搜索树BST的遍历：

二叉搜索树的插入：从上往下找直到子节点为None再插入即可

二叉搜索树的删除：

为叶子节点：直接删除

只有左子树/右子树：删除后把子树接到父节点即可

否则：找到该节点左子树中最大的/右子树中最小的节点，将其代替原有节点，最后删除该节点

特别的，二叉搜索树的中序遍历就是从小到大的顺序

1.给出二叉搜索树的前序遍历，要求输出后序遍历

```
#
class Tree:
    def __init__(self,v):
        self.left=None
        self.right=None
        self.value=v

def build_tree(pre):
    if not pre:
        return None

    root = pre[0]
    left_tree = []
    right_tree = []

    for i in range(1, len(pre)):
        if pre[i].value < root.value:
            left_tree.append(pre[i])#比根节点小的就加入左子树
        else:#比根节点大的加入右子树
            right_tree = pre[i:]
            break

    root.left = build_tree(left_tree)
    root.right = build_tree(right_tree)

    return root

def output(node,ans):
    if node.left:
        output(node.left,ans)
    if node.right:
        output(node.right,ans)
    ans.append(node.value)
    return  ans

n=int(input())
if n==0:
```

```
    print(None)
else:
    pre_tree=list(map(int,input().split()))
    for i in range(len(pre_tree)):
        pre_tree[i]=Tree(pre_tree[i])

    if len(pre_tree)>1:
        build_tree(pre_tree)
        ans=output(pre_tree[0],[])
        for i in range(len(ans)):
            ans[i]=str(ans[i])
        print(' '.join(ans))
    elif len(pre_tree)==1:
        print(pre_tree[0].value)
```

2.二叉搜索树的层次遍历

输入一个数字列表，建立二叉搜索树并按层次遍历

```
#
class Tree:
    def __init__(self,v):
        self.right=None
        self.left=None
        self.value=v


def insert(root,node):
    if not root:
        return node

    if node.value>root.value:
        root.right=insert(root.right,node)

    elif node.value<root.value:
        root.left=insert(root.left,node)

    return root

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

n=list(map(int,input().split()))
tree=[]
for i in n:
    if i not in tree:
```

```
        tree.append(i)
root=Tree(tree.pop(0))

while tree:
    node=tree.pop(0)
    node=Tree(node)
    root=insert(root,node)
output=level_traversal(root)
print(' '.join(map(str,output)))
```

树的转换：

把一棵 普通树 转化为 二叉树

核心方法：左儿子右兄弟

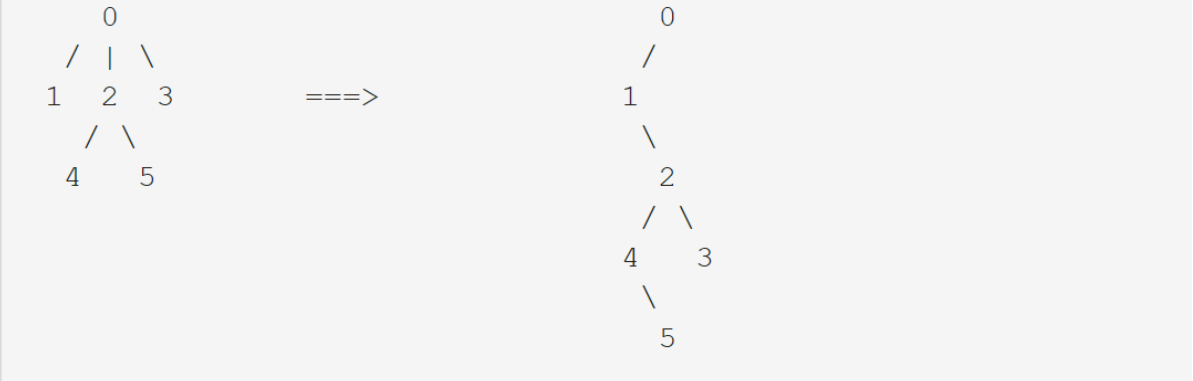
例题：

04081:树的转换

总时间限制：5000ms　单个测试点时间限制：1000ms　内存限制：65536kB

描述

我们都知道用"左儿子右兄弟"的方法可以将一棵一般的树转换为二叉树，如：



现在请你将一些一般的树用这种方法转换为二叉树，并输出转换前和转换后树的高度。

```
#
class Node:
    def __init__(self):
        self.children=[]
        self.parent=None
        self.left=None
        self.right=None

def build_tree(node):
    if node:
```

```
        x=Node()
        x.parent=node

        node.children.append(x)
        return x

def initial_tree_height(root,ans):
    global output1
    if root.children:
        for i in root.children:
            initial_tree_height(i,ans+1)
    else:
        output1.append(ans)

def binary_tree_height(node):
    if node is None:
        return -1
    left_height=binary_tree_height(node.left)
    right_height=binary_tree_height(node.right)
    return max(left_height,right_height)+1

def change_tree(root):
    if not root:
        return None

    if len(root.children) > 0:
        left_child = change_tree(root.children[0])
        root.left = left_child
        left_child.parent = root
        for i in range(1, len(root.children)):
            right_child = change_tree(root.children[i])
            left_child.right = right_child
            right_child.parent = left_child
            left_child = right_child
    return root

a=input()
#build initial tree
root=Node()
node=root
for i in range(len(a)):
    if a[i]=='d':
        node=build_tree(node)
    else:
        x=node
        node=x.parent

output1=[]
initial_tree_height(root,0)
initial_height=max(output1)#初始树高度
change_tree(root)
post_height=binary_tree_height(root)
print(initial_height,'=>',post_height)
```

把二叉树转换为普通树：逆过程

原理：每有一个左节点就代表有一层新的树，所有的右节点都是最左节点的兄弟

哈夫曼编码树

这段代码首先定义了一个 `Node` 类来表示哈夫曼树的节点。然后，使用最小堆来构建哈夫曼树，每次==从堆中取出两个频率最小的节点==进行合并，直到堆中只剩下一个节点，即哈夫曼树的根节点。接着，使用递归方法计算哈夫曼树的带权外部路径长度（weighted external path length）。最后，输出计算得到的带权外部路径长度。

题面要求：

连接左子节点的边代表0,连接右子节点的边代表1

权值小的节点算小。权值相同的两个节点，字符集里最小字符小的，算小。
例如 ({'c','k'},12) 和 ({'b','z'},12)，后者小。

合并两个节点时，小的节点必须作为左子节点

WPL值：哈夫曼树的 WPL（Weighted Path Length）值是指所有叶子节点的权重乘以它们到根节点的路径长度之和。路径长度是指从根节点到叶子节点的路径上所经过的边的数量。换句话说，WPL 是所有叶子节点的权重乘以它们的深度（根节点到该叶子节点的距离）之和。

```
import heapq

class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight
    #在这个方法中，首先检查两个对象的 weight 属性是否相等。如果相等，那么就比较它们的 char 属性，返回 self.char < other.char 的结果。如果 weight 属性不相等，就直接返回 self.weight < other.weight 的结果

def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight)#把两个最小的节点构建一个融合的新节点
        merged.left = left#初始化左右子节点
        merged.right = right
        heapq.heappush(heap, merged)
```

```
    return heap[0]#返回根节点

def encode_huffman_tree(root):
    #这是对哈夫曼树进行编码的函数。它使用递归方法遍历哈夫曼树的所有节点，并为每个叶子节点（即带有字符的节点）生成对应的编码
    codes = {}

    def traverse(node, code):
        if node.char:
            codes[node.char] = code
        else:#左子树+0，右子树+1
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes#最后返回一个字典，其中键是字符，值是对应的编码。

def huffman_encoding(codes, string):
    #它接受一个字典 codes，其中键是字符，值是对应的编码，以及一个字符串 string。它遍历字符串中的每个字符，将每个字符根据编码转换为对应的二进制字符串，并将所有二进制字符串连接起来，最后返回编码后的字符串。
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    #这是对哈夫曼编码进行解码的函数。它接受哈夫曼树的根节点 root 和一个编码后的字符串 encoded_string。它从根节点开始，根据编码的每一位 0 或 1 向左或向右遍历树，直到找到叶子节点，即带有字符的节点。然后将找到的字符添加到解码后的字符串中，继续从根节点开始下一轮遍历，直到解码完整个字符串。
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

        if node.char:
            decoded += node.char
            node = root
    return decoded

n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)


huffman_tree = build_huffman_tree(characters)


codes = encode_huffman_tree(huffman_tree)
```

```
strings = []
while True:
    try:
        line = input()
        if line:
            strings.append(line)
        else:
            break
    except EOFError:
        break

results = []

for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, string))
    else:
        results.append(huffman_encoding(codes, string))

for result in results:
    print(result)
```

collection库

```
#counter：计数字典子类，适用于list，str
#可以生成一个字典，其中key是对应的元素，value是该元素的出现次数
from collections import Counter
# 从可迭代对象创建 Counter
my_list = [1, 2, 3, 1, 2, 3, 4, 5]
my_counter = Counter(my_list)
print(my_counter)
# 输出: Counter({1: 2, 2: 2, 3: 2, 4: 1, 5: 1}

#也可直接自定义通过关键字参数创建 Counter
my_counter_explicit = Counter(a=2, b=3, c=1)
print(my_counter_explicit)
# 输出: Counter({'b': 3, 'a': 2, 'c': 1})

#特别地，访问不存在的元素不会引发 KeyError，而会返回 0
count_of_6 = my_counter[6]
print(count_of_6)
# 输出: 0

#常用的一些操作
# 获取所有元素
elements = my_counter.elements()
print(list(elements))
# 输出: [1, 1, 2, 2, 3, 3, 4, 5]
# 获取最常见的 n 个元素及其计数，会返回一个列表，其中包含计数最多的前 n 个元素及其计数。每个元素都表示为一个元组，第一个元素是元素本身，第二个元素是它在 Counter 对象中的计数。
most_common = my_counter.most_common(2)
print(most_common)
# 输出: [(1, 2), (2, 2)]
```

```
# 更新计数，即增加这些元素
my_counter.update([1, 2, 3, 4])
print(my_counter)
# 输出: Counter({1: 3, 2: 3, 3: 3, 4: 2, 5: 1})
```

其他捞分小tips

1.除法是否得到整数？（4/2=2.0）

2.缩进错误

3.用于调试的print是否删去

4.非一般情况的边界条件

5.递归中的return的位置

6.贪心是否最优

7.正难则反

8.审题是否准确？是否漏掉输出？