数据结构与算法

by 信息科学技术学院 吴一凡　　2024.6

一.逻辑结构

逻辑结构揭示了数据元素之间的逻辑关系。逻辑结构可分为“线性”和“非线性”两大类。线性结构比较直观，指数据在逻辑关系上呈线性排列；非线性结构则相反，呈非线性排列。

（1）线性数据结构：线性数据结构是一种按照线性顺序排列元素的数据结构。

其中包括：

1.列表（List）：列表是一种有序的可变序列，可以存储多个元素，并且可以根据索引访问和修改元素。

2.元组（Tuple）：元组是一种不可变序列，类似于列表，但是元组的元素不能被修改。

3.字符串（String）：字符串是一种有序的字符序列，可以通过索引访问和操作其中的字符。

（2）非线性数据结构：非线性数据结构是不按照线性顺序排列元素的数据结构。

其中包括：

1.字典（Dictionary）：字典是一种键值对的映射结构，可以根据键来访问和修改对应的值。

2.集合（Set）：集合是一种无序且不重复的数据结构，用于存储唯一的元素。

3.树（Tree）：树是一种层次结构，由节点和边组成，其中一个节点被称为根节点，每个节点可以有零个或多个子节点。

4.图（Graph）：图是由节点和边组成的数据结构，其中节点可以是任意对象，边表示节点之间的关系。

线性数据结构适用于按照顺序存储和访问元素的场景，而非线性数据结构适用于更复杂的数据关系和结构的表示和操作。

二.线性表

线性表的特点是元素之间按照一定的顺序排列，每个元素有且只有一个直接前驱和一个直接后继（除了第一个元素没有前驱，最后一个元素没有后继）。线性表又可以分为顺序表和链表。

（1）顺序表

顺序表使用连续的内存空间来存储元素。顺序表中的元素按照顺序依次存放，可以通过索引来访问和修改元素。列表（List）就是一种常见的顺序表实现。

```
class SequentialList:
    def __init__(self, n):
        self.data = list(range(n))

    def is_empty(self):
        return len(self.data) == 0

    def length(self):
        return len(self.data)
    def append(self, item):
        self.data.append(item)

    def insert(self, index, item):
        self.data.insert(index, item)

    def delete(self, index):
        if 0 <= index < len(self.data):
            del self.data[index]
        else:
            return IndexError('Index out of range')

    def get(self, index):
        if 0 <= index < len(self.data):
            return self.data[index]
        else:
            return IndexError('Index out of range')

    def set(self, index, target):
        if 0 <= index < len(self.data):
            self.data[index] = target
        else:
            return IndexError('Index out of range')

    def display(self):
        print(self.data)

lst = SequentialList(n)
```

顺序表的优点是可以快速地访问任意位置的元素，因为可以通过索引直接计算出元素的内存地址。此外，顺序表还可以动态地调整大小，即在需要时可以动态地添加或删除元素。

顺序表的缺点是在插入和删除元素时可能需要移动大量元素，这会带来较高的时间复杂度。另外，顺序表的大小是固定的，需要提前分配足够的内存空间，如果元素数量超过了分配的空间，就需要重新分配更大的空间并将现有元素复制到新的空间中。

（2）链表

链表使用离散的内存块来存储元素，每个元素由一个数据域和一个指针域组成。指针域指向下一个元素的内存地址，这样就形成了一个链式结构。

```
"""初始化链表"""
# 初始化各个节点
n0 = ListNode(1)
n1 = ListNode(3)
n2 = ListNode(2)
n3 = ListNode(5)
n4 = ListNode(4)
# 构建节点之间的引用
n0.next = n1
n1.next = n2
n2.next = n3
n3.next = n4

"""插入节点"""
```

```
def insert(n0: ListNode, P: ListNode):
    n1 = n0.next
    P.next = n1
    n0.next = P

"""删除节点"""
def remove(n0: ListNode):
    if not n0.next:
        return
    P = n0.next
    n1 = P.next
    n0.next = n1

"""访问节点"""
def access(head: ListNode, index: int) -> ListNode | None:
    for _ in range(index):
        if not head:
            return None
        head = head.next
    return head

"""查找节点"""
def find(head: ListNode, target: int) -> int:
    index = 0
    while head:
        if head.val == target:
            return index
        head = head.next
        index += 1
    return -1
```

常见的链表可分为以下几类：

（1）单链表：单链表是最简单的链表形式，每个节点包含一个数据元素和一个指向下一个节点的指针。节点之间的链接是单向的，即从头节点开始，每个节点只能访问到下一个节点。

（2）双链表：双向链表在节点中不仅包含指向下一个节点的指针，还包含指向前一个节点的指针。这样可以实现双向遍历，即可以从任意一个节点开始向前或向后遍历链表。

```
class ListNode:
    """双向链表节点类"""
    def __init__(self, val: int):
        self.val: int = val                # 节点值
        self.next: ListNode | None = None  # 指向后继节点的引用
        self.prev: ListNode | None = None  # 指向前驱节点的引用
```

（3）循环链表：循环链表是一种特殊的链表，其中最后一个节点的指针指向第一个节点，形成一个循环。这样可以在链表中实现循环访问，而不需要特别处理边界条件。

例1.颠倒链表

http://dsbpython.openjudge.cn/dspythonbook/P0040/

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

class LinkList:
    def __init__(self, lst):
        self.head = Node(lst[0])
        p = self.head
        for i in lst[1:]:
            node = Node(i)
            p.next = node
            p = p.next

    def reverse(self):
        prev = None
        curr = self.head
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node
        self.head = prev

    def print(self):
        p = self.head
        while p:
            print(p.data, end=" ")
            p = p.next
        print()

a = list(map(int, input().split()))
a = LinkList(a)
a.reverse()
a.print()
```

例2.判断是否为回文链表

找中间点反转再判断

```
def check(head: Node) -> bool:
    slow, fast = head, head
    while fast is not None and fast.next is not None:
        fast = fast.next.next
        slow = slow.next
    prev = None
    while slow:
        tmp = slow.next
        slow.next = prev
        prev = slow
        slow = tmp
    while prev is not None:
        if head.val != prev.val:
            return False
        head = head.next
        prev = prev.next
    return True
```

三.栈与队列

（1）栈

栈（stack）是一种遵循**先入后出**逻辑的线性数据结构

基本操作：

```
# 初始化栈
stack: list[int] = []

# 元素入栈
stack.append(1)
stack.append(3)
stack.append(2)
stack.append(5)
stack.append(4)

# 访问栈顶元素
peek: int = stack[-1]

# 元素出栈
pop: int = stack.pop()

# 获取栈的长度
size: int = len(stack)

# 判断是否为空
is_empty: bool = len(stack) == 0
```

栈的实现：

1.基于数组实现

使用数组实现栈时，我们可以将数组的尾部作为栈顶。入栈与出栈操作分别对应在数组尾部添加元素与删除元素，时间复杂度都为 $O(1)$ 。

```
class ArrayStack:
    """基于数组实现的栈"""
def __init__(self):
    """构造方法"""
    self._stack: list[int] = []

def size(self) -> int:
    """获取栈的长度"""
    return len(self._stack)

def is_empty(self) -> bool:
    """判断栈是否为空"""
    return self.size() == 0

def push(self, item: int):
    """入栈"""
    self._stack.append(item)
```

```python
    def pop(self) -> int:
        """出栈"""
        if self.is_empty():
            raise IndexError("栈为空")
        return self._stack.pop()

    def peek(self) -> int:
        """访问栈顶元素"""
        if self.is_empty():
            raise IndexError("栈为空")
        return self._stack[-1]

    def to_list(self) -> list[int]:
        """返回列表用于打印"""
        return self._stack
```

2.基于链表实现

使用链表实现栈时，我们可以将链表的头节点视为栈顶，尾节点视为栈底。对于入栈操作，我们只需将元素插入链表头部，这种节点插入方法被称为“头插法”。而对于出栈操作，只需将头节点从链表中删除即可。

```python
class LinkedListStack:
    """基于链表实现的栈"""
    def __init__(self):
        """构造方法"""
        self._peek: ListNode | None = None
        self._size: int = 0

    def size(self) -> int:
        """获取栈的长度"""
        return self._size

    def is_empty(self) -> bool:
        """判断栈是否为空"""
        return self._size == 0

    def push(self, val: int):
        """入栈"""
        node = ListNode(val)
        node.next = self._peek
        self._peek = node
        self._size += 1

    def pop(self) -> int:
        """出栈"""
        num = self.peek()
        self._peek = self._peek.next
        self._size -= 1
        return num

    def peek(self) -> int:
        """访问栈顶元素"""
        if self.is_empty():
            raise IndexError("栈为空")
        return self._peek.val

    def to_list(self) -> list[int]:
        """转化为列表用于打印"""
        arr = []
        node = self._peek
        while node:
            arr.append(node.val)
            node = node.next
        arr.reverse()
        return arr
```

（2）队列

队列（Queue）是一种线性数据结构，遵循**先进先出**的原则。队列允许在一端进行插入操作（队尾），在另一端进行删除操作（队头）。

基本操作：

```
from collections import deque
# 初始化队列

que: deque[int] = deque()

# 元素入队
que.append(1)
que.append(3)
que.append(2)
que.append(5)
que.append(4)

# 访问队首元素
front: int = que[0]

# 元素出队
pop: int = que.popleft()

# 获取队列的长度
size: int = len(que)

# 判断队列是否为空
is_empty: bool = len(que) == 0
```

队列的实现：

1.基于数组的实现

数组中包含元素的有效区间为 `[front, rear - 1]`。

- 入队操作：将输入元素赋值给 `rear` 索引处，并将 `size` 增加 1 。
- 出队操作：只需将 `front` 增加 1 ，并将 `size` 减少 1 。

这样，入队和出队操作都只需进行一次操作，时间复杂度均为 $O(1)$ 。

为解决在不断进行入队和出队的过程中，`front` 和 `rear` 都向右移动，当它们到达数组尾部时就无法继续移动的问题。我们可以将数组视为首尾相接的“环形数组”。

```python
class ArrayQueue:
    """基于环形数组实现的队列"""
def __init__(self, size: int):
    """构造方法"""
    self._nums: list[int] = [0] * size  # 用于存储队列元素的数组
    self._front: int = 0  # 队首指针，指向队首元素
    self._size: int = 0  # 队列长度

def capacity(self) -> int:
    """获取队列的容量"""
    return len(self._nums)

def size(self) -> int:
    """获取队列的长度"""
    return self._size

def is_empty(self) -> bool:
    """判断队列是否为空"""
    return self._size == 0

def push(self, num: int):
    """入队"""
    if self._size == self.capacity():
        raise IndexError("队列已满")
    # 计算队尾指针，指向队尾索引 + 1
    # 通过取余操作实现 rear 越过数组尾部后回到头部
    rear: int = (self._front + self._size) % self.capacity()
    # 将 num 添加至队尾
    self._nums[rear] = num
    self._size += 1

def pop(self) -> int:
    """出队"""
    num: int = self.peek()
    # 队首指针向后移动一位，若越过尾部，则返回到数组头部
    self._front = (self._front + 1) % self.capacity()
    self._size -= 1
    return num

def peek(self) -> int:
    """访问队首元素"""
    if self.is_empty():
        raise IndexError("队列为空")
    return self._nums[self._front]

def to_list(self) -> list[int]:
    """返回列表用于打印"""
    res = [0] * self.size()
    j: int = self._front
    for i in range(self.size()):
        res[i] = self._nums[(j % self.capacity())]
        j += 1
    return res
```

2.基于链表的实现

我们可以将链表的“头节点”和“尾节点”分别视为“队首”和“队尾”，规定队尾仅可添加节点，队首仅可删除节点。

```
class LinkedListQueue:
    """基于链表实现的队列"""
def __init__(self):
    """构造方法"""
    self._front: ListNode | None = None  # 头节点 front
    self._rear: ListNode | None = None  # 尾节点 rear
    self._size: int = 0

def size(self) -> int:
    """获取队列的长度"""
    return self._size

def is_empty(self) -> bool:
    """判断队列是否为空"""
    return self._size == 0

def push(self, num: int):
    """入队"""
    # 在尾节点后添加 num
    node = ListNode(num)
    # 如果队列为空，则令头、尾节点都指向该节点
    if self._front is None:
        self._front = node
        self._rear = node
    # 如果队列不为空，则将该节点添加到尾节点后
    else:
        self._rear.next = node
        self._rear = node
    self._size += 1

def pop(self) -> int:
    """出队"""
    num = self.peek()
    # 删除头节点
    self._front = self._front.next
    self._size -= 1
    return num

def peek(self) -> int:
    """访问队首元素"""
    if self.is_empty():
        raise IndexError("队列为空")
    return self._front.val

def to_list(self) -> list[int]:
    """转化为列表用于打印"""
    queue = []
    temp = self._front
    while temp:
        queue.append(temp.val)
        temp = temp.next
    return queue
```

（3）双向队列

在队列中，仅能删除头部元素或在尾部添加元素。**双向队列**（double-ended queue）提供了更高的灵活性，**允许在头部和尾部执行元素的添加或删除操作。**

基本操作：

```
from collections import deque

# 初始化双向队列
deq: deque[int] = deque()

# 元素入队
deq.append(2)      # 添加至队尾
deq.append(5)
deq.append(4)
deq.appendleft(3)  # 添加至队首
deq.appendleft(1)

# 访问元素
front: int = deq[0]  # 队首元素
rear: int = deq[-1]  # 队尾元素

# 元素出队
pop_front: int = deq.popleft()  # 队首元素出队
pop_rear: int = deq.pop()       # 队尾元素出队

# 获取双向队列的长度
size: int = len(deq)

# 判断双向队列是否为空
is_empty: bool = len(deq) == 0
```

双向队列的实现：

1.基于数组的实现

使用环形数组来实现双向队列

```
class ArrayDeque:
    """基于环形数组实现的双向队列"""
def __init__(self, capacity: int):
    """构造方法"""
    self._nums: list[int] = [0] * capacity
    self._front: int = 0
    self._size: int = 0

def capacity(self) -> int:
    """获取双向队列的容量"""
    return len(self._nums)

def size(self) -> int:
```

```python
        """获取双向队列的长度"""
        return self._size

    def is_empty(self) -> bool:
        """判断双向队列是否为空"""
        return self._size == 0

    def index(self, i: int) -> int:
        """计算环形数组索引"""
        # 通过取余操作实现数组首尾相连
        # 当 i 越过数组尾部后，回到头部
        # 当 i 越过数组头部后，回到尾部
        return (i + self.capacity()) % self.capacity()

    def push_first(self, num: int):
        """队首入队"""
        if self._size == self.capacity():
            print("双向队列已满")
            return
        # 队首指针向左移动一位
        # 通过取余操作实现 front 越过数组头部后回到尾部
        self._front = self.index(self._front - 1)
        # 将 num 添加至队首
        self._nums[self._front] = num
        self._size += 1

    def push_last(self, num: int):
        """队尾入队"""
        if self._size == self.capacity():
            print("双向队列已满")
            return
        # 计算队尾指针，指向队尾索引 + 1
        rear = self.index(self._front + self._size)
        # 将 num 添加至队尾
        self._nums[rear] = num
        self._size += 1

    def pop_first(self) -> int:
        """队首出队"""
        num = self.peek_first()
        # 队首指针向后移动一位
        self._front = self.index(self._front + 1)
        self._size -= 1
        return num

    def pop_last(self) -> int:
        """队尾出队"""
        num = self.peek_last()
        self._size -= 1
        return num

    def peek_first(self) -> int:
        """访问队首元素"""
        if self.is_empty():
            raise IndexError("双向队列为空")
        return self._nums[self._front]
```

```python
def peek_last(self) -> int:
    """访问队尾元素"""
    if self.is_empty():
        raise IndexError("双向队列为空")
    # 计算尾元素索引
    last = self.index(self._front + self._size - 1)
    return self._nums[last]

def to_array(self) -> list[int]:
    """返回数组用于打印"""
    # 仅转换有效长度范围内的列表元素
    res = []
    for i in range(self._size):
        res.append(self._nums[self.index(self._front + i)])
    return res
```

2.基于双向链表的实现

```python
class ListNode:
    """双向链表节点"""
def __init__(self, val: int):
    """构造方法"""
    self.val: int = val
    self.next: ListNode | None = None  # 后继节点引用
    self.prev: ListNode | None = None  # 前驱节点引用
class LinkedListDeque:
    """基于双向链表实现的双向队列"""
def __init__(self):
    """构造方法"""
    self._front: ListNode | None = None  # 头节点 front
    self._rear: ListNode | None = None  # 尾节点 rear
    self._size: int = 0  # 双向队列的长度

def size(self) -> int:
    """获取双向队列的长度"""
    return self._size

def is_empty(self) -> bool:
    """判断双向队列是否为空"""
    return self._size == 0

def push(self, num: int, is_front: bool):
    """入队操作"""
    node = ListNode(num)
    # 若链表为空，则令 front 和 rear 都指向 node
    if self.is_empty():
        self._front = self._rear = node
    # 队首入队操作
    elif is_front:
        # 将 node 添加至链表头部
        self._front.prev = node
        node.next = self._front
        self._front = node  # 更新头节点
    # 队尾入队操作
```

```python
        else:
            # 将 node 添加至链表尾部
            self._rear.next = node
            node.prev = self._rear
            self._rear = node  # 更新尾节点
        self._size += 1  # 更新队列长度

    def push_first(self, num: int):
        """队首入队"""
        self.push(num, True)

    def push_last(self, num: int):
        """队尾入队"""
        self.push(num, False)

    def pop(self, is_front: bool) -> int:
        """出队操作"""
        if self.is_empty():
            raise IndexError("双向队列为空")
        # 队首出队操作
        if is_front:
            val: int = self._front.val  # 暂存头节点值
            # 删除头节点
            fnext: ListNode | None = self._front.next
            if fnext != None:
                fnext.prev = None
                self._front.next = None
            self._front = fnext  # 更新头节点
        # 队尾出队操作
        else:
            val: int = self._rear.val  # 暂存尾节点值
            # 删除尾节点
            rprev: ListNode | None = self._rear.prev
            if rprev != None:
                rprev.next = None
                self._rear.prev = None
            self._rear = rprev  # 更新尾节点
        self._size -= 1  # 更新队列长度
        return val

    def pop_first(self) -> int:
        """队首出队"""
        return self.pop(True)

    def pop_last(self) -> int:
        """队尾出队"""
        return self.pop(False)

    def peek_first(self) -> int:
        """访问队首元素"""
        if self.is_empty():
            raise IndexError("双向队列为空")
        return self._front.val

    def peek_last(self) -> int:
        """访问队尾元素"""
        if self.is_empty():
            raise IndexError("双向队列为空")
        return self._rear.val

    def to_array(self) -> list[int]:
        """返回数组用于打印"""
        node = self._front
        res = [0] * self.size()
        for i in range(self.size()):
            res[i] = node.val
            node = node.next
        return res
```

例3.中序表达式转后序表达式

http://cs101.openjudge.cn/practice/24591/

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

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

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

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

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

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

例4.八皇后问题（用栈实现）

http://cs101.openjudge.cn/practice/02754

```python
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:    # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))
    return solutions

def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True

# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = queen_stack(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

例5.约瑟夫问题

http://cs101.openjudge.cn/practice/02746

①用list实现队列，O(n)

```python
# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = []
    for name in name_list:
        queue.append(name)
    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.pop(0))  # O(N)
        queue.pop(0)                                        # O(N)
    return queue.pop(0)                                 # O(N)

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

②用内置deque，O(1)

```
from collections import deque

# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = deque()
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.popleft()) # O(1)
        queue.popleft()
    return queue.popleft()

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

四.树

（1）树的概念和表示方法

树由**节点及连接节点的边**构成，有如下性质：

1.有一个根节点；

2.除根节点外，其他每个节点都与其唯一的父节点相连；

3.从根节点到其他每个节点都有且仅有一条路径；

二叉树：每个节点最多有两个子节点

按形态分类:

（1）完全二叉树——第n-1层全满，最后一层按顺序排列（靠左）

（2）满二叉树——二叉树的最下面一层元素全部满就是满二叉树

（3）avl树——左右子树高度差不超过1

（4）二叉查找树(二叉排序\搜索树)——左<中<右

例6.二叉树的深度

http://cs101.openjudge.cn/practice/06646/

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

def tree_depth(node):
    if node is None:
        return 0
    left_depth = tree_depth(node.left)
    right_depth = tree_depth(node.right)
    return max(left_depth, right_depth) + 1

n = int(input())  # 读取节点数量
nodes = [TreeNode() for _ in range(n)]

for i in range(n):
    left_index, right_index = map(int, input().split())
    if left_index != -1:
        nodes[i].left = nodes[left_index-1]
    if right_index != -1:
        nodes[i].right = nodes[right_index-1]

root = nodes[0]
depth = tree_depth(root)
print(depth)
```

例7.求二叉树的高度和叶子数目

http://cs101.openjudge.cn/practice/27638/

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

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

def count_leaves(node):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return 1
    return count_leaves(node.left) + count_leaves(node.right)

n = int(input())  # 读取节点数量
```

```
nodes = [TreeNode() for _ in range(n)]
has_parent = [False] * n  # 用来标记节点是否有父节点

for i in range(n):
    left_index, right_index = map(int, input().split())
    if left_index != -1:
        nodes[i].left = nodes[left_index]
        has_parent[left_index] = True
    if right_index != -1:
        #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}")
```

（2）树的基本操作

解析树的构建：

```
class Stack(object):
        def __init__(self):
            self.items = []
            self.stack_size = 0
    def isEmpty(self):
        return self.stack_size == 0

    def push(self, new_item):
        self.items.append(new_item)
        self.stack_size += 1

    def pop(self):
        self.stack_size -= 1
        return self.items.pop()

    def peek(self):
        return self.items[self.stack_size - 1]

    def size(self):
        return self.stack_size

class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

```python
def insertLeft(self, newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t

def insertRight(self, newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t

def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

def setRootVal(self, obj):
    self.key = obj

def getRootVal(self):
    return self.key

def traversal(self, method="preorder"):
    if method == "preorder":
        print(self.key, end=" ")
    if self.leftChild != None:
        self.leftChild.traversal(method)
    if method == "inorder":
        print(self.key, end=" ")
    if self.rightChild != None:
        self.rightChild.traversal(method)
    if method == "postorder":
        print(self.key, end=" ")
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
for i in fplist:
    if i == '(':
        currentTree.insertLeft('')
        pStack.push(currentTree)
        currentTree = currentTree.getLeftChild()
    elif i not in '+-*/)':
        currentTree.setRootVal(int(i))
        parent = pStack.pop()
        currentTree = parent
    elif i in '+-*/':
        currentTree.setRootVal(i)
```

```
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError("Unknown Operator: " + i)
    return eTree
```

例8.根据后序表达式建立队列表达式

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

```
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))
```

例9.根据二叉树中后序序列建树

http://cs101.openjudge.cn/practice/24750/

```
"""
后序遍历的最后一个元素是树的根节点。然后，在中序遍历序列中，根节点将左右子树分开。
可以通过这种方法找到左右子树的中序遍历序列。然后，使用递归地处理左右子树来构建整个树。
"""

def build_tree(inorder, postorder):
    if not inorder or not postorder:
        return []
    root_val = postorder[-1]
    root_index = inorder.index(root_val)

    left_inorder = inorder[:root_index]
    right_inorder = inorder[root_index + 1:]

    left_postorder = postorder[:len(left_inorder)]
    right_postorder = postorder[len(left_inorder):-1]

    root = [root_val]
    root.extend(build_tree(left_inorder, left_postorder))
    root.extend(build_tree(right_inorder, right_postorder))

    return root

def main():
    inorder = input().strip()
    postorder = input().strip()
    preorder = build_tree(inorder, postorder)
    print(''.join(preorder))

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

例10.根据二叉树前中序序列建树

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

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

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

```
    root.right = build_tree(preorder[1+root_index_inorder:], 
inorder[root_index_inorder+1:])
    return root

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

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

(3) 哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/

自下向上建树，逐渐向上拼凑出一棵完整的树

```
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, 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.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.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)

# 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 = []
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)
```

(4) 二叉堆

二叉堆是一种特殊的二叉树结构，它满足以下两个性质：

1. **堆序性质**：对于每个节点X，它的父节点的值要么小于等于X的值（最小堆），要么大于等于X的值（最大堆）。
2. **完全二叉树性质**：除了最底层之外，其他层的节点数必须达到最大，并且最底层的节点都依次从左到右排列。

简单最小堆示例：

```
class MinHeap:
    def __init__(self):
        self.heap = []
def parent(self, i):
    return i // 2

def left_child(self, i):
    return 2 * i

def right_child(self, i):
    return 2 * i + 1

def insert(self, value):
    self.heap.append(value)
    self.heapify_up(len(self.heap) - 1)

def heapify_up(self, i):
    while i > 0 and self.heap[i] < self.heap[self.parent(i)]:
        self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)],
self.heap[i]
        i = self.parent(i)

def extract_min(self):
    if len(self.heap) == 0:
        return None
    elif len(self.heap) == 1:
        return self.heap.pop()
    else:
        min_value = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return min_value

def heapify_down(self, i):
    while (left := self.left_child(i)) < len(self.heap):
        smallest = left
        right = self.right_child(i)
        if right < len(self.heap) and self.heap[right] < self.heap[left]:
            smallest = right
        if self.heap[i] <= self.heap[smallest]:
            break
        self.heap[i], self.heap[smallest] = self.heap[smallest], self.heap[i]
        i = smallest
```

例11.向下调整构建大顶堆

https://sunnywhy.com/sfbj/9/7

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            #print(f'i = {i}, {self.heapList}')
            self.percDown(i)
            i = i - 1
        #print(f'i = {i}, {self.heapList}')


n = int(input().strip())
heap = list(map(int, input().strip().split())) # [9, 5, 6, 2, 3]
heap = [-x for x in heap]

bh = BinHeap()
bh.buildHeap(heap)
ans = [-x for x in bh.heapList[1:]]
print(*ans)
```

（5）二叉搜索树（BST）

由于二叉搜索树的有序性，我们可以使用它来进行高效的查找、插入和删除操作。在二叉搜索树中，查询特定值的时间复杂度是**O(logN)**，其中N是树中节点的数量。

简单的二叉搜索树实现示例：

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

class BinarySearchTree:
    def __init__(self):
        self.root = None
def insert(self, value):
    if self.root is None:
        self.root = BinarySearchTreeNode(value)
    else:
        self._insert(self.root, value)

def _insert(self, node, value):
    if value < node.value:
        if node.left is None:
            node.left = BinarySearchTreeNode(value)
        else:
            self._insert(node.left, value)
    else:
        if node.right is None:
            node.right = BinarySearchTreeNode(value)
        else:
            self._insert(node.right, value)

def search(self, value):
    return self._search(self.root, value)

def _search(self, node, value):
    if node is None or node.value == value:
        return node
    if value < node.value:
        return self._search(node.left, value)
    else:
        return self._search(node.right, value)

def delete(self, value):
    self.root = self._delete(self.root, value)

def _delete(self, node, value):
    if node is None:
        return node
    if value < node.value:
        node.left = self._delete(node.left, value)
    elif value > node.value:
        node.right = self._delete(node.right, value)
    else:
        if node.left is None:
            return node.right
        elif node.right is None:
            return node.left
        else:
            min_node = self._find_min(node.right)
            node.value = min_node.value
            node.right = self._delete(node.right, min_node.value)
    return node

def _find_min(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

例12.二叉搜索树的遍历

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

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

def buildTree(preorder):
    if len(preorder) == 0:
        return None
    node = Node(preorder[0])

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

    return node

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

    return output

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

例13.二叉搜索树的层次遍历

http://cs101.openjudge.cn/practice/05455/

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

def insert(node, value):
    if node is None:
        return TreeNode(value)
    if value < node.value:
        node.left = insert(node.left, value)
    elif value > node.value:
        node.right = insert(node.right, value)
    return node

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

numbers = list(map(int, input().strip().split()))
numbers = list(dict.fromkeys(numbers))
root = None
for number in numbers:
    root = insert(root, number)
traversal = level_order_traversal(root)
print(' '.join(map(str, traversal)))
```

（5）AVL树（平衡二叉搜索树）

AVL树是一种**自平衡的二叉搜索树**，它在每次插入或删除节点时，会通过旋转操作来保持树的平衡。

AVL树具有以下特点：

①**平衡因子**：AVL树中每个节点都有一个平衡因子，它表示节点的左子树高度减去右子树高度的值。平衡因子可以是-1、0或1。

②**平衡性**：在AVL树中，每个节点的平衡因子必须满足平衡性要求，即平衡因子的绝对值不能超过1。

③**自平衡操作**：当插入或删除节点导致AVL树失去平衡时，需要进行自平衡操作来恢复平衡。自平衡操作通过旋转节点和更新平衡因子来完成。

简单的AVL树生成示例：

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

class AVLTree:
    def __init__(self):
        self.root = None
def insert(self, value):
    self.root = self._insert(self.root, value)

def _insert(self, node, value):
    if node is None:
        return AVLNode(value)
if value < node.value:
    node.left = self._insert(node.left, value)
else:
    node.right = self._insert(node.right, value)

node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
balance_factor = self._get_balance_factor(node)

# Left Left Case
if balance_factor > 1 and value < node.left.value:
    return self._right_rotate(node)
```

```
# Right Right Case
if balance_factor < -1 and value > node.right.value:
    return self._left_rotate(node)

# Left Right Case
if balance_factor > 1 and value > node.left.value:
    node.left = self._left_rotate(node.left)
    return self._right_rotate(node)

# Right Left Case
if balance_factor < -1 and value < node.right.value:
    node.right = self._right_rotate(node.right)
    return self._left_rotate(node)

return node
def _get_height(self, node):
    if node is None:
        return 0
    return node.height

def _get_balance_factor(self, node):
    if node is None:
        return 0
    return self._get_height(node.left) - self._get_height(node.right)

def _left_rotate(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 _right_rotate(self, z):
    y = z.left
    T3 = y.right
y.right = z
z.left = T3

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
```

例14.平衡二叉树的建立

https://sunnywhy.com/sfbj/9/5/359

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

```python
        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())))
```

（6）并查集（Disjoint Set）

在并查集中，每个元素都属于一个集合，并且这些集合之间是不相交的。为了高效地实现并查集操作，通常会使用树形结构来表示集合之间的关系。每个集合可以用一个树表示，其中树的根节点是集合的代表元素。使用邻接表来表示这种树形结构是一种常见的做法，其中每个节点存储其父节点的指针。

前缀树（Trie Tree）：前缀树是一种用于存储字符串集合的数据结构，通常用于快速地进行字符串匹配和搜索。在前缀树中，每个节点代表一个字符，从根节点到叶子节点的路径表示一个字符串。为了表示字符串的结构，通常会使用邻接表来表示前缀树，其中每个节点存储一个字符以及指向子节点的指针列表。

例15.食物链

http://cs101.openjudge.cn/practice/01182

```
class DisjointSet:
    def __init__(self, n):
        #设[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物。
        self.parent = [i for i in range(3 * n + 1)] # 每个动物有三种可能的类型，用 3 *
n 来表示每种类型的并查集
        self.rank = [0] * (3 * n + 1)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]
```

```python
    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv:
            return False
        if self.rank[pu] > self.rank[pv]:
            self.parent[pv] = pu
        elif self.rank[pu] < self.rank[pv]:
            self.parent[pu] = pv
        else:
            self.parent[pv] = pu
            self.rank[pu] += 1
        return True


def is_valid(n, k, statements):
    dsu = DisjointSet(n)

    def find_disjoint_set(x):
        if x > n:
            return False
        return True

    false_count = 0
    for d, x, y in statements:
        if not find_disjoint_set(x) or not find_disjoint_set(y):
            false_count += 1
            continue
        if d == 1:  # X and Y are of the same type
            if dsu.find(x) == dsu.find(y + n) or dsu.find(x) == dsu.find(y + 2 *
n):
                false_count += 1
            else:
                dsu.union(x, y)
                dsu.union(x + n, y + n)
                dsu.union(x + 2 * n, y + 2 * n)
        else:  # X eats Y
            if dsu.find(x) == dsu.find(y) or dsu.find(x + 2*n) == dsu.find(y):
                false_count += 1
            else: #[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物
                dsu.union(x + n, y)
                dsu.union(x, y + 2 * n)
                dsu.union(x + 2 * n, y + n)

    return false_count

if __name__ == "__main__":
    N, K = map(int, input().split())
    statements = []
    for _ in range(K):
        D, X, Y = map(int, input().split())
        statements.append((D, X, Y))
    result = is_valid(N, K, statements)
    print(result)
```

五.图

（1）部分定义概念：

完全图（简单完全图）：任意两个顶点之间都有边
连通图（一般指无向图）：图中任意两顶点连通
连通分量：无向图的极大连通子图（类似全集的概念）
极小连通分量：在保持连通的情况下使边数最少的子图（暗指无向图）
强连通图（特指有向图）：任意一对顶点都是强连通的
强连通分量：有向图中的极大强连通子图
生成树：包含图中全部顶点的一个极小连通子图

强连通分支：局部极大强连通子图，一个图中可能不止一个强连通分支

（2）判断无向图是否连通（有回路）

```
class Node:
def __init__(self, v):
        self.value = v
        self.joint = set()
"""判断是否连通"""
def connected(x, visited, num):
    visited.add(x)
    al = 1
    q = [x]
    while al != num and q:
        x = q.pop(0)
        for y in x.joint:
            if y not in visited:
                visited.add(y)
                al += 1
                q.append(y)
    return al == num
"""判断是否有环"""
def loop(x, visited, parent):
    visited.add(x)
    if x.joint:
        for a in x.joint:
            if a in visited and a != parent:
                return True
            elif a != parent and loop(a, visited, x):
                return True
    return False
n, m = map(int, input().split())
vertex = [Node(i) for i in range(n)]
for i in range(m):
    a, b = map(int, input().split())
    vertex[a].joint.add(vertex[b])
    vertex[b].joint.add(vertex[a])
if connected(vertex[0], set(), n):
    print('connected:yes')
else:
    print('connected:no')
```

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

（3）判断有向图是否有环：拓扑排序

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

如果最后output的长度为n则说明无环

```
class Node:
    def __init__(self, v):
        self.val = v
        self.to = []
from collections import deque
t = int(input())
for _ in range(t):
    n, m = map(int, input().split())
    node = [Node(i) for i in range(1, n + 1)]
    into = [0 for _ in range(n)]
    for _ in range(m):
        x, y = map(int, input().split())
        node[x - 1].to.append(node[y - 1])
        into[y - 1] += 1
    queue = deque([node[i] for i in range(n) if into[i] == 0])
    output = []
    while queue:
        a = queue.popleft()
        output.append(a)
        for x in a.to:
            num = x.val
            into[num - 1] -= 1
            if into[num - 1] == 0:
                queue.append(x)
    if len(output) == n:
        print('No')
    else:#否则说明有环
        print('Yes')
```

（4）图的遍历

①广度优先搜索（BFS）

例16.词梯

http://cs101.openjudge.cn/practice/28046/

```
from collections import deque
def check(a, b):
    for k in range(len(a)):
        if a[k] == '_':
            continue
        if a[k] != b[k]:
            return False
    return True

n = int(input())
graph = {}
degree = {}
vis = {}
for _ in range(n):
    word = input()
    vis[word] = False
    for p in range(4):
        tmp = word[:p]+'_'+word[p+1:]
        if word[:p]+'_'+word[p+1:] not in graph:
            graph[tmp] = [word]
            degree[tmp] = 1
        else:
            graph[tmp].append(word)
            degree[tmp] += 1

def bfs():
    start, ending = input().split()
    queue = deque()
    queue.append([start, [start]])
    vis[start] = True
    for p in range(4):
        tmp = start[:p] + '_' + start[p + 1:]
        degree[tmp] -= 1
    while queue:
        wd, now = queue.popleft()
        if wd == ending:
            print(*now)
            return
        for p in range(4):
            tmp = wd[:p] + '_' + wd[p + 1:]
            if degree[tmp] > 0:
                for wor in graph[tmp]:
                    if not vis[wor]:
                        vis[wor] = True
                        degree[tmp] -= 1
                        queue.append([wor, now + [wor]])
    print('NO')
bfs()
```

②深度优先搜索（DFS）

例17.马走日

http://cs101.openjudge.cn/practice/04123

```
import sys

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0
def add_vertex(self, key):
    self.num_vertices = self.num_vertices + 1
    new_ertex = Vertex(key)
    self.vertices[key] = new_ertex
    return new_ertex

def get_vertex(self, n):
    if n in self.vertices:
        return self.vertices[n]
    else:
        return None

def __len__(self):
    return self.num_vertices

def __contains__(self, n):
    return n in self.vertices

def add_edge(self, f, t, cost=0):
    if f not in self.vertices:
        nv = self.add_vertex(f)
    if t not in self.vertices:
        nv = self.add_vertex(t)
    self.vertices[f].add_neighbor(self.vertices[t], cost)

def getVertices(self):
    return list(self.vertices.keys())

def __iter__(self):
    return iter(self.vertices.values())
class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.disc = 0
        self.fin = 0
def __lt__(self,o):
    return self.key < o.key

def add_neighbor(self, nbr, weight=0):
    self.connectedTo[nbr] = weight

def get_neighbors(self):
    return self.connectedTo.keys()

def __str__(self):
    return str(self.key) + ":color " + self.color + ":disc " + str(self.disc) +
":fin " + str(
        self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
str(self.previous) + "]\n"
def knight_graph(board_size):
    kt_graph = Graph()
    for row in range(board_size):           #遍历每一行
        for col in range(board_size):       #遍历行上的每一个格子
            node_id = pos_to_node_id(row, col, board_size) #把行、列号转为格子ID
            new_positions = gen_legal_moves(row, col, board_size) #按照 马走日，返回
下一步可能位置
            for row2, col2 in new_positions:
                other_node_id = pos_to_node_id(row2, col2, board_size) #下一步的格
子ID
                kt_graph.add_edge(node_id, other_node_id)
    return kt_graph

def pos_to_node_id(x, y, bdSize):
    return x * bdSize + y

def gen_legal_moves(row, col, board_size):
    new_moves = []
    move_offsets = [                        # 马走日的8种走法
        (-1, -2),
        (-1, 2),
        (-2, -1),
        (-2, 1),
        (1, -2),
        (1, 2),
        (2, -1),
        (2, 1),
    ]
    for r_off, c_off in move_offsets:
        if (
            0 <= row + r_off < board_size
            and 0 <= col + c_off < board_size
        ):
            new_moves.append((row + r_off, col + c_off))
    return new_moves
```

```
def knight_tour(n, path, u, limit):
    u.color = "gray"
    path.append(u)              #当前顶点涂色并加入路径
    if n < limit:
        neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
        i = 0
    for nbr in neighbors:
        if nbr.color == "white" and \
            knight_tour(n + 1, path, nbr, limit):   #选择“白色”未经深入的点，层次加一，
递归深入
            return True
    else:                       #所有的“下一步”都试了走不通
        path.pop()              #回溯，从路径中删除当前顶点
        u.color = "white"       #当前顶点改回白色
        return False
else:
    return True
def ordered_by_avail(n):
    res_list = []
    for v in n.get_neighbors():
        if v.color == "white":
            c = 0
            for w in v.get_neighbors():
                if w.color == "white":
                    c += 1
            res_list.append((c,v))
    res_list.sort(key = lambda x: x[0])
    return [y[1] for y in res_list]

def main():
    def NodeToPos(id):
       return ((id//8, id%8))
bdSize = int(input())  # 棋盘大小
*start_pos, = map(int, input().split())  # 起始位置
g = knight_graph(bdSize)
start_vertex = g.get_vertex(pos_to_node_id(start_pos[0], start_pos[1], bdSize))
if start_vertex is None:
    print("fail")
    exit(0)

tour_path = []
done = knight_tour(0, tour_path, start_vertex, bdSize * bdSize-1)
if done:
    print("success")
else:
    print("fail")

exit(0)

# 打印路径
cnt = 0
for vertex in tour_path:
    cnt += 1
    if cnt % bdSize == 0:
        print()
    else:
        print(vertex.key, end=" ")
if __name__ == '__main__':
    main()
```

(5) 强连通图

```
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: Perform first DFS to get finishing times

   stack = []
   visited = [False] * len(graph)
   for node in range(len(graph)):
       if not visited[node]:
           dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

六.排序算法

(1）选择排序

```
def selection_sort(nums: list[int]):
    """选择排序"""
    n = len(nums)
    # 外循环：未排序区间为 [i, n-1]
    for i in range(n - 1):
        # 内循环：找到未排序区间内的最小元素
        k = i
        for j in range(i + 1, n):
            if nums[j] < nums[k]:
                k = j  # 记录最小元素的索引
        # 将该最小元素与未排序区间的首个元素交换
        nums[i], nums[k] = nums[k], nums[i]
```

（2）冒泡排序

```python
def bubble_sort(nums: list[int]):
    """冒泡排序"""
    n = len(nums)
    # 外循环：未排序区间为 [0, i]
    for i in range(n - 1, 0, -1):
        # 内循环：将未排序区间 [0, i] 中的最大元素交换至该区间的最右端
        for j in range(i):
            if nums[j] > nums[j + 1]:
                # 交换 nums[j] 与 nums[j + 1]
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

（3）插入排序

```python
def insertion_sort(nums: list[int]):
    """插入排序"""
    # 外循环：已排序区间为 [0, i-1]
    for i in range(1, len(nums)):
        base = nums[i]
        j = i - 1
        # 内循环：将 base 插入到已排序区间 [0, i-1] 中的正确位置
        while j >= 0 and nums[j] > base:
            nums[j + 1] = nums[j]  # 将 nums[j] 向右移动一位
            j -= 1
        nums[j + 1] = base  # 将 base 赋值到正确位置
```

（4）快速排序

```python
def partition(self, nums: list[int], left: int, right: int) -> int:
    """哨兵划分"""
    # 以 nums[left] 为基准数
    i, j = left, right
    while i < j:
        while i < j and nums[j] >= nums[left]:
            j -= 1  # 从右向左找首个小于基准数的元素
        while i < j and nums[i] <= nums[left]:
            i += 1  # 从左向右找首个大于基准数的元素
        # 元素交换
        nums[i], nums[j] = nums[j], nums[i]
    # 将基准数交换至两子数组的分界线
    nums[i], nums[left] = nums[left], nums[i]
    return i  # 返回基准数的索引
```

（5）归并排序

```python
def merge(nums: list[int], left: int, mid: int, right: int):
    """合并左子数组和右子数组"""
    # 左子数组区间为 [left, mid], 右子数组区间为 [mid+1, right]
    # 创建一个临时数组 tmp ，用于存放合并后的结果
    tmp = [0] * (right - left + 1)
    # 初始化左子数组和右子数组的起始索引
    i, j, k = left, mid + 1, 0
    # 当左右子数组都还有元素时，进行比较并将较小的元素复制到临时数组中
    while i <= mid and j <= right:
        if nums[i] <= nums[j]:
           tmp[k] = nums[i]
            i += 1
        else:
            tmp[k] = nums[j]
            j += 1
        k += 1
    # 将左子数组和右子数组的剩余元素复制到临时数组中
    while i <= mid:
        tmp[k] = nums[i]
        i += 1
        k += 1
        while j <= right:
        tmp[k] = nums[j]
        j += 1
        k += 1
    # 将临时数组 tmp 中的元素复制回原数组 nums 的对应区间
    for k in range(0, len(tmp)):
        nums[left + k] = tmp[k]
def merge_sort(nums: list[int], left: int, right: int):
    """归并排序"""
    # 终止条件
    if left >= right:
        return  # 当子数组长度为 1 时终止递归
    # 划分阶段
    mid = (left + right) // 2 # 计算中点
    merge_sort(nums, left, mid)  # 递归左子数组
    merge_sort(nums, mid + 1, right)  # 递归右子数组
    # 合并阶段
    merge(nums, left, mid, right)
```

(6) 堆排序

```
def sift_down(nums: list[int], n: int, i: int):
    """堆的长度为 n ，从节点 i 开始，从顶至底堆化"""
    while True:
        # 判断节点 i, l, r 中值最大的节点，记为 ma
        l = 2 * i + 1
        r = 2 * i + 2
        ma = i
        if l < n and nums[l] > nums[ma]:
            ma = l
        if r < n and nums[r] > nums[ma]:
            ma = r
        # 若节点 i 最大或索引 l, r 越界，则无须继续堆化，跳出
        if ma == i:
            break
        # 交换两节点
       nums[i], nums[ma] = nums[ma], nums[i]
        # 循环向下堆化
        i = ma
def heap_sort(nums: list[int]):
    """堆排序"""
    # 建堆操作：堆化除叶节点以外的其他所有节点
```

```python
    for i in range(len(nums) // 2 - 1, -1, -1):
        sift_down(nums, len(nums), i)
    # 从堆中提取最大元素，循环 n-1 轮
    for i in range(len(nums) - 1, 0, -1):
        # 交换根节点与最右叶节点（交换首元素与尾元素）
        nums[0], nums[i] = nums[i], nums[0]
        # 以根节点为起点，从顶至底进行堆化
        sift_down(nums, i, 0)
```

(7) 桶排序

```python
def bucket_sort(nums: list[float]):
    """桶排序"""
    # 初始化 k = n/2 个桶，预期向每个桶分配 2 个元素
    k = len(nums) // 2
    buckets = [[] for _ in range(k)]
    # 1. 将数组元素分配到各个桶中
    for num in nums:
        # 输入数据范围为 [0, 1)，使用 num * k 映射到索引范围 [0, k-1]
        i = int(num * k)
        # 将 num 添加进桶 i
        buckets[i].append(num)
    # 2. 对各个桶执行排序
    for bucket in buckets:
        # 使用内置排序函数，也可以替换成其他排序算法
        bucket.sort()
    # 3. 遍历桶合并结果
    i = 0
    for bucket in buckets:
        for num in bucket:
            nums[i] = num
            i += 1
```

(8) 计数排序

```python
def counting_sort_naive(nums: list[int]):
    """计数排序"""
    # 简单实现，无法用于排序对象
    # 1. 统计数组最大元素 m
    m = 0
    for num in nums:
        m = max(m, num)
    # 2. 统计各数字的出现次数
    # counter[num] 代表 num 的出现次数
    counter = [0] * (m + 1)
    for num in nums:
        counter[num] += 1
    # 3. 遍历 counter ，将各元素填入原数组 nums
    i = 0
    for num in range(m + 1):
        for _ in range(counter[num]):
            nums[i] = num
            i += 1
```

（9）基数排序

```python
def digit(num: int, exp: int) -> int:
    """获取元素 num 的第 k 位，其中 exp = 10^(k-1)"""
    # 传入 exp 而非 k 可以避免在此重复执行昂贵的次方计算
    return (num // exp) % 10
def counting_sort_digit(nums: list[int], exp: int):
    """计数排序（根据 nums 第 k 位排序）"""
    # 十进制的位范围为 0~9 ，因此需要长度为 10 的桶数组
    counter = [0] * 10
    n = len(nums)
    # 统计 0~9 各数字的出现次数
    for i in range(n):
        d = digit(nums[i], exp)  # 获取 nums[i] 第 k 位，记为 d
        counter[d] += 1  # 统计数字 d 的出现次数
    # 求前缀和，将“出现个数”转换为“数组索引”
    for i in range(1, 10):
        counter[i] += counter[i - 1]
    # 倒序遍历，根据桶内统计结果，将各元素填入 res
    res = [0] * n
    for i in range(n - 1, -1, -1):
        d = digit(nums[i], exp)
        j = counter[d] - 1  # 获取 d 在数组中的索引 j
        res[j] = nums[i]  # 将当前元素填入索引 j
        counter[d] -= 1  # 将 d 的数量减 1
    # 使用结果覆盖原数组 nums
    for i in range(n):
        nums[i] = res[i]
def radix_sort(nums: list[int]):
    """基数排序"""
    # 获取数组的最大元素，用于判断最大位数
    m = max(nums)
    # 按照从低位到高位的顺序遍历
    exp = 1
    while exp <= m:
        # 对数组元素的第 k 位执行计数排序
        # k = 1 -> exp = 1
        # k = 2 -> exp = 10
        # 即 exp = 10^(k-1)
        counting_sort_digit(nums, exp)
        exp *= 10
```

结束！！！