一、基础算法

1.二分查找

求解最小化最大值（minMax）或最大化最小值（maxMin），使用**左闭右开**写法避免出错

08210.河中跳房子

```
l,n,m=map(int,input().split())
rocks=[]
for _ in range(n):
    rocks.append(int(input()))
rocks.append(l)

#检验目前的min_distance是否可以达到
def can_reach(min_distance,m,rocks):
    last_rock=0
    remove_num=0
    for rock in rocks:
        if rock-last_rock<min_distance:
            remove_num+=1
            if remove_num > m:
                return False

            continue
        else:
            last_rock=rock
            continue
    return True

#二分查找可能的最大的最小距离
left,right=0,l+1  #最小距离范围[left,right)
ans=-1
while left<right:
    mid=(left+right)//2
    if can_reach(mid,m,rocks):
        ans=mid
        left=mid+1 #继续尝试更大的值
    else:
        right=mid

print(ans)
```

2.math

（1）最大公因数（GCD）

```
import math

def find_gcd(x, y):
    return math.gcd(x, y)
```

（2）栈的合法出栈序列计数问题（卡特兰数）

统计从 1 到 n 所有数字的 **合法出栈序列数量**。等价于：合法括号匹配数量，栈操作中，任何时刻出栈次数不得超过入栈次数，从坐标 (0, 0) 到 (n, n) 的路径中，不越过对角线 y = x 的路径数量

✅ 最终结论（卡特兰数）

合法出栈序列的总数 = 第 n 个 **卡特兰数**：

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Python 实现：

```
import math

def catalan_number(n):
    return math.comb(2 * n, n) // (n + 1)
```

（3）保留x位小数

使用 f-string

```
number = 3.141592653589793
formatted_number = f"{number:.2f}"
print(formatted_number)  # 输出 "3.14"
```

3.读取不定行输入

```
lines = []
while True:
    try:
        line = input()
        if line:  # 如果输入非空，则添加到列表中
            lines.append(line)
        else:  # 遇到空行则停止读取
            break
    except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
        break
```

4.缓存器

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

1. **输入处理**：将每一行的高度列表转换为元组（tuple），因为Python的`lru_cache`要求参数是可哈希的（hashable），而列表不是可哈希的，但元组是。

5.欧拉筛(ES)找质数

```
def ES(n):
  isprime=[True for _ in range(n+1)]
  prime=[]
  for i in range(2,n+1):
      if isprime[i]:
          prime.append(i)
      for j in range(len(prime)):
          if  i*prime[j]>n:break
          isprime[i*prime[j]]=False
          if i%prime[j]==0 :break

  return prime
```

二、线性数据结构

1、链表

单向链表模板

```
class LinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data = data  # Store data
            self.next = next  # Point to the next node

    def __init__(self):
        self.head = None  # Initialize head as None
        self.tail = None  # Initialize tail as None
        self.size = 0  # Initialize size to 0

    def print(self):  #输出链表
        ptr = self.head
        while ptr is not None:
            if ptr != self.head:  # Avoid printing a comma before the first
element
                print(',', end='')
            print(ptr.data, end='')
            ptr = ptr.next
        print()  # Move to the next line after printing all elements

    def insert_after(self, p, data):    #在p后面插入元素
        nd = LinkList.Node(data)
        if p is None:  # If p is None, insert at the beginning
            self.pushFront(data)
        else:
            nd.next = p.next
            p.next = nd
            if p == self.tail:  # Update tail if necessary
                self.tail = nd
            self.size += 1

    def delete_after(self, p):    #删除p后面的元素
```

```
        if p is None or p.next is None:  #p是尾部
            return  # Nothing to delete
        if self.tail is p.next:  # Update tail if necessary
            self.tail = p
        p.next = p.next.next
        self.size -= 1

    def popFront(self):  #从前端弹出
        if self.head is None:
            raise Exception("Popping front from empty link list.")
        else:
            data = self.head.data
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            return data

    def pushFront(self, data):  #从后端弹出
        nd = LinkList.Node(data, self.head)
        self.head = nd
        if self.size == 0:
            self.tail = nd
        self.size += 1

    def pushBack(self, data):  #从尾部加入
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert_after(self.tail, data)

    def clear(self):  #清空链表
        self.head = None
        self.tail = None
        self.size = 0

    #以下两部分代码用于将链表转化为可迭代对象（可以在for……in……中使用）
    def __iter__(self):
        self.ptr = self.head
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
```

常见链表操作

反转链表

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_linked_list(head: ListNode) -> ListNode:
    prev = None
    curr = head
    while curr is not None:
        next_node = curr.next  # 暂存当前节点的下一个节点
        curr.next = prev       # 将当前节点的下一个节点指向前一个节点
        prev = curr            # 前一个节点变为当前节点
        curr = next_node       # 当前节点变更为原先的下一个节点
    return prev
```

合并两个有序链表

dummy哨兵节点，先人为创造一个链表头

```
def merge_sorted_lists(l1, l2):
    dummy = Node(0)
    tail = dummy
    while l1 and l2:
        if l1.data < l2.data:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    if l1:
        tail.next = l1
    else:
        tail.next = l2
    return dummy.next
```

快慢指针找链表中点

```
def find_middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next  #奇数个元素，slow停在中间；偶数个元素，slow停在中间两个中靠后的
那个
        fast = fast.next.next #奇数个元素，fast停在最后；偶数个元素，fast停在None
    return slow
```

2.栈

括号匹配

```
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            s.append(symbol) # push(symbol)，遇到左括号，压入栈
        elif symbol in ')]}':
            top = s.pop()
            if not matches(top, symbol):
                balanced = False
        index += 1
        #if balanced and s.is_empty():
        if balanced and not s:
            return True
        else:
            return False

def matches(open, close):
    opens = "([{"
    closes = ")]}"
    return opens.index(open) == closes.index(close)

print(par_checker('{{}}[]]'))

# output: False
```

进制转化

使用常用的一直除的方法，把余数依次入栈，最后依次pop出来，正好就是倒着的了

前序、中序、后序表达式

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

中序转后序算法Shunting Yard

思路：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：

        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - > 当我们看到一个左括号时，我们会将其保存以指示即将出现一个高优先级的操作符。那个操作符需要等待直到出现相应的右括号来标明它的位置（回忆完全括号化的方法）。当右括号出现时，操作符可以从栈中弹出。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

oj24591.中序表达式转后序表达式

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}  #优先级
    stack = []
    postfix = []
    number = ''  #number buffer用于接收浮点数

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

    if number:  #弹出剩余的数字
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:  #弹出剩余操作符
        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))
```

后序表达式求值

思路：扫描后缀表达式时，**看到操作数，压入栈**，等待操作符。每当在输入中**看到一个操作符时，使用最近的两个操作数进行求值，并将结果压回栈中。**

OJ24588.后序表达式求值

```
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in '+-*/':
            # 弹出栈顶的两个元素
            right_operand = stack.pop()
            left_operand = stack.pop()
            # 执行运算
            if token == '+':
                stack.append(left_operand + right_operand)
            elif token == '-':
                stack.append(left_operand - right_operand)
            elif token == '*':
                stack.append(left_operand * right_operand)
            elif token == '/':
                stack.append(left_operand / right_operand)
        else:
            # 将操作数转换为浮点数后入栈
            stack.append(float(token))
    # 栈顶元素就是表达式的结果
    return stack[0]

# 读取输入行数
n = int(input())
# 对每个后序表达式求值
for _ in range(n):
    expression = input()
    result = evaluate_postfix(expression)
    # 输出结果，保留两位小数
    print(f"{result:.2f}")
```

前序表达式求值

思路：**从后往前扫**表达式, 遇到数字压入栈, 遇到操作符从栈中弹出两个操作数 (注意顺序) 做计算, 运算完再压入栈, 扫完一遍表达式刚好运算完, 栈内唯一元素即运算结果；这是一个 **bottom-up** 的过程.代码和后序类似

3.哈希表

一种**根据关键字（key）直接访问数据**的数据结构。它的核心思想是：通过一个**哈希函数（Hash Function）将关键字映射为数组中的一个索引位置**，然后把数据存储在这个位置上，从而实现快速查找。

一、基本组成

1. **哈希函数（Hash Function）**
   把关键字转换为散列表的数组下标。常见的方法包括除留余数法、乘法散列法等。
2. **哈希表（Hash Table）**
   一个数组，数组的每个位置称为一个**桶**或**槽位**，用于存储数据项。
3. **处理冲突的方法（Collision Resolution）**
   不同的关键字可能会被映射到同一个索引，这称为**哈希冲突**。常见的解决方法有：
   - 开放地址法（Open Addressing）：如线性探测、二次探测等
   - 链地址法（Chaining）：每个槽位维护一个链表，冲突的元素依次挂在链表上

常见应用：编程语言中的字典或映射（如Python 的 `dict`）、数据库中的索引、缓存（如 LRU 缓存中用 Hash Table 存 key-value 映射）

二、开放地址法（闭散列法）

开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 key 的**初始散列地址 H0 = H(key)**发生冲突时，**以 H0 为基础，采取合适方法计算得到另一个地址 H1**，如果 H1 仍然发生冲突，**以 H1 为基础再求**下一个地址 H2，若 H2 仍然冲突，再求得 H3。依次类推，直至 Hk 不发生冲突为止，则 Hk 为该记录在表中的散列地址。

上述方法可用如下公式表示：

$$H_i = (H_{i-1}(key) + d_i) \mod m$$

其中，H(key)为散列函数，m 为散列表表长，d为增量序列。根据d取值的不同，可以分为以下3种探测方法。

(1) 线性探测法 di = 1, 2, 3, ..., m-1

(2) 二次探测法 di=12,−12,22,−22,32,....,+k2,−k2 (k≤m/2)

(3)伪随机探测法 di = 伪随机数序列

4.KMP（字符串比对）

📘 题目1：字符串乘方（练习02406）

> 给字符串 `s`，判断是否可以写成某个子串 `a` 重复多次：`s = a^n`，求最大的 `n`

✅ 核心公式

- 构建 KMP 的 `next` 数组
- 最小循环节长度 `p = len(s) - next[-1]`
- 若 `len(s) % p == 0`，则 `n = len(s) // p`

否则 `n = 1`

✅ 模板代码

```
def get_next(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
    return next

while True:
    s = input().strip()
    if s == '.':
        break
    next = get_next(s)
    p = len(s) - next[-1]
    if len(s) % p == 0:
        print(len(s) // p)
    else:
        print(1)
```

📗 题目2：前缀中的周期（练习01961 / POJ1961）

> 给一个字符串 `s`，判断它每一个前缀是否能表示成某个子串重复 `K > 1` 次。若能，输出前缀长度和重复次数

✅ 判断条件

- 构建 `next` 数组
- 每个前缀 `s[0:i]`，最小循环节长度为 `k = i - next[i - 1]`
- 若 `i % k == 0` 且 `i // k > 1`，则为周期串，输出 `i` 和 `i // k`

✅ 模板代码

```
def get_next(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
    return next

case = 0
while True:
    n = int(input())
    if n == 0:
        break
    s = input().strip()
    case += 1
    print("Test case #{}".format(case))
    next = get_next(s)
    for i in range(2, len(s) + 1):
        k = i - next[i - 1]
        if i % k == 0 and i // k > 1:
```

```
            print(i, i // k)
    print()
```

✅ **总结对比表**

题目	目标	输出内容	判断条件
字符串乘方	整个字符串是否可重复构成	最大重复次数 n	`len % (len - next[-1]) == 0`
前缀中的周期	每个前缀是否可由重复子串构成	长度 i 和次数 K	`i % (i - next[i-1]) == 0` 且 `K > 1`

🧠 记忆口诀

- `next[i]` 是：s[0:i+1] 的最长**相等前后缀**
- `循环节长度 = i - next[i-1]`
- **可整除且重复次数 > 1** → 有周期！

三、树

2.递归的基本操作

遍历

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

# 前序遍历
def preorder_traversal(node):
    if node:
        print(node.value, end=" ")
        preorder_traversal(node.left)
        preorder_traversal(node.right)

# 中序遍历
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.value, end=" ")
        inorder_traversal(node.right)

# 后序遍历
def postorder_traversal(node):
    if node:
        postorder_traversal(node.left)
        postorder_traversal(node.right)
```

```
        print(node.value, end=" ")

#层序遍历
from collections import  deque
def levelOrder(root):
    if not root:
        return []

    q=deque()
    q.append(root)
    ans=[]
    while q:
        level_size=len(q)
        level=[]
        for _ in range(level_size):
            node=q.popleft()
            level.append(node.value)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
        ans.append(level)

    return ans
```

求树的高度（注意题目定义）

OJM27638: 求二叉树的高度和叶子数目

> 本题二叉树**高度**定义：从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的结点数减1为树的高度。只有一个结点的二叉树，高度是0。

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

def tree_height(node):
    if node is None:
        return -1  # 根据定义，空树高度为-1
    return max(tree_height(node.left), tree_height(node.right)) + 1

def count_leaves(node):
    if node is None:
        return 0  #空节点，没东西
    if node.left is None and node.right is None:
        return 1  #叶子节点
    return count_leaves(node.left) + count_leaves(node.right)

n = int(input())  # 读取节点数量
nodes = [TreeNode() for _ in range(n)]
has_parent = [False] * n  # 用来标记节点是否有父节点

for i in range(n):
    left_index, right_index = map(int, input().split())
```

```
    if left_index != -1:
        nodes[i].left = nodes[left_index]
        has_parent[left_index] = True
    if right_index != -1:
        #print(right_index)
        nodes[i].right = nodes[right_index]
        has_parent[right_index] = True

# 寻找根节点，也就是没有父节点的节点
root_index = has_parent.index(False)
root = nodes[root_index]

# 计算高度和叶子节点数
height = tree_height(root)
leaves = count_leaves(root)

print(f"{height} {leaves}")
```

判断两棵树是否相同

```
def is_same_tree(p, q):
    if not p and not q:
        return True
    if not p or not q:
        return False
    return (p.val == q.val and
            is_same_tree(p.left, q.left) and
            is_same_tree(p.right, q.right))
```

翻转二叉树

```
def invert_tree(root):
    if root:
        root.left, root.right = invert_tree(root.right), invert_tree(root.left)
    return root
```

BST找最小值/最大值

```
def find_min(root):
    if not root.left:
        return root.val
    return find_min(root.left)

def find_max(root):
    if not root.right:
        return root.val
    return find_max(root.right)
```

判断是否是平衡二叉树AVL

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_balanced(root):
    def check_height(node):
        if not node:
            return 0

        left_height = check_height(node.left)
        if left_height == -1:
            return -1  # Left subtree is unbalanced

        right_height = check_height(node.right)
        if right_height == -1:
            return -1  # Right subtree is unbalanced

        if abs(left_height - right_height) > 1:
            return -1  # Current node is unbalanced

        return max(left_height, right_height) + 1

    return check_height(root) != -1
```

3.树的表示

24729: 括号嵌套树

可以用括号嵌套的方式来表示一棵树。表示方法如下：

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

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

思路：对于括号嵌套树，使用stack记录进行操作中的父节点，node记录正在操作的节点。每当遇见一个字母，将其设为node，并存入stack父节点中；遇到'('，即对当前node准备添加子节点，将其append入stack中，node重新设为None；遇到')'，stack父节点操作完毕，将其弹出并作为操作中的节点node，不断重复建立树，同时最后返出的父节点为树的根root。

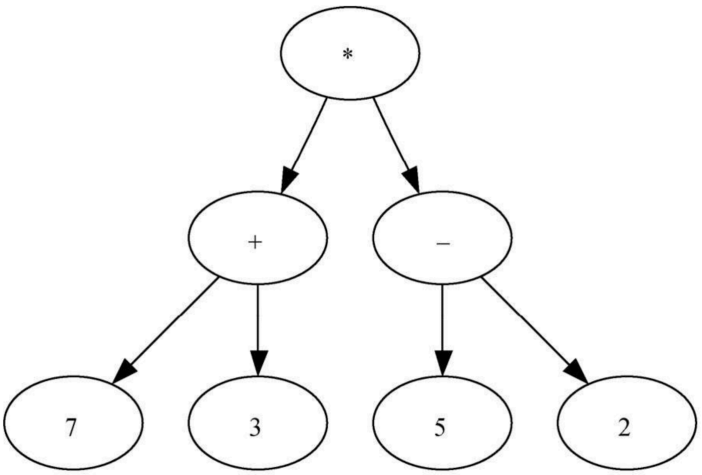
```
class TreeNode:
    def __init__(self, value): #类似字典
        self.value = value
        self.children = []

def parse_tree(s):
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
```

```
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点


def preorder(node):
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def postorder(node):
    output = []
    for child in node.children:
        output.extend(postorder(child))
    output.append(node.value)
    return ''.join(output)

# 主程序
def main():
    s = input().strip()
    s = ''.join(s.split())  # 去掉所有空白字符
    root = parse_tree(s)  # 解析整棵树
    if root:
        print(preorder(root))  # 输出前序遍历序列
        print(postorder(root))  # 输出后序遍历序列
    else:
        print("input tree string error!")

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

4.树的应用

（1）AST解析树

我们可以将`( ( 7 + 3) * ( 5 - 2 ) )`这样的数学表达式表示成解析树



- **中序表达式构建解析树**

  (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。

M25140: 根据后序表达式建立队列表达式

提示：建立起表达式树，按层次遍历表达式树的结果前后颠倒就得到队列表达式

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

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

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

n = int(input().strip())
for _ in range(n):
```

```
    postfix = input().strip()
    root = build_tree(postfix)
    queue_expression = level_order_traversal(root)[::-1]
    print(''.join(queue_expression))
```

（2）Huffman编码

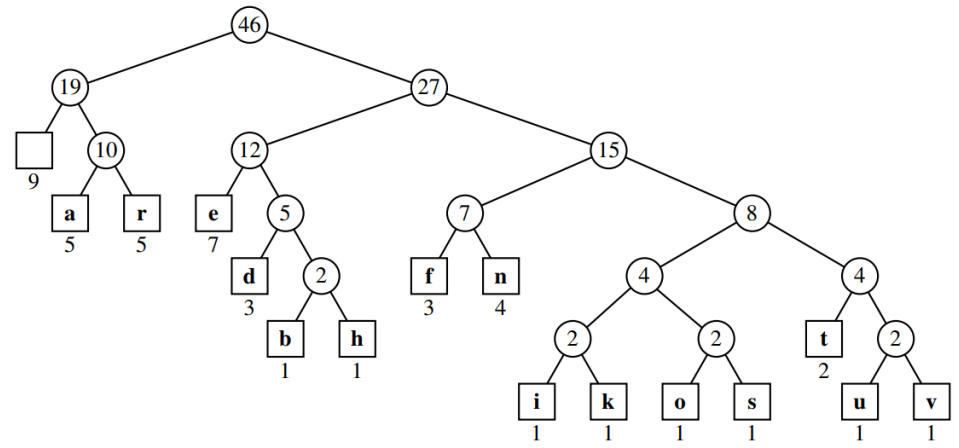
功能：可用于**求最小外部路径长度**（外部路径长度是**衡量编码效率**的一个指标，Huffman编码可以使它最小）。

Huffman编码树的本身用途——**编码和转译**，计算最优长度前缀码，左孩子0，右孩子1

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)



(a) X中每个字符的频率；(b) 字符串X的霍夫曼树T。字符c的编码是通过从T的根节点追踪到存储c的叶子节点的路径获得的，其中**左孩子边用0表示，右孩子边用1表示。**例如，字符“r”的编码是011，而字符“h”的编码是10111。

T22161: 哈夫曼编码树

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

思路：

建树：主要利用**最小堆**，**每次取出weight最小的两个节点**，weight相加后创建节点，连接左右孩子，再入堆，直至堆中只剩一个节点.

编码：跟踪每一步走的是左还是右，用0和1表示，直至遇到有char值的节点，说明到了叶子节点，将01字串添加进字典.（字符->01编码）

解码：根据01字串决定走左还是右，直至遇到有char值的节点，将char值取出.（01编码->字符）

```
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):  #在node类中重新定义<的含义
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight

#建树
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))   #把所有字符和其权值加入最小堆,最小堆内部全都是treenode类，使用类中定义的方式比大小，即先比较weight，weight相同时比较字符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，其实是啥都无所谓
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

#构建字符和01编码的对应字典
def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:  #是叶子节点，有编码的字符
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

#字符转01
def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

#01转字符
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 = {}  #字符：权值字典
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)

# 构建字符-01表
codes = encode_huffman_tree(huffman_tree)

#读取不定行输入
strings = []
while True:
    try:
        line = input()
        strings.append(line)

    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)
```

(3) 二叉堆

直接import heapq，需要实现最大堆，把数值取负数即可

(4) 二叉搜索树BST

性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。基于这一特性，**二叉搜索树的中序遍历是有序的，即从小到大的顺序**

M22275: 二叉搜索树的遍历

给出一棵二叉搜索树的前序遍历，求它的后序遍历（**BST——已知中序遍历**）

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


def build(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_val = preorder[0]
    root = Node(root_val)
    root_index = inorder.index(root_val)
    root.left = build(preorder[1:root_index + 1], inorder[:root_index])
    root.right = build(preorder[root_index + 1:], inorder[root_index + 1:])
    return root


def postorder(root):
    if not root:
        return []
    if root.left is None and root.right is None:
        return [root.val]
    result = []
    result += postorder(root.left)
    result += postorder(root.right)
    result += [root.val]
    return result


input()
preorder = list(map(int, input().split()))
inorder = sorted(preorder)
root = build(preorder, inorder)
result = postorder(root)
print(' '.join(map(str, result)))
```

(5) 平衡二叉搜索树（AVL）

```
class Node:
    def __init__(self, value):
        self.value = value   # 节点存储的值
        self.left = None     # 左子节点
        self.right = None    # 右子节点
        self.height = 1      # 节点高度（初始为1，因为单个节点高度为1）
```

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

    def insert(self, value):
        """公共插入方法：向AVL树中插入新值"""
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        """私有递归插入方法"""
        # 1. 执行标准的BST插入
        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)

        # 2. 更新当前节点的高度
        node.height = 1 + max(self._get_height(node.left),
                              self._get_height(node.right))

        # 3. 获取当前节点的平衡因子
        balance = self._get_balance(node)

        # 4. 根据平衡因子进行旋转操作，恢复平衡
        # 情况1：左子树比右子树高超过1（左重）
        if balance > 1:
            if value < node.left.value:
                # LL情况：新节点插入在左子树的左边
                return self._rotate_right(node)  # 右旋转
            else:
                # LR情况：新节点插入在左子树的右边
                node.left = self._rotate_left(node.left)  # 先对左子树左旋转
                return self._rotate_right(node)  # 再对当前节点右旋转

        # 情况2：右子树比左子树高超过1（右重）
        if balance < -1:
            if value > node.right.value:
                # RR情况：新节点插入在右子树的右边
                return self._rotate_left(node)  # 左旋转
            else:
                # RL情况：新节点插入在右子树的左边
                node.right = self._rotate_right(node.right)  # 先对右子树右旋转
                return self._rotate_left(node)  # 再对当前节点左旋转

        # 如果不需要旋转，直接返回当前节点
        return node

    def _get_height(self, node):
        """获取节点高度（处理空节点情况）"""
        if not node:
```

```
            return 0  # 空节点高度为0
        return node.height  # 非空节点返回存储的高度值

    def _get_balance(self, node):
        """计算节点的平衡因子"""
        if not node:
            return 0  # 空节点的平衡因子为0
        # 平衡因子 = 左子树高度 - 右子树高度
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        """左旋转操作（处理右右不平衡）"""
        # z: 不平衡的节点
        y = z.right  # y: z的右子节点（将成为新的根）
        T2 = y.left  # T2: y的左子树（需要重新连接）

        y.left = z
        # 将T2连接到z的右边
        z.right = T2

        # 更新旋转后节点的高度（先更新子节点z，再更新父节点y）
        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):
        """右旋转操作（处理左左不平衡）"""
        # y: 不平衡的节点
        x = y.left  # x: y的左子节点（将成为新的根）
        T2 = x.right  # T2: x的右子树（需要重新连接）

        x.right = y
        # 将T2连接到y的左边
        y.left = T2

        # 更新旋转后节点的高度（先更新子节点y，再更新父节点x）
        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
```

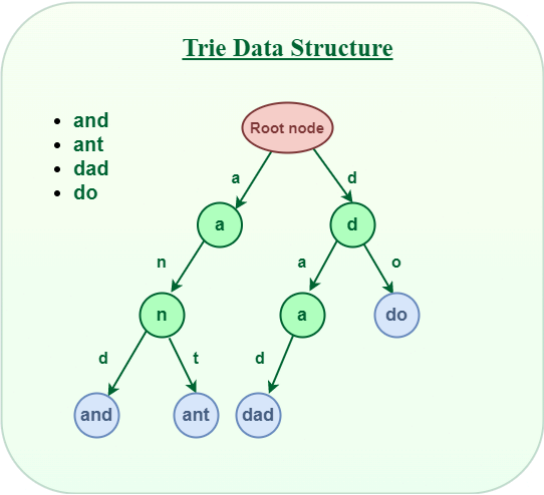
(6) 并查集（disjoint set)

```
class DisJointSet():
    def __init__(self,n):
        self.rank = [1] * n  #秩，当前树的高度
        self.parent = [i for i in range(n)]
        #self.size=[1]*n  按大小合并用

    def find(self,x):
```

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

    #按秩合并，试图最小化合并后树的高度
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx==rooty:
            return

        elif rootx!=rooty:
            if self.rank[rootx]>self.rank[rooty]:
                self.father[rooty]=rootx
            elif self.rank[rootx]<self.rank[rooty]:
                self.father[rootx]=rooty
            else:
                self.father[rootx]=rooty
                self.rank[rooty]+=1

    #或者按大小合并
    def unionBySize(self, i, j):
        irep = self.find(i)
        jrep = self.find(j)
        if irep == jrep:
            return

        isize = self.Size[irep]
        jsize = self.Size[jrep]
        if isize < jsize:
            self.Parent[irep] = jrep
            self.Size[jrep] += self.Size[irep]
        else:
            self.Parent[jrep] = irep
            self.Size[irep] += self.Size[jrep]
```

(7)Trie

本质是**字典套字典**



1. **插入（Insert）**：

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False  #是否是叶子节点（已经拼完的完整单词）

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```

2. **查找（Search）**：

```
def search(self, word):
    node = self.root
    for char in word:
        if char not in node.children:
            return False
        node = node.children[char]
    return node.is_end_of_word
```

3. **前缀查询（StartsWith）**：

```
def starts_with(self, prefix):
    node = self.root
    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]
    return True
```

四、图

1.图的表示

邻接表，defaultdict 字典套列表，好用

```
#假设输入已经转化成list，包含所有（起点，终点）元组
from collections import defaultdict
def build_graph(l):
    graph=defaultdict(list)
    for start,end in l:
        graph[start].append(end)
    return graph
```

2.图算法

（1）基础遍历

bfs

28046: 词梯

```
from collections import deque, defaultdict

def construct_graph(words):
    graph = {}
    for word in words:
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            graph[pattern].append(word)
    return graph

def bfs(start, end, graph):
    queue = deque([(start, [start])])
    visited = set([start])

    while queue:
        word, path = queue.popleft()
        if word == end:
            return path
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            if pattern in graph:
                neighbors = graph[pattern]
                for neighbor in neighbors:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, path + [neighbor]))
    return None

def word_ladder(words, start, end):
    graph = construct_graph(words)
    return bfs(start, end, graph)

n = int(input())
words = [input().strip() for _ in range(n)]
start, end = input().strip().split()

result = word_ladder(words, start, end)

if result:
    print(' '.join(result))
else:
    print("NO")
```

dfs

T28050: 骑士周游

思路：本以为是和马走日差不多的dfs模板题，但是发现会超时……算法优化方法学习题解采用 **Warnsdorff's Rule 进行搜索，优先选择出度最小的下一步路径**，从而提高找到完整骑士周游路径的成功率。需要**回溯**的功能来确保即使某个方向走不通，仍然可以回到上一步，尝试其他可能的路径。**结合了 Warnsdorff 规则 和 回溯**，确保最大程度提高找到骑士周游的概率

代码：

```
dx=[1,1,2,2,-1,-1,-2,-2]
dy=[2,-2,1,-1,2,-2,1,-1]

n=int(input())
maze=[[0]*n for _ in range(n)]
can_reach=False
sr,sc=map(int,input().split())
maze[sr][sc]=1

def is_valid(x,y):
    return 0<=x<n and 0<=y<n and maze[x][y]==0

def get_degree(x,y):  #获取棋盘上每个位置的出度
    degree=0
    for i in range(8):
        nx=x+dx[i]
        ny=y+dy[i]
        if is_valid(nx,ny):
            degree+=1
    return degree

def dfs(x,y):
    global can_reach

    if can_reach:
        return
    if sum(sum(row) for row in maze)==n*n:
        can_reach=True
        return

    next_move=[]
    for i in range(8):
        nx=x+dx[i]
        ny=y+dy[i]
        if is_valid(nx,ny):
            degree=get_degree(nx,ny)
            next_move.append((degree,nx,ny))
    next_move.sort()  #将下一步的位置按照度的大小排序，优先选择可以到达的点最少的位置作为下一个位置，提高算法效率
    for _,nx,ny in next_move:
        maze[nx][ny]=1
        dfs(nx,ny)
        if can_reach:
            return
        maze[nx][ny]=0  #回溯
```

```
dfs(sr,sc)
if can_reach:
    print('success')
else:
    print('fail')
```

（2）拓扑排序

拓扑排序（Topological Sort） 是对一个 **有向无环图（DAG）** 的所有节点进行排序，使得对图中每一条有向边 `u → v`，节点 `u` 在排序中出现在 `v` 之前。

方法一：入度法（Kahn's Algorithm）

1. **建图（邻接表）**
2. **统计每个节点的「入度」（有多少条边指向它）**
3. **把入度为 0 的节点加入队列**
4. **每次从队列中取一个节点，把它加到结果中，并将它的后继节点入度 -1**
5. **如果后继节点入度变为 0，加入队列**
6. **最终如果结果数组长度等于图的节点数，就是合法的拓扑排序**

方法二：DFS 逆后序遍历

适合递归思维。每次从未访问的节点出发 DFS，所有相邻节点遍历完后才加入结果数组（后加入的排前面）。

M04084: 拓扑排序

思路：入度法太好用了！邻接表也是快忘完了。一直很迷惑怎么固定成题目要求的那种输出，然后发现原来弹出一个点就要重新排序一遍才能和题目要求的输出长得一样，我还以为一开始就入度0和操作一次入度0不是同等地位呢，用的deque（……）题干你能再表述不清一点吗，改成heap了

```
from collections import defaultdict
import heapq

def build_graph(v,l):
    graph=defaultdict(list)
    for i in range(1,v+1):
        graph[i]=[]
    for (come,go) in l:
        graph[come].append(go)
    for i in range(1, v + 1):
        graph[i].sort()
    return graph

def topo_sort(graph,v):
    indegree=defaultdict(int)
    heap=[]
    heapq.heapify(heap)
    ans=[]

    #计算每个点的入度
    for i in range(1, v + 1):
        indegree[i]=0
```

```
    for i in range(1,v+1):
        for j in graph[i]:
            indegree[j]+=1

    #让入度为0的顶点入队
    for i in range(1,v+1):
        if indegree[i]==0:
            heapq.heappush(heap,i)

    #排序
    while heap:
        i=heapq.heappop(heap)
        ans.append(i)
        for j in graph[i]:
            indegree[j]-=1
            if indegree[j]==0:
                heapq.heappush(heap,j)

    output=' '.join('v'+str(i) for i in ans)
    return output

v,a=map(int,input().split())
l=[]
for _ in range(a):
    come,go=map(int,input().split())
    l.append((come,go))

graph=build_graph(v,l)
#print(graph)
ans=topo_sort(graph,v)
print(ans)
```

（3）判断无向图、有向图是否有环

无向图中判断是否有环

1. DFS + visited + parent

DFS遍历图，每次 DFS 时，记录当前节点的“父亲节点”。如果访问到了已经访问过的节点，且不是当前节点的父亲节点，说明存在环。

```
def has_cycle_undirected(graph):
    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
```

```
            if dfs(node, -1):
                return True
    return False
```

---

2. 并查集（Union-Find）

初始每个点属于不同的集合。每条边连接两个点，如果两个点已经在一个集合中，说明成环。**适合稠密图，边比较多时效率较高。**

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]

    def union(self, x, y):
        root_x, root_y = self.find(x), self.find(y)
        if root_x == root_y:
            return False  # 同一集合，成环
        self.parent[root_y] = root_x
        return True

def has_cycle_union_find(n, edges):
    uf = UnionFind(n)
    for u, v in edges:
        if not uf.union(u, v):
            return True
    return False
```

有向图中判断是否有环

1. DFS + recursion stack

类似无向图的 DFS，但这里需要用一个额外的 **递归栈** 记录当前路径上的节点。如果在当前 DFS 过程中再次访问到了路径上的某个节点，就说明存在环。

```
def has_cycle_directed(graph):
    visited = set()
    rec_stack = set()

    def dfs(node):
        visited.add(node)
        rec_stack.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in rec_stack:
                return True
        rec_stack.remove(node)
```

```
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False
```

---

2. 拓扑排序（Topological Sort）

- **拓扑排序只能用于有向图。**适合检测整个图是否为 DAG（有向无环图）
- 将所有入度为 0 的点加入队列，每次移除一个点并减少邻接点的入度。**最后如果还有剩余点，说明存在环。**

```
from collections import deque, defaultdict

def has_cycle_topo_sort(graph):
    indegree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    queue = deque([node for node in graph if indegree[node] == 0])
    visited_count = 0

    while queue:
        node = queue.popleft()
        visited_count += 1
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return visited_count != len(graph)
```

如果一个有向图中存在环，那么在执行拓扑排序时，**某些节点可能永远不会被访问到，因为它们是环的一部分**，无法确定一个有效的拓扑顺序。这会**导致** `visited_count` **小于图中节点的总数。**

图类型	方法	核心思想	时间复杂度	特点
无向图	DFS + parent	visited + 父节点检测	O(V + E)	常规方法，易实现
无向图	并查集	检查是否连通重复	O(α(n))	高效用于边多情况
有向图	DFS + 递归栈	检查当前路径回溯是否重复	O(V + E)	通用方法
有向图	拓扑排序	是否能全部排序	O(V + E)	适用于 DAG 判定

Msy382: 有向图判环

思路：参考讲义做的，如果在访问过程中，遇到了一个已经在当前正在访问的路径中的节点，那么就存在一个"以该点起止的"环。**可以使用一个颜色数组来跟踪每个节点的状态：未访问（0），正在访问（1），已访问（2）。**要求判断是当前有向图里有环就行。（用**颜色标识是否正在访问和recursion stack同理**）

```
def has_cycle(n,edge):
    graph=[[] for _ in range(n)]
    for u,v in edge:
        graph[u].append(v)
    color=[0]*n

    def dfs(node):
        if color[node]==1:  #重复遇到正在访问的节点，有环
            return True
        if color[node]==2:  #该节点已经访问过没有结果，说明该节点不在环里
            return False

        color[node]=1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node]=2

    for i in range(n):
        if dfs(i):
            return 'Yes'
    return 'No'

n,m=map(int,input().split())
edge=[]
for _ in range(m):
    edge.append(tuple(map(int,input().split())))
print(has_cycle(n,edge))
```

（4）强连通分量（SCC）

在**有向图**中，一个**强连通分量**（Strongly Connected Component, SCC）是一个**极大**的顶点集合，满足：

- 任意两个顶点 `u` 和 `v`，都有：
  - 从 `u` 可以到达 `v`；
  - 从 `v` 也可以到达 `u`。

即集合内任意点都**可互达**，不能再加任何点进去。

⚙️ Kosaraju 算法（时间复杂度 O(V + E)）

1. **第一次 DFS（正向图）**
   - 遍历图，记录每个节点的**结束时间**（即完成搜索的时间点）；
   - 将节点按结束时间压入栈中（用于第二步排序）。

> 如果我们在正向图中做 DFS，把每个节点的“结束时间”记录下来，**谁最后结束，就说明它和很多其它节点是连通的，可能是整个图的“出口”或汇点。**

2. **转置图**
   - 所有边方向反转，得到**反向图。**
3. **第二次 DFS（反向图）**
   - 按照第一步中栈的**出栈顺序（结束时间逆序）**，对反向图做 DFS；
   - 每次 DFS 访问到的一整组节点就是一个 SCC。

🧩 Python 实现

```python
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: 正向 DFS，记录结束时间顺序
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: 构造转置图
    transposed = [[] for _ in range(len(graph))]
    for u in range(len(graph)):
        for v in graph[u]:
            transposed[v].append(u)

    # Step 3: 反向图 DFS，找出 SCC
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            component = []
            dfs2(transposed, node, visited, component)
            sccs.append(component)
    return sccs
```

（5）最短路径问题

总结：无权图bfs，有权图dijkstra，有负权的图/检出负权环Bellman Ford，多源最短路径（任意两点）Floyd-Warshall

Dijkstra

20106: 走山路

思路：和bfs最大的区别在于每一步的权重不都是1，而是不同的非负数！进队的条件需要改成到达该点的体力值更小，而不是没有进过对就行，同时允许重复进队以反复更新最小体力值

```
import heapq
directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
def dijkstra(xs, ys, xe, ye):  # 走山路
    if region[xs][ys] == "#" or region[xe][ye] == "#":
        return "NO"
    if (xs, ys) == (xe, ye):
        return 0
    pq = []  # 初始化堆
    heapq.heappush(pq, (0, xs, ys))  # (体力消耗, x坐标, y坐标)
    visited = set()  # 已访问坐标集
    efforts = [[float('inf')] * n for _ in range(m)]  # 到达图中位置需要的体力,初始化为inf,即不可达到
    efforts[xs][ys] = 0  # 初始化体力记录表
    while pq:
        current_effort, x, y = heapq.heappop(pq)  # 取出堆顶元素
        if (x, y) in visited: continue  # 若访问过,跳过这个节点
        visited.add((x, y))  # 未访问过,加入已访问
        if (x, y) == (xe, ye):  return current_effort  # 达到终点则返回
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if 0 <= nx < m and 0 <= ny < n and (nx, ny) not in visited:
                if region[nx][ny] == "#":  continue  # 无法达到,跳过
                effort = current_effort + abs(int(region[x][y]) - int(region[nx][ny]))
                if effort < efforts[nx][ny]:
                    efforts[nx][ny] = effort
                    heapq.heappush(pq, (effort, nx, ny))  # 放入堆中
    return "NO"
```

M07735:道路

思路：差点忘了最短路径可以用**heapq**了，这样就不用纠结绕路到得了但是入队条件要求距离更短的问题了，**能到得了的都加入堆，最先弹出来的就是能到达前提下路程最短的。**太久没写Dijkstra，不怎么会写了。

```
import heapq

def Dijkstra(graph,n,k):
    q=[]
    cost=[[float('inf'),float('inf')] for _ in range(n+1)]  #路程，路费
    cost[1]=[0,0]
    heapq.heappush(q,(0,0,1) ) #当前已走路程，当前已花费路费,当前城市
    while q:
```

```
        length,money,city=heapq.heappop(q)
        if city==n:
            return length

        for (ncity,nl,nt) in graph[city]:
            if money+nt<=k:
                nlength=length+nl
                nmoney=money+nt
                #管他是不是最短路径，能走到为上，反正heapq最先弹出来的就是最短的
                heapq.heappush(q,(nlength,nmoney,ncity))

    return -1

def main():
    k=int(input())
    n=int(input())
    r=int(input())
    graph={i:[] for i in range(1,n+1)}
    for _ in range(r):
        s,d,l,t=map(int,input().split())
        graph[s].append((d,l,t))  #终点，道路长度，路费
    #print(graph)

    print(Dijkstra(graph,n,k))

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

Bellman Ford

Bellman-Ford 算法 是一种用于求解 **单源最短路径（Single-Source Shortest Path）** 的经典图算法。它可以在 **存在负权边** 的图中工作，而且还能检测是否存在 **负权环（Negative Weight Cycle）**。

给定一个图 `G = (V, E)` 和一个起点 `s`：

1. 初始化：所有点的最短距离为无穷大，起点距离为 0。
2. **对所有边进行 `V-1` 次松弛操作（Relaxation）：**
   - 如果从 u 到 v 的边权重为 w，且 `dist[u] + w < dist[v]`，则更新 `dist[v] = dist[u] + w`
3. 第 `V` 次遍历检查是否还可以继续松弛：
   - 如果可以，说明图中存在 **负权环**，无法求出最短路径；
   - 否则，得到的是从起点出发到各点的最短路径。

M787.K站中转内最便宜的航班

思路：感觉和月考的道路一题，比较像，不知道Bellman Ford是什么但是先尝试用Dijkstra写一下，会超时，发现超时的测试数据是一个到达不了的数据，就加了一个首先判断起点和终点之间有没有路能到，发现就过了。

感觉本题用Dijkstra更合适吧，因为路费都是正的,但是为什么Bellman更快呢，因为Bellman松弛操作的次数就可以用来控制k，而Dijkstra不行吗？

Dijkstra代码：

```
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst:
int, k: int) -> int:
        import heapq

        def dijkstra(graph,src,dst,n,k):
            q=[]
            heapq.heappush(q,(0,-1,src))  #钱，换乘次数，起点

            while q:
                money,exchange,i=heapq.heappop(q)
                if i==dst and exchange<=k:
                    return money

                for to,price in graph[i]:
                    nm=money+price
                    ne=exchange+1
                    if exchange<=k:
                        heapq.heappush(q,(nm,ne,to))
            return -1

        graph={i:[] for i in range(n)}
        for come,to,price in flights:
            graph[come].append((to,price))

        #检查起点和终点之间有没有路，没有直接return -1
        can_reach=[src]
        visited=set()
        while can_reach:
            place=can_reach.pop()
            visited.add(place)
            for to,_ in graph[place]:
                if to not in visited:
                    can_reach.append(to)
        if dst not in visited:
            return -1

        #dijkstra
        ans=dijkstra(graph,src,dst,n,k)
        return ans
```

Bellman Ford代码:

```
#bellman ford
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst:
int, k: int) -> int:
        def bellman_ford(src):
            # 初始化距离数组
            pre= [float('inf')] * n
            pre[src] = 0
            cur=[float('inf')] * n
            cur[src] = 0

            # 松弛所有边 n-1 次,保证所有点的距离都是最小的
```

```
            #本题要求最多中转k次，所以最多可以松弛k+1次，松弛一次就向外拓展一步，注意要全部边遍历一遍过后再更新权值，不然就不止k次中转了
            for _ in range(min(n-1,k+1)):
                updated = False
                for u, v, w in flights:
                    if pre[u] != float('inf') and pre[u] + w < cur[v] :
                        cur[v] = pre[u] + w
                        updated = True
                pre=cur.copy()

                if not updated:
                    break  # 提前退出优化

            # 检测负权环,本题不需要
            #for u, v, w in flights:
                #if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    #print("图中存在负权环！")
                    #return None

            # 返回最短路径结果
            return cur

        cost=bellman_ford(src)
        if cost[dst]==float('inf'):
            return -1
        else:
            return cost[dst]
```

Floyd Washall

思想：动态规划 + 三重循环

状态定义：`dist[i][j]` **表示 i 到 j 的最短路径长度**

转移方程：

`dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j])`

表示**是否通过中间点 k 能让路径更短**

最终得出任意两点之间的最短路径

> ✅ 关键：中间点枚举（引入“桥梁”路径）

具体步骤如下：

1. 初始化一个二维数组`dist`，用于存储任意两个顶点之间的最短距离。初始时，`dist[i][j]`表示顶点i到顶点j的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新`dist`数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通过顶点k可以使得从顶点i到顶点j的路径变短，则更新`dist[i][j]`为更小的值。

   `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新`dist`数组。最终，`dist`数组中存储的就是所有顶点之间的最短路径。

代码实现

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

> 函数将返回一个二维数组，其中`dist[i][j]`表示从顶点i到顶点j的最短路径长度。

（6）MST最小生成树

MST（Minimum Spanning Tree）最小生成树 是指一个**连通加权无向图**中：**连接所有顶点的边的子集**，且**总权重最小**，并且**不能形成回路。**例如用最短的距离连接所有城市

1. Kruskal 算法（并查集）

思路：**将边按权重从小到大排序，逐个添加到生成树中，前提是不形成环。**（形成环代表有多余的边，不是最小）

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))

    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):
        fx, fy = self.find(x), self.find(y)
        if fx == fy:
            return False
        self.parent[fy] = fx
        return True

def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])  # 按权重排序
    uf = UnionFind(n)
    mst_weight = 0
    mst_edges = []

    for u, v, weight in edges:
        if uf.union(u, v):
```

```
            mst_weight += weight
            mst_edges.append((u, v, weight))
    return mst_weight, mst_edges
```

2. Prim 算法（heapq）

思路：**从某个点出发，逐步将离已生成树最近的点加入。**

```
import heapq
from collections import defaultdict

def prim(n, edges):
    graph = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((w, v))
        graph[v].append((w, u))

    visited = [False] * n
    min_heap = [(0, 0)]  # (权重, 节点)
    mst_weight = 0
    mst_edges = []

    while min_heap and sum(visited) < n:
        weight, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        mst_weight += weight
        for w, v in graph[u]:
            if not visited[v]:
                heapq.heappush(min_heap, (w, v))
    return mst_weight
```

01258：农业网

Prim算法代码：

```
#Prim算法
import heapq
from collections import defaultdict

def build_graph(n,distance):
    graph=defaultdict(list)
    for i in range(n):
        for j in range(i+1,n):
            w=distance[i][j]
            graph[i].append((w,j))  #权重/距离，节点
            graph[j].append((w,i))
    return graph

def prim(n,graph):
    visited=[False]*n  #是否访问过该节点
    min_heap=[(0,0)]  #权重，节点
    mst_weight=0
```

```
    while min_heap and sum(visited)<n:
        weight,u=heapq.heappop(min_heap)  #弹出离当前已连接部分距离最近的城市
        if visited[u]:
            continue
        visited[u]=True
        mst_weight+=weight
        for w,v in graph[u]:  #将u城市能连接到的城市加入最小堆
            if not visited[v]:
                heapq.heappush(min_heap,(w,v))
    return mst_weight

def main():
    #读取输入
    lines = []
    while True:
        try:
            line = input()
            if line:  # 如果输入非空，则添加到列表中
                lines.append(line)
            else:  # 遇到空行则停止读取
                break
        except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
            break

    idx=0
    while idx<len(lines):
        n=int(lines[idx])
        idx+=1
        distance=[]
        for _ in range(n):
            row=list(map(int,lines[idx].split()))
            idx+=1
            if len(row)<n:
                row.extend(list(map(int,lines[idx].split())))
                idx+=1
            distance.append(row)

        graph=build_graph(n,distance)
        mst_weight=prim(n,graph)
        print(mst_weight)

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

Kruskal算法代码：

```
#Kruskal算法

class UnionFind:
    def __init__(self,n):
        self.parent=[i for i in range(n)]  #用于记录每个结点的父节点，初始都是他们自身（各自独立，都没连在一起）
```

```
    def find(self,x):  #查找x属于哪个集合
        if self.parent[x]!=x:
            self.parent[x]=self.find(self.parent[x])
        return self.parent[x]

    def union(self,x,y):
        fx,fy=self.find(x),self.find(y)
        if fx==fy:
            return False  #二者属于同一个集合
        self.parent[fy]=fx
        return True  #把y所属集合加到x所属集合里面

def kruskal(n,edges):
    uf=UnionFind(n)
    mst_weight=0

    for w,u,v in edges:
        if uf.union(u,v):
            mst_weight+=w
    return mst_weight

def main():
    #读取输入
    lines = []
    while True:
        try:
            line = input()
            if line:  # 如果输入非空，则添加到列表中
                lines.append(line)
            else:  # 遇到空行则停止读取
                break
        except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退
出循环
            break

    idx=0
    while idx<len(lines):
        n=int(lines[idx])
        idx+=1
        distance=[]
        for _ in range(n):
            row=list(map(int,lines[idx].split()))
            idx+=1
            if len(row)<n:
                row.extend(list(map(int,lines[idx].split())))
                idx+=1
            distance.append(row)

        edges=[]
        for i in range(n):
            for j in range(i+1,n):
                w=distance[i][j]
                edges.append((w,i,j))
        edges.sort()
        mst_weight=kruskal(n,edges)
        print(mst_weight)
```

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

（7）关键路径

基本概念

关键路径是**在一个有向无环图（DAG）中的最长路径**，用于表示一个工程中从开始到结束最久需要多少时间。

名称	英文缩写	含义
最早开始时间	`ve[i]` (earliest)	从起点出发，到达 `i` 的最早时间
最晚开始时间	`vl[i]` (latest)	不影响总工期的前提下，到达 `i` 的最晚时间
活动的最早/最晚开始时间	`e(k)` / `l(k)`	对边 `k: i→j` 而言，最早开始是 `ve[i]`，最晚开始是 `vl[j] - w(k)`

关键活动是指那些最早开始时间和最晚开始时间相等的活动。即对于边 `(u, v)`，如果 `EST[u] + weight(u, v) == LST[v]`，则 `(u, v)` 是关键活动。

通过检查所有边来确定哪些是关键活动，并根据这些关键活动构建关键路径。

要点

1. **拓扑排序 -> 正向更新 ve[u]，如果 `EST[u] + weight(u, v)` 大于 `EST[v]`，则更新 `EST[v] = EST[u] + weight(u, v)`。**
2. **逆拓扑排序 -> 反向更新 vl[u]，如果 `LST[u] - weight(v, u)` 小于 `LST[v]`，则更新 `LST[v] = LST[u] - weight(v, u)`。**
3. **若某边 (u→v) 满足 ve[u] == vl[v] - w，则是关键活动**
4. **最长路径 ve[终点] = 项目总工期**

代码

```
from collections import defaultdict, deque

class Edge:
    def __init__(self, v, w):
        self.v = v
        self.w = w

def topo_sort(n, G, in_degree):
    q = deque([i for i in range(n) if in_degree[i] == 0])
    ve = [0] * n
    topo_order = []

    while q:
        u = q.popleft()
        topo_order.append(u)
        for edge in G[u]:
            v = edge.v
```

```
            in_degree[v] -= 1
            if in_degree[v] == 0:
                q.append(v)
            if ve[u] + edge.w > ve[v]:  #拓扑排序，同时更新ve
                ve[v] = ve[u] + edge.w

    if len(topo_order) == n:
        return ve, topo_order
    else:  #异常情况排除（有环ect.）
        return None, None

def get_critical_path(n, G, in_degree):
    ve, topo_order = topo_sort(n, G, in_degree.copy())
    if ve is None:
        return -1, []

    #初始化终点的 LST 为其 EST 值
    maxLength = max(ve)
    vl = [maxLength] * n

    for u in reversed(topo_order):
        for edge in G[u]:
            v = edge.v
            if vl[v] - edge.w < vl[u]:
                vl[u] = vl[v] - edge.w

    activity = defaultdict(list)
    for u in G:
        for edge in G[u]:
            v = edge.v
            e, l = ve[u], vl[v] - edge.w
            if e == l:
                activity[u].append(v)

    return maxLength, activity

# Main
n, m = map(int, input().split())
G = defaultdict(list)
in_degree = [0] * n
for _ in range(m):  #建图+算入度
    u, v, w = map(int, input().split())
    G[u].append(Edge(v, w))
    in_degree[v] += 1

maxLength, activity = get_critical_path(n, G, in_degree)
if maxLength == -1:
    print("No")
else:
    print("Yes")
    print(f"Critical Path Length: {maxLength}")
    # 打印所有关键路径
    def print_critical_path(u, activity, path=[]):
        path.append(u)
        if u not in activity or not activity[u]:
            print("->".join(map(str, path)))
```

```
        else:
            for v in sorted(activity[u]):
                print_critical_path(v, activity, path.copy())
        path.pop()

    for i in range(n):
        if in_degree[i] == 0:
            print_critical_path(i, activity)
```