常用语法

```
math.ceil(a) #向上取整

b = 110000000.0
print("%.2f"%(b)) #可以补齐
print(round(b,7))
print(format(b,".2f"))
print(round(5.264,2)) #round() 函数，四舍五入并保留指定位数

num = 3.50300
print("{:g}".format(num)) #字符串的格式化，去除.0
print(eval(str(num))) #将str转化成数字并自动抹0

num = 3.1400
num_str = str(num) # 将浮点数转换为字符串
result = num_str.rstrip("0").rstrip(".") # 删除末尾的零和小数点

aim = 'www'
string = 'as de as www d'
print(string.count(aim))
print(string.find(aim))
ls = [1,2,3]
ls.insert(0,4)
print(ls.index(2))
a = set()
a.add(str(1)) a.add('e') a.update(str(2)) a.update('b') a.remove('a') b = a.copy()

print(ord('a'))
print(chr(98))

#2,8,16进制 bin,oct,hex
int(num,n) #把num（n进制）化为十进制

a.isnumeric() a.isalpha()

for line in ls :
    print(' '.join(map(str,line))) #矩阵输出
for i in matrix:
    print(*i)  #矩阵输出

print('\n'.join(map(str, ans))) #将列表中的元素一次性逐行输出

ls = list(map(int, input().strip().split()))
ls = list(dict.fromkeys(ls))  #删除重复元素

ls.sort(key=lambda x: x[2])
#ls为一列表，其中每个元素有三个子元素，此为按照第三个元素排序

#给keys排序
d=dict(sorted(d.items(),key=lambda x:x[0]))
#给values排序
d=dict(sorted(d.items(), key=lambda item: item[1]))

#.setdefault(key, default) 是字典（dict）的一个方法，
#用于获取指定键 key 对应的值，如果键不存在，则将键和默认值 default 添加到字典中，并返回该默认值。
#.get(key, default) 是字典（dict）的一个方法，用于
#获取指定键 key 对应的值。如果键存在于字典中，则返回对应的值；如果键不存在，则返回指定的默认值 default

#用在当def函数中的return不在末尾时，判断后面是主程序用的
if __name__ == "__main__":

#将初始值设为负无穷
startValue = float("-inf")
```

```python
#接受形如(10,90) (20,180) (30,270)的数据并对数据两两求和
pairs = [i[1:-1] for i in input().split()]
distances = [ sum(map(int,i.split(','))) for i in pairs]

#埃拉托斯特尼筛法 寻找素数！！！
def eratosthenes(n):
    # 创建一个包含了n个元素都为True的列表
    primes = [True for _ in range(n+1)]
    p = 2
    while p**2 <= n:
        # 如果primes[p]没有变为False，那么它就是一个素数
        if primes[p] == True:
            # 更新所有p的倍数为False
            for i in range(p**2, n+1, p):
                primes[i] = False
        p += 1
    # 获得所有素数
    primes_list = [p for p in range(2, n+1) if primes[p]]
    return primes_list
print(eratosthenes(100))

#正则表达式，用来找特定字符串！！！
import re
# 定义一个字符串
text = "Hello, my name is John. I live in ABC Street."
# 定义一个正则表达式模式
pattern = r"John"
# 使用re模块中的search函数查找匹配的部分
match = re.search(pattern, text)
# 如果找到匹配，则打印出匹配的内容
if match:
    print("找到匹配:", match.group())
else:
    print("未找到匹配")

#LRU Cache,限制缓存空间
from functools import lru_cache
# 使用LRU Cache装饰器来缓存函数的结果
@lru_cache(maxsize=3)  # 最多缓存3个结果
def expensive_function(n):
    print(f"Calculating {n}")
    return n * n
# 访问函数的结果，第一次计算并缓存
print(expensive_function(2))
# 访问函数的结果，使用缓存
print(expensive_function(2))
# 计算并缓存不同的输入
print(expensive_function(3))
# 计算并缓存不同的输入，导致最早的结果被淘汰
print(expensive_function(4))

#一个自己的二分查找
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid  # 找到目标，返回索引
        elif arr[mid] < target:
            left = mid + 1  # 目标在右侧
        else:
            right = mid - 1  # 目标在左侧
```

```
    return -1  # 目标不在列表中

#求最大公约数
def gcd(m,n):
    while m%n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm%oldn
    return n

#zip函数，将不同列表的对应元素打包成元组
list1 = [4, 2, 1, 3]
list2 = ['a', 'b', 'c', 'd']
list3 = ['x', 'y', 'z', 'w']
zipped = zip(list1, list2, list3)  # 使用 zip() 打包这三个列表
ls = sorted(list(zipped))
print(ls)
for item in zipped:
    print(item)

from collections import deque #双端队列
import heapq #堆
import sys
.isalnum() 是否只包含数字和字母  .isnumeric() 是否只包含数字
sys.exit(0) #直接结束程序！！！
#第一行加# pylint: skip-file跳过检查
```

栈，队列，链表

```
###定义一个“栈”
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

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

    def size(self):
        return len(self.items)

s = Stack()

print(s.is_empty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.is_empty())
s.push(8.4)
print(s.pop())
print(s.size())
###
###可用栈处理“括号匹配”，“进制转换”，“中缀转后缀”，“后序表达式求值”，“八皇后”
```

```
###单调栈，找出右手边第一个大于自己的元素的位置，保证栈中的元素是递减的
n = int(input())
arr = list(map(int,input().split()))
stack = []
f = [0] * n

for i in range(n):
    while stack and arr[stack[-1]] < arr[i]:
        f[stack.pop()] = i + 1

    stack.append(i)

while stack:
    f[stack.pop()] = 0

print(*f)
###

###定义一个“队列”
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

q = Queue()

q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
print(q.items)
q.dequeue()
print(q.items)
###

###定义一个“双端队列”
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)
```

```
    def size(self):
        return len(self.items)

d = Deque()
print(d.isEmpty())
d.addRear(4)
d.addRear('dog')
d.addFront(True)
print(d.size())
print(d.isEmpty())
d.addRear(8.4)
print(d.removeRear())
print(d.removeFront())
###

###定义一个“单向链表”
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def delete(self, value):
        if self.head is None:
            return

        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()

# 使用示例
linked_list = LinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()  # 输出：1 2 3
linked_list.delete(2)
linked_list.display()  # 输出：1 3
```

```
###

###定义一个“双向链表”
class Node:
    def __init__(self, value):
        self.value = value
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_before(self, node, new_node):
        if node is None:  # 如果链表为空，将新节点设置为头部和尾部
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = node
            new_node.prev = node.prev
            if node.prev is not None:
                node.prev.next = new_node
            else:  # 如果在头部插入新节点，更新头部指针
                self.head = new_node
            node.prev = new_node

    def display_forward(self):
        current = self.head
        while current is not None:
            print(current.value, end=" ")
            current = current.next
        print()

    def display_backward(self):
        current = self.tail
        while current is not None:
            print(current.value, end=" ")
            current = current.prev
        print()

# 使用示例
linked_list = DoublyLinkedList()
# 创建节点
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
# 将节点插入链表
linked_list.insert_before(None, node1)  # 在空链表中插入节点1
linked_list.insert_before(node1, node2)  # 在节点1前插入节点2
linked_list.insert_before(node1, node3)  # 在节点1前插入节点3
# 显示链表内容
linked_list.display_forward()  # 输出：2 3 1
linked_list.display_backward()  # 输出：1 3 2
###
```

排序

```
###（直接）插入排序
### j从第2个开始,其左边的只要比它大就往右走一个,不比它大就break,将第j个放在当前位置。
def insertionSort(A):
    for j in range(1, len(A)):
        key = A[j]
        # Insert A[i] into the
        # sorted sequence A[0..j-1]
```

```
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i + 1] = A[i]
            i -= 1
        A[i + 1] = key

arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print(' '.join(map(str, arr)))
# Time Complexity: O(N^2)
# Auxiliary Space: O(1)
# Stability: YES
###

###冒泡排序
###从第1个开始,左边比右边大就交换,依次从后往前排好。
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break

if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]
    bubbleSort(arr)
    print(' '.join(map(str, arr)))
# Time Complexity: O(N^2)
# Auxiliary Space: O(1)
# Stability: YES
###

###选择排序
###从第1个开始,找到剩余队列中最小的,放在排列好的后的第一个,依次从前往后排好。
A = [64, 25, 12, 22, 11]
# Traverse through all array elements
for i in range(len(A)):
    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[j] < A[min_idx] :
            min_idx = j
    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

print(' '.join(map(str, A)))
# Time Complexity: O(N^2)
# Auxiliary Space: O(1)
# Stability: NO
###

###快速排序
###以最后一个为pivot,双指针，从左边找比pivot大的,从右边找比pivot小的,交换,直到指针重合,将pivot放中间,再对左半和右半重复。
```

```python
def quicksort(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort(arr, left, partition_pos - 1)
        quicksort(arr, partition_pos + 1, right)

def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i

arr = [22, 11, 88, 66, 55, 77, 33, 44]
quicksort(arr, 0, len(arr) - 1)
print(arr)
# Time Complexity: O(N^2)
# Auxiliary Space: O(1)
# Stability: NO
###

###合（归）并排序 (merge sort)
###直接分成最小单元,依次排好左最小两个、次小...左半、右最小两个、次小...右半,最后合并排好两半。
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2

        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # Into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    mergeSort(arr)
```

```
    print(' '.join(map(str, arr)))
# Time Complexity: O(N*log(N))
# Auxiliary Space: O(N)
# Stability: YES
###

###merge sort 并且记录了逐个交换的次数
def merge_sort(lst):
    if len(lst) <= 1:
        return lst, 0

    middle = len(lst) // 2
    left, inv_left = merge_sort(lst[:middle])
    right, inv_right = merge_sort(lst[middle:])

    merged, inv_merge = merge(left, right)

    return merged, inv_left + inv_right + inv_merge

def merge(left, right):
    merged = []
    inv_count = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i

    merged += left[i:]
    merged += right[j:]

    return merged, inv_count

while True:
    n = int(input())
    if n == 0:
        break

    lst = []
    for _ in range(n):
        lst.append(int(input()))

    _, inversions = merge_sort(lst)
    print(inversions)
###

###希尔排序（直接插入的改进版）
###gap=n/2,j从gap右侧第一个开始,与gap左侧前j个比较,交换;gap减半,重复,至gap=1,对整个arr比较交换一次。
def shellSort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap # This will keep help in maintain gap value
            while i >= 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:
```

```
                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]
                i = i - gap # To check left side also
            # If the element present is greater than current element
            j += 1
        gap = gap // 2

arr2 = [12, 34, 54, 2, 3]
shellSort(arr2, len(arr2))
print(' '.join(map(str, arr2)))
# Time Complexity: O(N^2)
# Auxiliary Space: O(1?)
# Stability: NO
###

###堆排序 构建最大堆+依次弹出堆顶元素放至队尾,并维护剩余堆
def heapify(arr, n, i):
    largest = i  # 初始化最大元素的索引为父节点索引
    left = 2 * i + 1  # 左子节点索引
    right = 2 * i + 2  # 右子节点索引

    # 如果左子节点存在且大于父节点，则更新最大元素的索引
    if left < n and arr[left] > arr[largest]:
        largest = left

    # 如果右子节点存在且大于当前最大元素，则更新最大元素的索引
    if right < n and arr[right] > arr[largest]:
        largest = right

    # 如果最大元素不是父节点，则交换父节点和最大元素，并继续递归调整子树
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # 构建最大堆 O(log(N))
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # 交换堆顶元素与最后一个元素，并调整堆 O(N*log(N))
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # 将最大元素放到末尾
        heapify(arr, i, 0)  # 调整堆

    return arr

# 示例
arr = [12, 11, 13, 5, 6, 7]
sorted_arr = heap_sort(arr)
print(sorted_arr)
# Time Complexity: O(N*log(N))
# Auxiliary Space: O(1)
# Stability: NO
###
```

树

```
###求二叉树的高度和叶子数目（node未按顺序输入，从0开始）
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}")
###

###求二叉树的深度（node按顺序输入，从1开始）
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)
###

###输入括号嵌套树，输出其前序遍历和后续遍历
class TreeNode:
    def __init__(self, value):
        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): #前序遍历
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def postorder(node): #后序遍历，输出一串数字
    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()
###

###栈+树构建+将中序表达式构建为树+输出三种遍历+用树计算+还原括号
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

    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

exp = "( ( 7 + 3 ) * ( 5 - 2 ) )"
pt = buildParseTree(exp)
for mode in ["preorder", "postorder", "inorder"]:
    pt.traversal(mode)
    print()

"""
* + 7 3 - 5 2
7 3 + 5 2 - *
7 + 3 * 5 - 2
"""

# 代码清单6-10（与下同方法）
import operator

def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}

    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()

    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()

print(evaluate(pt))
# 30

#代码清单6-14 后序求值（与上同方法）
def postorder_evaluate(tree):
    opers = {'+':operator.add, '-':operator.sub,
             '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postorder_evaluate(tree.getLeftChild())
        res2 = postorder_evaluate(tree.getRightChild())
        if res1 and res2:
```

```
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()

print(postorder_evaluate(pt))
# 30

#代码清单6-16 中序还原完全括号表达式
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal

print(printexp(pt))
# (((7)+3)*((5)-2))
###

###最全面的中转后，后转中（可用来删去多余括号），用re处理多位数和小数点
import re

def f(number): #处理小数点用，将标记处cha改为f(cha)即可
    """Remove trailing zeros from a decimal number."""
    if '.' in number:
        number = number.rstrip('0').rstrip('.')
    return number

class Node:
    def __init__(self,value):
        self.value = value
        self.children = []

def pri(x):
    if x == '*' :
        return 2
    if x == '+' :
        return 1
    return 0

def midtopost(mid):
    post = []
    stack = []
    for cha in mid:
        if re.match(r'\d+(\.\d+)?', cha): #if cha.isnumeric():
            post.append(cha) #
        else:
            if cha == '(' :
                stack.append(cha)
            elif cha == ')' :
                while stack and stack[-1] != '(' :
                    post.append(stack.pop())
                stack.pop()
            else:
                while stack and pri(stack[-1]) >= pri(cha) and stack[-1] != '(' :
                    post.append(stack.pop())
                stack.append(cha)
    while stack:
        post.append(stack.pop())
    return post

def posttomid(post):
    stack = []
    for cha in post:
```

```
        if re.match(r'\d+(\.\d+)?', cha): #if cha.isnumeric():
            stack.append(cha) #
        elif cha in ['+','*'] :
            if cha == '*' :
                if '+' in stack[-2]:
                    stack[-2] = '(' + stack[-2] + ')'
                if '+' in stack[-1]:
                    stack[-1] = '(' + stack[-1] + ')'
            op2 = stack.pop()
            op1 = stack.pop()
            stack.append(op1 + cha + op2)
    return stack.pop()

while True:
    try:
        s = input()
        mid = [token for token in re.split(r'(\d+\.\d+|\d+|[\+\*\(\)])', s) if token]
        post = midtopost(mid) #若不需要re处理，上行可直接删去
        print(posttomid(post))
    except EOFError:
        break
###

###输入表达式，转后序，根据后序建树，输出“树”的图形，求树高，赋值并计算
import operator as op

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

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

def transfer(mid): #中序转后序
    post = []
    stack = [] #operation_stack
    for cha in mid:
        if cha.isalpha():
            post.append(cha)
        else:
            if cha == '(' :
                stack.append(cha)
            elif cha == ')' :
                while stack and stack[-1] != '(' :
                    post.append(stack.pop())
                stack.pop()
            else:
                while stack and priority(stack[-1]) >= priority(cha) and stack[-1] != '(' :
                    post.append(stack.pop())
                stack.append(cha)
    while stack:
        post.append(stack.pop())
    return post

def buildtree(post): #根据后序建树
    stack = [] #node_stack
    for cha in post:
        if cha.isalpha():
            node = TreeNode(cha)
```

```
        else:
            node = TreeNode(cha)
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]

def get_depth(node): #得到的值减1即为树高
    left_depth = get_depth(node.left) if node.left else 0
    right_depth = get_depth(node.right) if node.right else 0
    return max(left_depth,right_depth) + 1

def printtree(root,depth): #输出“树”的图形
    graph = [' '*(2**depth-1) + root.value + ' '*(2**depth-1)]
    graph.append(' '*(2**depth-2) + ('/' if root.left else ' ')
                 + ' ' + ('\\' if root.right else ' ') + ' '*(2**depth-2))

    if depth == 0:
        return root.value

    depth -= 1

    if root.left:
        left = printtree(root.left,depth)
    else:
        left = [' '*(2**(depth+1)-1)]*(2*depth+1)
    if root.right:
        right = printtree(root.right,depth)
    else:
        right = [' '*(2**(depth+1)-1)]*(2*depth+1)

    for i in range(2*depth+1):
        graph.append(left[i] + ' ' + right[i])

    return graph

def get_value(node,value_dict): #根据value_dict赋值并计算
    if node.value in '+-*/' :
        operator = {'+': op.add, '-': op.sub, '*': op.mul, '/': op.floordiv}
        return operator[node.value](get_value(node.left,value_dict) ,
get_value(node.right,value_dict))
    else:
        return value_dict[node.value]

mid = input()
n = int(input()) #字母个数
value_dict = {}
for i in range(n):
    alpha,num = input().split()
    value_dict[alpha] = int(num)

post = transfer(mid)
root = buildtree(post)
depth = get_depth(root) - 1
graph = printtree(root,depth)
answer = get_value(root,value_dict)

print(''.join(post))
for line in graph:
    print(line)
print(answer)
###

###根据前序和中序建树，输出后序
class Treenode:
```

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

def buildtree(pre,mid):
    if not pre or not mid:
        return None

    rootvalue = pre[0]
    point = mid.find(rootvalue)

    leftmid = mid[:point]
    rightmid = mid[point+1:]

    leftpre = pre[1:len(leftmid)+1]
    rightpre = pre[len(leftmid)+1:]

    node = Treenode(rootvalue)
    node.left = buildtree(leftpre,leftmid)
    node.right = buildtree(rightpre,rightmid)

    return node

def postorder(root): #输出一串字符
    if root is None:
        return ''
    return postorder(root.left) + postorder(root.right) + root.value

while True:
    try:
        pre = input()
        mid = input()
        root = buildtree(pre,mid)
        print(''.join(postorder(root)))
    except EOFError:
        break
###

###哈夫曼编码树（节点为char，附上对应的二进制编码） #所有节点的带权路径和最小
import heapq

class Node:
    def __init__(self,char,weight):
        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 Buildtree(dic):
    heap = []
    for char,weight in dic.items():
        heapq.heappush(heap,Node(char,weight))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None,left.weight + right.weight)
        merged.left = left
        merged.right = right
```

```
        heapq.heappush(heap,merged)

    return heap[0]

def encode(root):  #给节点附上二进制表示
    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 char_to_num(codes,string):  #把字符串转成数字编码
    ans = ''
    for char in string:
        ans += codes[char]
    return ans

def num_to_char(root,string):  #把数字编码转成字符串
    ans = ''
    node = root
    for num in string:
        if num == '0':
            node = node.left
        else:
            node = node.right

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

n = int(input())
dic = {}
for i in range(n):
    char,weight = input().split()
    dic[char] = int(weight)
huffmantree = Buildtree(dic)
codes = encode(huffmantree)
while True:
    try:
        string = input()
        if string[0] in('0','1'):
            print(num_to_char(huffmantree,string))
        else:
            print(char_to_num(codes,string))
    except EOFError:
        break
###

###二叉搜索树（BST）的建立和逐行输出 #左小右大
class Treenode:
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None

def buildtree(ls):
    root = None
    for num in ls:
```

```
        root = insert(root,num)
    return root

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

def traversal(node):
    stack = [node]
    ans = []
    while stack:
        nd = stack.pop(0)
        ans.append(str(nd.value))
        if nd.left:
            stack.append(nd.left)
        if nd.right:
            stack.append(nd.right)
    return ans

ls = list(map(int, input().strip().split()))
ls = list(dict.fromkeys(ls))
tree = buildtree(ls)
ans = traversal(tree)
print(' '.join(ans))
###

###平衡二叉搜索树（AVL）有多少节点，即斐波那契数列
from functools import lru_cache

@lru_cache(maxsize=None)
def avl_min_nodes(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return avl_min_nodes(n-1) + avl_min_nodes(n-2) + 1

n = int(input())
min_nodes = avl_min_nodes(n)
print(min_nodes)
###

###平衡二叉搜索树有多少层（高度）
from functools import lru_cache

@lru_cache(maxsize=None)
def min_nodes(h):
    if h == 0:
        return 0
    if h == 1:
        return 1
    return min_nodes(h-1) + min_nodes(h-2) + 1

def max_height(n):
    h = 0
    while min_nodes(h) <= n:
        h += 1
    return h - 1
```

```
n = int(input())
print(max_height(n))
###
###AVL树的高度都等于节点数取对数再乘以一个常数（1.44）（n>10可用）

###平衡二叉树的实现，并输出前序遍历
class Node:
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

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

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

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

        node.height = 1 + max(self.get_height(node.left),self.get_height(node.right))

        balance = self.get_balance(node)

        if balance > 1:
            if value < node.left.value:  #LL
                return self.rotate_right(node)
            else: #LR
                node.left = self.rotate_left(node.left)
                return self.rotate_right(node)

        if balance < -1:
            if value > node.right.value:  #RR
                return self.rotate_left(node)
            else:  #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())
ls = list(map(int,input().split()))

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

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

###遍历树，按该节点及子节点的值从小到大遍历输出值
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def traverse_print(root, nodes):
    if root.children == []:
        print(root.value)
        return
    pac = {root.value: root}
    for child in root.children:
        pac[child] = nodes[child]
    for value in sorted(pac.keys()):
        if value in root.children:
            traverse_print(pac[value], nodes)
        else:
            print(root.value)

n = int(input())
nodes = {}
children_list = []
for i in range(n):
    info = list(map(int, input().split()))
    nodes[info[0]] = TreeNode(info[0])
    for child_value in info[1:]:
        nodes[info[0]].children.append(child_value)
        children_list.append(child_value)
root = nodes[[value for value in nodes.keys() if value not in children_list][0]]
traverse_print(root, nodes)
###

###字典树Trie,查找是否有相同的前序
```

```
class Trienode:
    def __init__(self):
        self.child = {}

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

    def insert(self,v):
        current = self.root
        for x in v:
            if x not in current.child:
                current.child[x] = Trienode()
            current = current.child[x]

    def search(self,v_):
        current = self.root
        for x in v_:
            if x not in current.child:
                return 0
            current = current.child[x]
        return 1

for i in range(int(input())):

    nums = []
    for j in range(int(input())):
        nums.append(str(input()))
    nums.sort(reverse = True)

    trie = Trie()
    s = 0

    for num in nums:
        s += trie.search(num)
        trie.insert(num)
    if s > 0 :
        print('NO')
    else:
        print('YES')
###

###树的镜面映射，输入“伪满二叉树”，输出原树的镜面映射的层级遍历
from collections import deque

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

def buildtree(ls, index):
    node = Treenode('')
    node.value = ls[index][0]
    if ls[index][1] == '0':

        index += 1
        child, index = buildtree(ls, index)
        node.left = child

        index += 1
        child, index = buildtree(ls, index)
        node.right = child

    return node, index
```

```
def printtree(node):
    queue, stack = deque(), deque()

    while node is not None:
        if node.value != '$':
            stack.append(node)
        node = node.right

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

    while queue:
        node = queue.popleft()
        print(node.value, end = ' ')

        if node.left:
            node = node.left
            while node is not None:
                if node.value != '$':
                    stack.append(node)
                node = node.right

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

if __name__ == "__main__":
    n = int(input())
    root, index = buildtree(list(input().split()), 0)
    printtree(root)
###

###左儿子右兄弟建树加输出深度（输入du序列）
class TreeNode:
    def __init__(self):
        self.children = []
        self.first_child = None
        self.next_sibling = None

def buildtree(string):
    root = TreeNode()
    depth = 0
    stack = [root]

    for i in string:
        current_node = stack[-1]

        if i == 'd':
            new_node = TreeNode()

            if not current_node.children:
                current_node.first_child = new_node
            else:
                current_node.children[-1].next_sibling = new_node

            current_node.children.append(new_node)
            stack.append(new_node)
            depth = max(depth,len(stack)-1)
        else:
            stack.pop()
    return root, depth

def tra_depth(node):
    if not node:
        return 0
```

```
    return max(tra_depth(node.first_child), tra_depth(node.next_sibling)) + 1

string = input()
root, ori_depth = buildtree(string)
tra_d = tra_depth(root)
print(f'{ori_depth} => {tra_d -1}')
###

###02775文件结构“图”，主要学习这个奇怪输出的写法！
class File:
    def __init__(self):
        self.name = 'ROOT'
        self.files = []
        self.dirs = []

    def __str__(self):
        return '\n'.join([self.name]+['|     '+s for d in self.dirs for s in
str(d).split('\n')]+sorted(self.files))

    def build(self,parent,s):
        if s[0] == 'f':
            parent.files.append(s)
        else:
            dir_ = File()
            dir_.name = s
            parent.dirs.append(dir_)
            while True:
                s = input()
                if s == ']' :
                    break
                dir_.build(dir_,s)

x = 0
while True:
    s = input()
    if s == '#' :
        break
    x += 1
    root = File()
    while s != '*' :
        root.build(root,s)
        s = input()
    print('DATA SET '+str(x)+':')
    print(root)
    print()
###
```

散列表，并查集

```
###散列表的实现
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data #replace
```

```
            else:
                nextslot = self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot,len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot] = key
                    self.data[nextslot] = data
                else:
                    self.data[nextslot] = data #replace

    def hashfunction(self,key,size):
        return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size

    def get(self,key):
        startslot = self.hashfunction(key,len(self.slots))

        data = None
        stop = False
        found = False
        position = startslot
        while self.slots[position] != None and not found and not stop:
                if self.slots[position] == key:
                    found = True
                    data = self.data[position]
                else:
                    position=self.rehash(position,len(self.slots))
                    if position == startslot:
                        stop = True
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)

H=HashTable()
H[54]="cat" H[26]="dog" H[93]="lion" H[17]="tiger" H[77]="bird" H[31]="cow" H[44]="goat"
H[55]="pig"
H[20]="chicken"
print(H.slots)
print(H.data)

print(H[20])
print(H[17])

H[20] = 'duck'
print(H[20])
print(H.data)
print(H[99])
###

###并查集，并根据高度Union
class DisjSet:
    def __init__(self, n):
        self.rank = [1] * n
        self.parent = [i for i in 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):
        xset = self.find(x)
        yset = self.find(y)
        if xset == yset:
            return
        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[yset] = xset  #这里把y接入x，实际无所谓
            self.rank[xset] = self.rank[xset] + 1

obj = DisjSet(5) obj.Union(0, 2) obj.Union(4, 2) obj.Union(3, 1)
if obj.find(4) == obj.find(0):
    print('Yes')
else:
    print('No')
if obj.find(1) == obj.find(0):
    print('Yes')
else:
    print('No')
###

###并查集，并根据元素数量Union
class UnionFind:
    def __init__(self, n):
        self.Parent = list(range(n))
        self.Size = [1] * n

    def find(self, i):
        if self.Parent[i] != i:
            self.Parent[i] = self.find(self.Parent[i])
        return self.Parent[i]

    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]

n = int(input())
unionFind = UnionFind(n)

unionFind.unionBySize(0, 1) unionFind.unionBySize(2, 3) unionFind.unionBySize(0, 4)
###

###用并查集输出班级个数（n为人数，m为关系数）注意用set!
def Find(i):
    if parent[i] != i:
        return Find(parent[i])
    return parent[i]
```

```
def Union(i,j):
    ipar = Find(i)
    jpar = Find(j)
    parent[ipar] = jpar

case = 0
while True:
    case += 1

    n,m = map(int,input().split())
    if n == m == 0 :
        break
    parent = [_ for _ in range(n+1)]

    for k in range(m):
        i,j = map(int,input().split())
        Union(i,j)

    aset = set(Find(x) for x in range(1,n+1))
    print('Case '+str(case)+': '+str(len(aset)))
###
```

图

```
###图的实现，Vertex和Graph（笔试用）
###不要用v.getId之类的，因为输出是method不是数，直接v.id即可
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

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

    #def __str__(self):
        #return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])
    #def getConnections(self):
        #return self.connectedTo.keys()
    #def getId(self):
        #return self.id
    #def getWeight(self,nbr):
        #return self.connectedTo[nbr]

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    #def getVertex(self,n):
        #if n in self.vertList:
            #return self.vertList[n]
        #else:
            #return None
    #def __contains__(self,n):
        #return n in self.vertList
    #def getVertices(self):
        #return self.vertList.keys()

    def addEdge(self,f,t,weight=0):
```

```
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def __iter__(self):
        return iter(self.vertList.values())
###
###BFS,较近的点较快;DFS,稀疏图和较远点较快

###DFS最大联通面积
dx = [-1,0,1,1,-1,-1,0,1]
dy = [-1,-1,-1,0,0,1,1,1]

def can_visit(x,y) :
    return 0 <= x < n and 0 <= y < m and matrix[x][y] == 'W' and not in_queue[x][y]

def dfs(x,y) :
    global count
    in_queue[x][y] = True
    for i in range(8) :
        nx = x + dx[i]
        ny = y + dy[i]
        if can_visit(nx,ny) :
            count += 1
            dfs(nx,ny)

t = int(input())
for a in range(t) :

    n,m = map(int,input().split())
    matrix = [list(input()) for _ in range(n)]
    in_queue = [[False] * m for _ in range(n)]

    count = end = 0
    for i in range(n) :
        for j in range(m) :
            if matrix[i][j] == 'W' and not in_queue[i][j] :
                count = 1
                dfs(i,j)
                end = max(end,count)
    print(end)
###

###马走日，找可能的遍历方法数，DFS
move = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]

def check(q,p,x,y,visited):
    return 0<=x<q and 0<=y<p and not visited[x][y]

def dfs(q,p,x, y, visited):
    global ans
    global step
    for i in range(8):
        nx, ny = x + move[i][0], y + move[i][1]
        if check(q,p,nx,ny,visited):
            visited[nx][ny] = True
            step += 1
            if step == p * q:
                ans += 1
            dfs(q,p,nx, ny, visited)
            visited[nx][ny] = False
            step -= 1
```

```python
n = int(input())
for m in range(n):
    q, p, sx, sy = map(int, input().split())
    ans = 0
    step = 1
    visited = [[False] * p for _ in range(q)]
    visited[sx][sy] = True
    dfs(q,p,sx, sy, visited)
    print(ans)
###骑士周游（马走日的优化）DFS
def knight_tour(n, sr, sc):
    moves = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
             (1, -2), (1, 2), (2, -1), (2, 1)]

    visited = [[False] * n for _ in range(n)]

    def is_valid_move(row, col):
        return 0 <= row < n and 0 <= col < n and not visited[row][col]

    def count_neighbors(row, col):
        count = 0
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(next_row, next_col):
                count += 1
        return count

    def sort_moves(row, col):
        neighbor_counts = []
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(next_row, next_col):
                count = count_neighbors(next_row, next_col)
                neighbor_counts.append((count, (next_row, next_col)))
        neighbor_counts.sort()
        sorted_moves = [move[1] for move in neighbor_counts]
        return sorted_moves

    visited[sr][sc] = True
    tour = [(sr, sc)]

    while len(tour) < n * n:
        current_row, current_col = tour[-1]
        sorted_next_moves = sort_moves(current_row, current_col)
        if not sorted_next_moves:
            return "fail"
        next_row, next_col = sorted_next_moves[0]
        visited[next_row][next_col] = True
        tour.append((next_row, next_col))

    return "success"

n = int(input())
sr, sc = map(int, input().split())
print(knight_tour(n, sr, sc))
###骑士变体，输出字典序最小路径，舍弃优化，用数组（可覆盖）存储路径！！！
move = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]

def is_valid(x, y, q, p, visited):
    return 0<=x<q and 0<=y<p and not visited[x][y]

def dfs(x, y, step, q, p, visited, ans):
    if step == p * q:
        return True
    for dx,dy in move:
```

```
        nx, ny = x + dx, y + dy
        if is_valid(nx, ny, q, p, visited):
            visited[nx][ny] = True
            ans[step] = chr(nx + 65) + str(ny+1)
            if dfs(nx, ny, step + 1, q, p, visited, ans):
                return True
            visited[nx][ny] = False
    return False

n = int(input())
for m in range(n):
    p, q = map(int, input().split())
    ans = ["" for _ in range(p * q)]
    visited = [[False] * p for _ in range(q)] #(q,p)
    visited[0][0] = True
    ans[0] = 'A1'
    if dfs(0, 0, 1, q, p, visited, ans):
        result = "".join(ans)
    else:
        result = "impossible"
    print(f"Scenario #{m+1}:")
    print(result)
    print()
###

###经典DFS，迷宫问题，寻找多少种出去方法
dx = [-1, 0, 1, 0]
dy = [0, 1, 0, -1]
def dfs(maze, x, y):
    global cnt
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if maze[nx][ny] == 'e':
            cnt += 1
            continue
        if maze[nx][ny] == 0:
            maze[x][y] = 1      #这三行很重要！！！
            dfs(maze, nx, ny)   #这三行很重要！！！
            maze[x][y] = 0      #这三行很重要！！！
    return
n, m = map(int, input().split())
maze = []
maze.append([-1 for x in range(m+2)])
for _ in range(n):
    maze.append([-1] + [int(_) for _ in input().split()] + [-1])
maze.append([-1 for x in range(m+2)])
maze[1][1] = 's'
maze[n][m] = 'e'
cnt = 0
dfs(maze, 1, 1)
print(cnt)
###

###八皇后(n皇后)(栈)
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)
###

###走山路（BFS），堆
from heapq import heappop, heappush

def bfs(x1, y1):
    q = [(0, x1, y1)]
    v = set()
    while q:
        t, x, y = heappop(q)
        v.add((x, y))
        if x == x2 and y == y2:
            return t
        for dx, dy in dir:
            nx, ny = x+dx, y+dy
            if 0 <= nx < m and 0 <= ny < n and ma[nx][ny] != '#' and (nx, ny) not in v:
                nt = t+abs(int(ma[nx][ny])-int(ma[x][y]))
                heappush(q, (nt, nx, ny))
    return 'NO'

m, n, p = map(int, input().split())
ma = [list(input().split()) for _ in range(m)]
dir = [(1, 0), (-1, 0), (0, 1), (0, -1)]
for _ in range(p):
    x1, y1, x2, y2 = map(int, input().split())
    if ma[x1][y1] == '#' or ma[x2][y2] == '#':
        print('NO')
        continue
    print(bfs(x1, y1))
###

###BFS受限层号的顶点数（本题有向图，无向图稍加修改即可）
from collections import deque

def bfs(n, m, s, k, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
```

```
        graph[u].append(v)  # 只按照输入的方向添加边

    distance = [-1] * n
    distance[s] = 0

    queue = deque([s])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if distance[neighbor] == -1:
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)

    return sum(1 for d in distance if d <= k and d != -1)

n, m, s, k = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

count = bfs(n, m, s, k, edges)
print(count)
###

###词梯BFS（两个步骤，建图，bfs建轨迹列表）
from collections import deque

def buildgraph(ls):
    g = {}
    for w in ls:
        for i in range(len(w)):
            w_ = w[:i] + '_' + w[i+1:]
            g.setdefault(w,[]).append(w_)
            g.setdefault(w_,[]).append(w)
    return g

def bfs(sv,ev,g):
    stack = deque([(sv,[sv])])
    visited = set(sv)

    while stack:
        word, path = stack.popleft()
        if word == ev:
            return path

        for i in range(len(word)):
            word_ = word[:i] + '_' + word[i+1:]
            if word_ in g:
                for children in g[word_]:
                    if children not in visited:
                        visited.add(children)
                        stack.append([children,path+[children]])
    return

def find(sv,ev,ls):
    g = buildgraph(ls)
    return bfs(sv,ev,g)

ls = []
for _ in range(int(input())):
    ls.append(input())

start_v, end_v = input().split()
```

```
mark = find(start_v,end_v,ls)
if mark:
    print(' '.join(mark))
else:
    print('NO')
###

###判断无向图是否连通有无回路
from collections import deque

def connected(graph,n):
    visited = [False] * n
    stack = deque()
    stack.append(0)

    while stack:
        v = stack.popleft()
        visited[v] = True

        for nei in graph[v]:
            if visited[nei]:
                continue

            visited[nei] = True
            stack.append(nei)

    return all(visited)

def loop(graph,n):
    visited = [0] * n

    def dfs(u,visited,graph,father):
        if visited[u] == 1:
            return True

        if visited[u] == 2:
            return False

        visited[u] = 1
        for nei in graph[u]:
            if nei != father:
                if dfs(nei,visited,graph,u):
                    return True
        visited[u] = 2
        return False

    for i in range(n):
        if visited[i] == 0:
            if dfs(i,visited,graph,-1):
                return True

    return False

n,m = map(int,input().split())
graph = [[] for _ in range(n)]
for i in range(m):
    u,v = map(int,input().split())
    graph[u].append(v)
    graph[v].append(u)

connected = connected(graph,n)
loop = loop(graph,n)
###

###Dijkstra,用堆计算从一个固定顶点出发到图中所有其他顶点的最短路径。
```

```python
import heapq

def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges: #两顶点及边权重
        graph[u].append((v, w))
        graph[v].append((u, w))

    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
    distances[s] = 0

    while pq:
        dist, node = heapq.heappop(pq)
        if node == t:
            return dist
        if node in visited:
            continue
        visited.add(node)
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                new_dist = dist + weight
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))
    return -1 #无法到达，输出-1

# Read input 顶点数，边数，起始序号，终止序号
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]

# Solve the problem and print the result
result = dijkstra(n, edges, s, t)
print(result)
###

###Dijkstra变体，《电话线路》，使路径上第k+1个数最小，并输出这个数。
import heapq

def d(g,dists):
    q = []
    dists[1][0] = 0
    heapq.heappush(q,(0,1,0)) #dist,node,mark
    while q:
        dist,node,mark = heapq.heappop(q)
        if vis[node][mark]:
            continue
        vis[node][mark] = True
        for nei,w in g[node]:
            if dists[nei][mark] > max(dist,w):
                dists[nei][mark] = max(dist,w)
                heapq.heappush(q,(dists[nei][mark],nei,mark))
            if mark < k and dists[nei][mark+1] > dist:
                dists[nei][mark+1] = dists[node][mark]
                heapq.heappush(q,(dists[nei][mark+1],nei,mark+1))

n,p,k = map(int,input().split())
g = {_:[] for _ in range(n+1)}
vis = [[False]*(k+1) for _ in range(n+1)]
dists = [[float('inf')]*(k+1) for _ in range(n+1)]
for i in range(p):
    a,b,l = map(int,input().split())
    g[a].append((b,l))
    g[b].append((a,l))
```

```
d(g,dists)
ans = float('inf')
for i in range(k+1):
    ans = min(dists[n][i],ans)
print(ans if ans != float('inf') else -1)
###

###Prime(稠密图),用堆构建最小生成树，即连接图中所有顶点的一棵树，使得树的边权重之和最小。
import heapq

def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, vertex)
    min_spanning_tree_cost = 0

    while min_heap:
        weight, vertex = heapq.heappop(min_heap)

        if visited[vertex]:
            continue

        visited[vertex] = True
        min_spanning_tree_cost += weight

        for neighbor, neighbor_weight in graph[vertex]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_weight, neighbor))

    return min_spanning_tree_cost if all(visited) else -1 #不连通输出-1

def main():
    n, m = map(int, input().split()) #顶点数，边数
    graph = [[] for _ in range(n)]

    for _ in range(m):
        u, v, w = map(int, input().split()) #两顶点及边权重
        graph[u].append((v, w))
        graph[v].append((u, w))

    min_spanning_tree_cost = prim(graph, n)
    print(min_spanning_tree_cost)

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

###Kruskal(稀疏图),用并查集构建最小生成树。
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x] #注意不要偷懒写成x！！！

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[px] = py
            if self.rank[px] == self.rank[py]:
                self.rank[py] += 1
```

```
def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    res = 0
    for u, v, w in edges: #两顶点及边权重
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            res += w
    if len(set(uf.find(i) for i in range(n))) > 1:
        return -1
    return res

n, m = map(int, input().split()) #顶点数，边数
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))
print(kruskal(n, edges))
###

###拓扑排序，判断有向图是否存在环并输出学习课程顺序。
from collections import defaultdict

def courseSchedule(n, edges):
    graph = defaultdict(list)
    indegree = [0] * n
    for u, v in edges: #边的起点和终点
        graph[u].append(v)
        indegree[v] += 1

    queue = [i for i in range(n) if indegree[i] == 0]
    queue.sort()
    result = []

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

    if len(result) == n:
        return "Yes", result #输出学习顺序
    else:
        return "No", n - len(result) #输出不能学习的课程数目

n, m = map(int, input().split()) #顶点数，边数
edges = [list(map(int, input().split())) for _ in range(m)]
res, courses = courseSchedule(n, edges)
print(res)
if res == "Yes":
    print(*courses)
else:
    print(courses)
###
```

单调栈

```
###经典单调栈-接雨水
###三个量：i,index,stack[-1]
n = int(input())
ls = list(map(int,input().split()))

if n == 0:
    print(0)
else:
    stack = [] #存储序号
    ans = 0

    for i in range(n):
        while stack and ls[i] > ls[stack[-1]]:
            index = stack.pop()
            if not stack:
                break

            l = i - stack[-1] -1
            h = min(ls[stack[-1]] , ls[i]) - ls[index]
            ans += l*h

        stack.append(i)
    print(ans)
###

###经典单调栈-求最大矩形面积
###先转换成固定底的求最大矩形面积问题，再用单调栈解决
m,n = map(int,input().split()) #m行n列
g = [list(map(int,input().split())) for _ in range(m)]
end = 0

for i in range(n):
    #构建以i为底的高度list
    ls = [0 for _ in range(m)]
    for x in range(m):
        h = 0
        for y in range(i,n):
            if g[x][y] == 0:
                h += 1
            else:
                break
        ls[x] = h

    #求最大矩形面积问题
    def max_S(ls,m):
        left_tail = [0 for _ in range(m)] #左边界
        right_tail = [m-1 for _ in range(m)] #右边界

        stack_l = [] #存储序号
        stack_r = [] #存储序号

        for a in range(m):
            #找右边界
            while stack_r and ls[a] < ls[stack_r[-1]]:
                index = stack_r.pop()
                right_tail[index] = a
            stack_r.append(a)
            #找左边界
            b = m - 1 - a
            while stack_l and ls[b] < ls[stack_l[-1]]:
                index = stack_l.pop()
                left_tail[index] = b
            stack_l.append(b)
```

```
        ans = 0
        for j in range(m):
            cur = ls[j] * (right_tail[j] - left_tail[j] -1)
            ans = max(ans,cur)
        return ans

    end = max(end,max_S(ls,m))
print(end)
###

###奶牛排队，要求找出左低右高的最长序列（中间不与端点相同,不要求大小顺序）
N = int(input())
heights = [int(input()) for _ in range(N)]

left_bound = [-1] * N
right_bound = [N] * N

stack = []  #存储索引

#求左侧第一个≥h[i]的奶牛位置
for i in range(N):
    while stack and heights[stack[-1]] < heights[i]:
        stack.pop()

    if stack:
        left_bound[i] = stack[-1]

    stack.append(i)

#求右侧第一个≤h[i]的奶牛位
stack = []
for i in range(N-1, -1, -1):
    while stack and heights[stack[-1]] > heights[i]:
        stack.pop()

    if stack:
        right_bound[i] = stack[-1]

    stack.append(i)

ans = 0

for i in range(N):  # 枚举右端点 B寻找 A，更新 ans
    for j in range(left_bound[i] + 1, i):
        if right_bound[j] > i:
            ans = max(ans, i - j + 1)
            break
print(ans)
###
```

DP，二分查找

```
#汉诺塔，3根柱子时最少移动2^n-1次
def hanoi(n,a,b,c) :
    if n > 3 :
        b1,c1 = c,b
        hanoi(n-1,a,b1,c1)
        print(str(n)+':'+a+'->'+b1)
        a2,b2,c2 = b,a,c
        hanoi(n-1,a2,b2,c2)
    else:
        print(str(n-2)+':'+a+'->'+c)
        print(str(n-1)+':'+a+'->'+b)
        print(str(n-2)+':'+c+'->'+b)
```

```
        print(str(n)+':'+a+'->'+c)
        print(str(n-2)+':'+b+'->'+a)
        print(str(n-1)+':'+b+'->'+c)
        print(str(n-2)+':'+a+'->'+c)

ls = list(input().split())
n = int(ls[0])
a,b,c = ls[1],ls[2],ls[3]
hanoi(n,a,b,c)
#4柱汉诺塔的dp解
def hanoi4(n):
    h_list = [0] * (n + 1)

    def f(m):
        if h_list[m]: #这个if似乎并不需要?
            return h_list[m]
        result = 2 ** m - 1
        for x in range(1, m):
            result = min(result, 2 * f(x) + 2 ** (m - x) - 1)
        h_list[m] = result  #记录算过的值
       return result

    return f(n)
print(hanoi4(int(input())))

#dp经典例题，求某数的最大积分解
def max_product_partition(s):
    # 创建一个数组来存储最大乘积
    max_product = [0] * (s + 1)

    # 初始化数组，最大乘积为1
    max_product[1] = 1

    # 从2开始遍历到s
    for i in range(2, s + 1):
        # 遍历求解最大乘积
        for j in range(1, i):
            # 更新最大乘积
            max_product[i] = max(max_product[i], max(j, max_product[j]) * max(i - j,
max_product[i - j]))

    # 构造分解结果
    result = []
    i = s
    while i > 0:
        # 将当前最大乘积的因子加入结果数组
        result.append(max_product[i])
        # 更新i
        i -= max_product[i]

    # 输出结果
    print(*result)

# 例子输入
max_product_partition(7)

#小偷背包的经典问题（二维DP）
n,b = map(int,input().split())
values = list(map(int,input().split()))
weights = list(map(int,input().split()))
matrix = [[0 for i in range(b+1)] for j in range(n+1)]
for i in range(b+1) :
    matrix[0][i] = 0
for i in range(1,n+1) :
    for j in range(1,b+1) :
```

```
        if weights[i-1] <= j :
            matrix[i][j] = max(matrix[i-1][j],values[i-1]+matrix[i-1][j-weights[i-1]])
        else:
            matrix[i][j] = matrix[i-1][j]
print(matrix[-1][-1])

#月度开销，用二分查找找到最小的maxmax
def check(mid):
    num,cnt = 1,0
    for i in range(n):
        if cnt + ls[i] > mid:
            num += 1
            cnt = ls[i]
        else:
            cnt += ls[i]
    return False if num > m else True

n,m = map(int,input().split())
ls = [int(input()) for _ in range(n)]
maxmax = sum(ls)
minmax = max(ls)
while minmax < maxmax:
    mid = (minmax + maxmax)//2
    if check(mid):
        maxmax = mid
    else:
        minmax = mid + 1
print(maxmax)
#河中跳房子，同理，找出最大的maxmin
def check(mid):
    num,left = 0,0
    for i in range(N+1):
        if ls[i] - left <= mid:
            num += 1
        else:
            left = ls[i]

    return True if num <= M else False

L,N,M = map(int,input().split())
ls = [int(input()) for _ in range(N)] + [L]
minmin = 0
maxmin = L
while minmin < maxmin:
    mid = (minmin + maxmin)//2
    if check(mid):
        minmin = mid + 1
    else:
        maxmin = mid
print(maxmin)
```

笔试

#数据项是数据的最小单位，而数据元素（基本单位）是由一个或多个数据项组成的，它们共同构成了数据结构中的数据。

#存储密度=数据元素占的空间/（数据元素占的空间+指针占的存储空间）。

#把n个元素建立成一个单链表，O(n^2)。

#串是一种特殊的线性表，其特殊性体现在“数据元素是一个字符”。

#串“abcde”是由5个数据元素构成的，每个数据元素含一个字符，D意味着‘ab’或‘cde’作为一个数据元素，这是不正确的。

#二次聚集：争夺下一个地址。

#二次探测法：当冲突时，我们会尝试在哈希值加上探测次数的平方（即 $H(key)+i^2$ ）的位置插入元素，其中i是探测次数。

#二叉堆BinHeap，考试时直接import heapq即可。

#堆调整：把最后的拉上来，再调整。

#快排先排左边再排右边，每轮走1/n个数组，希尔每轮走一整个数组。

#二分查找中栈的使用情况：栈的容量即为递归次数，为$\lceil \log_2(N) \rceil + 1$ 。

#1到n的二叉搜索树的中序排列就是1,2,3,4,5...n。

#二叉搜索树实现快排，即输出二叉搜索树的中序表达式。

#二叉树公式：$n_0 = n_2 + 1, n = n_0 + n_1 + n_2$ 联立这两个方程即可。

#森林->二叉树：先把树变成二叉树，再把每个树接在前一个树根的右子结点处。

#对于二叉搜索树（BST），如果树是平衡的，那么在平均情况下，查找、插入和删除的时间复杂度也是 $O(\log n)$。但是，如果树不平衡，例如退化为链表，那么时间复杂度可能会变为 $O(n)$。

#平衡二叉树：平衡因子：左减右。

#m阶B-树：非叶结点至少有ceil(m/2)棵子树。

#广度优先用队列，深度优先用栈。

#回溯算法：DFS，用栈。

#Prime 构造稠密图的最小生成树

#Kruskal 构造稀疏图的最小生成树

#Dijkstra 求非负权图最短路径

#Floyd 求负权图最短路径

#考察某个具体问题是否适合应用动态规划算法，必须判定它是否具有最优子结构性质。