并查集的类定义

```
class disjointset():
    def __init__(self,size):
        self.parent=list(range(size))
        self.rank=[0]*(size)
    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):
        x_root=self.find(x)
        y_root=self.find(y)
        if x_root==y_root:
            return
        elif self.rank[x_root]>self.rank[y_root]:
            self.parent[y_root]=x_root
        elif self.rank[x_root]<self.rank[y_root]:
            self.parent[x_root]=y_root
        else:
            self.parent[x_root]=y_root
            self.rank[y_root]+=1
    def connected(self,x,y):
        return self.find(x)==self.find(y)
```

二叉树/森林的类定义与相关函数

```
class Treenode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
class Tree:
    def __init__(self,val):
        self.val=val
        self.children=[]
# 前中后序表达式
def preorder_traversal(root):
    if root is None:
        return ''
    return root.val+preorder_traversal(root.left)+preorder_traversal(root.right)
def inorder_traversal(root):
    if root is None:
        return ''
    return inorder_traversal(root.left)+root.val+inorder_traversal(root.right)
def postorder_traversal(root):
    if root is None:
        return ''
    return
postorder_traversal(root.left)+postorder_traversal(root.right)+root.val
# 二叉树的深度
def depth(root):
    if root is None:
        return 0
    l=depth(root.left)
    r=depth(root.right)
    return max(l,r)+1
# 层序遍历
def level_traversal(root):
    level=[root]
    ans=[]
    while level:
        lst_val=[]
        next_level=[]
        for node in level:
            lst_val.append(node.val)
            if node.left:
                next_level.append(node.left)
            if node.right:
                next_level.append(node.right)
        level=next_level
        ans+=lst_val
    return ' '.join(map(str,ans))
# 建立一颗二叉搜索树
def build_tree(lst):
    if not lst:
        return None
    root=TreeNode(lst[0])
    num=lst[0]
    id=len(lst)
    for i in range(1,len(lst)):
        if lst[i]>num:
            id=i
            break
    left_lst=lst[1:id]
    right_lst=lst[id:]
    root.left=build_tree(left_lst)
    root.right=build_tree(right_lst)
    return root
# 根据前中序建树
def buildTree(preorder,inorder):
    if not preorder:
        return None
    root=Treenode(preorder[0])
    for i in range(len(inorder)):
        if inorder[i]==root.val:
            id=i
            break
    left_preorder=preorder[1:id+1]
    right_preorder=preorder[id+1:]
    left_inorder=inorder[:id]
    right_inorder=inorder[id+1:]
    root.left=buildTree(left_preorder,left_inorder)
    root.right=buildTree(right_preorder,right_inorder)
    return root
# 根据后序表达式建树 （？）
def build_tree(s):
    stack=[]
    for char in s:
```

```python
        if 97<=ord(char)<=122:
            stack.append(TreeNode(char))
        else:
            right_node=stack.pop()
            left_node=stack.pop()
            root=TreeNode(char)
            root.left=left_node
            root.right=right_node
            stack.append(root)
    return root
# 求叶子数目
def count_leaves(node):
    if not node:
        return 0
    if not node.left and not node.right:
        return 1
    return count_leaves(node.left)+count_leaves(node.right)
# 最近公共祖先，递归
def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if root in (None,p,q):
            return root
        left=self.lowestCommonAncestor(root.left,p,q)
        right=self.lowestCommonAncestor(root.right,p,q)
        if left and right:
            return root
        return left or right

# 路径总和
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        ans=0
        cnt=defaultdict(int)
        cnt[0]=1
        def dfs(node,s):
            nonlocal ans
            if node is None:
                return
            s+=node.val
            ans+=cnt[s-targetSum]
            cnt[s]+=1
            dfs(node.left,s)
            dfs(node.right,s)
            cnt[s]-=1
        dfs(root,0)
        return ans
```

树状dp

```python
# 最大路径和
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        ans = -inf

        def dfs(node):
            nonlocal ans
            if node is None:
                return 0
            l_val = dfs(node.left)
            r_val = dfs(node.right)
            ans = max(ans, l_val + r_val + node.val)
            return max(max(l_val, r_val) + node.val, 0)

        dfs(root)
        return ans
# 宝藏二叉树
def find_max_value(root):
    if root is None:
        return 0
    a = find_max_value(root.left) + find_max_value(root.right)
    b = root.val
    if root.left:
        b += find_max_value(root.left.left) + find_max_value(root.left.right)
    if root.right:
        b += find_max_value(root.right.left) + find_max_value(root.right.right)
    return max(a, b)
```

字典树

```python
class TrieNode:
    def __init__(self):
        self.children={}
        self.is_end_of_word=False

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

    def insert(self,word):
        node=self.root
        for char in word:
            if char not in node.children:
                node.children[char]=TrieNode()
            node=node.children[char]
        node.is_end_of_word=True
    def start_with(self,word):
        node=self.root
        for char in word:
            if char not in node.children:
                return False
            node=node.children[char]
        return True
     def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
    def find_all_with_prefix(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]
        return self._dfs(node, prefix)
    def _dfs(self, node, current_word):
        result = []
        if node.is_end_of_word:
            result.append(current_word)
        for char, child_node in node.children.items():
            result.extend(self._dfs(child_node, current_word + char))
        return result
```

huffman编码树

```
import heapq
class Node:
    def __init__(self,weight,char):
        self.left=None
        self.right=None
        self.char=char
        self.weight=weight
    def __lt__(self,other):
        if self.weight==other.weight:
            return self.char<other.char
        return self.weight<other.weight

def buildHuffmanTree(lst):
    q=[]
    for weight,char in lst:
        heapq.heappush(q,Node(weight,char))
    while len(q)>=2:
        left=heapq.heappop(q)
        right=heapq.heappop(q)
        merged=Node(left.weight+right.weight,min(left.char,right.char))
        merged.left=left
        merged.right=right
        heapq.heappush(q,merged)
    return q[0]

def encodeHuffmanTree(root):
    codes={}
    def traversal(node,code):
        if node.left is None and node.right is None:
            codes[node.char]=code
        else:
            traversal(node.left,code+'0')
            traversal(node.right,code+'1')
    traversal(root,'')
    return codes

def decodeHuffmanTree(root,s):
    ans=''
    node=root
    for bit in s:
        if bit=='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

lst=[]
for _ in range(int(input())):
    char,weight=input().split()
    lst.append((int(weight),char))
HuffmanTree=buildHuffmanTree(lst)
codes=encodeHuffmanTree(HuffmanTree)

while True:
    try:
        s=input()
        if s[0] in {'0','1'}:
            print(decodeHuffmanTree(HuffmanTree,s))
        else:
            print(''.join(codes[c] for c in s))
    except EOFError:
        break
```

实现堆结构

```
class Heap:
    def __init__(self):
        self.q=[0]
        self.size=0

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

    def insert(self,num):
        self.q.append(num)
        self.size+=1
        self.percUp(self.size)

    def minChild(self,i):
        if 2*i+1>self.size:
            return 2*i
        if self.q[2*i]<self.q[2*i+1]:
            return 2*i
        return 2*i+1

    def percDown(self,i):
        while 2*i<=self.size:
            k=self.minChild(i)
            if self.q[i]>self.q[k]:
                self.q[i],self.q[k]=self.q[k],self.q[i]
            i=k

    def popMin(self):
        self.q[1],self.q[self.size]=self.q[self.size],self.q[1]
        m=self.q.pop()
        self.size-=1
        self.percDown(1)
        return m
```

*linked list:

```
#快慢指针:
slow,fast=head,head
while fast.next and fast.next.next:
    slow=slow.next
    fast=fast.next.next
#slow此时是中偏左位置
slow=slow.next
```

```
# 反转链表:
# 单链表:
def fan(head):
    pre,cur=None,head
    while cur:
        tmp=cur.next
        cur.next=pre
        pre=cur
        cur=tmp
    return pre
# 双链表:
def fan(head):
    pre,nt=None,None
    while head!=None:
        nt=head.next
        head.next=pre
        head.last=nt#last表示上一个
        pre=head
        head=nt
    return pre
```

```
# 链表判断环:
def detectCycle(head):
    if head is None or head.next is None or head.next.next is None:
        return None
    slow=head.next
    fast=head.next.next
    while slow!=fast:
        if fast.next is None or fast.next.next is None:
            return None
        slow=slow.next
        fast=fast.next.next
    fast=head
    while slow!=fast:
        slow=slow.next
        fast=fast.next
    return slow
```

最小生成树

```
#prim算法，用于边密集，从一个点开始
import heapq
while True:
    try:
        n = int(input())
        distance = []
        for _ in range(n):
            distance.append(list(map(int, input().split())))
        visited = [False] * n
        visited[0] = True
        q=[]
        mst = []
        for v in range(1, n):
            heapq.heappush(q, (distance[0][v], v))
        while len(mst) < n - 1:
            dist, u = heapq.heappop(q)
            if not visited[u]:
                visited[u] = True
                mst.append(dist)
                for v in range(n):
                    if v != u and not visited[v]:
                        heapq.heappush(q, (distance[u][v], v))
        print(sum(mst))
    except EOFError:
        break
# Krustal算法，用于边稀疏
class disjointset:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size

    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):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return False
        elif self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        elif self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[x_root] = y_root
            self.rank[y_root] += 1
        return True

n = int(input())
obj = disjointset(n)
edge = []
for _ in range(n - 1):
    lst = list(input().split())
    u = lst[0]
    while len(lst) > 2:
        dist = lst.pop()
        v = lst.pop()
        edge.append((int(dist), ord(u) - 65, ord(v) - 65))
ans, cnt = 0, 0
edge.sort()
for dist, u, v in edge:
    if obj.union(u, v):
        ans += dist
        cnt += 1
    if cnt == n - 1:
        break
print(ans)
```

三色标记法判断图中是否有有向圈

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        neighbors={k:[] for k in range(numCourses)}
        for a,b in prerequisites:
            neighbors[a].append(b)
        visited=[0]*numCourses
        ans=[]
        def dfs(x):
            visited[x]=1
            for y in neighbors[x]:
                if visited[y]==1 or visited[y]==0 and dfs(y):
                    return True
            visited[x]=2
            ans.append(x)
            return False
        for x,visit in enumerate(visited):
            if visit==0 and dfs(x):
                return []
        return ans
```

单调栈

```python
# 最小新整数
def removeKDigits(num, k):
    stack = []
    for digit in num:
        while k and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    while k:
        stack.pop()
        k -= 1
    return int("".join(stack))


t = int(input())
results = []
for _ in range(t):
    n, k = input().split()
    results.append(removeKDigits(n, int(k)))
for result in results:
    print(result)

# 求柱状图中的最大矩形
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n=len(heights)
        left=[-1]*len(heights)
        stack=[]
        for i in range(n):
            while stack and heights[i]<=heights[stack[-1]]:
                stack.pop()
            if stack:
                left[i]=stack[-1]
            stack.append(i)
        right=[n]*len(heights)
        stack=[]
        for i in range(n-1,-1,-1):
            while stack and heights[i]<=heights[stack[-1]]:
                stack.pop()
            if stack:
                right[i]=stack[-1]
            stack.append(i)
        ans=0
        for h,l,r in zip(heights,left,right):
            ans=max(ans,h*(r-l-1))
        return ans
```

归并排序求逆序对

```
def merge(a,b):
    c,cnt,i,j=[],0,0,0
    while i<len(a) and j<len(b):
        if a[i]>=b[j]:
            c.append(b[j])
            j+=1
            cnt+=len(a)-i
        else:
            c.append(a[i])
            i+=1
    c.extend(a[i:])
    c.extend(b[j:])
    return c,cnt

def merge_sort(lst,ll,rr):
    if rr-ll<=1:
        return 0
    mid=(ll+rr)//2
    l_cnt=merge_sort(lst,ll,mid)
    r_cnt=merge_sort(lst,mid,rr)
    lst[ll:rr],cnt=merge(lst[ll:mid],lst[mid:rr])
    return l_cnt+r_cnt+cnt
```

骑士周游+Warnsdoff

```
def dfs(x, y, n, path, visited):
    if len(path) == n**2:
        return "success"
    candidates=[]
    for dx, dy in direction:
        nx, ny = x + dx, y + dy
        if 0 <= nx <= n - 1 and 0 <= ny <= n - 1 and not visited[nx][ny]:
            degree=0
            for ddx,ddy in direction:
                nnx,nny=nx+ddx,ny+ddy
                if 0<=nnx<=n-1 and 0<=nny<=n-1 and not visited[nnx][nny]:
                    degree+=1
            candidates.append((degree,nx,ny))
    candidates.sort()
    for degree,nx,ny in candidates:
        path.append((nx, ny))
        visited[nx][ny] = True
        result = dfs(nx, ny, n, path, visited)
        if result != "fail":
            return result
        nx, ny = path.pop()
        visited[nx][ny] = False
    return "fail"
```

中序表达式转后序表达式

```python
def split_expression(string):
    separators=['(',')','+','-','*','/']
    result=[]
    num=''
    for char in string:
        if char in separators:
            if num:
                result.append(num)
                num=''
            result.append(char)
        else:
            num+=char
    if num:
        result.append(num)
    return result

def transform(string):
    result=[]
    string=split_expression(string)
    priority={'(':1,')':1,'*':3,'/':3,'+':2,'-':2}
    operator = ["(", ")", "+", "-", "*", "/"]
    opstack=[]
    for char in string:
        if char in operator:
            if char=='(':
                opstack.append(char)
            elif char==')':
                while opstack[-1]!='(': # 此时要把括号内的所有运算符全部输出后再继续操作
                    result.append(opstack.pop())
                opstack.pop()
            else:
                while opstack and priority[char]<=priority[opstack[-1]]: # 弹出栈
顶优先级更高的运算符
                    result.append(opstack.pop())
                opstack.append(char)
        else:
            result.append(char)
    while opstack:
        result.append(opstack.pop())
    return result

n=int(input())
lst=[]
for _ in range(n):
    lst.append(input())
for expression in lst:
    print(' '.join(transform(expression)))
```

拓扑排序+dp

```
n, m = map(int, input().split())
neighbors, parents = {k: [] for k in range(n)}, {k: [] for k in range(n)}
degree = [0] * n
for _ in range(m):
    a, b = map(int, input().split())
    neighbors[a].append(b)
    parents[b].append(a)
    degree[b] += 1
see = set()
ans = 100 * n
topology_order = []
while len(topology_order) < n:
    lst = []
    for u in range(n):
        if degree[u] == 0 and u not in topology_order:
            lst.append(u)
    for u in lst:
        topology_order.append(u)
        for v in neighbors[u]:
            degree[v] -= 1
dp = [0] * n
for u in topology_order[::-1]:
    for v in parents[u]:
        if dp[v] <= dp[u]:
            dp[v] = dp[u] + 1

ans += sum(dp)
print(ans)
```

回溯几个模板

```
# 解数独
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        nums={'1','2','3','4','5','6','7','8','9'}
        row=[set() for _ in range(9)]
        col=[set() for _ in range(9)]
        lattice=[[set() for _ in range(3)] for _ in range(3)]
        blank=[]
        for i in range(9):
            for j in range(9):
                num=board[i][j]
                if num=='.':
                    blank.append((i,j))
                else:
                    row[i].add(num)
                    col[j].add(num)
                    lattice[i//3][j//3].add(num)
        def dfs(n):
            if n==len(blank):
                return True
            i,j=blank[n]
            lst=nums-row[i]-col[j]-lattice[i//3][j//3]
            if not lst:
                return False
            for num in lst:
                board[i][j]=num
                row[i].add(num)
                col[j].add(num)
                lattice[i//3][j//3].add(num)
                if dfs(n+1):
                    return True
                row[i].remove(num)
                col[j].remove(num)
                lattice[i//3][j//3].remove(num)
        dfs(0)
# 全排列
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        ans,path=[],[]
        def dfs(i,s):
            if i==len(nums):
                ans.append(path[:])
                return
            for x in s:
                path.append(x)
                dfs(i+1,s-{x})
                path.pop()
        dfs(0,set(nums))
        return ans
# 单词查找
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        m,n=len(board),len(board[0])
        direction=[(1,0),(0,1),(-1,0),(0,-1)]
        def dfs(x,y,k):
            if board[x][y]!=word[k]:
                return False
            if k==len(word)-1:
                return True
            board[x][y]="."
            for dx,dy in direction:
                nx,ny=x+dx,y+dy
                if 0<=nx<=m-1 and 0<=ny<=n-1 and dfs(nx,ny,k+1):
                    return True
            board[x][y]=word[k]
            return False
        return any(dfs(i,j,0) for i in range(m) for j in range(n))
# 分割回文串
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        ans,path=[],[]
        n=len(s)
        def dfs(i):
            if i==n:
                ans.append(path[:])
            for j in range(i,n):
                t=s[i:j+1]
                if t==t[::-1]:
                    path.append(t)
                    dfs(j+1)
                    path.pop()
        dfs(0)
        return ans
# 八皇后
def is_ok(placed, row):  # 第i+1行放置的皇后位置为placed[i]+1,判断是否能在col+1行row+1列放置皇后
    col = len(placed)
    for i in range(col):
        if placed[i] == row or placed[i] == col + row - i or placed[i] == row - col + i:
            return False
    return True

def queen(placed):
    if len(placed) == 8:
        ans.append("".join(str(placed[i] + 1) for i in range(8)))
        return

    for row in range(8):
        if is_ok(placed, row):
            placed.append(row)
            queen(placed)
            placed.pop()
```

floyd（可以处理负权边，但不能处理负权环，负权环用bellman ford判定）

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [row[:] for row in graph]  # 深拷贝初始图矩阵

    for k in range(n):        # 中间点
        for i in range(n):    # 起点
            for j in range(n):  # 终点
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

任意多组数据接收

```
while True:
    try:

    except EOFError:
        break
```

质数筛法

```
def prime_set(max_):
    is_prime=[True]*(max_+1)
    is_prime[0]=is_prime[1]=False
    p=2
    while p*p<=max_:
        if is_prime[p]:
            for i in range(p*p,max_+1,p):
                is_prime[i]=False
        p+=1
    return {a for a, b in enumerate(is_prime) if b}
```

kmp

```
def build_next(pattern):
    next_array = [0] * len(pattern)
    j = 0  # 当前最长前缀长度
    for i in range(1, len(pattern)):
        while j > 0 and pattern[i] != pattern[j]:
            j = next_array[j - 1]  # 回溯
        if pattern[i] == pattern[j]:
            j += 1
        next_array[i] = j
    return next_array
def kmp_search(text, pattern):
    next_array = build_next(pattern)
    i, j = 0, 0  # i: 主串指针, j: 模式串指针
    while i < len(text):
        if j == 0 or text[i] == pattern[j]:
            i += 1
            j += 1
            if j == len(pattern):  # 匹配成功
                return i - j
        else:
            j = next_array[j - 1]  # 根据next数组调整模式串位置
    return -1  # 匹配失败

# 前缀中的周期
import sys

def build_next(pattern):
    n = len(pattern)
    next_array = [0] * n
    j = 0
    for i in range(1, n):
        while j > 0 and pattern[i] != pattern[j]:
            j = next_array[j - 1]
        if pattern[i] == pattern[j]:
            j += 1
        next_array[i] = j
    return next_array

def main():
    test_case = 1
    while True:
        n_line = sys.stdin.readline()
        if not n_line:
            break
        n = int(n_line.strip())
        if n == 0:
            break
        s = sys.stdin.readline().strip()
        next_array = build_next(s)
        print(f"Test case #{test_case}")
        for i in range(2, n + 1):
            len_val = next_array[i - 1]
            if len_val == 0:
                continue
            d = i - len_val
            if i % d == 0:
                print(f"{i} {i // d}")
        print()
        test_case += 1

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

*Kadane's(最大子数组)

```
def max_subarray_sum(arr):
    if not arr:
        return 0
    max_current=max_global=arr[0]
    for num in arr[1:]:
        max_current =max(num,max_current+num)
        if max_current>max_global:
            max_global= max_current
    return max_global
```

推广：最大子矩阵

```
'''
为了找到最大的非空子矩阵，可以使用动态规划中的Kadane算法进行扩展来处理二维矩阵。
基本思路是将二维问题转化为一维问题：可以计算出从第i行到第j行的列的累计和，
这样就得到了一个一维数组。然后对这个一维数组应用Kadane算法，找到最大的子数组和。
通过遍历所有可能的行组合，我们可以找到最大的子矩阵。
'''
def max_submatrix(matrix):
    def kadane(arr):
        # max_ending_here 用于追踪到当前元素为止包含当前元素的最大子数组和。
        # max_so_far 用于存储迄今为止遇到的最大子数组和。
        max_end_here = max_so_far = arr[0]
        for x in arr[1:]:
            # 对于每个新元素，我们决定是开始一个新的子数组（仅包含当前元素 x），
            # 还是将当前元素添加到现有的子数组中。这一步是 Kadane 算法的核心。
            max_end_here = max(x, max_end_here + x)
            max_so_far = max(max_so_far, max_end_here)
        return max_so_far

    rows = len(matrix)
    cols = len(matrix[0])
    max_sum = float('-inf')

    for left in range(cols):
        temp = [0] * rows
        for right in range(left, cols):
            for row in range(rows):
                temp[row] += matrix[row][right]
            max_sum = max(max_sum, kadane(temp))
    return max_sum

n = int(input())
nums = []

while len(nums) < n * n:
    nums.extend(input().split())
matrix = [list(map(int, nums[i * n:(i + 1) * n])) for i in range(n)]

max_sum = max_submatrix(matrix)
print(max_sum)
```

*Manacher:

```
def manacher(s):
    ss= '#' + '#'.join(s) + '#'
    n=len(ss)
    p=[0]*n
    ans=0
    c,r=0,0
    for i in range(n):
        length=min(p[2*c-i],r-i) if r>i else 1
        while i+length<n and i-length>=0 and ss[i + length]==ss[i - length]:
            length+=1
        if i+length>r:
            r=i+length
            c=i
        ans=max(ans,length)
        p[i]=length
    return ans-1
```

常用库

```
#队列(default字典)
from collections import deque(defaultdict)
#递归上限
from sys import setrecursionlimit
#数学***math库***：最常用的sqrt,对数log(x[,base])、三角sin()、反三角asin()也都有；还有e,pi等常数，inf表示无穷大；返回小于等于x的最大整数floor（）,大于等于ceil（）,判断两个浮点数是否接近isclose（a，b，*， rel_tol=1e-09， abs_tol=0.0）；一般的取幂pow（x，y）,阶乘factorial（x）如果不符合会ValueError,组合数comb（n，k）`math.radians()`将度数转换为弧度，或者使用`math.degrees()`将弧度转换为度数。
import math
#二分库
import bisect
bisect.bisect_right(a,6)#返回在a列表中若要插入6的index（有重复数字会插在右边）
bisect.insort(a,6)#返回插入6后的列表a
#conuter
from collections import Counter
```

小数输出

```
print(f"{ans:.3f}")
```

修改递归深度&lru

```
import sys
sys.setrecursionlimit(10**6)

from functools import lru_cache
@lru_cache(maxsize=None)
def dfs():
    ...
```

lambda函数

1. 按字符串的长度排序

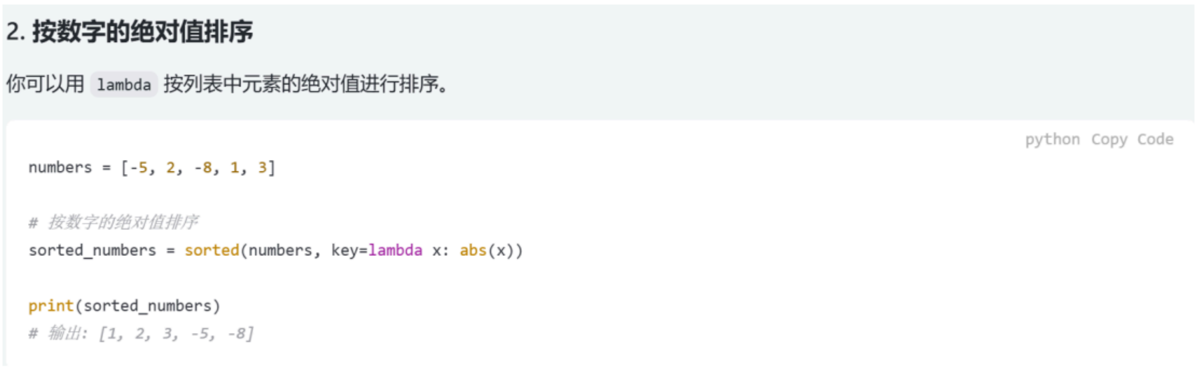
如果你有一个包含字符串的列表，可以使用 `lambda` 按字符串的长度来排序。

python Copy Code

```python
words = ['apple', 'banana', 'cherry', 'date']

# 按照字符串的长度排序
sorted_words = sorted(words, key=lambda x: len(x))

print(sorted_words)
# 输出: ['date', 'apple', 'cherry', 'banana']
```



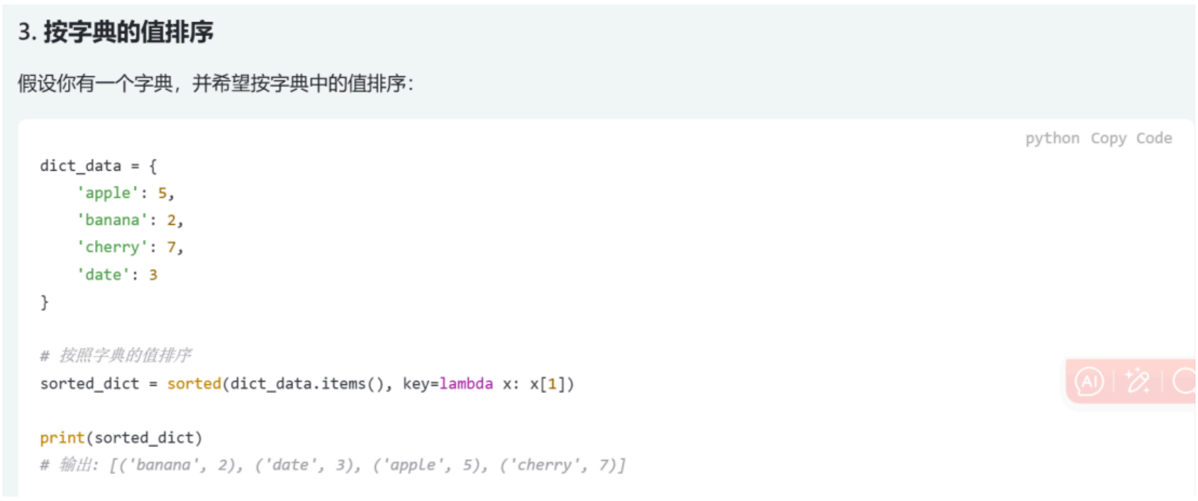
2. 按数字的绝对值排序

你可以用 `lambda` 按列表中元素的绝对值进行排序。

```python
numbers = [-5, 2, -8, 1, 3]

# 按数字的绝对值排序
sorted_numbers = sorted(numbers, key=lambda x: abs(x))

print(sorted_numbers)
# 输出: [1, 2, 3, -5, -8]
```



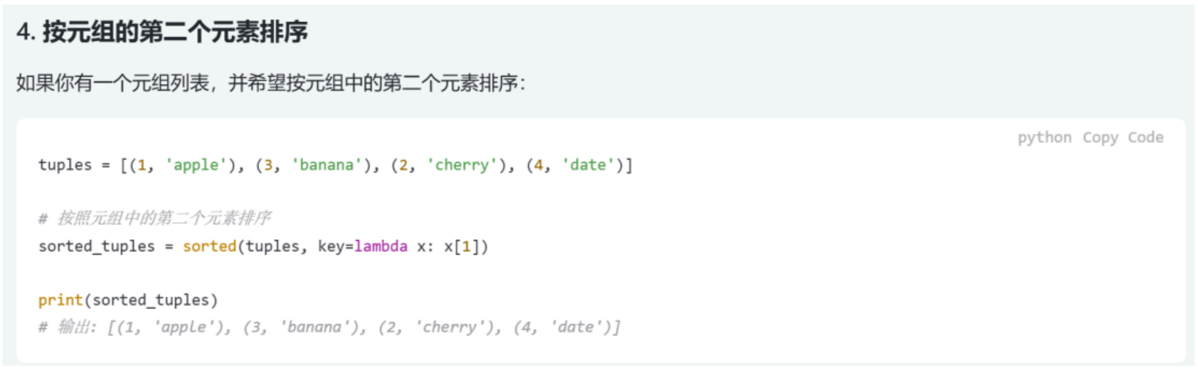
3. 按字典的值排序

假设你有一个字典，并希望按字典中的值排序：

```python
dict_data = {
    'apple': 5,
    'banana': 2,
    'cherry': 7,
    'date': 3
}

# 按照字典的值排序
sorted_dict = sorted(dict_data.items(), key=lambda x: x[1])

print(sorted_dict)
# 输出: [('banana', 2), ('date', 3), ('apple', 5), ('cherry', 7)]
```



4. 按元组的第二个元素排序

如果你有一个元组列表，并希望按元组中的第二个元素排序：

```python
tuples = [(1, 'apple'), (3, 'banana'), (2, 'cherry'), (4, 'date')]

# 按照元组中的第二个元素排序
sorted_tuples = sorted(tuples, key=lambda x: x[1])

print(sorted_tuples)
# 输出: [(1, 'apple'), (3, 'banana'), (2, 'cherry'), (4, 'date')]
```

OJ的pylint是静态检查，有时候报的compile error不对。解决方法有两种，如下：

1）第一行加# pylint: skip-file

2）方法二：如果函数内使用全局变量（变量类型是immutable，如int），则需要在程序最开始声明一下。如果是全局变量是list类型，则不受影响。