

求逆序对:

```
c=0
def f(t):
    global c
    if len(t)==1:
        return t
    if len(t)==2:
        if t[0]>t[1]:
            c+=1
            return [t[1],t[0]]
        else:
            return t
    x=len(t)//2
    res=[]
    a,b=f(t[:x+1]),f(t[x+1:])
    if b[0]>=a[-1]:
        return a+b
    if a[0]>b[-1]:
        c+=len(a)*len(b)
        return b+a
    if a[0]==b[-1]:
        i=0
        while i<len(a) and a[i]==a[0]:
            i+=1
        j=-1
        while j>=-len(b) and b[j]==b[-1]:
            j-=1
        o=i*(len(b)+j+1)+(len(a)-i)*len(b)
        c+=o
        return b+a
    i,j=0,0
    while i<x+1 and j<len(t)-x-1:
        if a[i]>b[j]:
            res.append(b[j])
            j+=1
        else:
            c+=j
            res.append(a[i])
            i+=1
    if i==x+1:
        res.extend(b[j:])
    else:
        c+=(x+1-i)*len(b)
        res.extend(a[i:])
    return res
n=int(input())
l=[int(input()) for _ in range(n)]
l=list(reversed(l))
f(l)
print(c)
```

## 栈（2个经典题）

#22068: 合法出栈序列

```
from collections import deque
x=input()
while 1:
    try:
        s=input()
        i=0
        if len(x)!=len(s):
            print('NO')
            continue
        a=[];q=deque(s)
        i=0
        while 1:
            try:
                if a and a[-1]==q[0]:
                    a.pop()
                    q.popleft()
                else:
                    a.append(x[i])
                    i+=1
            except:
                break
        if a: print('NO') else: print('YES')
    except EOFError:
        break
```

#04077: 出栈序列统计（卡特兰数）

```
from math import comb
n=int(input())
print(int(comb(2*n, n)/(n+1)))
```

**单调栈（每日温度）：**单调递减栈求的是一个元素右侧第一个比它大的和左侧第一个比它大的，单调递增栈求的是一个元素右侧第一个比它小的和左侧第一个比它小的（都是从左往右遍历）

**5:每日温度**

总时间限制：1000ms    内存限制：65536kB

**描述**

现在有一组数据表示一段时间内每日的温度。你要做的是统计这段时间内，每一天起的下一个更高的温度出现在几天后。如果气温在这一天之后都不会升高，请在该位置用 0 来代替

**输入**

输入包括两行  
第一行是一个整数 n，表示天数（n 不超过 500000）  
第二行是一行整数，表示这段时间内每天的温度

**输出**

输出一行整数，每个整数表示对应的日子下一个更高温度出现在几天后

**样例输入**

```
4
73 74 72 77
```

**样例输出**

```
1 2 1 0
```

**提示**

暴力解法存在超时的可能，可以考虑使用栈结构来解决

**思路：**也就是说针对每只牛，啊，不，每头牛求它到它右边第一个比它高（或身高相等）的牛之间有多少头牛，然后将求得的结果相加就是最后的答案。朴素的做法是针对每头牛去寻找右边比它高的牛的位置，时间复杂度为 $O(n^2)$ ，如果用单调栈的话就是 $O(n)$ 。

**具体实现：**利用根据单调递增栈解决，如果栈为空或入栈元素小于栈顶元素则入栈，否则会破坏栈的单调性，则将栈顶元素出栈并更新结果，直到栈为空或碰到一个小于入栈元素的值。然后将当前元素入栈。

设数组的最后一个元素为最大值，也就相当于在最右边的牛的右边设了一个高度无限高的牛。这样做的目的是，最后让栈内的元素全部出栈。

**PS：**我的代码中的单调栈保存的是牛的位置。

```
#include <iostream>
#include<queue>
#include<string>
#include<vector>
using namespace std;

int main(){
```

```

int n;
cin>>n;
int l[500002];
int ans[500002];
vector<int> stack;
for(int i=0;i<n;i++){
    cin>>l[i];
}
for(int i=n-1;i>=0;i--){
    while((!stack.empty()) && l[stack[stack.size()-1]]<=l[i]){
        stack.pop_back();
    }
    if(stack.empty()){ans[i]=0;}
    else{
        ans[i]=stack[stack.size()-1]-i;
    }
    stack.push_back(i);
}
for(int i=0;i<n;i++){
    cout<<ans[i]<<' ';
}
}

```

#### #27205 护林员盖房子（经典）

```

m,n=map(int,input().split())
l=[list(map(int,input().split())) for _ in range(m)]
dp=[[0]*(n+1) for _ in range(m)]
for i in range(m):
    for j in range(n):
        if l[i][j]:
            dp[i][j]=0
        else:
            dp[i][j]=dp[i-1][j]+1
ans=0
for i in range(m):
    fl=[[0,0] for _ in range(n+1)]
    stack=[]
    for j in range(n+1):
        ans=max(ans,dp[i][j])
        while stack and dp[i][j]<dp[i][stack[-1]]:
            a=stack.pop()
            fl[a][1]=j
        if stack:
            fl[j][0]=stack[-1]
        else:
            fl[j][0]=-1
        stack.append(j)
    for j in range(n):
        ans=max(ans,(fl[j][1]-fl[j][0]-1)*dp[i][j])
print(ans)

```

## 调度场算法:

```
#中序转后序
d={'+':1,'-':1,'*':2,'/':2}
for _ in range(int(input())):
    s=input()
    l=[]
    j=0
    for i in range(len(s)):
        if s[i] in '+-*/()':
            l.append((s[j:i]))
            l.append(s[i])
            j=i+1
    l.append((s[j:]))
    ans=[]
    op=[]
    for u in l:
        if u==' ' or u==' ':
            continue
        if u not in '+-*/()':
            ans.append(u)
        elif u=='(':
            op.append(u)
        elif u==')':
            while op:
                x=op.pop()
                if x=='(':
                    break
            ans.append(x)
        else:
            while op and op[-1]!='(' and d[u]<=d[op[-1]]:
                x=op.pop()
                ans.append(x)
            op.append(u)
    while op:
        ans.append(op.pop())
    print(*ans)
```

```
#布尔表达式
def ShuntingYard(l:list):
    stack,output=[],[]
    for i in l:
        if i==" ":continue
        if i in 'VF':output.append(i)
        elif i=='(':stack.append(i)
        elif i in '&|!':
            while True:
                if i=='!':break
                elif not stack:break
                elif stack[-1]=="(":
                    break
            else:output.append(stack.pop())
        stack.append(i)
    elif i==')':
```

```

        while stack[-1]!='(':
            output.append(stack.pop())
            stack.pop()
        if stack:output.extend(reversed(stack))
        return output

def Bool_shift(a):
    if a=='V':return True
    elif a=='F':return False
    elif a==True:return 'V'
    elif a==False:return 'F'

def cal(a,operate,b=None):
    if operate=='&':return Bool_shift(Bool_shift(a) and Bool_shift(b))
    if operate=='|':return Bool_shift(Bool_shift(a) or Bool_shift(b))
    if operate=='!':return Bool_shift(not Bool_shift(a))

def post_cal(l:list):
    stack=[]
    for i in l:
        if i in 'VF':stack.append(i)
        elif i in "&|!":
            if i=="!":
                stack.append(cal(stack.pop(),'!'))
            else:
                a,b=stack.pop(),stack.pop()
                stack.append(cal(a,i,b))
    return stack[0]

while True:
    try:print(post_cal(ShuntingYard(list(input()))))
    except EOFError:break

```

## 堆:

### 1.实现堆结构

```

l=[0]
for _ in range(int(input())):
    s=input()
    if s[0]=='1':
        a,b=map(int,s.split())
        l.append(b)
        t=len(l)-1
        while t!=1 and l[t]<l[t//2]:
            l[t],l[t//2]=l[t//2],l[t]
            t=t//2
    else:
        print(l[1])
        x=l.pop()
        if len(l)>1:
            l[1]=x
            t=1

```

```

while (2*t<len(l) and l[t]>l[2*t]) or (2*t+1<len(l) and l[t]>l[2*t+1]):
    if (2*t<len(l) and l[t]>l[2*t]) and (2*t+1<len(l) and l[t]>l[2*t+1]):
        if l[2*t]<l[2*t+1]:
            l[t],l[2*t]=l[2*t],l[t]
            t=2*t
        else:
            l[t],l[2*t+1]=l[2*t+1],l[t]
            t=2*t+1
    elif (2*t<len(l) and l[t]>l[2*t]):
        l[t],l[2*t]=l[2*t],l[t]
        t=2*t
    elif (2*t+1<len(l) and l[t]>l[2*t+1]):
        l[t],l[2*t+1]=l[2*t+1],l[t]
        t=2*t+1

```

## 2.一个堆题 洛谷P6044

### 分析题意

#### 简要题意

即有  $n$  条线段，选出  $k$  条线段，使得他们的交集长度最大并输出任意一种方案。

易知最终得到的区间是**某一线段的左端点与另一线段的右端点**。

### 解题思路

#### 贪心

若先判断左端点，那将左端点按值从小到大排序，然后从前遍历即可。

#### 优先队列

剩下只用判断右端点，需要维护  $k$  条线段的右端点值，那么用优先队列，可以比较方便的得出。针对该题，我们使用小根堆。

#### 如何获得答案

维护合法右端点减去左端点的最大值，最后输出即可。

对于线段的编号，需要顺带维护最大长度的左端点与右端点，记为  $l$  和  $r$ 。我们重新遍历一边，找到左端点小于等于  $l$  和右端点大于等于  $r$  的线段，输出即可。

#### 代码

### 并查集：

#### 1.反集：（用于：

- 一个人的朋友的朋友是朋友
  - 一个人的敌人的敌人是朋友
- )

```

#include<iostream>
using namespace std;
int a[2010];
int n, m;

```

```

int findd(int x) { // 查找模板
    return x == a[x] ? x : a[x]=findd(a[x]);
}
void merge(int x, int y) { // 并查集模板
    x = findd(x), y = findd(y);
    if(x != y) a[x] = y;
}
int main() {
    cin >> n >> m;
    for(int i = 1; i <= 2*n; ++i) // 2倍数据
        a[i] = i;
    int x, y;
    char c;
    for(int i = 0; i != m; ++i) {
        cin >> c >> x >> y;
        if(c == 'F') {
            merge(x, y); // 普通合并
        } else {
            // 不能直接合并
            merge(x+n, y); // 先用反集分别保存一下
            merge(y+n, x);
            //如果a和b是敌人，合并n+b和a，n+a和b
            //如果c和a是敌人，合并n+c和a，n+a和c
            //那么b和c就并在一起了

            // 注意这里的合并祖先是 在 1~n
            // 如果写成merge(x, y+n) 祖先就在n~2*n里面了
            // 因为merge里面 a[x] = y
        }
    }
    int ans = 0;
    for(int i = 1; i <= n; ++i)
        if(a[i] == i) ans++;
    cout << ans << endl;
    return 0;
}

```

## 2.带权并查集（注意更新方式，尤其是下面第4,5行不能交换）

```

def f(x):
    y=0
    while x!=dic[x][0]:
        y=(y+dic[x][1])%2
        x=dic[x][0]
        dic[x]=[dic[dic[x][0]][0],(dic[x][1]+dic[dic[x][0]][1])%2]
    return x,y
n=int(input())
dic={i:[i,0] for i in range(1,n+1)}
h=1
for i in range(1,n+1):
    a,b=map(int,input().split())
    a1,a2=f(a)
    i1,i2=f(i)
    if a1==i1 and ((b==1 and a2!=i2) or (b==0 and a2==i2)):
        h=0

```



```

    if (b==0 and a2!=i2) or (b==1 and a2==i2):
        dic[a1]=[i1,0]
    else:
        dic[a1]=[i1,1]
res={}
for i in range(1,n+1):
    a,b=f(i)
    if a not in res:
        res[a]=[0,0]
        res[a][b]+=1
    else:
        res[a][b]+=1
if h==0:
    print('No answer')
    exit()
print(2**len(res))
a,b=0,0
for u in res:
    a+=max(res[u])
    b+=min(res[u])
print(a)
print(b)

```

### 3.并查集求最小环

#### 题目描述

有  $n$  个同学（编号为 1 到  $n$ ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为  $i$  的同学的信息传递对象是编号为  $T_i$  的同学。

游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息，但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮？

#### 输入格式

输入共 2 行。

第一行包含 1 个正整数  $n$ ，表示  $n$  个人。

第二行包含  $n$  个用空格隔开的正整数  $T_1, T_2, \dots, T_n$ ，其中第  $i$  个整数  $T_i$  表示编号为  $i$  的同学的信息传递对象是编号为  $T_i$  的同学， $T_i \leq n$  且  $T_i \neq i$ 。

#### 输出格式

共一行一个整数，表示游戏一共可以进行多少轮。

把每个同学看成一个点，信息的传递就是在他们之间连有向边，游戏轮数就是求最小环。

图论求最小环，我在里面看到了并查集。

假如说信息由A传递给B，那么就连一条由A指向B的边，同时更新A的父节点，A到它的父节点的路径长也就是B到它的父节点的路径长+1。

这样我们就建立好了一个图，之后信息传递的所有环节都按照这些路径。游戏结束的轮数，也就是这个图里最小环的长度。

如果有两个点祖先节点相同，那么就可以构成一个环，长度为两个点到祖先节点长度之和+1

```
#include<cstdio>
#include<iostream>
using namespace std;
int f[200002],d[200002],n,minn,last;    //f保存祖先节点，d保存点到其祖先节点的路径长。
int fa(int x)
{
    if (f[x]!=x)                        //查找时沿途更新祖先节点和路径长。
    {
        int last=f[x];                //记录父节点（会在递归中被更新）。
        f[x]=fa(f[x]);                //更新祖先节点。
        d[x]+=d[last];                //更新路径长（原来连在父节点上）。
    }
    return f[x];
}
void check(int a,int b)
{
    int x=fa(a),y=fa(b);               //查找祖先节点。
    if (x!=y) {f[x]=y; d[a]=d[b]+1;}    //若不相连，则连接两点，更新父节点和路径长。
    else minn=min(minn,d[a]+d[b]+1);    //若已连接，则更新最小环长度。
    return;
}
int main()
{
    int i,t;
    scanf("%d",&n);
    for (i=1;i<=n;i++) f[i]=i;          //祖先节点初始化为自己，路径长为0。
    minn=0x7777777;
    for (i=1;i<=n;i++)
    {
        scanf("%d",&t);
        check(i,t);                    //检查当前两点是否已有边相连接。
    }
    printf("%d",minn);
    return 0;
}
```

## 树

建树：注意节点名字可能会重复

```
#扩展二叉树（完全二叉树的先序遍历）
from collections import defaultdict
i=1
def f(a):
    global i
    child[a].append(s[i])
    if s[i]=='.':
        i+=1
    return
t=s[i]
i+=1
```

```

f(t)
f(t)
def pre(a):
def post(a):
s=input()
child=defaultdict(list)
f(s[0])
f(s[0])
print(pre(s[0]))
print(post(s[0]))

```

#文本二叉树（用栈解决，到叶子节点回溯到上一个可插入的节点）

```

from collections import defaultdict
dic=defaultdict(list)
l=[]
root=''
for _ in range(int(input())):
    while 1:
        s=input()
        if s=='0':
            break
        if not root:
            root=s[-1]
        if not l:
            l.append([s[-1],len(s)])
        else:
            if len(s)==l[-1][1]+1:
                dic[l[-1][0]].append(s[-1])
                l.append([s[-1],len(s)])
            else:
                while len(s)!=l[-1][1]+1:
                    l.pop()
                dic[l[-1][0]].append(s[-1])
                l.append([s[-1],len(s)])

```

## 1.树状数组

#实现是很容易掌握的：注意几点：1.数组大小是+1 2.不要操作索引为0的情况，否则会陷入死循环 3.求区间和利用了前缀和思想

```

class FenwickTree:
    def __init__(self, size):
        self.size = size
        self.tree = [0] * (size + 1)

    def update(self, index, value):
        while index <= self.size:
            self.tree[index] += value
            index += index & -index

    def query(self, index):
        result = 0
        while index > 0:
            result += self.tree[index]
            index -= index & -index

```

```
return result
```

```
def range_query(self, start, end):  
    return self.query(end) - self.query(start - 1)
```

## 2.关于3种遍历

### 1.

先序和后序不能唯一确定树的形状（找只有一个子树的节点数量即可）



青丝、暮成雪 创建时间：2016-10-25 11:05:17

[查看文章](#)

只有一个儿子的节点 才会在知道 前序后序 的情况下有不同的中序遍历，所以将题目转化成找 只有一个儿子的节点个数。

可以很容易的找出这类节点在前序后序中出现的规律。（前序中出现AB，后序中出现BA，则这个节点只有一个儿子）

每个这类节点有两种中序遍历（及儿子在左，儿子在右）根据乘法原理中序遍历数为  $2^{\text{节点个数}}$  种

```
#include<stdio>  
#include<cstring>  
#include<cmath>  
#include<algorithm>  
using namespace std;  
int ans;  
char str1[233],str2[233];  
int main()  
{  
    scanf("%s",str1);  
    scanf("%s",str2);  
    for(int i=0;i<strlen(str1);i++)  
        for(int j=1;j<strlen(str2);j++)  
            if(str1[i]==str2[j]&&str1[i+1]==str2[j-1])  
                ans++;  
    printf("%d",1<<ans);  
    return 0;  
}
```

👍 337    💬 146 条评论

[收起](#) ^

#薛定谔的二叉树

```
def g(a,b):  
    if a[0]!=b[-1]:  
        return 0  
    if len(set(a))!=len(a) or len(set(b))!=len(b):  
        return 0  
    if len(a)!=len(b):  
        return 0  
    for u in a:  
        if u not in b:  
            return 0  
    return 1  
def f(a,b):  
    c=-1  
    if not g(a, b):#检查是否合法
```

```

        return -1
    if len(a)==1:
        return 0
    i=0
    while i<len(a) and a[i]==b[-1-i]:#排去前面完全相同的部分
        i+=1
        c+=1#因为只有一义的话可以在左也可以在右
    if i==len(a):
        return c
    if i>1:
        x=f(a[i-1:],b[:1-i])
        if x==-1:
            return -1
        return c+x
    u=0
    for t in range(i,len(a)):
        if g(a[i:t+1],b[:t+1-i]) and g(a[t+1:],b[t+1-i:-1]):
            u=1
            break
    if not u:
        return -1
    x=f(a[i:t+1],b[:t+1-i])
    y=f(a[t+1:],b[t+1-i:-1])
    if x==-1 or y==-1: return -1
    c+=x+y
    return c
while 1:
    try:
        a,b=input().split()
        k=f(a, b)
        if k==-1:
            print(0)
        else:
            print(2**k)
    except:
        break

```

## 2.前中序找后序 | 中后序找前序 ： 简单递归

```

#22158 根据二叉树前中序序列建树
def f(a,b):
    if len(a)==0:
        return ''
    if len(a)==1:
        return a
    else:
        x=a[0]
        i=b.find(x)
        return f(a[1:i+1],b[:i])+f(a[i+1:],b[i+1:])+x
while 1:
    try:
        a=input()
        b=input()
        print(f(a, b))
    except:

```

```
break
```

### 3.一些递归dp题（计算可能种类）

1.输入 $n(1 < n < 13)$ ，求 $n$ 个结点的二叉树有多少种形态

```
#P0720 其实是卡特兰数
l=[1,1,2]
for i in range(10):
    t=i+3
    res=0
    for j in range(t):
        res+=l[j]*l[t-1-j]
    l.append(res)
n=int(input())
print(l[n])
```

2.avl树

```
#27625 输入 $n(0 < n < 50)$ ，输出一个 $n$ 层的AVL树至少有多少个结点
l=[1,2]
for i in range(50):
    l.append(l[-1]+l[-2]+1)
print(l[int(input())-1])
```

```
#27626  $n$ 个结点的AVL树最多有多少层？
l=[1,2]
for i in range(100):
    l.append(l[-1]+l[-2]+1)
n=int(input())
for i in range(100):
    if l[i]<=n<l[i+1]:
        print(i+1)
        break
```

### 4.关于表达式

```
#24591:中序表达式转后序表达式
1.树写法
class Node:
    def __init__(self,val,l,r):
        self.val=val
        self.left=l
        self.right=r
def dfs(x):
    try:
        x.left
    except:
        return x
    if x.val in {'*','/','+','-'}:
        if type(x.left)==str:
            r=x.left+' '
        else:
```

```

        r=dfs(x.left)+' '
        if type(x.right)==str:
            r+=x.right
        else:
            r+=dfs(x.right)
        return r+' '+x.val
    else:
        return dfs(x.left)+' '+dfs(x.right)+' '+x.val
def f(t):
    if '(' not in t:
        while ('*' in t or '/' in t):
            for i in range(len(t)):
                u=t[i]
                if u=='*' or u=='/':
                    s=Node(u, t[i-1], t[i+1])
                    t=t[:i-1]+s+t[i+2:]
                    break
            if len(t)==1:
                return t[0]
        p=0;i=0
        while i<len(t):
            u=t[i]
            if u=='+':
                p=Node('+', p, t[i+1])
                i+=2
                if i==len(t):
                    return p
            elif u=='-':
                p=Node('-', p, t[i+1])
                i+=2
                if i==len(t):
                    return p
            else:
                p=u
                i+=1
        else:
            a=t.index(')')
            for i in range(a-1,-1,-1):
                if t[i]=='(':
                    break
            return f(t[:i]+[f(t[i+1:a])]+t[a+1:])
for _ in range(int(input())):
    s=input()
    t=[]
    i=0
    for uu in range(len(s)):
        u=s[uu]
        if u in {'+', '*', '(', ')', '-', '/'}:
            t.append(s[i:uu])
            t.append(u)
            i=uu+1
    t.append(s[i:])
    o=[]
    for u in t:
        if u:
            o.append(u)

```

```
k=f(o)
print(dfs(k))
2. 调度场算法在前面
```

#### #24588: 后序表达式求值

```
for _ in range(int(input())):
    l=list(input().split())
    s=[]
    for u in l:
        try:
            float(u)
            s.append(float(u))
        except:
            b=float(s.pop())
            a=float(s.pop())
            if u=='+':
                s.append(a+b)
            elif u=='-':
                s.append(a-b)
            elif u=='*':
                s.append(a*b)
            else:
                s.append(a/b)
    print('%.2f'%s[0])
```

#### #02694: 波兰表达式

描述

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。

```
def f(x):
    try:
        float(x)
        return 1
    except:
        return 0
l=list(input().split())
stack=[]
for u in l[::-1]:
    if f(u):
        stack.append(u)
    else:
        a=float(stack.pop())
        b=float(stack.pop())
        if u=='+':
            stack.append(a+b)
        if u=='-':
            stack.append(a-b)
        if u=='*':
            stack.append(a*b)
        if u=='/':
            stack.append(a/b)
print('%.6f'%stack[0])
```

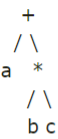


## 5.一些需要建树的题（递归写法，写Node的太麻烦仅作练习）

```
#扩展二叉树
from collections import defaultdict
i=1
def f(a):
    global i
    child[a].append(s[i])
    if s[i]=='.':
        i+=1
        return
    t=s[i]
    i+=1
    f(t)
    f(t)
def g(a):
    if a=='.':
        return ''
    s=g(child[a][0])
    s+=a
    s+=g(child[a][1])
    return s
def h(a):
    if a=='.':
        return ''
    s=h(child[a][0])
    s+=h(child[a][1])+a
    return s
s=input()
child=defaultdict(list)
f(s[0])
f(s[0])
print(g(s[0]))
print(h(s[0]))
```

描述

众所周知，任何一个表达式，都可以用一棵表达式树来表示。例如，表达式`a+b*c`，可以表示为如下的表达式树：



现在，给你一个中缀表达式，这个中缀表达式用变量来表示（不含数字），请你将这个中缀表达式用表达式二叉树的形式输出出来。

输入

输入分为三个部分。  
第一部分为一行，即中缀表达式(长度不大于50)。中缀表达式可能含有小写字母代表变量（a-z），也可能含有运算符（+、-、\*、/、小括号），不含有数字，也不含有空格。  
第二部分为一个整数n(n < 10)，表示中缀表达式的变量数。  
第三部分有n行，每行格式为C x，C为变量的字符，x为该变量的值。

输出

输出分为三个部分，第一个部分为该表达式的逆波兰式，即该表达式树的后根遍历结果。占一行。  
第二部分为表达式树的显示，如样例输出所示。如果该二叉树是一棵满二叉树，则最底部的叶子结点，分别占据横坐标的第1、3、5、7.....个位置（最左边的坐标是1），然后它们的父结点的横坐标，在两个子结点的中间。如果不是满二叉树，则没有结点的地方，用空格填充（但请略去所有的行末空格）。每一行父结点与子结点中隔开一行，用斜杠 (/) 与反斜杠 (\) 来表示树的关系。/出现的横坐标位置为父结点的横坐标偏左一格，\出现的横坐标位置为父结点的横坐标偏右一格。也就是说，如果树高为m，则输出就有2m-1行。  
第三部分为一个整数，表示将值代入变量之后，该中缀表达式的值。需要注意的一点是，除法代表整除运算，即舍弃小数点后的部分。同时，测试数据保证不会出现除以0的现象。

样例输入

```
a+b*c
3
a 2
b 7
c 5
```

样例输出

```
abc*+
+
/\
a  *
  /\
  b c
37
```

'''  
表达式树是一种特殊的二叉树。对于你的问题，需要先将中缀表达式转换为后缀表达式（逆波兰式），然后根据后缀表达式建立表达式树，最后进行计算。

首先使用stack进行中缀到后缀的转换，然后根据后缀表达式建立表达式二叉树，再通过递归和映射获取表达式的值。  
最后，打印出整棵树

```
'''
#from collections import deque as q
import operator as op
#import os

class Node:
    def __init__(self, x):
        self.value = x
        self.left = None
```

```

        self.right = None

def priority(x):
    if x == '*' or x == '/':
        return 2
    if x == '+' or x == '-':
        return 1
    return 0

def infix_trans(infix):
    postfix = []
    op_stack = []
    for char in infix:
        if char.isalpha():
            postfix.append(char)
        else:
            if char == '(':
                op_stack.append(char)
            elif char == ')':
                while op_stack and op_stack[-1] != '(':
                    postfix.append(op_stack.pop())
                op_stack.pop()
            else:
                while op_stack and priority(op_stack[-1]) >= priority(char) and
op_stack[-1] != '(':
                    postfix.append(op_stack.pop())
                op_stack.append(char)
    while op_stack:
        postfix.append(op_stack.pop())
    return postfix

def build_tree(postfix):
    stack = []
    for item in postfix:
        if item in '+-*/':
            node = Node(item)
            node.right = stack.pop()
            node.left = stack.pop()
        else:
            node = Node(item)
            stack.append(node)
    return stack[0]

def get_val(expr_tree, var_vals):
    if expr_tree.value in '+-*/':
        operator = {'+': op.add, '-': op.sub, '*': op.mul, '/': op.floordiv}
        return operator[expr_tree.value](get_val(expr_tree.left, var_vals),
get_val(expr_tree.right, var_vals))
    else:
        return var_vals[expr_tree.value]

```

# 计算表达式树的深度。它通过递归地计算左右子树的深度，并取两者中的最大值再加1，得到整个表达式树的深度。

```
def getDepth(tree_root):  
    #return max([self.child[i].getDepth() if self.child[i] else 0 for i in  
range(2)]) + 1  
    left_depth = getDepth(tree_root.left) if tree_root.left else 0  
    right_depth = getDepth(tree_root.right) if tree_root.right else 0  
    return max(left_depth, right_depth) + 1
```

'''

首先，根据表达式树的值和深度信息构建第一行，然后构建第二行，该行包含斜线和反斜线，用于表示子树的链接关系。接下来，如果当前深度为0，表示已经遍历到叶子节点，直接返回该节点的值。否则，递减深度并分别获取左子树和右子树的打印结果。最后，将左子树和右子树的每一行拼接在一起，形成完整的树形打印图。

打印表达式树的函数。表达式树是一种抽象数据结构，它通过树的形式来表示数学表达式。在这段程序中，函数printExpressionTree接受两个参数：tree\_root表示树的根节点，d表示树的总深度。

首先，函数会创建一个列表graph，列表中的每个元素代表树的一行。第一行包含根节点的值，并使用空格填充左右两边以保持树的形状。第二行显示左右子树的链接情况，使用斜杠/表示有左子树，反斜杠\表示有右子树，空格表示没有子树。

接下来，函数会判断深度d是否为0，若为0则表示已经达到树的最底层，直接返回根节点的值。否则，将深度减1，然后递归调用printExpressionTree函数打印左子树和右子树，并将结果分别存储在left和right中。

最后，函数通过循环遍历2倍深度加1次，将左子树和右子树的每一行连接起来，存储在graph中。最后返回graph，即可得到打印好的表达式树。

'''

```
def printExpressionTree(tree_root, d): # d means total depth  
  
    graph = [" "*(2**d-1) + tree_root.value + " "*(2**d-1)]  
    graph.append(" "*(2**d-2) + ("/" if tree_root.left else " ")  
                + " " + ("\" if tree_root.right else " ") + " "*(2**d-2))  
  
    if d == 0:  
        return tree_root.value  
    d -= 1  
    '''  
  
    应该是因为深度每增加一层，打印宽度就增加一倍，打印行数增加两行  
    '''  
  
    #left = printExpressionTree(tree_root.left, d) if tree_root.left else [  
    #     " "*(2**(d+1)-1)]*(2*d+1)  
    if tree_root.left:  
        left = printExpressionTree(tree_root.left, d)  
    else:  
        #print("left_d",d)  
        left = [" "*(2**(d+1)-1)]*(2*d+1)  
        #print("left_left",left)  
  
    right = printExpressionTree(tree_root.right, d) if tree_root.right else [  
        " "*(2**(d+1)-1)]*(2*d+1)  
  
    for i in range(2*d+1):  
        graph.append(left[i] + " " + right[i])
```

```

        #print('graph=',graph)
        return graph

infix = input().strip()
n = int(input())
vars_vals = {}
for i in range(n):
    line = input().split()
    vars_vals[line[0]] = int(line[1])

'''
infix = "a+(b-c*d*e)"
#infix = "a+b*c"
n = 5
vars_vals = {'a': 2, 'b': 7, 'c': 5, 'd':1, 'e':1}
'''

postfix = infix_trans(infix)
tree_root = build_tree(postfix)
print(''.join(str(x) for x in postfix))
expression_value = get_val(tree_root, vars_vals)

for line in printExpressionTree(tree_root, getDepth(tree_root)-1):
    print(line.rstrip())

print(expression_value)

```

#25140:根据后序表达式建立表达式树

#做法就是构建栈，然后不断地将计算的步骤转化成构建的新节点，最后返回根节点即可

```

class Node:
    def __init__(self, val, l, r):
        self.val = val
        self.left = l
        self.right = r

class Tree:
    def create(self, s):
        stack = []
        for u in s:
            if u.isupper():
                a = stack.pop()
                b = stack.pop()
                t = Node(u, b, a)
                stack.append(t)
            else:
                stack.append(u)
        return stack[0]

    def dfs(self, x, layer):
        if type(x) != Node:
            al[layer].append(x)
            return
        al[layer].append(x.val)
        self.dfs(x.left, layer+1)
        self.dfs(x.right, layer+1)

for _ in range(int(input())):
    s = input()

```

```

t=Tree()
x=t.create(s)
a1=[] for _ in range(100)]
t.dfs(x,0)
ans=[]
for u in a1:
    ans.extend(u)
print(*reversed(ans),sep='')

```

#P0680:括号嵌套二叉树

```

class Node():
    def __init__(self,v,p):
        self.val=v
        self.left=None
        self.right=None
        self.par=p
class Tree:
    def __init__(self,v):
        self.root=Node(v,None)
    def struct(self,s):
        cur=self.root
        r=len(s)
        for i in range(1,r):
            if s[i].isalpha() or s[i]=='*':
                if s[i-1]=='(':
                    cur.left=Node(s[i], cur)
                    if s[i+1]=='(':
                        cur=cur.left
                elif s[i-1]==',':
                    cur.right=Node(s[i], cur)
                    if s[i+1]=='(':
                        cur=cur.right
            if s[i]==')':
                cur=cur.par
    def a(self,x):
        if x==None or x.val=='*':
            return ''
        else:
            return x.val+self.a(x.left)+self.a(x.right)
    def b(self,x):
        if x==None or x.val=='*':
            return ''
        else:
            return self.b(x.left)+x.val+self.b(x.right)

for _ in range(int(input())):
    s=input()
    t=Tree(s[0])
    t.struct(s)
    print(t.a(t.root))
    print(t.b(t.root))

```

#二叉搜索树

```

l=list(map(int,input().split()))

```

```

class Node:
    def __init__(self, val, left=None, right=None):
        self.val=val
        self.left=left
        self.right=right
class Tree:
    def __init__(self, root):
        self.root=Node(root)
    def insert(self, w):
        cur=self.root
        while 1:
            if w<cur.val:
                if cur.left==None:
                    cur.left=Node(w)
                    break
                else:
                    cur=cur.left
            elif w>cur.val:
                if cur.right==None:
                    cur.right=Node(w)
                    break
                else:
                    cur=cur.right
            else:
                break
    def predfs(self, a):
        if a==None:
            return []
        return [a.val]+self.preds(a.left)+self.preds(a.right)
s=Tree(1[0])
for u in 1[1:]:
    s.insert(u)
print(*s.preds(s.root))

```

#03720: 文本二叉树

```

class Node:
    def __init__(self, v, layer, par):
        self.val=v
        self.left=None
        self.right=None
        self.layer=layer
        self.parent=par
class Tree:
    def __init__(self, v, la):
        self.root=Node(v, la, None)
    def insert(self, l):
        cur=self.root
        for u in l[1:]:
            t=len(u)-1
            v=u[-1]
            if v=='*':
                h=Node(v, t, cur)
                cur.left=h
            elif t==cur.layer+1:
                if cur.left==None:

```

```

        cur.left=Node(v, t,cur)
        cur=cur.left
    else:
        cur.right=Node(v, t,cur)
        cur=cur.right
    else:
        while cur.layer!=t-1:
            cur=cur.parent
        if cur.left==None:
            cur.left=Node(v, t,cur)
            cur=cur.left
        else:
            cur.right=Node(v, t,cur)
            cur=cur.right

def dfs1(self,p):
    if p==None or p.val=='*':
        return ''
    return p.val+self.dfs1(p.left)+self.dfs1(p.right)
def dfs2(self,p):
    if p==None or p.val=='*':
        return ''
    return self.dfs2(p.left)+p.val+self.dfs2(p.right)
def dfs3(self,p):
    if p==None or p.val=='*':
        return ''
    return self.dfs3(p.left)+self.dfs3(p.right)+p.val

n=int(input())
for _ in range(n):
    l=[]
    while 1:
        x=input()
        if x=='0':
            break
        else:
            l.append(x)
    s=Tree(l[0],0)
    s.insert(l)
    print(s.dfs1(s.root))
    print(s.dfs3(s.root))
    print(s.dfs2(s.root))
    print()

```

## 5.字典树

```

class Node:
    def __init__(self):
        self.children = dict()
        self.isEnd = False
    # 字符节点
    # 初始化字符节点
    # 初始化子节点
    # isEnd 用于标记单词结束

class Trie:
    # 字典树
    # 初始化字典树
    def __init__(self):
        self.root = Node()
    # 初始化根节点（根节点不保存字符）

```



```

# 向字典树中插入一个单词
def insert(self, word: str) -> None:
    cur = self.root
    for ch in word:
        if ch not in cur.children:
            cur.children[ch] = Node()
        cur = cur.children[ch]
    cur.isEnd = True

# 查找字典树中是否存在一个单词
def search(self, word: str) -> bool:
    cur = self.root
    for ch in word:
        if ch not in cur.children:
            return False
        cur = cur.children[ch]
    return cur is not None and cur.isEnd

# 查找字典树中是否存在一个前缀
def startswith(self, prefix: str) -> bool:
    cur = self.root
    for ch in prefix:
        if ch not in cur.children:
            return False
        cur = cur.children[ch]
    return cur is not None

```

的节点  
 子节点  
 下一个字符  
 结束  
 的节点  
 查找下一个字符  
 结束标记  
 的节点  
 查找下一个字符  
 功

# 遍历单词中的字符  
 # 如果当前节点的子节点中，不存在键为 ch  
 # 建立一个节点，并将其保存到当前节点的  
 # 令当前节点指向新建立的节点，继续处理  
 # 单词处理完成时，将当前节点标记为单词  
 # 直接返回 False  
 # 令当前节点指向新建立的节点，然后继续  
 # 判断当前节点是否为空，并且是否有单词  
 # 直接返回 False  
 # 令当前节点指向新建立的节点，然后继续  
 # 判断当前节点是否为空，不为空则查找成

## 6.avl树

```

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)

```

```

        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        balance = self._get_balance(node)
        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else: # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)
        if balance < -1:
            if value > node.right.value: # 树形是 RR
                return self._rotate_left(node)
            else: # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

```

```

def preorder(self):
    return self._preorder(self.root)

def _preorder(self, node):
    if not node:
        return []
    return [node.value] + self._preorder(node.left) +
self._preorder(node.right)

n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder()))))

```

## 7.哈夫曼编码树

根据字符使用频率(权值)生成一棵唯一的哈夫曼编码树。生成树时需要遵循以下规则以确保唯一性:

选取最小的两个节点合并时，节点比大小的规则是:

- 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。
- 2) 合并两个节点时，小的节点必须作为左子节点
- 3) 连接左子节点的边代表0,连接右子节点的边代表1

然后对输入的串进行编码或解码

### 输入

第一行是整数n，表示字符集有n个字符。

接下来n行，每行是一个字符及其使用频率（权重）。字符都是英文字母。

再接下来是若干行，有的是字母串，有的是01编码串。

### 输出

对输入中的字母串，输出该字符串的编码

对输入中的01串,将其解码，输出原始字符串

### 样例输入

```

3
g 4
d 8
c 10
dc
110

```

### 样例输出

```

110
dc

```

```

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

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) #note: 合并后, char 字段默认值是空
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

```

```

        #if node.char:
            if node.left is None and node.right is None:
                decoded += node.char
                node = root
            return decoded

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

#string = input().strip()
#encoded_string = input().strip()

# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)

# 编码和解码
codes = encode_huffman_tree(huffman_tree)

strings = []
while True:
    try:
        line = input()
        strings.append(line)

    except EOFError:
        break

results = []
#print(strings)
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)

```



建图非常关键：1.单双向边 2.注意连接2点的道路如果超过1条，如何存储的问题，可以用字典或邻接矩阵存最小值，但是像road这样的题可能会时间更长但路费少，所以必须全部存下

## 1.dijkstra

```

#roads (1: 没有vis 2: 控制条件的k可能会陷入存一维的思维陷阱，但是这里只是加了判断，实际速度反而快了)
from collections import deque
import heapq
k=int(input())
n=int(input())

```

```

r=int(input())
p=[[list() for _ in range(102)] for _ in range(102)]
for _ in range(r):
    s,d,a,b=map(int,input().split())
    p[s][d].append((a,b))
t=[]
for i in range(2,n+1):
    for u in p[1][i]:
        heapq.heappush(t, (u[0],u[1],i))

ans=-1
while t:
    a,b,c=heapq.heappop(t)
    #print(a,b,c)
    if b>k:
        continue
    if c==n:
        print(a)
        exit()
    for i in range(1,n+1):
        for u in p[c][i]:
            heapq.heappush(t, (a+u[0],b+u[1],i))
print(ans)

```

## 2.最小生成树

prim算法：需要defaultdict建边，需要vis记录已访问，需要heapq把每次可扩展的边加入

kruskal算法：需要字典用来并查集，先开始记录了边不需要邻接矩阵，需要heapq（或排序好的边），不需要vis但需要记录已经加了多少条边（和最短路区别是第一个元素是边的权重，而最短路是到那个点的距离）

```

def prim(graph,start):
    pq = PriorityQueue()
    for vertex in graph:
        vertex.distance = sys.maxsize
        vertex.previous = None
    start.distance = 0
    pq.buildHeap([(v.distance,v) for v in graph])
    while pq:
        distance, current_v = pq.delete()
        for next_v in current_v.get_eighbors():
            new_distance = current_v.get_neighbor(next_v)
            if next_v in pq and new_distance < next_v.distance:
                next_v.previous = current_v
                next_v.distance = new_distance
                pq.change_priority(next_v,new_distance)
#
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices

    def find(self, x):

```

```

    if self.parent[x] != x:
        self.parent[x] = self.find(self.parent[x])
    return self.parent[x]

def union(self, x, y):
    root_x = self.find(x)
    root_y = self.find(y)

    if root_x != root_y:
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_x] = root_y
            self.rank[root_y] += 1

def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree

```

3.如走山路，鸣人等，都是多加一维的dp题。一般的bfs储存走过的vis其实是知道第一次到达某个点一定是最小值了，其他时候往往加入其他维度来控制，或者用dp存储已知最小步数

```

#20106 走山路
import heapq
m,n,p=map(int,input().split())
mp=[list(input().split()) for _ in range(m)]

```

```

for _ in range(p):
    a,b,c,d=map(int,input().split())
    l=[(0,a,b)]
    dp=[[1e9]*n for __ in range(m)]
    heapq.heapify(l)
    ans='NO'
    if mp[a][b]=='#' or mp[c][d]=='#':
        print(ans)
        continue
    while l:
        num,x,y=heapq.heappop(l)
        if x==c and y==d:
            ans=num
            break
        for u in [(x-1,y),(x+1,y),(x,y-1),(x,y+1)]:
            xx,yy=u
            if 0<=xx<m and 0<=yy<n and mp[xx][yy]!='#':
                t=abs(int(mp[xx][yy])-int(mp[x][y]))
                if num+t<dp[xx][yy]:
                    dp[xx][yy]=num+t
                    heapq.heappush(l, (num+t,xx,yy))
    print(ans)

```

#### #01376:Robot

```

from collections import deque
dic={'north':0,'west':1,'south':2,'east':-1}
dr={0:(-1,0),1:(0,-1),2:(1,0),-1:(0,1)}
while 1:
    m,n=map(int,input().split())
    if m==0:
        break
    l=[]
    for _ in range(m):
        t=list(map(int,input().split()))
        l.append(t)
    a,b,c,d,e=input().split()
    a,b,c,d=int(a),int(b),int(c),int(d)
    fl=[[1e9]*n for _ in range(m)] for __ in range(4)]
    fl[dic[e]][a][b]=0
    q=deque([(a,b,dic[e],0)])
    ans=1e9
    while q:
        a,b,e,f=q.popleft()
        if a==c and b==d:
            ans=min(ans,f)
        for u in range(1,4):
            aa,bb=a+u*dr[e][0],b+u*dr[e][1]
            if 1<=aa<m and 1<=bb<n and 1[aa][bb]==1[aa][bb-1]==1[aa-1][bb]==1[aa-1][bb-1]==0:
                if fl[e][aa][bb]>f+1:
                    fl[e][aa][bb]=f+1
                    q.append((aa,bb,e,f+1))
        else:
            break
    for u in range(-1,3):

```



```

        if u==e:
            continue
        if fl[u][a][b]>f+min(4-abs(u-e),abs(u-e)):
            fl[u][a][b]=f+min(4-abs(u-e),abs(u-e))
            q.append((a,b,u,f+min(4-abs(u-e),abs(u-e))))
    if ans==1e9:
        print(-1)
    else:
        print(ans)

```

```

//Saving Tang Monk
#include<iostream>
#include<queue>
#include<cstring>
#include<map>
using namespace std;
char maze[105][105];
int n,m;
struct node{
    int step,x,y,num,s;
    bool operator>(const node& other) const{
        return step>other.step;
    }
};
map<pair<int,int>,int> snake;
bool record[105][105][10];
int dirx[]={0,0,1,-1};
int diry[]={1,-1,0,0};
int main(){
    while(cin>>n>>m){
        if(n==0&&m==0) break;
        priority_queue<node,vector<node>,greater<node> > queue;
        memset(record,0,sizeof(record));
        node start={0,0,0,0,0};
        int cnt=0;
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                cin>>maze[i][j];
                if(maze[i][j]=='K'){
                    start={0,i,j,0,0};
                    queue.push(start);
                    record[i][j][0]=1;
                }
                else if(maze[i][j]=='S'){
                    snake[make_pair(i,j)]=cnt++;
                }
            }
        }
        bool flag=false;
        while(!queue.empty()){
            node t=queue.top();
            queue.pop();
            int x=t.x;int y=t.y;int step=t.step;int num=t.num;int s=t.s;
            if(maze[x][y]=='T'&&num==m){
                cout<<step<<endl;
            }

```

```

        flag=true;
        break;
    }
    for(int i=0;i<4;i++){
        int nx=x+dirx[i];int ny=y+diry[i];
        if(0<=nx&&nx<n&&0<=ny&&ny<n&&maze[nx][ny]!='#'){
            int tmp;
            if(maze[nx][ny]-'0'-num==1) tmp=maze[nx][ny]-'0';
            else tmp=num;
            if(record[nx][ny][tmp]==0){
                if(maze[nx][ny]=='S'){
                    if((s>>snake[make_pair(nx,ny)]&1)==0){
                        queue.push({step+2,nx,ny,tmp,s|
(1<<(snake[make_pair(nx,ny)]))});
                        record[nx][ny][tmp]=1;
                    }
                    else{
                        queue.push({step+1,nx,ny,tmp,s});
                        record[nx][ny][tmp]=1;
                    }
                }
                else{
                    queue.push({step+1,nx,ny,tmp,s});
                    record[nx][ny][tmp]=1;
                }
            }
        }
    }
}
if(!flag) cout<<"impossible"<<endl;
}
return 0;
}

```

5.拓扑排序：给出的是这些事件的逻辑顺序，所以在dp里为保证后效性，先用拓扑排序然后以此为顺序dp常见

```

from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

```

```

# 执行拓扑排序
while queue:
    u = queue.popleft()
    result.append(u)

    for v in graph[u]:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)

# 检查是否存在环
if len(result) == len(graph):
    return result
else:
    return None

# 示例调用代码
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}

sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")

```

```

#sorting it all out (判定是否可确定序列，看每次入度为0的节点是否为1)
import heapq
from collections import defaultdict
while 1:
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    dic={}
    already=1
    for u in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[:n]:
        dic[u]=0
    t=defaultdict(list)#记录小于的字母
    cycle=0
    sure=0
    al=set()
    for _ in range(m):
        s=input()
        if s in al:
            continue
        f=1#检验是否determined
        al.add(s)
        a,b=s[0],s[2]

```

```

t[a].append(b)
dic[b]+=1
l=[]
p=dic.copy()
vis=set()
res=[]
c=0
for u in p:
    if p[u]==0:
        heapq.heappush(l, u)
        vis.add(u)
        res.append(u)
        c+=1
if c>1:
    f=0
while l and len(vis)<n:
    a=heapq.heappop(l)
    vis.add(a)
    c=0
    for u in t[a]:
        p[u]-=1
        if u not in vis and p[u]==0:
            heapq.heappush(l, u)
            res.append(u)
            vis.add(u)
            c+=1
    if c>1:
        f=0
if already and len(vis)<n:
    cycle+=1
    print('Inconsistency found after %d relations.'%cycle)
    already=0
if already and f and cycle==0 and sure==0:
    sure+=1
    print('Sorted sequence determined after %d relations: %s.'%
(sure, ''.join(res)))
    already=0
if already:
    print('Sorted sequence cannot be determined.')

```

## 附录（其他算法内容）

### 2.辅助栈，维护min数组

```

//P0510:快速堆猪
int main() {
    string s;
    stack<int> a;
    vector<int> m;
    while (cin >> s) {
        if (s == "pop") {
            if (!a.empty()) {
                a.pop();
            }
        }
    }
}

```

```

        if (!m.empty()) m.pop_back();
    }
} else if (s == "min") {
    if (!m.empty()) cout << m.back() << endl;
} else {
    int h;
    cin >> h;
    a.push(h);
    if (m.empty()) m.push_back(h);
    else {
        int k = m.back();
        m.push_back(min(k, h));
    }
}
}
return 0;
}

```

#### 4.dp(最佳加法表达式)

```

#04152
while 1:
    try:
        m=int(input())
        s=input()
        t=len(s)
        if m==0:
            print(s)
            continue
        dp=[([0]+[float('inf')])*m for _ in range(t)]
        for i in range(t):
            for j in range(1,m+1):
                if j==1:
                    dp[i][j]=int(s[:i+1])
                    continue
                for h in range(i):
                    dp[i][j]=min(dp[i][j],dp[h][j-1]+int(s[h+1:i+1]))
        ans=float('inf')
        for i in range(t-1):
            ans=min(ans,dp[i][-1]+int(s[i+1:]))
        print(ans)
    except Exception as e:
        break

```

#### 5.dp (核电站)

```

#include<stdio.h>
int main(){
    int n,m,i;
    scanf("%d%d",&n,&m);
    long long dp[n+1];
    dp[0]=1;//n>1,m>1;
    for(i=1;i<=n;i++){
        if(i<m) dp[i]=dp[i-1]*2;
        else if(i==m) dp[i]=dp[i-1]*2-1;
        else dp[i]=dp[i-1]*2-dp[i-m-1];
    }
    printf("%lld",dp[n]);
}

```

6.dp(上机): 这题dp难在每个状态还会由他后面所影响, 那么这题就考虑多种情况把后面的状态也进行分类, 那么后面状态dp的时候就要考虑只能由前面标定的后面状态来推。像酒鬼, 核电站这种题, 难在当前状态会随前面不同的状态而改变递推方式, 所以也要多情况考虑。

```

n=int(input())
a=list(map(int,input().split()))
b=list(map(int,input().split()))
c=list(map(int,input().split()))
dp=[[0,0,0,0]for _ in range(n)]
dp[0][1]=dp[0][3]=-99
dp[0][0]=a[0]
dp[0][3]=b[0]
for i in range(1,n):
    dp[i][0]=max(dp[i-1][2],dp[i-1][3])+a[i]
    dp[i][1]=max(dp[i-1][1],dp[i-1][0])+b[i]
    dp[i][2]=max(dp[i-1][2],dp[i-1][3])+b[i]
    dp[i][3]=max(dp[i-1][1],dp[i-1][0])+c[i]
print(max(dp[-1][0],dp[-1][1]))

```

## 7.欧拉筛

```

def euler_sieve(n):
    is_prime, primes = [True] * (n + 1), []
    for i in range(2, n + 1):
        if is_prime[i]: primes.append(i); (for j in range(i * i, n + 1, i):
        is_prime[j] = False)
    return primes

```