数据结构与算法笔记

1.二分的时候注意精度问题，如果能转为整型优先转为整型。注意自己写的是开区间还是闭区间。2.如果要取整，注意是怎么个取整法，{:.2f}是保留两位小数。3.出bug时记得检查有没有变量混用。

一些用法

```
import functools
a=[]
ans=k=p=0
s=""
def cmp(x,y):
    if x+y>y+x:return -1#return -1是x在前面
    elif x+y<y+x:return 1
    else:return 0
a.sort(key=functools.cmp_to_key(cmp))#自定义比较函数
import random
random.randomint(1,10)#两侧都是闭区间
random.uniform(1,10)#均匀分布
from collections import Counter
cnt=Counter([(k-1,0)])
for i,c in enumerate(s):
    sum+=1 if a[ord(c)-ord('a')] else -1
    p=(i%k,sum)
    ans+=cnt[p]
    cnt[p]+=1
s=input()
t=input()
c1=Counter(s)
c2=Counter(t)
c3=c2-c1
#Counter是特殊的字典，使用前可以不判断有没有，且可以作差
from functools import lru_cache#记忆化搜索
import bisect
b=0
index=bisect.bisect_left(b,i, l,r)#二分搜索,后两个参数可以指定查找区间
import itertools
n = int(input())
a = list(map(int, input().split()))
result = set(itertools.permutations(a))
for permutation in sorted(result):
    print(" ".join(map(str, permutation)))#全排列
import heapq
class node:
    def __init__(self,x,y,step):
        self.step = step
        self.x = x
        self.y = y
    def __lt__(self,other):
        return self.step < other.step#可以自定义比较函数，类似结构体
import sys
data=sys.stdin.read()
input=sys.stdin.read().split()#以空白字符分割为一个列表
```

```
my_list = ['a', 'b', 'c', 'd']
value = 'c'
try:
    index = my_list.index(value)
    print(f"{value} 在列表中的索引为 {index}")
except ValueError:
    print(f"{value} 不在列表中")#用index查找是否存在，可以返回索引
import re#正则表达式
ret=re.match("[hH].\\d","hel9")#match只能从开头开始匹配
#在末尾加*表示上一个字符出现0或多次，+是至少一次，？是0次或1次，{m,n}是m次到n次
ret1 = re.match("[a-zA-Z0-9_]{8,20}","1ad12f23s34455ff66")#匹配8到20位的密码
ret2 = re.match("([^-]*)-(\\d+)","010-12345678")
print(ret2.group())#010-12345678
print(ret2.group(1))#010
print(ret2.group(2))#12345678
ret3=re.search("[1-9]*","123534")#serach不一定从开头开始
ret4=re.findall("[1-9]*","123534")#找到所有，返回一个列表
ret5 = re.split(r":| ","info:xiaoZhang 33 shandong")#分割，不加参数默认无限次
ret6 = re.sub(r"\d+", '998', "python = 997")#替换，不加参数默认无限次
data = [(1, 3), (2, 2), (3, 1)]
sorted_data = sorted(data, key=lambda x: x[1])#按元组的第二个元素排序
x=y=m=1
pow(x, y, m)#快速幂计算x^ymodm
import re
s1=input()
s2=input()
s1=re.escape(s1)
s1=s1.replace(r'\?','.').replace(r'\*','.*')
print('matched' if re.match(s1,s2) else 'not matched')#带通配符的字符匹配，先转义
```

一、队列、栈、堆、排序、KMP

1.当前队列中位数（懒标记）

```
import heapq
from collections import defaultdict,deque
class Midlist:
    def __init__(self):
        self.q1=[]
        self.q2=[]
        self.l1=0
        self.l2=0
        self.q=deque()
        self.mp=defaultdict(int)
    def remove(self):
        while self.q1 and self.mp[self.q1[0]]>0:
            self.mp[self.q1[0]]-=1
            heapq.heappop(self.q1)
        while self.q2 and self.mp[-self.q2[0]]>0:
            self.mp[-self.q2[0]]-=1
            heapq.heappop(self.q2)
    def balance(self):
        while self.l1 > self.l2 + 1:
            temp = heapq.heappop(self.q1)
            if self.mp[temp] > 0:
```

```
                self.mp[temp] -= 1
            else:
                heapq.heappush(self.q2, -temp)
                self.l1 -= 1
                self.l2 += 1
        while self.l1 < self.l2:
            temp = -heapq.heappop(self.q2)
            if self.mp[temp] > 0:
                self.mp[temp] -= 1
            else:
                heapq.heappush(self.q1, temp)
                self.l1 += 1
                self.l2 -= 1
        self.remove()
    def insert(self, val):
        if not self.q1 or val >= self.q1[0]:
            heapq.heappush(self.q1, val)
            self.l1 += 1
        else:
            heapq.heappush(self.q2, -val)
            self.l2 += 1
        self.q.append(val)
        self.balance()
    def delete(self):
        self.balance()
        val = self.q.popleft()
        self.mp[val] += 1
        if val >= self.q1[0]:
            self.l1 -= 1
        else:
            self.l2 -= 1
        self.balance()
    def query(self):
        self.remove()
        if (self.l1 + self.l2) & 1:
            return self.q1[0]
        else:
            return (self.q1[0] - self.q2[0]) / 2
n=int(input())
t=Midlist()
for _ in range(n):
    s=input().split()
    if s[0][0]=='a':
        t.insert(int(s[1]))
    elif s[0][0]=='d':
        t.delete()
    else:
        temp=t.query()
        print(int(temp) if temp==int(temp) else "{:.1f}".format(temp))
```

2.给定m个数字序列，每个序列包含n个非负整数。我们从每一个序列中选取一个数字组成一个新的序列，显然一共可以构造出n^m个新序列。接下来我们对每一个新的序列中的数字进行求和，一共会得到n^m个和，请找出最小的n个和（两两合并）

```
import heapq
```

```
n=0
def merge(a,b):
    q=[]
    res=[]
    vis=set()
    vis.add((0,0))
    heapq.heappush(q,(a[0]+b[0],0,0))
    while len(res)<n:
        s,i,j=heapq.heappop(q)
        res.append(s)
        if i+1<n and (i+1,j) not in vis:
            vis.add((i+1,j))
            heapq.heappush(q,(a[i+1]+b[j],i+1,j))
        if j+1<n and (i,j+1) not in vis:
            vis.add((i,j+1))
            heapq.heappush(q,(a[i]+b[j+1],i,j+1))
    return res
t=int(input())
for _ in range(t):
    m,n=map(int,input().split())
    ans=list(map(int,input().split()))
    ans.sort()
    for __ in range(m-1):
        temp=list(map(int,input().split()))
        ans=merge(ans,sorted(temp))
    print(*ans)
```

3.归并排序求逆序数

```
n=int(input())
a=list(map(int,input().split()))
ans=0
def mergesort(b):
    global ans
    if len(b)>1:
        mid=len(b)//2
        L=b[:mid]
        R=b[mid:]
        mergesort(L)
        mergesort(R)
        i=j=k=0
        while i<len(L) and j < len(R):
            if L[i]<=R[j]:
                b[k]=L[i]
                i+=1
            else:
                b[k]=R[j]
                ans+=len(L)-i
                j+=1
            k+=1
        while i<len(L):
            b[k]=L[i]
            i+=1
            k+=1
        while j<len(R):
```

```
            b[k]=R[j]
            j+=1
            k+=1
mergesort(a)
print(ans)
```

4.中序转后序(优先级大于等于的pop出来)

```
def change(s):
    mp={'+':1,'-':1,'*':2,'/':2}
    stack=[]
    postfix=[]
    number=""
    for x in s:
        if x.isnumeric() or x=='.':number+=x
        else:
            if number:
                postfix.append(int(number) if '.' not in number else float(number))
                number=""
            if x in mp:
                while stack and stack[-1]in mp and mp[stack[-1]]>=mp[x]:
                    postfix.append(stack.pop())
                stack.append(x)
            elif x=='(':stack.append(x)
            elif x==')':
                while stack and stack[-1]!='(':postfix.append(stack.pop())
                stack.pop()
    if number:postfix.append(int(number) if '.' not in number else float(number))
    while stack:
        postfix.append(stack.pop())
    return ' '.join(map(str,postfix))
n=int(input())
for _ in range(n):
    print(change(input()))
```

5.可以用模拟法直接验证是否是合法出栈序列,序列总数事实上是comb(2*n,n)//(n+1),这也是n个节点的二叉树数量。

```
s=input()
def check(x):
    if len(x)!=len(s):
        return False
    stack=[]
    i=0
    for ch in x:
        if stack and ch==stack[-1]:
            stack.pop()
            continue
        while i<len(s) and ch!=s[i]:
            stack.append(s[i])
            i+=1
        if i<len(s) and ch==s[i]:
            i+=1
        else:
```

```
            return False
    return True
while 1:
    try:
        temp=input()
        print('YES' if check(temp) else 'NO')
    except EOFError:
        break
```

6.奶牛排队（单调栈）：奶牛在熊大妈的带领下排成了一条直队。显然，不同的奶牛身高不一定相同……

现在，奶牛们想知道，如果找出一些连续的奶牛，要求最左边的奶牛 A 是最矮的，最右边的 B 是最高的，且 B 高于 A 奶牛。中间如果存在奶牛，则身高不能和 A,B 奶牛相同。问这样的奶牛最多会有多少头?

从左到右给出奶牛的身高，请告诉它们符合条件的最多的奶牛数（答案可能是 0,2，但不会是 1）。

```
import bisect
n=int(input())
s1,s2=[-1],[-1]
ans=0
h=[]
for i in range(n):
    h.append(int(input()))
    while len(s1)>1 and h[s1[-1]]>=h[-1]:
        s1.pop()#s1是严格单调递增栈，栈顶就是往左第一个能作为到当前下标的区间最小值的位置
    while len(s2)>1 and h[s2[-1]]<h[-1]:
        s2.pop()#s2是单调递减栈，栈顶是往左第一个不低于当前高度的下标
    pos=bisect.bisect_right(s1,s2[-1])#找到第一个能够作为起始点的下标
    if pos<len(s1):
        ans=max(ans,i-s1[pos]+1)
    s1.append(i)
    s2.append(i)
print(ans)
```

7.KMP算法（字符串匹配）

next[i]是使得s[0…k]与s[i-k…i]相等的最大的k的值，如果没有为-1.求解时先令j=next[0]=-1,历遍i,不断令j=next[j],直到j==-1 or s[i]==s[j+1],如果s[i]==s[j+1]则令j+=1,最后next[i]=j，给定文本串text,模式串pattern,i指向text的待比较位置，j指向pattern已经被匹配的最后一位，匹配时如果失配就让j=next[j]直到j==-1 or text[i]==pattern[j+1].如果是统计patternd的出现次数（可以重叠），则在匹配成功后也让j=next[j]即可.

最简单的统计出现次数（可以重叠）

```
s1=input()
s2=input()#要生成next的
n=len(s2)
next=[-1]*n
j=-1
for i in range(1,n):
    while j!=-1 and s2[i]!=s2[j+1]:
        j=next[j]
    if s2[i]==s2[j+1]:
        j+=1
```

```
    next[i]=j
ans=0
j=-1
for i in range(len(s1)):
    while j!=-1 and s1[i]!=s2[j+1]:
        j=next[j]
    if s1[i]==s2[j+1]:
        j+=1
    if j==n-1:
        ans+=1
        j=next[j]
print(ans)
```

现有一个字符串，计算它最多是由多少个相同子串拼成的.利用的性质是如果一个字符串的最小循环节长度是n-next[n-1]-1（如果能整除）（next的定义会差个1），否则不循环.

```
import sys
while True:
    s = sys.stdin.readline().strip()
    if s == '.':
        break
    len_s = len(s)
    next = [0] * len(s)
    j = 0
    for i in range(1, len_s):
        while j > 0 and s[i] != s[j]:
            j = next[j - 1]
        if s[i] == s[j]:
            j += 1
        next[i] = j
    base_len = len(s)-next[-1]
    if len(s) % base_len == 0:
        print(len_s // base_len)
    else:
        print(1)
```

8.统计原子数量（利用栈和Counter),输入如H2O

```
from collections import Counter
class Solution:
    def countOfAtoms(self, formula: str) -> str:
        n = len(formula)
        def get_name(i):
            res = formula[i]
            i += 1
            while i < n and formula[i].islower():
                res += formula[i]
                i += 1
            return res, i
        def get_num(i):
            if i >= n or not formula[i].isdigit():
                return 1,i
            num = int(formula[i])
            i += 1
```

```
            while i < n and formula[i].isdigit():
                num = num * 10 + int(formula[i])
                i += 1
            return num, i
        stack = [Counter()]
        i = 0
        while i < n:
            ch = formula[i]
            if ch == '(':
                stack.append(Counter())
                i += 1
            elif ch == ')':
                i += 1
                mul, i = get_num(i)
                temp = stack.pop()
                for atom, cnt in temp.items():
                    stack[-1][atom] += cnt * mul
            else:
                atom, i = get_name(i)
                cnt, i = get_num(i)
                stack[-1][atom] += cnt
        ans = []
        mp = stack.pop()
        for key in sorted(mp.keys()):
            ans.append(key)
            if mp[key] > 1:
                ans.append(str(mp[key]))
        return ''.join(ans)
```

9.用二次探查法建立散列表 探查增量序列依次为1,-1,4,-4…,注意重复关键字要映射到同一个位置，并且这里要快读.

```
import sys
input=sys.stdin.read
data=input().split()
pos=0
n=int(data[pos])
pos+=1
m=int(data[pos])
pos+=1
a=[int(x) for x in data[pos:pos+n]]
mp={}
visited=[False]*m
for x in a:
    if x in mp:
        continue
    index=x%m
    if not visited[index]:
        visited[index]=True
        mp[x]=index
    else:
        i=1
        while 1:
            index1=(index-i*i)%m
            index2=(index+i*i)%m
```

```
            if not visited[index2]:
                visited[index2]=True
                mp[x]=index2
                break
            elif not visited[index1]:
                visited[index1]=True
                mp[x]=index1
                break
            i+=1
b=[mp[x] for x in a]
print(' '.join(map(str,b)))
```

10.给你一个字符串 `s` ，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 **返回结果的字典序最小**（要求不能打乱其他字符的相对位置）。单调栈(维护字符单增，但是要注意字符可能被用完)

```
class Solution:
    def removeDuplicateLetters(self, s: str) -> str:
        stack=[]
        mp=defaultdict(int)
        visited=defaultdict(bool)
        for x in s:
            mp[x]+=1
        for x in s:
            if visited[x]:
                mp[x]-=1
                continue
            while stack and x<stack[-1] and mp[stack[-1]]:
                visited[stack.pop()]=False
            stack.append(x)
            visited[x]=True
            mp[x]-=1
        return ''.join(stack)
```

11.最大矩形给定一个仅包含 `0` 和 `1` 、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 `1` 的最大矩形，并返回其面积。（维护一个单调递增的栈）

```
class Solution:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        m,n=len(matrix),len(matrix[0])
        stack=[]
        h=[0]*(n+1)
        h[n]=-float('inf')
        ans=0
        for i in range(m):
            for j in range(n+1):
                if j<n:
                    if matrix[i][j]=='1':
                        h[j]+=1
                    else:
                        h[j]=0
                while stack and h[j]<h[stack[-1]]:
                    height=h[stack.pop()]
                    left=stack[-1] if stack else -1
                    ans=max(ans,height*(j-left-1))
```

```
                stack.append(j)
            stack.pop()
        return ans
```

二、树

1.根据二叉树前中序序列建树,每次找到中序历遍序列中对应的节点作为左右子树的分界。

```
class Node():
    def __init__(self, val=""):
        self.val=val
        self.left=None
        self.right=None
def create(prel,prer,inl,inr):
    if prer<prel:
        return None
    node=Node(preorder[prel])
    k=inl
    while k<=inr:
        if inorder[k]==preorder[prel]:
            break
        k+=1
    numleft=k-inl
    node.left=create(prel+1,prel+numleft,inl,k-1)
    node.right=create(prel+numleft+1,prer,k+1,inr)
    return node
def postorder(root):
    if not root:
        return
    postorder(root.left)
    postorder(root.right)
    res.append(root.val)
while 1:
    try:
        preorder=input()
        inorder=input()
        n=len(preorder)
        root=create(0,n-1,0,n-1)
        res=[]
        postorder(root)
        print(''.join(res))
    except EOFError:
        break
```

2.猜二叉树（中后序历遍建树并层序历遍）

```
from collections import deque
n=int(input())
class TreeNode():
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
def find(inL,inR,x):
```

```
    for i in range(inL,inR+1):
        if s1[i]==x:return i
def create(postL,postR,inL,inR):
    if postL>postR:return None
    root=TreeNode(s2[postR])
    k=find(inL,inR,s2[postR])
    num_left=k-inL
    root.left=create(postL,postL+num_left-1,inL,k-1)
    root.right=create(postL+num_left,postR-1,k+1,inR)
    return root
def level_order(root):
    q=deque()
    q.append(root)
    while q:
        p=q.popleft()
        print(p.val,end="")
        if p.left:q.append(p.left)
        if p.right:q.append(p.right)
for _ in range(n):
    s1=input()
    s2=input()
    root=create(0,len(s1)-1,0,len(s1)-1)
    level_order(root)
    print()
```

3.由层序遍历序列和中序遍历序列确定二叉树

- 根据层序遍历序列第一个结点确定根结点；
- 根据根结点在中序遍历序列中分割出左右子树的中序序列；
- 根据分割出的左右子树的中序序列从层序序列中提取出对应的左右子树的层序序列；
- 对左子树和右子树分别递归使用相同的方式继续分解；
- ```
  class Node:
      def __init__(self, val=0):
          self.val = val
          self.left = None
          self.right = None
  def preorder(root):
      if not root:
          return
      print(root.val, end=" ")
      preorder(root.left)
      preorder(root.right)
  def create(level_order, inorder, inL, inR, pos_map):
      if not level_order or inL > inR:
          return None
      root_val = level_order[0]
      pos = pos_map[root_val]
      # 将剩余层序按在中序中位置分到左右子树
      left_levels, right_levels = [], []
      for v in level_order[1:]:
          if pos_map[v] < pos:
              left_levels.append(v)
          else:
  ```
  ```

```
            right_levels.append(v)
    node = Node(root_val)
    node.left  = create(left_levels,  inorder, inL,     pos-1, pos_map)
    node.right = create(right_levels, inorder, pos+1, inR,     pos_map)
    return node
if __name__ == "__main__":
    n = int(input())
    level_order = input().split()
    inorder     = input().split()
    # 建立 值→中序下标 的哈希表
    pos_map = { val:i for i,val in enumerate(inorder) }
    root = create(level_order, inorder, 0, n-1, pos_map)
    preorder(root)
    print()
```

4.利用完全括号表达式建立解析式树时，遵循以下规则：

(1) 如果当前标记是(，就为当前节点添加一个左子节点，并下沉至该子节点； (2) 如果当前标记在列表 ['+', '-', '/', ' * '] 中，就将当前节点的值设为当前标记对应的运算符；为当前节点 添加一个右子节点，并下沉至该子节点； (3) 如果当前标记是数字，就将当前节点的值设为这个数并返回至父节点； (4) 如果当前标记是)，就跳到当前节点的父节点。

5.根据后序表达式建立表达式树（利用后序表达式建立表达式树，其层序历遍结果颠倒后就是队列表达式）

队列表达式是从左到右扫描，碰到数就入队，碰到操作符就取队首两个数运算后入队（先出队的是第二个运算数）

```
from collections import deque
class Node():
    def __init__(self, val=""):
        self.val = val
        self.left = None
        self.right = None
n=int(input())
for _ in range(n):
    s=input()
    stack=[]
    for x in s:
        if x.islower():
            stack.append(Node(x))
        else:
            new_node=Node(x)
            new_node.right=stack.pop()
            new_node.left=stack.pop()
            stack.append(new_node)
    res=[]
    q=deque()
    q.append(stack[-1])
    while q:
        p=q.popleft()
        if not p:
            continue
        res.append(p.val)
        q.append(p.left)
        q.append(p.right)
```

```
        print(''.join(reversed(res)))
```

6.printExp（逆波兰表达式建树），输出时不能有多余的括号。

```
class Node():
    def __init__(self, val=""):
        self.val=val
        self.left=None
        self.right=None
def change(string):#中缀转后缀
    res=[]
    op=[]
    postlist=[]
    inlist=string.split()
    mp={'(':0,'or':1,'and':2,'not':3}
    for x in inlist:
        if x=='(':
            op.append(x)
        elif x=='True' or x=='False':
            postlist.append(x)
        elif x==')':
            while op and op[-1]!='(':
                postlist.append(op.pop())
            op.pop()
        else:
            while op and mp[op[-1]]>=mp[x]:
                postlist.append(op.pop())
            op.append(x)
    while op:
        postlist.append(op.pop())
    return postlist
def create(infix):
    postlist=change(infix)
    stack=[]
    for x in postlist:
        if x=='not':
            new_node=Node(x)
            new_node.left=stack.pop()
            stack.append(new_node)
        elif x=='True' or x=='False':
            stack.append(Node(x))
        else:
            right,left=stack.pop(),stack.pop()
            new_node=Node(x)
            new_node.right=right
            new_node.left=left
            stack.append(new_node)
    return stack[-1]
def output(root):
    if root.val=='or':
        return output(root.left)+['or']+output(root.right)
    elif root.val=='not':
        return ['not']+(['(']+output(root.left)+[')']) if root.left.val not in
['True','False'] else ['not']+output(root.left)
    elif root.val=='and':
```

```
        leftpart=['(']+output(root.left)+[')'] if root.left.val=='or' else
output(root.left)
        rightpart=['(']+output(root.right)+[')'] if root.right.val=='or' else
output(root.right)
        return leftpart+['and']+rightpart
    else:
        return [root.val]
s=input()
root=create(s)
print(' '.join(output(root)))
```

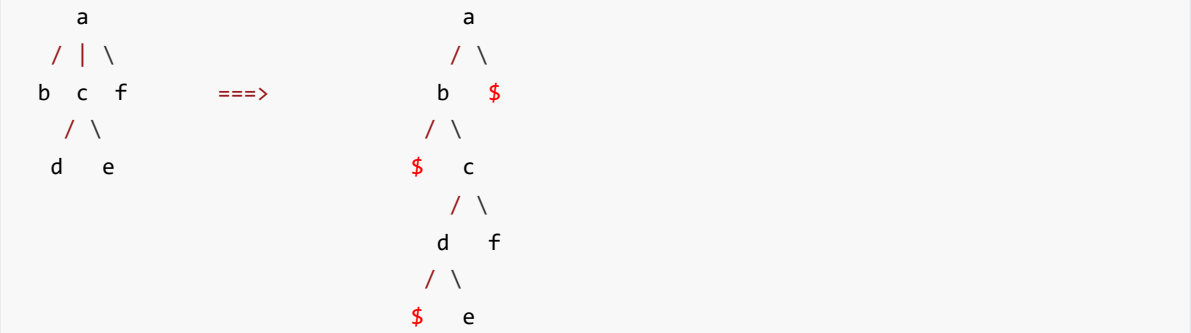
7.哈夫曼树

```
import heapq
class Node():
    def __init__(self, char="",val=0):
        self.val=val
        self.char=char
        self.left=None
        self.right=None

    def __lt__(self,other):
        if self.val==other.val:
            return self.char<other.char
        return self.val<other.val
n=int(input())
q=[]
a=[]
for _ in range(n):
    s,x=input().split()
    a.append(x)
    x=int(x)
    heapq.heappush(q,Node(s,x))
while len(q)>1:
    x,y=heapq.heappop(q),heapq.heappop(q)
    new_node=Node(min(x.char,y.char),x.val+y.val)
    new_node.left=x
    new_node.right=y
    heapq.heappush(q,new_node)
char_to_num,num_to_char={},{}
root=q[0]
def dfs(root,path):
    if not root:
        return
    if not root.left and not root.right:
        char_to_num[root.char]=path
        num_to_char[path]=root.char
    dfs(root.left,path+'0')
    dfs(root.right,path+'1')
dfs(root,"")
while 1:
    try:
        s=input()
        if s.isdigit():
            temp=""
```

```
            pos=0
            res=""
            while pos<len(s):
                temp+=s[pos]
                if temp in num_to_char:
                    res+=num_to_char[temp]
                    temp=""
                pos+=1
            print(res)
        else:
            res=[]
            for x in s:
                res.append(char_to_num[x])
            print(''.join(res))
    except EOFError:
        break
```

8.树的镜面映射(涉及翻转，层序历遍，二叉树与多叉树的转化)




```
from collections import deque
class Node():
    def __init__(self, val=""):
        self.val=val
        self.left=None
        self.right=None
        self.children=[]
def lever_order(root):
    q=deque()
    res=[]
    q.append(root)
    while q:
        p=q.popleft()
        if not p:
            continue
        res.append(p.val)
        for x in p.children:
            q.append(x)
    return res
def add_children(root):
    if root.left and root.left.val!='$':
        cur=root.left
        while cur and cur.val!='$':
            root.children.append(cur)
            cur=cur.right
```

```
def create(root):
    if not root or root.val=='$':
        return
    add_children(root)
    create(root.left)
    create(root.right)
def flip(root):
    if not root:
        return
    for x in root.children:
        flip(x)
    root.children=root.children[::-1]
n=int(input())
s=input().split()
stack=[]
root=Node(s[0][0])
root.left=Node()
stack.append(root)
root=root.left
for i in range(1,len(s)):
    if s[i][1]=='0':
        root.val=s[i][0]
        root.left=Node()
        stack.append(root)
        root=root.left
    else:
        root.val=s[i][0]
        root=stack.pop()
        while stack and root.right:
            root=stack.pop()
        if i==len(s)-1:
            break
        root.right=Node()
        stack.append(root)
        root=root.right
create(root)
flip(root)
print(' '.join(lever_order(root)))
```

9.向下调整建堆(此时只要历遍非叶子节点)

```
n=int(input())
a=list(map(int,input().split()))
a.insert(0,-1)
i=len(a)//2
def down_adjust(index):
        while 2*index<len(a):
            pos=index*2 if 2*index+1>=len(a) or a[2*index]>a[2*index+1] else index*2+1
            if a[index]<a[pos]:
                a[index],a[pos]=a[pos],a[index]
                index=pos
            else:
                break
while i>0:
    down_adjust(i)
```

```
    i-=1
print(' '.join(map(str,a[1::])))
```

10.向上调整建堆

```
n=int(input())
a=list(map(int,input().split()))
a.insert(0,-1)
def up_adjust(index):
    while index>1 and a[index]>a[index//2]:
        a[index],a[index//2]=a[index//2],a[index]
        index//=2
for i in range(2,n+1):
    up_adjust(i)
print(' '.join(map(str,a[1::])))
```

11.堆排序(先建堆，然后每次都交换节点，等价于pop出来其实)

```
n=int(input())
a=list(map(int,input().split()))
a.insert(0,-1)
def down_adjust(index,l):
    while 2*index<=l:
        pos=2*index if 2*index+1>l or a[2*index]>a[2*index+1] else 2*index+1
        if a[index]<a[pos]:
            a[index],a[pos]=a[pos],a[index]
            index=pos
        else:
            break
for i in range(len(a)//2,0,-1):
    down_adjust(i,n)
for i in range(n,1,-1):
    a[i],a[1]=a[1],a[i]
    down_adjust(1,i-1)
print(' '.join(map(str,a[1::])))
```

12.根据二叉搜索树的前序历遍还原二叉树

```
class Node():
    def __init__(self,val=0):
        self.val=val
        self.left=None
        self.right=None
def create(preL,preR):
    if preR<preL:
        return None
    root=Node(a[preL])
    pos=preR+1
    for i in range(preL+1,preR+1):
        if a[i]>a[preL]:
            pos=i
            break
    root.left=create(preL+1,pos-1)
    root.right=create(pos,preR)
```

```
    return root
res=[]
def postorder(root):
    if not root:
        return
    postorder(root.left)
    postorder(root.right)
    res.append(root.val)
n=int(input())
a=list(map(int,input().split()))
root=create(0,n-1)
postorder(root)
print(' '.join(map(str,res)))
```

13.n层的AVL树最少节点数$a_n = a_{n-1} + a_{n-2} + 1$,与之相对应，自然也可以计算n个节点的AVL树的最大层数。

14.食物链

带权并查集

```
n,k=map(int,input().split())
num=0
root=list(range(n+1))
dist=[0]*(n+1)
def find_root(x):
    if x!=root[x]:
        t=root[x]
        root[x]=find_root(root[x])
        dist[x]=(dist[x]+dist[t])%3
    return root[x]
for _ in range(k):
    d,x,y=map(int,input().split())
    if x>n or y>n or (d==2 and x==y):num+=1
    else:
        rx,ry=find_root(x),find_root(y)
        if rx==ry:
            if d-1!=(dist[x]-dist[y]+3)%3:num+=1
        else:
            root[rx]=ry
            dist[rx]=(-dist[x]+dist[y]+d+2)%3
print(num)
```

开三倍空间

```
n,k=map(int,input().split())
root=list(range(3*n+1))#1-n表示同类，n+1-2n表示x吃的动物，2n+1-3n表示吃x的动物
def find_root(x):
    if x!=root[x]:
        root[x]=find_root(root[x])
    return root[x]
ans=0
for _ in range(k):
    d,x,y=map(int,input().split())
    if x>n or y>n or (x==y and d==2):
```

```
            ans+=1
            continue
        rx1,ry1=find_root(x),find_root(y)
        rx2,ry2=find_root(x+n),find_root(y+n)
        rx3,ry3=find_root(x+2*n),find_root(y+2*n)
        if d==1:
            if rx1==ry2 or rx1==ry3:
                ans+=1
                continue
            root[rx1]=ry1
            root[rx2]=ry2
            root[rx3]=ry3
        else:
            if rx1==ry1 or rx3==ry1:
                ans+=1
                continue
            root[rx2]=ry1
            root[rx1]=ry3
            root[rx3]=ry2
    print(ans)
```

15.蚂蚁王国越野跑（统计逆序数）（树状数组）(离散化)

```
lowbit=[i&-i for i in range(100001)]
while 1:
    try:
        n=int(input())
        a=[]
        for _ in range(n):
            a.append(int(input()))
        sorted_a=sorted(set(a))
        v_to_index={v:i+1 for i,v in enumerate(sorted_a)}
        m=len(sorted_a)
        tr=[0]*(m+1)
        def cal(x):
            num=0
            while x>0:
                num+=tr[x]
                x-=lowbit[x]
            return num
        def update(index,val):
            while index<=m:
                tr[index]+=val
                index+=lowbit[index]
        res=0
        for x in a:
            index=v_to_index[x]
            res+=cal(index-1)
            update(index,1)
        print(res)
        input()
    except EOFError:
        break
```

16.将一般的树转为二叉树（左长子右兄弟）

```
class Node:
    def __init__(self,val=0):
        self.val=val
        self.children=[]
class Bnode:
    def __init__(self,val=0):
        self.val=val
        self.left=None
        self.right=None
s=input()
stack=[]
root=Node()
for x in s:
    if x=='d':
        root.children.append(Node())
        stack.append(root)
        root=root.children[-1]
    else:
        root=stack.pop()
def find_h(root):
    if not root:
        return -1
    res=-1
    for x in root.children:
        res=max(res,find_h(x))
    return res+1
h0=find_h(root)
def create(root):
    if not root:
        return None
    broot=Bnode()
    if root.children:
        broot.left=create(root.children[0])
    cur=broot.left
    for x in root.children[1:]:
        cur.right=create(x)
        cur=cur.right
    return broot
def find_bh(broot):
    if not broot:
        return -1
    return 1+max(find_bh(broot.left),find_bh(broot.right))
print("{} => {}".format(h0,find_bh(create(root))))
```

17.火星车勘探 火星车初始处于二叉树地形的根节点，对二叉树进行前序遍历。当火星车遇到非空节点时，则采样一定量的泥土样本，记录下样本数量；当火星车遇到空节点，使用一个标记值`#`进行记录。给定一串以空格分隔的序列，验证它是否是火星车对于二叉树地形正确的前序遍历结果。

其实只要统计空节点个数，维护count即可，最后count一定要是0

```
def check(s):
    count=1
    for x in s:
        if count<=0:
            return False
```

```
        if x=='#':
            count-=1
        else:
            count+=1
    return count==0
while 1:
    n=int(input())
    if not n:
        break
    temp=input().split()
    print('T' if check(temp) else 'F')
```

18.递归生成给定节点数的所有二叉搜索树（递归函数中返回一个列表，这样就不会涉及那些要修改啥的问题）

```
class Solution:
    def generateTrees(self, n: int) -> List[Optional[TreeNode]]:
        if not n:
            return []
        def create(left,right):
            all_trees=[]
            if left>right:
                return [None]
            for i in range(left,right+1):
                left_trees=create(left,i-1)
                right_trees=create(i+1,right)
                for l in left_trees:
                    for r in right_trees:
                        cur=TreeNode(i)
                        cur.left=l
                        cur.right=r
                        all_trees.append(cur)
            return all_trees
        return create(1,n)
```

19.倍增法查找第k近祖先，进而O(logn)查询两两节点间距离

```
class Solution:
    def assignEdgeWeights(self, edges: List[List[int]], queries: List[List[int]]) ->
List[int]:
        n=len(edges)+1
        m=n.bit_length()
        a=[[] for _ in range(n)]
        for x,y in edges:
            x-=1
            y-=1
            a[x].append(y)
            a[y].append(x)
        depth=[0]*n
        pa=[[-1]*m for _ in range(n)]

        def dfs(x,fa):
            pa[x][0]=fa
            for y in a[x]:
```

```
                if y!=fa:
                    depth[y]=depth[x]+1
                    dfs(y,x)
        dfs(0,-1)
        for i in range(m-1):
            for x in range(n):
                if (p:=pa[x][i])!=-1:
                    pa[x][i+1]=pa[p][i]

        def get_k(node,k):
            for i in range(k.bit_length()):
                if k>>i&1:
                    node=pa[node][i]
            return node

        def get_lca(x,y):
            if depth[x]>depth[y]:
                x,y=y,x
            y=get_k(y,depth[y]-depth[x])
            if x==y:
                return x
            for i in range(len(pa[x])-1,-1,-1):
                px,py=pa[x][i],pa[y][i]
                if px!=py:
                    x,y=px,py
            return pa[x][0]
        def get_dis(x,y):
            return depth[x]+depth[y]-depth[get_lca(x,y)]*2
        return [pow(2,get_dis(x-1,y-1)-1,10**9+7) if x!=y else 0 for x,y in queries]
```

三、图

1.电话号码（字典树）

```
delta=False
class Trie:
    def __init__(self):
        self.children=[None]*10
        self.isEnd = False

    def insert(self,word):
        global delta
        is_create=False
        p=self
        for x in word:
            x=int(x)
            if p.isEnd:
                delta=True
            if p.children[x] is None:
                is_create=True
                p.children[x]=Trie()
            p=p.children[x]
        p.isEnd=True
        if not is_create:
            delta=True
```

```
t=int(input())
for _ in range(t):
    n=int(input())
    root=Trie()
    delta=False
    for __ in range(n):
        s=input()
        root.insert(s)
    print('YES' if not delta else 'NO')
```

2.词梯（桶排序）

```
from collections import deque,defaultdict
n=int(input())
a=[[] for _ in range(n)]
s=[]
pre=[-1]*n
buckets=defaultdict(list)
for i in range(n):
    s.append(input())
    for j in range(4):
        buckets[s[i][:j]+'.'+s[i][j+1:]].append(i)
for bucket in buckets.values():
    for i in range(len(bucket)):
        for j in range(i+1,len(bucket)):
            a[bucket[i]].append(bucket[j])
            a[bucket[j]].append(bucket[i])
x,y=input().split()
u,v=-1,-1
for i in range(n):
    if s[i]==x:
        u=i
    elif s[i]==y:
        v=i
def bfs(start,end):
    delta=False
    visited=[False]*n
    q=deque()
    q.append(start)
    visited[start]=True
    while q:
        p=q.popleft()
        if p==end:
            delta=True
            break
        for pos in a[p]:
            if visited[pos]:
                continue
            visited[pos] = True
            q.append(pos)
            pre[pos]=p
    if not delta:
        return 'NO'
    path=[]
    while end!=-1:
```

```
        path.append(s[end])
        end=pre[end]
    return ' '.join(path[::-1])
print(bfs(u,v))
```

3.先导课程（用heapq保证先学序号小的）拓扑排序,拓扑排序是否唯一可以通过检查队列长度实现。（不是这题）

```
import heapq
n,m=map(int,input().split())
a=[[] for _ in range(n)]
indegree=[0]*n
for _ in range(m):
    u,v=map(int,input().split())
    a[u].append(v)
    indegree[v]+=1
q=[]
path=[]
for i in range(n):
    if indegree[i]==0:
        heapq.heappush(q,i)
while q:
    p=heapq.heappop(q)
    path.append(p)
    for x in a[p]:
        indegree[x]-=1
        if not indegree[x]:
            heapq.heappush(q,x)
if len(path)==n:
    print('Yes')
    print(' '.join(map(str,path)))
else:
    print('No')
    print(n-len(path))
```

4.DAG最短路（dp) 按字典序输出

```
from functools import lru_cache
n,m=map(int,input().split())
a=[[] for _ in range(n)]
af=[-1]*n
for _ in range(m):
    u,v,w=map(int,input().split())
    a[u].append((v,w))
for i in range(n):
    a[i].sort()
@lru_cache(maxsize=None)
def dp(i):
    res=0
    for x,w in a[i]:
        temp=dp(x)
        if temp+w>res:
            res=temp+w
            af[i]=x
```

```
    return res
ans=0
pos=-1
for i in range(n):
    temp=dp(i)
    if temp>ans:
        ans=temp
        pos=i
path=[]
while pos!=-1:
    path.append(pos)
    pos=af[pos]
print('->'.join(map(str,path)))
```

5.固定终点的DAG最长路(把除了终点外其他的初始值设置为负无穷)

```
n,m,t=map(int,input().split())
a=[[] for _ in range(n)]
for _ in range(m):
    u,v,w=map(int,input().split())
    a[u].append((v,w))
from functools import lru_cache
@lru_cache(maxsize=None)
def dp(i):
    if i==t:
        return 0
    res=-float('inf')
    for x,w in a[i]:
        res=max(res,dp(x)+w)
    return res
ans=0
for i in range(n):
    ans=max(ans,dp(i))
print(ans)
```

6.Dijkstra(同时统计最短路径)

```
import heapq
n,m,s,t=map(int,input().split())
a=[[] for _ in range(n)]
d=[float('inf')]*n
d[s]=0
num=[0]*n
num[s]=1
for _ in range(m):
    u,v,w=map(int,input().split())
    a[u].append((v,w))
    a[v].append((u,w))
q=[]
heapq.heappush(q,(0,s))
pre=[[] for _ in range(n)]
while q:
    temp=heapq.heappop(q)
    p=temp[1]
```

```
    for x in a[p]:
        if d[p]+x[1]<d[x[0]]:
            num[x[0]]=num[p]
            d[x[0]]=d[p]+x[1]
            pre[x[0]].clear()
            pre[x[0]].append(p)
            heapq.heappush(q,(x[1],x[0]))
        elif d[p]+x[1]==d[x[0]]:
            if p not in pre[x[0]]:
                num[x[0]]+=num[p]
                pre[x[0]].append(p)
ways=[]
def dfs(index,path):
    if index==s:
        ways.append(list(reversed(path)))
        return
    for x in pre[index]:
        tep=path.copy()
        tep.append(x)
        dfs(x,tep)
dfs(t,[t])
ways.sort()
print(len(ways))
for x in ways:
    print('->'.join(str(y) for y in x))
```

7.出现负权边的时候，Dijkstra算法失效，用Bellman-Ford算法，并且可以检测有无从源点出发可达的负环,改进后为SPFA算法.在统计最短路径条数的时候如果相同要重新检查，pre数组要用集合，其他情况都和Dijkstra类似.如果要检测负环，只需要检查有没有入队次数大于等于n的节点即可.

```
n,m,s,t=map(int,input().split())
a=[[] for _ in range(n)]
for _ in range(m):
    u,v,w=map(int,input().split())
    a[u].append((v,w))
    a[v].append((u,w))
from collections import deque
q=deque()
inq=[False]*n
inq[s]=True
q.append(s)
dist=[float('inf')]*n
dist[s]=0
num=[0]*n
num[s]=1
pre=[set() for _ in range(n)]
while q:
    p=q.popleft()
    inq[p]=False
    for x,w in a[p]:
        if dist[p]+w<dist[x]:
            dist[x]=dist[p]+w
            if not inq[x]:
                inq[x]=True
                q.append(x)
```

```
            num[x]=num[p]
            pre[x].clear()
            pre[x].add(p)
        elif dist[p]+w==dist[x]:
            pre[x].add(p)
            num[x]=0
            for y in pre[x]:
                num[x]+=num[y]
            if not inq[x]:
                inq[x]=True
                q.append(x)
print(dist[t],num[t])
```

8.Floyd算法以O(n^3)的时间复杂度解决全源（任意两点）最短路问题.一般建邻接表.

现有一个共n个顶点（代表城市）、m条边（代表道路）的**无向连通图**（假设顶点编号为从`0`到`n-1`），每条边有各自的边权，代表两个城市之间的距离。为了促进城市间交流，需要从k个备选城市中选择其中一个城市作为交通枢纽，满足从这个交通枢纽出发到达其他所有城市的最短距离之和最小。

```
n,m,k=map(int,input().split())
d=[[float('inf')]*n for _ in range(n)]
for _ in range(m):
    u,v,w=map(int,input().split())
    d[u][v]=d[v][u]=w
a=list(map(int,input().split()))
for i in range(n):
    d[i][i]=0
for k in range(n):
    for i in range(n):
        for j in range(n):
            if d[i][k]+d[k][j]<d[i][j]:
                d[i][j]=d[i][k]+d[k][j]
a.sort()
index=-1
s=float('inf')
for x in a:
    if sum(d[x])<s:
        s=sum(d[x])
        index=x
print(index,s)
```

9.最小生成树

prim算法，贪心加边

```
import heapq
n=int(input())
a=[[] for _ in range(n)]
for i in range(n-1):
    s=input().split()
    for j in range(2,len(s),2):
        a[i].append((int(s[j+1]),ord(s[j])-ord('A')))
        a[ord(s[j])-ord('A')].append((int(s[j+1]),i))
num=0
q=[(0,0)]
```

```
vis=[False]*n
m=0
while q and m<n:
    w,u=heapq.heappop(q)
    if vis[u]:continue
    vis[u]=True
    m+=1
    num+=w
    for x,y in a[u]:
        if not vis[y]:
            heapq.heappush(q,(x,y))
print(num)
```

krusal算法(这题是瞎破坏，破坏的最多就是留下的边权最小，转为最小生成树)这个熟知，不写了。

10.强连通分量

1. 第一次DFS：遍历整个图，记录每个节点的完成时间，并将节点按照完成时间排序后压入栈中。 2. 图的转置：将原图中的边反转，得到转置图。 3. 第二次DFS：按照栈中节点的顺序，对转置图进行DFS，从而找到强连通分量。 最后，输出找到的强连通分量。

```
import sys
sys.setrecursionlimit(10**7)
n,m=map(int,input().split())
a=[[] for _ in range(n+1)]
b=[[] for _ in range(n+1)]
for _ in range(m):
    u,v=map(int,input().split())
    a[u].append(v)
    b[v].append(u)
stack=[]
visited1=[False]*(n+1)

def dfs1(pos):
    for x in a[pos]:
        if not visited1[x]:
            visited1[x]=True
            dfs1(x)
    stack.append(pos)
def dfs2(pos):
    for x in b[pos]:
        if not visited2[x]:
            visited2[x]=True
            dfs2(x)
ans=0
for i in range(1,n+1):
    if not visited1[i]:
        visited1[i]=True
        dfs1(i)
visited2=[False]*(n+1)
while stack:
    pos=stack.pop()
    if not visited2[pos]:
        visited2[pos]=True
        dfs2(pos)
```

```
        ans+=1
print(ans)
```

11.Floyd算法，只存边。货币套现

```
n,m,s,b=map(float,input().split())
n,m,s=int(n),int(m),int(s)
a=[]
for _ in range(m):
    u,v,r1,c1,r2,c2=map(float,input().split())
    u=int(u)
    v=int(v)
    a.append((u,v,r1,c1))
    a.append((v,u,r2,c2))
amount=[0]*(n+1)
amount[s]=b
for _ in range(n):
    is_update=False
    for u,v,rate,com in a:
        temp=(amount[u]-com)*rate
        if temp>amount[v]:
            amount[v]=temp
            is_update=True
    if not is_update:
        break
for u,v,rate,com in a:
    temp=(amount[u]-com)*rate
    if temp>amount[v]:
        print('YES')
        exit()
print('YES' if amount[s]>b else 'NO')
```