1.排序算法

```
#插入排序（稳定）
def insertion_sort(arr):
    for i in range(1,len(arr)):
        j = i
        while j >= 1 and arr[j-1] > arr[j]:
            arr[j-1],arr[j] = arr[j],arr[j-1]
            j = j-1
#冒泡排序（稳定）
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        swap = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                swap = True
                arr[j],arr[j+1] = arr[j+1],arr[j]
        if swap is False:
            break
#选择排序（不稳定）
def selection_sort(arr):
    n = len(arr)
    for i in range(n-1):
        idx = i
        for j in range(i+1,n):
            if arr[idx] > arr[j]:
                idx = j
        arr[i],arr[idx] = arr[idx],arr[i]
#快速排序（不稳定）
def quick_sort(left,right,arr):
    if left<right:
        p = partition(left,right,arr)
        quick_sort(left,p-1,arr)
        quick_sort(p+1,right,arr)
def partition(arr,left,right):
    pivot = arr[right]
    i,j = left,right-1
    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
#归并排序（稳定）
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        l,r = arr[:mid],arr[mid:]
        merge_sort(l)
```

```python
        merge_sort(r)
        i = j = k = 0
        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
        while i < len(l):
            arr[k] = l[i]
            i += 1
            k += 1
        while j < len(r):
            arr[k] = r[j]
            j += 1
            k += 1
#希尔排序（不稳定）
def shell_sort(arr):
    n = len(arr)
    gap = n//2
    while gap > 0:
        j = gap
        while j<n:
            i = j-gap
            while i>=0 and arr[i]>arr[i+gap]:
                arr[i],arr[i+gap] = arr[i+gap],arr[i]
                i -= gap
            j += 1
        gap //= 2
#基数排序(稳定)
def radix_sort(arr):
    max_value = max(arr)
    digit = 1
    while digit <= max_value: #确认最大位数
        temp = [[] for i in range(10)]
        for i in arr:
            t = i // digit % 10  #取出当前位数的数字
            temp[t].append(i) #入桶
        arr.clear()
        for bucket in temp:
            arr.extend(bucket) #从0到9、先进先出顺序将元素重排
        digit *= 10
    return arr



#基于归并排序的逆序数问题
def merge_sort(arr: list, l, r): #初始l = 0,r = len(arr)
    #将arr[l: r]排好，并返回该切片（左闭右开）内的逆序数
    #最终结果：返回这一区间逆序数，且对这一区间arr排序完成
    if r - l == 1:
        return 0
    mid = (l + r) // 2
    inv = 0
```

```python
    inv += merge_sort(arr, l, mid)
    inv += merge_sort(arr, mid, r)
    temp = []
    i = l
    j = mid
    while i < mid and j < r:
        if arr[i] <= arr[j]:
            temp.append(arr[i])
            i += 1
        else:
            temp.append(arr[j])
            j += 1
            inv += mid - i  # 此步计算逆序数
            #单纯对于j而言，从i到mid-1都大于j，构成逆序
    while i < mid:
        temp.append(arr[i])
        i += 1
    while j < r:
        temp.append(arr[j])
        j += 1
    for k in range(l, r):
        arr[k] = temp[k - l]
    return inv
```

2.数据结构算法（栈，队列，链表）

2.1 栈:中缀转后缀（shunting yard）

```python
def in_to_post(expression):
    mydict = {"+":1,"-":1,"*":2,"/":2}
    stack = []
    post = []
    num = ""
    for char in expression:
        if char.isdigit() or char == ".":
            num += char
            continue
        if num:
            post.append(num)
            num = ""
        if char == "(":stack.append(char)
        elif char == ")":
            while stack and stack[-1] != "(":
                post.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1] in "+-*/" and mydict[stack[-1]] >= mydict[char]:
                post.append(stack.pop())
            stack.append(char)
    if num:
        post.append(num)
    while stack:
        post.append(stack.pop())
```

```
    return " ".join(str(x) for x in post)
```

2.2 栈：后序表达式求值

```
def evaluate_post(expression):
    stack = []
    for char in expression:
        if char in "+-*/":
            num2 = stack.pop()
            num1 = stack.pop()
            stack.append(str(eval(num1+char+num2)))
        else:
            stack.append(char)
    return float(stack[0])
for _ in range(int(input())):
    print(f"{evaluate_post(input().split()):.2f}")
```

2.3 栈：后序表达式转完全括号表达式

```
def post_to_in(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(char)
        else:
            op2 = stack.pop()
            op1 = stack.pop()
            ex = "(" + op1 + char + op2 + ")"
            stack.append(ex)
    return stack.pop()
```

2.4 栈：实现dfs处理八皇后

```
def is_valid(row,col,queens):
    for r in range(row):
        if queens[r] == col or abs(queens[r]-col) == row-r:
            return False
    return True
def queen_stack(n):
    stack = []
    solutions = []
    stack.append((0,[]))
    while stack:
        row,cols = stack.pop()
        if row == n:
            solutions.append(cols)
            break
        for i in range(n):
            if is_valid(row,i,cols):
                stack.append((row+1,cols+[i]))
```

```python
    return solutions
def get_queen_string(b,n):#(这种做法深度回溯是反过来的)
    solutions = queen_stack(n)
    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
```

2.5 队列：约瑟夫问题

```python
from collections import deque
while True:
    n,m = map(int,input().split())
    if n == 0 and m == 0:
        break
    queue = deque([i for i in range(1,n+1)])
    for i in range(n-1):
        for i in range(m-1):
            queue.append(queue.popleft())
        queue.popleft()
    print(queue[0])
```

2.6 链表实现

```python
#双向链表实现
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
```

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

#循环链表实现
class Node:
    def __init__(self,data,next=None):
        self.data,self.next = data,next
class LinkList:
    def __init__(self):
        self.tail,self.size = None,0
    def pushFront(self,data):
        nd = Node(data)
        if self.tail is None:
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1
    def pushBack(self,data):
        self.pushFront(data)
        self.tail = self.tail.next
    def popFront(self):
        if self.size == 0:
            return
        nd = self.tail.next
        self.size -= 1
        if self.size == 0:
            self.tail = None
        else:
            self.tail.next = nd.next
        return nd.data
    def remove(self,data):
        if self.size == 0:return
        ptr = self.tail
        while ptr.next.data != data:
            ptr = ptr.next
            if ptr == self.tail:
                return False
        self.size -= 1
        if ptr.next == self.tail:
            self.tail = ptr
        ptr.next = ptr.next.next
        return True
```

3.二叉树基础

3.1 二叉树的高度与叶子数目

```python
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
#根结点的寻找：建立has_parent,建树的时候完成查找
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)
```

3.2 括号嵌套树解析

```python
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 = Node(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
            stack.append(node)  # 把当前节点推入栈中
        elif char == ')':  # 遇到右括号，子节点列表结束
            node = stack.pop()  # 弹出当前节点
    return node  # 根节点
```

3.3 完全括号表达式转二叉树

```python
#注：二叉树转前中后序表达式：前中后序遍历
#二叉树求值：递归
def buildParseTree(fplist):
    root = Node("")
    current_node = root
    stack = [root]
    for i in fplist:
        if i == "(":
            new_node = Node("")
            current_node.left = new_node
            current_node = current_node.left
        elif i not in "+-*/)":
            current_node.val = i
            current_node = stack.pop()
        elif i in "+-*/":
            current_node.val = i
            new_node = Node("")
            current_node.right = new_node
            stack.append(current_node)
            current_node = current_node.right
        elif i == ")":
            current_node = stack.pop()
```

```
    return root
```

3.4 后序表达式转二叉树

```
def build_tree(postfix):
    stack = []
    for char in postfix:
        node = Node(char)
        if char.isupper(): #表示操作符，大写字母
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]
```

注：二叉树转前中后序表达式：前中后序遍历；二叉树求值：递归

3.5 树的遍历

```
#按层次遍历
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

#前序遍历
def preorder(node):
    result = ""
    if node:
        result = node.key + preorder(node.left) + preorder(node.right)
    return result

#中序遍历
def in_order(node):
    result = ""
    if node:
        result = in_order(node.left) + node.key + in_order(node.right)
    return result

#后序遍历
def post_order(node):
    result = ""
    if node:
        result = post_order(node.left) + post_order(node.right) + node.key
```

```
    return result
```

3.6 二叉树推理

```
#根据二叉树前中序序列建树
def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_value = preorder[0]
    root = Node(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

#根据二叉树中后序序列建树
def build_tree(inorder, postorder):
    if not postorder or not inorder:
        return None
    root_value = postorder[-1]
    root = Node(root_value)
    root_index_inorder = inorder.index(root_value)
    root.left = build_tree(inorder[:root_index_inorder],postorder[:root_index_inorder])
    root.right = build_tree(inorder[root_index_inorder+1:],postorder[root_index_inorder:-1])
    return root
```

3.7 多叉树转二叉树

```
class Node:
        """多叉树节点"""
    def __init__(self):
        self.children = []
        self.next_sibling = None
class Node2:
        """二叉树节点"""
    def __init__(self):
        self.left = None
        self.right = None

#如果在建多叉树时为每个节点存储了`next_sibling`, 则很容易根据递归模版建树:
def convert(root: Node) -> Node2:
        """将以root为根的多叉树转换为二叉树, 并返回该二叉树的根节点"""
    # step 1: build root
    new_root = Node2()
    # step 2: connect subtrees
    if root.children:
        new_root.left = convert(root.children[0])
    if root.next_sibling:
        new_root.right = convert(root.next_sibling)
    # step 3:  return
    return new_root
```

```
#如果没有存`next_sibling,按原多叉树的特点进行:
def _convert(root: Node) -> Node2:
        # base case
    if not root:
        return None
        # step 1: 建根
    binary_root = Node2()
    # step 2: 将原多叉树的第一个子树（转换出的二叉树）连接为二叉树的左子树
    if root.children:
        binary_root.left = convert(root.children[0])
        # step 3: 将原多叉树的其他子树（转换出的二叉树），按顺序（类似链表）连接为前一棵子树（转换出的二叉
    curr = binary_root.left
    for i in range(1, len(root.children)):
        curr.right = convert(root.children[i])
        curr = curr.right
        # step 4: return
    return binary_root
```

4.二叉堆与二叉搜索树

4.1 手搓二叉堆

注：二叉堆排序不稳定

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0#初始化列表元素为0
    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                self.heapList[i],self.heapList[i//2] = self.heapList[i//2],self.heapList[i]
            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]:
                self.heapList[i],self.heapList[mc] = self.heapList[mc],self.heapList[i]
            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):
```

```python
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize -= 1
        self.heapList.pop() #先相等再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):
            self.percDown(i)
            i = i - 1
```

4.2 哈夫曼编码实现(权重+字符字典序)

```python
import heapq
class Node:
    def __init__(self,val,char):
        self.value = val
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self,other):
        if self.value == other.value:
            return ord(self.char) < ord(self.char)
        return self.value < other.value
def decode(ini_root,wait_string):
    now_node = ini_root
    result = ""
    for char in wait_string:
        if char == "1":
            now_node = now_node.right
        else:
            now_node = now_node.left
        if now_node.left is None:
            result += now_node.char
            now_node = ini_root
    return result
def encode(ini_root):
    codes = {}
    def parse(node,code):
        if node.left is None:
            codes[node.char] = code
        else:
            parse(node.left,code+"0")
            parse(node.right,code+"1")
    parse(ini_root,"")
    return codes
n = int(input())
mylist = []
for _ in range(n):
    word,freq = input().split()
    freq = int(freq)
```

```
    current_node = Node(freq,word)
    mylist.append(current_node)
heapq.heapify(mylist)
for i in range(len(mylist)-1):
    small = heapq.heappop(mylist)
    big = heapq.heappop(mylist)
    add_node = Node(small.value+big.value,small.char if ord(small.char)< ord(big.char) else b
    add_node.left = small
    add_node.right = big
    heapq.heappush(mylist,add_node)
root = mylist[0]
code_dict =encode(root)
```

4.3 二叉搜索树BST建树

```
#性质：左子结点小，右子结点大，中序遍历为由小到大的顺序序列
#利用中序遍历可以实现快排
#1.根据BST前序遍历建树
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

#2.根据插入顺序建树
def insert(node, value):
    if node is None:
        return Node(value)
    if value < node.value:
        node.left = insert(node.left, value)
    elif value > node.value:
        node.right = insert(node.right, value)
    return node
root = None
for number in numbers:
    root = insert(root, number)
```

4.4 手搓平衡二叉搜索树AVL

```
#n层AVL最小节点个数满足递推式：an = an-1 + an-2 + 1
#a0 = 1,a1 = 2(层=高度=边个数)
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())))
```

5.二叉树进阶

5.1 并查集算法

```
class DisjSet:
    def __init__(self, n):
        self.rank = [1] * n
        self.parent = [i for i in range(n)]
        self.size = [1]*n
    # 查找代表元素
    def find(self, x):
        if (self.parent[x] != x):
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    #按秩作连接
    def Union(self, x, y):
        #parent[find(x)] = find(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
            self.rank[xset] = self.rank[xset] + 1
    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]
# sets = set(find(x) for x in range(1, n + 1))
# print(len(sets))
```

5.2 线段树算法 Segment tree

```
#下标均为列表索引，n=len(arr)个元素存储在n至2n-1的位置(i+n)
#线段树有助于做列表/序列频繁更新情况下求部分和
N = 100000
# Max size of tree
tree = [0] * (2 * N)
# function to build the tree
def build(arr):
    # insert leaf nodes in tree
    for i in range(n):
        tree[n+i] = arr[i]
    for i in range(n-1, 0, -1):
        tree[i] = tree[i*2] + tree[i*2+1]
def updateTreeNode(p, value):
    tree[p+n] = value
    p = p + n
    i = p
    while i > 1: # move upward and update parents
        # ^表示异或运算符，分别比较两个数的二进制的每一位：若该位的值不相同，该位结果为1，否则该位结果为0
        # 这里^用以判断与i在一棵树下的另一个节点的坐标
        tree[i//2] = tree[i] + tree[i^1]
        i //= 2
# function to get sum on interval [l, r) #表示索引位置
def query(l, r):
    res = 0
    l += n
    r += n
    while l < r:
        #这是因为只有奇数的情况下，我们才需要进行操作，这是因为偶数是左子结点，往上走一定可以被容纳到
        if l % 2 == 1:
            res += tree[l]
            l += 1
        if r % 2 == 1:
            r -= 1
            res += tree[r]
        l //= 2
        r //= 2
    return res
```

5.3 前缀树算法Trie

```
#以26个小写字母为例
class TrieNode:
    def __init__(self):
        self.childNode = [None] * 26
        self.wordCount = 0
def insert_key(root, key):
```

```python
    # Initialize the currentNode pointer with the root node
    currentNode = root
    for c in key:
        # Check if the node exist for the current character in the Trie.
        if not currentNode.childNode[ord(c) - ord('a')]:
            # If node for current character does not exist
            # then make a new node
            newNode = TrieNode()
            # Keep the reference for the newly created node.
            currentNode.childNode[ord(c) - ord('a')] = newNode
        currentNode = currentNode.childNode[ord(c) - ord('a')]
    # Increment the wordEndCount for the last currentNode
    currentNode.wordCount += 1
def search_key(root, key):
    currentNode = root
    for c in key:
        if not currentNode.childNode[ord(c) - ord('a')]:
            # Given word does not exist in Trie
            return False
        currentNode = currentNode.childNode[ord(c) - ord('a')]
    return currentNode.wordCount > 0
def delete_key(root, word):
    currentNode = root
    lastBranchNode = None
    lastBrachChar = 'a'
    for c in word:
        if not currentNode.childNode[ord(c) - ord('a')]:
            return False
        else:
            count = 0
            for i in range(26):
                if currentNode.childNode[i]:
                    count += 1
            if count > 1:
                lastBranchNode = currentNode
                lastBrachChar = c
            currentNode = currentNode.childNode[ord(c) - ord('a')]
    count = 0
    for i in range(26):
        if currentNode.childNode[i]:
            count += 1
    # Case 1: The deleted word is a prefix of other words in Trie.
    if count > 0:
        currentNode.wordCount -= 1
        return True
    # Case 2: The deleted word shares a common prefix with other words in Trie.
    if lastBranchNode:
        lastBranchNode.childNode[ord(lastBrachChar) - ord('a')] = None
        return True
    # Case 3: The deleted word does not share any common prefix with other words in Trie.
    else:
        root.childNode[ord(word[0]) - ord('a')] = None
        return True
```

6 图基础

三类问题辨析：

bfs：一种应用是拓扑排序，有其本身算法。其他大多数求某种最优路径问题，应该在入队列时完成visit/distance访问，但根据约束条件的不同，visit/distance的维度可能会增加。如果要回溯路径，需要注意对prev的构建。如果最优化目标无法同队列的构建顺序保持一致，则问题在某种程度上划分为dijkstra算法，最小堆会用得上

dfs：回溯到就标记已经访问。如果并非简单的判断连通等问题，可能还需要回溯（如骑士周游/马走日），则完成节点探索后需要将visit标记取消。dfs判环/拓扑排序中，采用正在访问和已经完成访问两种标记，碰到正在访问说明有环

贪心算法：主要是dijkstra求加权最短路径和prim求总路径最优，采用heap做维护，distance常更新，vis只有从heap中弹出再确认

6.1 手搓图的类实现

```
import sys
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connected = {} #点类：权重
        #self.previous = None
        #self.distance = sys.maxsize
        #self.color = white
    def add_neighbor(self,neighbor,weight=0):
        #每一个点(传入)的neighbor存储都是：点类（非id）：权重
        self.connected[neighbor] = weight
    def get_neighbors(self):
        #得到的是一群点类
        return self.connected.keys()
    def get_id(self):
        return self.id
    def get_weight(self,neighbor):
        #传入的参数均为点类
        return self.connected[neighbor]
class Graph:
    def __init__(self):
        self.vertices = {}
        #构建：id：点类
        self.num = 0
    def add_vertex(self,key):
        self.num += 1
        new = Vertex(key)
        self.vertices[key] = new
        return new
    def get_vertex(self,key):
        #得到点类
        if key in self.vertices:
            return self.vertices[key]
        else:
```

```
            return None
    def addEdge(self,key1,key2,weight = 0):
        if key1 not in self.vertices:
            self.add_vertex(key1)
        if key2 not in self.vertices:
            self.add_vertex(key2)
        self.vertices[key1].add_neighbor(self.vertices[key2],weight)
    def getVertices(self):
        #这里的定义是返回所有的id名
        return self.vertices.keys()
```

6.2 BFS广度优先搜索算法：基础+词梯问题

```
#标准代码
from collections import deque
def bfs(start):
    start.distance = 0
    start.previous = None
    vert_queue = deque()
    vert_queue.append(start)
    while vert_queue:
        current = vert_queue.popleft()  # 取队首作为当前顶点
        for neighbor in current.get_neighbors():
            #遍历当前顶点的邻接顶点,标准写法
            #如果用嵌套数组表示graph就访问对应的neighbor
            if neighbor.color == "white":
                neighbor.color = "black"
                #表示已经被访问
                neighbor.distance = current.distance + 1
                neighbor.previous = current
                vert_queue.append(neighbor)

#词梯问题
#考虑了start与end不在字典中的可能性,以及多次数据的可能性
from collections import defaultdict,deque

for _ in range(int(input())):
    n = int(input())
    buckets = defaultdict(set)
    graph = defaultdict(list)
    for _ in range(n):
        word = input()
        for i in range(len(word)):
            bucket = word[:i] + "_" + word[i + 1:]
            buckets[bucket].add(word)
    start, end = input().split()
    for word in [start, end]:
        for i in range(len(word)):
            bucket = word[:i] + "_" + word[i + 1:]
            buckets[bucket].add(word)
    for wordlist in buckets.values():
        for word1 in wordlist:
            for word2 in wordlist - {word1}:
                graph[word1].append(word2)
```

```
    if start not in graph or end not in graph:
        print("NO")
        continue
    visited = {start}
    queue = deque([(start, [start])])
    cnt = 0
    while queue:
        node, path = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                #bfs做优化，不会回溯，直接加入visited
                if neighbor == end:
                    result = path + [neighbor]
                    cnt = 1
                    print(*result)
                    queue = []
                    break
                queue.append((neighbor, path + [neighbor]))
    if cnt == 0:
        print("NO")
```

6.3 DFS广度优先搜索算法：基础+骑士周游问题

```
#标准代码
def dfs(v):
    visited.add(v)
    for neighbor in graph[v]:
        if neighbor not in visited:
            dfs(neighbor)
#在这种情况下，每个节点被访问过后，如果还有路径可以到达，dfs不会再访问
#骑士周游问题中，每个节点被访问过后，还有可能在另一个路径上被访问
#一般考虑路径问题，需要回溯；而讨论是否连通，无需回溯
#手搓类的代码
def dfs_visit(start_vertex,Graph):
    start_vertex.color = "gray"
    Graph.time = Graph.time + 1
    for next_vertex in start_vertex.get_neighbors():
        if next_vertex.color == "white":
            next_vertex.previous = start_vertex
            Graph.dfs_visit(next_vertex,Graph)
    start_vertex.color = "black"
    #需要进一步回溯：white
    Graph.time += 1
    start_vertex.closing_time = Graph.time
    #记录在【只考虑访问一次】的情况下，完成访问时间
    #输出路径：node = end,while node,node=node.previous
    #另一种：参考骑士周游，存储path

#骑士周游问题
def valid(x,y):
    if 0<=x<n and 0<=y<n:
        return True
    return False
```

```python
def order(x,y):
    #核心步骤：先访问可能路径少的节点
    #为此，对于合法且未访问的节点，可能的路径做排序
    result = []
    for nx,ny in graph[x][y]:
        if visited[nx][ny] is False:
            cnt = 0
            for x2,y2 in graph[nx][ny]:
                if visited[x2][y2] is False:
                    cnt += 1
            result.append((nx,ny,cnt))
    result.sort(key = lambda x:x[2])
    return [(y[0],y[1]) for y in result]
def dfs(x,y,t):
    visited[x][y] = True
    #在dfs中，访问到这个节点了，先做True
    if t == n**2:
        return True
    for nx,ny in order(x,y):
        if dfs(nx,ny,t+1):
            return True
    visited[x][y] = False
    #核心步骤：访问结束的回溯，该结点在这条路径上不再访问，回归False
    return False
n = int(input())
sx,sy = map(int,input().split())
visited = [[False for j in range(n)] for i in range(n)]
graph = [[[] for j in range(n)] for i in range(n)]
#核心步骤：先建图，避免每次都讨论是否这个东西是合法的
for i in range(n):
    for j in range(n):
        for dx,dy in [(-2,-1), (-2,1), (-1,2), (-1,-2), (1,2), (1,-2), (2,1), (2,-1)]:
            if valid(i+dx,j+dy):
                graph[i][j].append((i+dx,j+dy))
if dfs(sx,sy,1):
    print("success")
else:
    print("fail")
#路径输出：dfs传入path，每次append和pop
```

6.4 一些应用

确定是否连通/有几个连通块：用dfs/bfs，dfs不用回溯就可以

确定连通块权值：dfs/bfs，每次记录权重，dfs把函数设置成return 权值

最小层数：bfs，用一个distance列表/节点属性标记层数

路径问题：path进入迭代/在不回溯（只访问一次情况下）可使用prev数组或属性保存

判断环：对于无向图，dfs过程中碰到已经访问过的说明有环；对于有向图，dfs过程中碰到正在访问的说明有环，可以用visited/visiting表示，或者采用color标记

7. 图进阶算法

7.1 拓扑排序topological sorting

```python
#dfs实现拓扑排序(有向图)
def dfs(node_list):
    def _dfs(node):
        nonlocal time
        visited[node] = 1 #visiting
        time += 1
        for neighbor in graph[node]:
            if neighbor in visited:
                if visited[neighbor] == 1:
                    return True
            else:
                _dfs(neighbor)
        time += 1
        times[node] = time#也可以改用栈：先入栈的说明先完成，再取逆
        visited[node] = 2
    time = 0
    times = {}
    visited = {}
    for node in node_list:
        if node not in visited:
            if _dfs(node):
                return False #成环
    result = sorted(times.keys(),key = lambda x:times[x],reverse = True)
    return result

#Kahn算法：bfs实现拓扑排序，用于判环
#无向图判环：第一次将所有<=1的节点入队；当相邻节点的度减到1，将其入队
from collections import deque, defaultdict
def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()
    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0: #无向图为<=1
            queue.append(u)
    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0: #无向图为1
                queue.append(v)
    # 检查是否存在环
    if len(result) == len(graph):
        return result
```

```python
    else:
        return None

#无向图成环的判断：如果遍历到了已经访问过的元素，且该元素不是prev(直接父节点，说明成环)
def loop(node,prev):
    visit[node] = True
    for element in graph[node]:
        if visit[element] is False:
            if loop(element,node):
                return True
        elif element != prev:
            return True
    return False
```

7.2 强连通分量SCC（一定针对有向图）

1. 2DFS(Kosaraju算法)

```python
#注意：graph为字典
#思想：先找到拓扑排序序列，然后对图做转置，找scc
#注意：visited的设法要注意（目前是序号0至n-1）
#注意：visited中包含全部节点，因此graph也要对应包含全部
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)):#目前采用0至n-1标号
        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)):#目前采用0至n-1标号
        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)#目前采用0至n-1标号
    sccs = []
    while stack:
        #栈顶是最后完成的，按照完成时间的逆序（即拓扑排序的正序）
        node = stack.pop()
        if not visited[node]:
```

```
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

2. Tarjan算法

```
def tarjan(graph):
    def dfs(node):
        nonlocal index, stack, indices, low_link, on_stack, sccs
        index += 1
        indices[node] = index#分配搜索次序和最低链接值入栈
        low_link[node] = index
        stack.append(node)
        on_stack[node] = True
        for neighbor in graph[node]:
            if indices[neighbor] == 0:  # Neighbor not visited yet
                dfs(neighbor)
                #回溯过程中，更新当前顶点的最低链接值
                low_link[node] = min(low_link[node], low_link[neighbor])
            elif on_stack[neighbor]:  # Neighbor is in the current SCC
                low_link[node] = min(low_link[node], indices[neighbor])
        if indices[node] == low_link[node]:
            scc = []#如果最低链接值=搜索次序：弹出从当前顶点开始的栈中顶点，构成scc
            while True:
                top = stack.pop()
                on_stack[top] = False
                scc.append(top)
                if top == node:
                    break
            sccs.append(scc)
    index = 0
    stack = []
    indices = [0] * len(graph)
    low_link = [0] * len(graph)
    on_stack = [False] * len(graph)
    sccs = []
    for node in range(len(graph)):
    #从图中选择一个未访问的顶点开始dfs
        if indices[node] == 0:
            dfs(node)
    return sccs
```

7.3 有权图的最短路径算法（Dijkstra算法）

提供从一个顶点到其他所有顶点的最短路径(无法处理负权边)

有向图与无向图的唯一不同在于初始graph对于边的构建

```
import heapq
import sys
```

```
#graph用字典嵌套：{node:{neighbor:weight}}
#distance、prev首选字典(均为连续标号用列表)
#visited用集合/列表
def dijkstra(graph,start):
    mylist = []
    distance[start] = 0 #其他的提前设定好为sys.maxsize/float("inf")
    heapq.heappush(mylist,(0,start)) #一定要注意是距离在前
    visited = set() #在dijkstra算法中，加入堆的不加入visited
    #只有当确定距离（从堆中首次弹出）才入visited

    while mylist:
        current_dist,node = heapq.heappop(mylist)
        if node in visited:continue #非首次弹出说明有已经确定的最小距离
        visited.add(node)
        #if node == end:  (...) break #如果有必要，找到对应点后输出正确答案，弹出
        for neighbor,inter_dist in graph[node].items():
            new_dist = current_dist + inter_dist#新距离
            if new_dist < distance[neighbor]: #一个条件判断足够，无需vis
                distance[neighbor] = new_dist #更新距离
                heapq.heappush(mylist,(new_dist,neighbor)) #列表，(距离，点)
                #prev[neighbor] = node 如果有需要，更新previous链条方便追索路径
    #如果有必要，输出没有找到的内容
```

Bellman-Ford算法：解决单源最短路径问题，可以处理带有负权边的图。

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    def bellman_ford(self, src):
        # 初始化距离数组，表示从源点到各个顶点的最短距离
        dist = [float('inf')] * self.V
        dist[src] = 0
        # 迭代 V-1 次，每次更新所有边
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
        # 检测负权环
        for u, v, w in self.graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                return "Graph contains negative weight cycle"
        return dist
```

SPFA算法：可以处理存在负权边的情况,也可以判断负权环路

```
from collections import deque
def SPFA(graph,start):
    queue = deque()
    queue.append(start)
```

```
    distance[start] = 0
    #初始化其他节点的最短距离为无穷大
    #cnt = [0] 判断负环: 初始路径长度为0
    visiting[start] = True
    while queue:
        node = queue.popleft()
        for neighbor,weight in graph[node].items():
            if distance[neighbor] > distance[node] + weight:
                distance[neighbor] = distance[node]+weight
                # cnt[neighbor] = 1 + cnt[node]
                # if cnt[neighbor] > n-1: 无法成环
                if visiting[neighbor] is False:
                    visiting[neighbor] = True
                    queue.append(neighbor)
        visiting[start] = False
    return distance
```

多源最短路径Floyd-Warshall算法

```
def floyd_warshall(edges,n):
    dist = [[float("inf")]*n for _ in range(n)]
    for word1,word2,weight in edges:
        dist[word1][word2] = weight
        #此处设定标号0至n-1
        #dist[word2][word1] = weight （无向图）
    for i in range(n):
        dist[i][i] = 0
    for k in range(n): #k表示可能的以之为中间节点的更新路径
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j],dist[i][k]+dist[k][j])
    return dist
```

7.4 有权图求最小权重和（最小生成树MST）： Prim算法

最小生成树T是边集的无环子集，连接V中的所有顶点，且边集合的权重最小

注意：prim算法只能用于无向图！！！ 构建graph时要加两次边

prim算法适用于稠密图（边多）

```
import heapq
import sys
#graph用字典嵌套：{node:{neighbor:weight}}
#distance、prev首选字典(均为连续标号用列表)
#visited用集合/列表
def prim(graph,start):
    mylist = [(0,start)] #一定要注意是距离在前
    distance[start] = 0 #其他的提前设定好为sys.maxsize/float("inf")
    visited = set()
    #prim同dijkstra算法一样，只有首次弹出最小距离才记visited
```

```python
    while mylist:
        current_dist,node = heapq.heappop(mylist)
        #如果算权重和，把所有的current_dist
        if node in visited: continue
        #说明已经有更小的同在MST中节点的距离了
        visited.add(node)
        #if node == end: (...)break #if有必要，找到目标点位弹出
        for neighbor,inter_dist in graph[node].items():
            if neighbor not in visited and distance[neighbor]>inter_dist:
                #需要加上不在visited中的条件
                distance[neighbor] = inter_dist
                #prev[neighbor] = node 如果有需要，更新previous链条方便追索路径
                heapq.heappush(mylist,(inter_dist,neighbor))#列表，(距离，点)
    #如果有必要，输出没找到的内容
```

Kruskal算法：适用于稀疏图。每次取出全局最小的距离，并查集确认不成环后添入

kruskal算法中，无向图的边加入一次就可以（可以用于有向图）

```python
class DisjointSet:
    def __init__(self, num_vertices):#表示点的数量
        #这里的点标号为0至n-1
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1
def kruskal(num_vertices,edges):
    # 按照权重排序
    edges.sort(key=lambda x: x[2])
    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)
    # 构建最小生成树的边集
    minimum_spanning_tree = []
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
    return minimum_spanning_tree
```

8 其他算法

8.1 单调栈

```
# 单调栈：模版（找右边第一个大于ai的下标，i=1..n)
def monotonic_stack(arr,n):#n = len(arr)
    i = 0
    stack = []
    ans = [0 for _ in range(n)] #不存在用0表示
    while i < n:
        while stack and arr[i]>arr[stack[-1]]:
            ans[stack.pop()] = i+1
        stack.append(i)
        i += 1
    return ans

# 找左边第一个比自己(严格)小的元素/右边第一个比自己小的元素,结果都以标号形式输出
# 维护(严格)递增栈
def first_min(arr):
    stack = [-1]
    left_first_min = [-1 for i in range(len(arr))] #最终结果-1表示没有比自己小的
    right_first_min = [len(arr) for i in range(len(arr))]
    #最终结果len(arr)表示没有比自己小的
    for i in range(len(arr)):
        while len(stack) > 1 and arr[i] <= arr[stack[-1]]:
            #加等号，左侧严格递增，右侧非严格
            right_first_min[stack[-1]] = i
            left_first_min[stack[-1]] = stack[-2]
            stack.pop()
        stack.append(i)
    for i in range(1,len(stack)):
        left_first_min[stack[i]] = stack[i-1]
    return left_first_min,right_first_min
#求总的（不比之小的）范围，相减-1即可

# 找左边第一个比自己(严格)大的元素/右边第一个比自己大的元素,结果都以标号形式输出
# 维护(严格)递减栈
def first_max(arr):
    stack = [-1]
    left_first_max = [-1 for i in range(len(arr))] #最终结果-1表示没有比自己大的
    right_first_max = [len(arr) for i in range(len(arr))]
    #最终结果len(arr)表示没有比自己大的
    for i in range(len(arr)):
        while len(stack) > 1 and arr[i] >= arr[stack[-1]]:
            #加等号，左侧严格递减，右侧非严格
            right_first_max[stack[-1]] = i
            left_first_max[stack[-1]] = stack[-2]
            stack.pop()
        stack.append(i)
    for i in range(1,len(stack)):
        left_first_max[stack[i]] = stack[i-1]
    return left_first_max,right_first_max
#求总的（不比之大的）范围，相减-1即可
```

8.2 二分查找

二分逼近算法：以月度开销为例

```
def valid(mid,m):
    result = 0
    cumulate = 0
    for i in mylist:
        if cumulate + i > mid:
            result += 1
            cumulate = i
        else:
            cumulate += i
    if cumulate != 0: result += 1
    return result <= m
n,m = map(int,input().split())
h = 0
l = 0
mylist = []
for _ in range(n):
    num = int(input())
    mylist.append(num)
    h += num
    l = max(l,num)
mid = (h+l)//2
while mid > l:
    if valid(mid,m):
        h = mid
    else:
        l = mid
    mid = (h+l)//2
if valid(l,m):
    print(l)
else:
    print(l+1)
```

二分查找标准库

```
import bisect
#一定是已经从小到大排好序的序列
arr = [0,2,2,2,5]
#bisect_left 寻找目标元素出现的最左侧位置的索引
#若不存在，结果为大于目标元素的第一个元素的索引
#当元素大于max时，返回len(arr);小于等于min时，返回0
position1 = bisect.bisect_left(arr,2) #1
position2 = bisect.bisect_left(arr,3) #4

#bisect_right  寻找目标元素出现的最右侧位置的索引+1
#若不存在，结果为大于目标元素的第一个元素的索引
#当元素>=max时，返回len(arr)；小于min时，返回0
position3 = bisect.bisect_right(arr,2) #4
position4 = bisect.bisect_right(arr,3) #4
```

```
#insort_left 将新元素插入到目标元素出现的最左侧位置的左侧
#当不存在，插入的位置为其对应的大小位置
#bisect.insort_left(arr,2.0) #[0, 2.0, 2, 2, 2, 5]

#insort_right 将新元素插入到目标元素出现的最右侧位置的右侧
#当不存在，插入的位置为其对应的大小位置
bisect.insort_right(arr,2.0) #[0, 2, 2, 2, 2.0, 5]

#lo参数与hi参数
#默认为0和len(arr),当自设定时可以修改查找范围
#查找/插入范围是从索引lo开始到索引hi-1的所有
```

8.3 KMP算法（字符串匹配）

```python
# 计算pattern字符串的next数组
def compute_next(pattern):
    m = len(pattern)
    next = [0]*m
    length = 0
    for i in range(1,m):
        while length > 0 and pattern[i] != pattern[length]:
            length = next[length-1]
        if pattern[i] == pattern[length]:
            length += 1
        next[i] = length
        #做下一个比对时，直接去比对索引为length即可
        #length也是可以跳过的字符个数
    return next
def kmp(text,pattern):
    n,m = len(text),len(pattern)
    if m == 0:
        return 0
    next = compute_next(pattern)
    matches = []
    j = 0 #pattern索引
    for i in range(n):
        while j>0 and text[i] != pattern[j]:
            j = next[j-1] #看前一个next数组的值
            #j表示可以跳过匹配的字符个数
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i-j+1)
            j = next[j-1]
    return matches
```

8.4 前缀和与前缀积问题

```python
#前缀和问题（和为k的连续子数组个数）
def subarray_num_add(nums,k):
    count = 0 #统计答案个数
    sums = 0 #存储遍历到的所有数组总和
```

```python
    mydict = {} #存储累积和的个数(和: 个数)
    mydict[0] = 1 #便于输出，在当前累计和=k时，自动加1
    for i in range(len(nums)):
        sums += nums[i]
        count += mydict.get(sums-k,0) #得到等于sums-k的个数，默认0
        mydict[sums] = mydict.get(sums,0) + 1
    return count

#前缀积问题
def subarray_num_multi(nums,k):
    count = 0
    sums = 1
    mydict = {}
    mydict[1] = 1
    for i in range(len(nums)):
        sums *= nums[i]
        count += mydict.get(sums/k,0)
        mydict[sums] = mydict.get(sums,0) + 1
    return count

#后缀和与后缀积算法同理
#若希望前缀不带有本身，向后移一位即可，第一个元素设为0/1 (和/积)
#后缀向前移一位即可，最后一位设定同理
```

一维前缀和

```
初始化S[0] = 0;
S[i] = a[1] + a[2] + ... a[i];
作用: 快速求出数组a (下标) 在区间[l, r]内的部分和
a[l] + ... + a[r] = S[r] - S[l - 1];
```

二维前缀和

```
S[i][j] = 第i行j列元素a[i][j]及左上部分所有元素的和
初始化S[0][j] = s[i][0] = 0;
S[i][j] = S[i - 1][j] + S[i][j - 1] - S[i - 1][j - 1] + a[i][j];
作用: 以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵的和为:
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
```

8.5 dp问题

```python
# 最大上升子序列求和
b = [int(x) for x in input().split()]
n = len(b)
dp = [0] * n
for i in range(n):
    dp[i] = b[i]
    for j in range(i):
        if b[j] < b[i]:
            dp[i] = max(dp[j] + b[i], dp[i])
```

```
print(max(dp))
#状态转移方程,背包问题:
#a[i][j] = max(a[i-1][j-t]+value[t],a[i-1][j])
```

note:一些不常用的方法&注意

0.pycharm撤回：ctrl+Z/ctrl+shift+Z(恢复)

1.python递归增加层数的方式

```
import sys
sys.setrecursionlimit(1000000)
```

2.以空间省时间（dp）

条件：传进函数参数，不可变类型（可做key，可哈希对象）；不能有参数相同但结果不同的情况

可哈希：数字类型，字符串类型，布尔类型，元组（列表字典集合不可以）

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

3.数据接收

```
import sys
input = sys.stdin.read
data = input().split()
```

4.常用包

```
#defaultdict
from collections import defaultdict
#避免KeyError：引用key不存在时，直接创建对应type的空元素作为value并返回
d = defaultdict(type) #表示value为这个type

#copy:拷贝最好用深拷贝，深拷贝得到的对象与原对象完全独立
#浅拷贝：两个前后对象会共享可变元素（可变子对象），直接修改对象两者不会同步改变
from copy import deepcopy
a = [[1,2],(30,40),"kkk"]
b = a.copy()
c = deepcopy(a) #需要调用包

#math包
#计算最大公因式
from math import gcd
x = gcd(15,20,25)
print(x)
```

```
## 5
math.pow(m,n)   #计算m的n次幂。
math.log(m,n)   #计算以n为底的m的对数(logn_m)

#eval()函数：将字符串当成有效的表达式来求值，并返回计算结果
result = eval("1 + 1")
print(result)  # 2
result = eval("'+' * 5")
print(result)  # +++++

# 排列组合问题
from itertools import permutations,combinations
l = [1,2,3]
print(list(permutations(l,2))) #全排列
print(list(combinations(l,2))) # 组合数，输出：[(1, 2), (1, 3), (2, 3)]

# 笛卡尔积
from itertools import product
a,b,c,d = [1,2,3],["x","y","z","w"],[4,5,6],[7,8,9,10]
prod = product(a,b,c,d) #每个集合中取一个
for p in prod: print(p) #打印方式：生成元组

#counter:等价于字典的计数包
from collections import Counter
mylist = ["a","b","a","o","b","a"]#创建一个Counter对象
count = Counter(mylist) #Counter({'a':3,'b':2,'o':1})
print(count["a"]) #3.访问不存在的元素返回0
count.update(["g","a"]) #Counter({'a':4, 'b':2, 'o':1, 'g':1})

#reduce:通过函数对迭代器对象中的元素进行遍历操作，返回计算结果
from functools import reduce
arr = [4,2,3,1] #可以自定义函数
product1 = reduce(lambda x,y:x*y,arr) #累乘，24
product2 = reduce(lambda x,y:x+y,arr) #累加，10
product3 = reduce(lambda x,y:x if x<y else y,arr) #min，1
```

5.列表&字符串

int(str,n)：将用n进制表示的字符串转化为十进制整数

查找某个元素/子串首次出现的索引：字符串用find&index，列表用index(find没找到-1，index报错)。确认出现次数两者均可用count

列表排序：reverse表示逆序【b = list(reversed(a))】;sort(倒序：.sort(reverse = True))，remove在列表中可以删除第一次出现位置的值，insert(position,item)插入值,pop删除还会返回该元素

list(zip(a,b)) 将两个列表元素一一配对，生成元组的列表。

字符串大小写：upper(),lower()

清空：.clear()函数将列表/字典/集合元素清空

`str.lstrip()` / `str.rstrip()`：移除字符串左侧/右侧的空白字符。

`str.replace(old, new)`：将字符串中的 `old` 子字符串替换为 `new` 。

`str.isalpha()` / `str.isdigit()` / `str.isalnum()`：检查字符串是否全部由字母/数字/字母和数字组成

min、max操作不能对空列表使用

ASCII:转字符chr();转整数：ord().A = 65, a = 97

6.集合&字典&元组

集合：并|,交&，差-，包含判断<=。增加元素：set.add(item),删除元素set.remove(item),增加多个set.update()

字典：dict.get(k,alt)(不存在value返回,默认None/alt)；

dic.setdefault(key,[]).append(value) 常用在字典中加入元素的方式（如果没有key就建空表，有key就直接添加value）【原始版本：mydict.setdefault(key,default_value)】

元组：不可变类型。增加元素()+()

7.魔术方法：

add,str,eq,ne(不相等),lt,le(小于等于),gt,ge

8.时间复杂度：

logn：10^18; n:10^7; nlogn:10^6;n^2:10^4;n^3:10^2