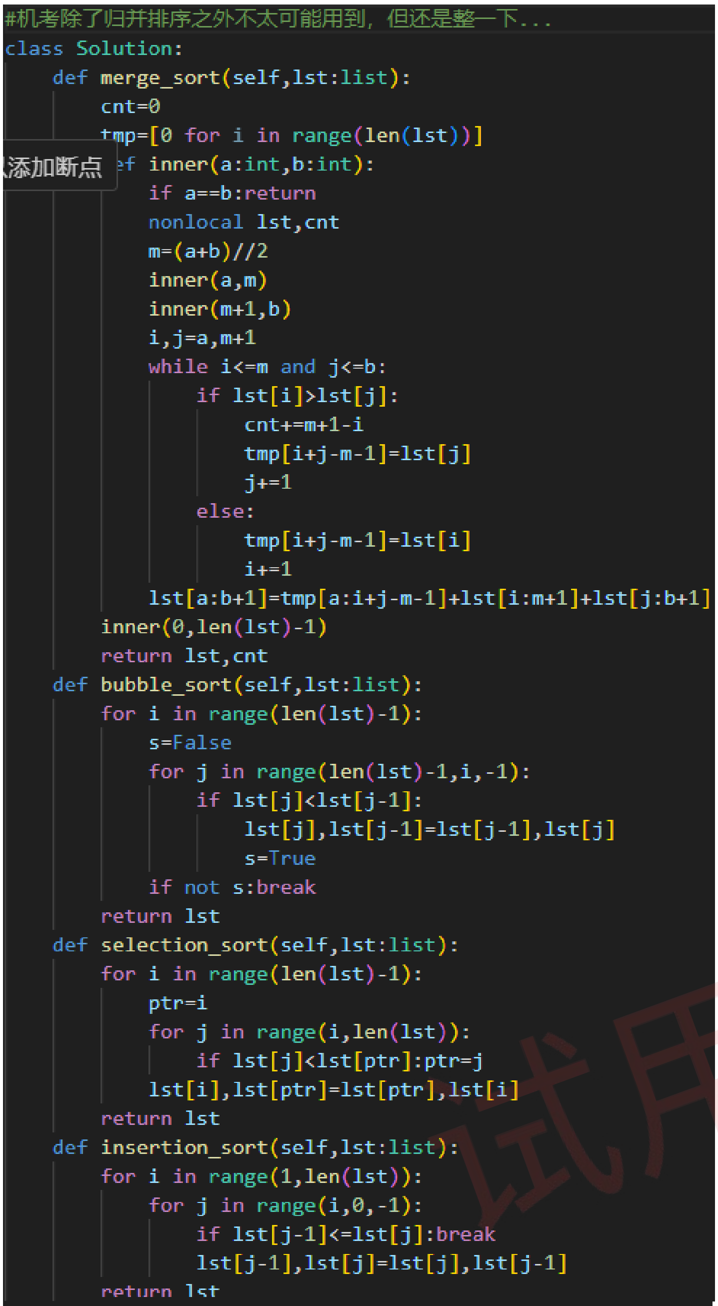
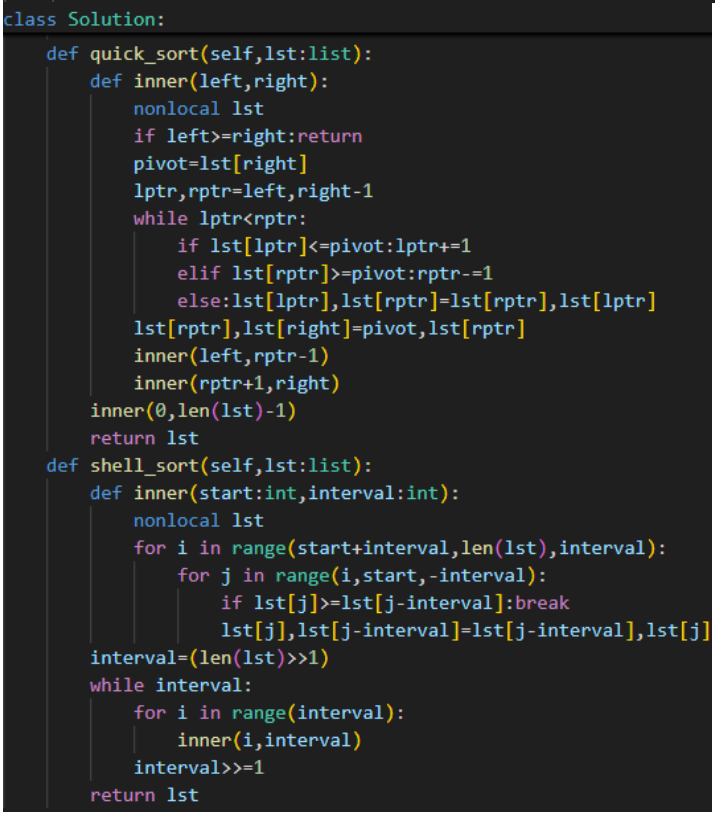
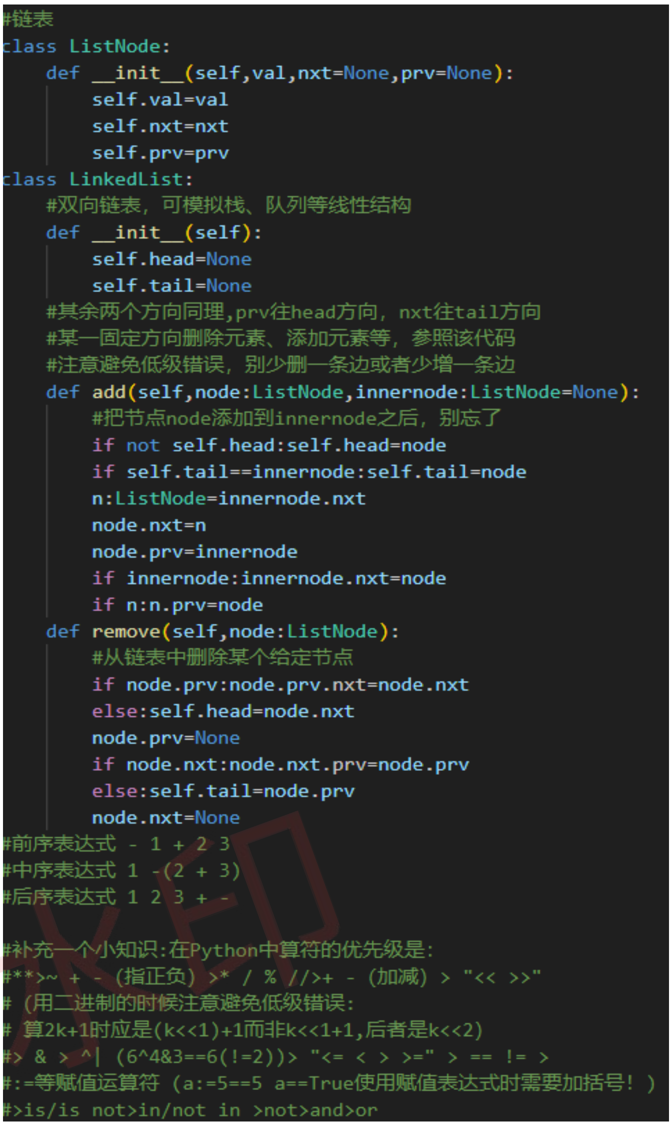


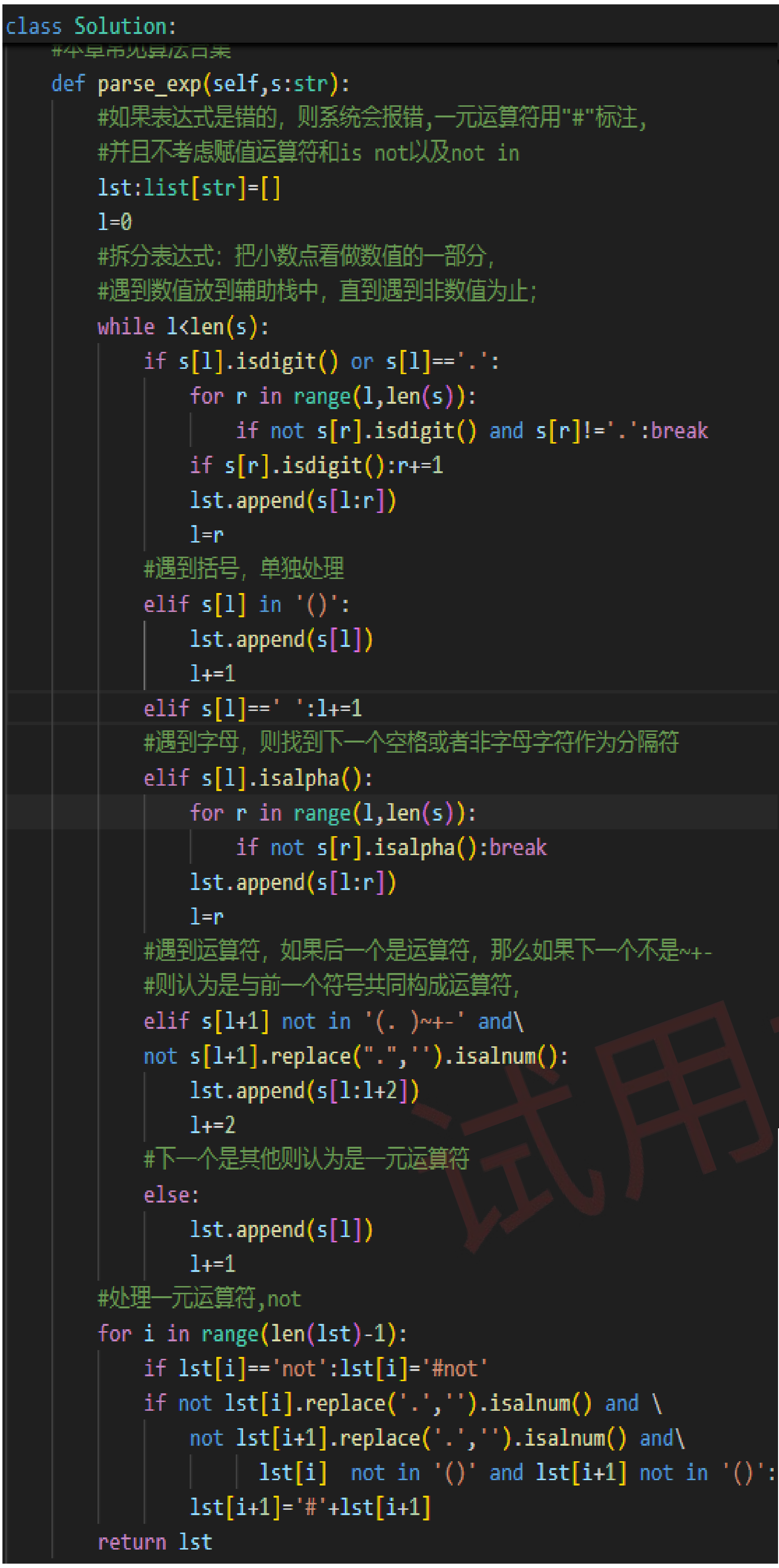
```
#机考除了归并排序之外不太可能用到，但还是整一下...
class Solution:
    def merge_sort(self,lst:list):
        cnt=0
        tmp=[0 for i in range(len(lst))]
        def inner(a:int,b:int):
            if a==b:return
            nonlocal lst,cnt
            m=(a+b)//2
            inner(a,m)
            inner(m+1,b)
            i,j=a,m+1
            while i<=m and j<=b:
                if lst[i]>lst[j]:
                    cnt+=m+1-i
                    tmp[i+j-m-1]=lst[j]
                    j+=1
                else:
                    tmp[i+j-m-1]=lst[i]
                    i+=1
            lst[a:b+1]=tmp[a:i+j-m-1]+lst[i:m+1]+lst[j:b+1]
        inner(0,len(lst)-1)
        return lst,cnt
    def bubble_sort(self,lst:list):
        for i in range(len(lst)-1):
            s=False
            for j in range(len(lst)-1,i,-1):
                if lst[j]<lst[j-1]:
                    lst[j],lst[j-1]=lst[j-1],lst[j]
                    s=True
            if not s:break
        return lst
    def selection_sort(self,lst:list):
        for i in range(len(lst)-1):
            ptr=i
            for j in range(i,len(lst)):
                if lst[j]<lst[ptr]:ptr=j
            lst[i],lst[ptr]=lst[ptr],lst[i]
        return lst
    def insertion_sort(self,lst:list):
        for i in range(1,len(lst)):
            for j in range(i,0,-1):
                if lst[j-1]<=lst[j]:break
                lst[j-1],lst[j]=lst[j],lst[j-1]
        return lst
```



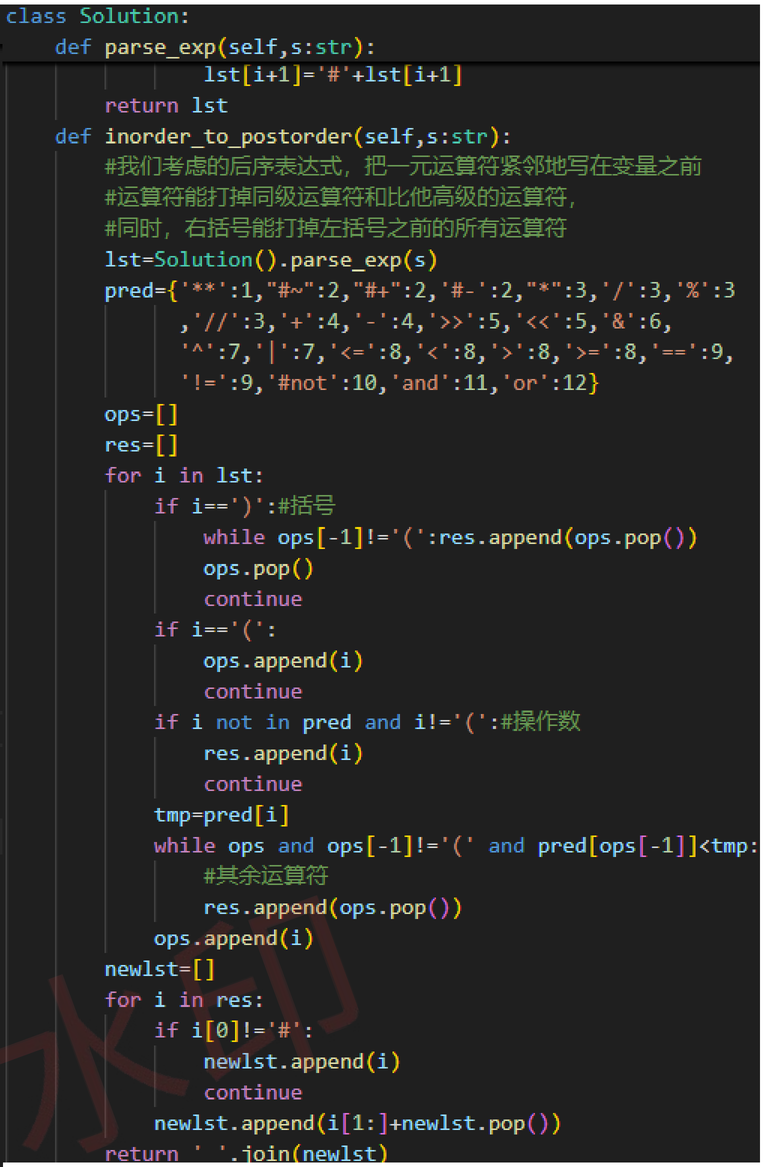
```
class Solution:
    def quick_sort(self,lst:list):
        def inner(left,right):
            nonlocal lst
            if left>=right:return
            pivot=lst[right]
            lptr,rptr=left,right-1
            while lptr<rptr:
                if lst[lptr]<=pivot:lptr+=1
                elif lst[rptr]>=pivot:rptr-=1
                else:lst[lptr],lst[rptr]=lst[rptr],lst[lptr]
            lst[rptr],lst[right]=pivot,lst[rptr]
            inner(left,rptr-1)
            inner(rptr+1,right)
        inner(0,len(lst)-1)
        return lst
    def shell_sort(self,lst:list):
        def inner(start:int,interval:int):
            nonlocal lst
            for i in range(start+interval,len(lst),interval):
                for j in range(i,start,-interval):
                    if lst[j]>=lst[j-interval]:break
                    lst[j],lst[j-interval]=lst[j-interval],lst[j]
        interval=(len(lst)>>1)
        while interval:
            for i in range(interval):
                inner(i,interval)
            interval>>=1
        return lst
```



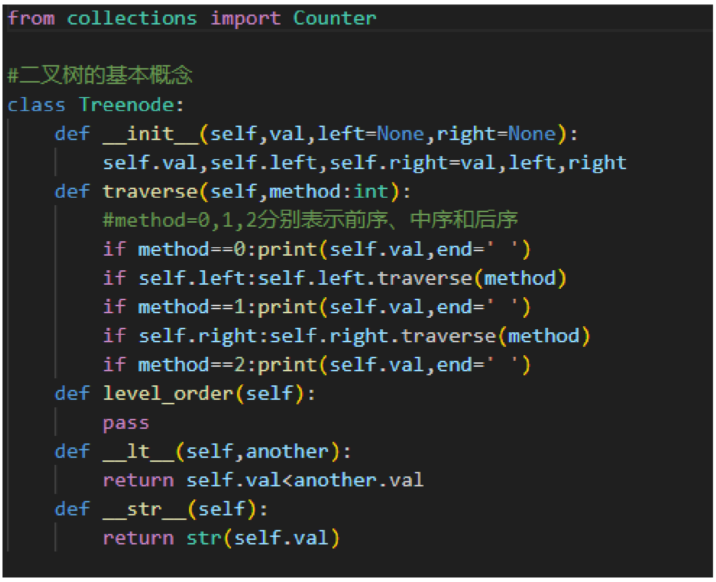
```
#链表
class ListNode:
    def __init__(self,val,nxt=None,prv=None):
        self.val=val
        self.nxt=nxt
        self.prv=prv
class LinkedList:
    #双向链表，可模拟栈、队列等线性结构
    def __init__(self):
        self.head=None
        self.tail=None
    #其余两个方向同理,prv往head方向，nxt往tail方向
    #某一固定方向删除元素、添加元素等，参照该代码
    #注意避免低级错误，别少删一条边或者少增一条边
    def add(self,node:ListNode,innernode:ListNode=None):
        #把节点node添加到innernode之后，别忘了
        if not self.head:self.head=node
        if self.tail==innernode:self.tail=node
        n:ListNode=innernode.nxt
        node.nxt=n
        node.prv=innernode
        if innernode:innernode.nxt=node
        if n:n.prv=node
    def remove(self,node:ListNode):
        #从链表中删除某个给定节点
        if node.prv:node.prv.nxt=node.nxt
        else:self.head=node.nxt
        node.prv=None
        if node.nxt:node.nxt.prv=node.prv
        else:self.tail=node.prv
        node.nxt=None
#前序表达式 - 1 + 2 3
#中序表达式 1 -(2 + 3)
#后序表达式 1 2 3 + -

#补充一个小知识:在Python中算符的优先级是:
#**>~ + - (指正负) >* / % //>+ - (加减) > "<< >>"
# (用二进制的时候注意避免低级错误:
# 算2k+1时应是(k<<1)+1而非k<<1+1,后者是k<<2)
#> & > ^| (6^4&3==6(!=2))> "<= < > >=" > == != >
#:=等赋值运算符 (a:=5==5 a==True使用赋值表达式时需要加括号! )
#>is/is not>in/not in >not>and>or
```



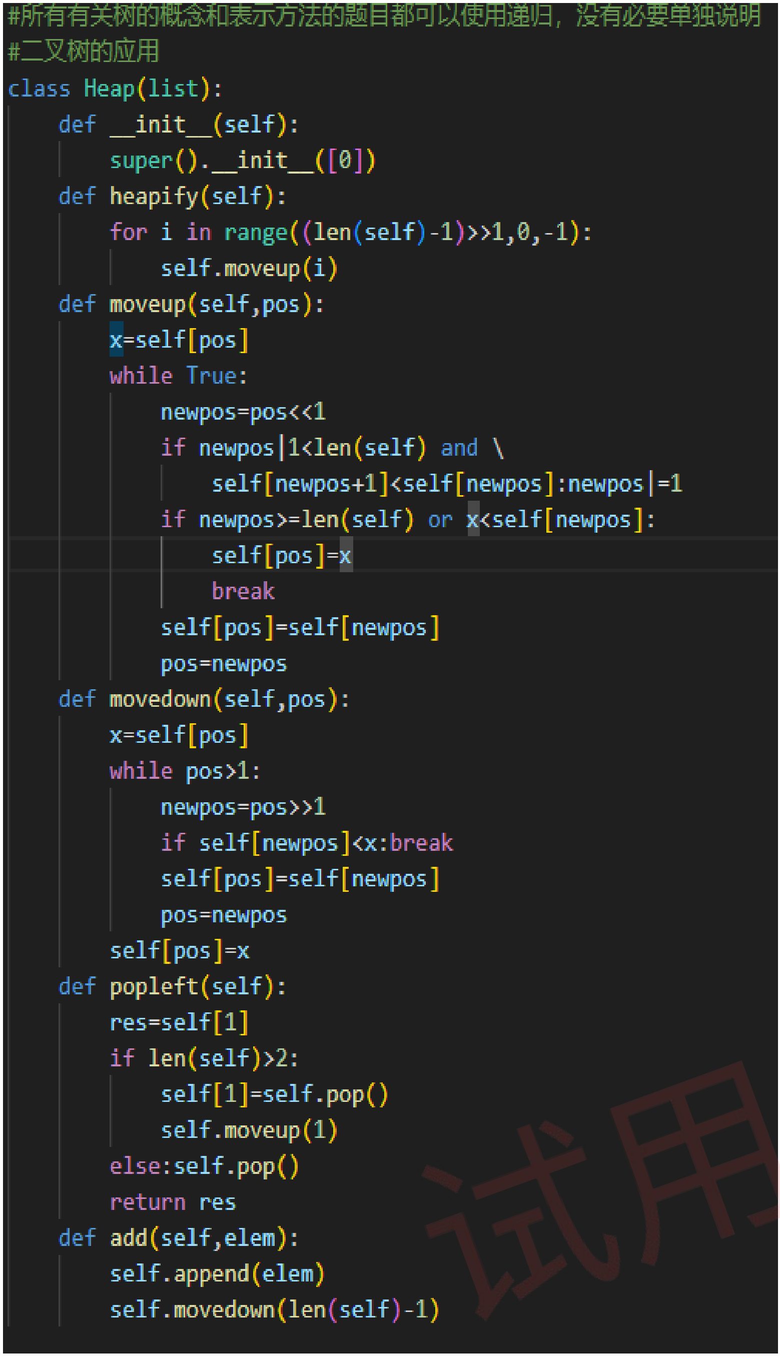
```
class Solution:
    #本章常见算法合集
    def parse_exp(self,s:str):
        #如果表达式是错的，则系统会报错,一元运算符用"#"标注,
        #并且不考虑赋值运算符和is not以及not in
        lst:list[str]=[]
        l=0
        #拆分表达式：把小数点看做数值的一部分，
        #遇到数值放到辅助栈中，直到遇到非数值为止;
        while l<len(s):
            if s[l].isdigit() or s[l]=='.':
                for r in range(l,len(s)):
                    if not s[r].isdigit() and s[r]!='.':break
                if s[r].isdigit():r+=1
                lst.append(s[l:r])
                l=r
            #遇到括号，单独处理
            elif s[l] in '()':
                lst.append(s[l])
                l+=1
            elif s[l]==' ':l+=1
            #遇到字母，则找到下一个空格或者非字母字符作为分隔符
            elif s[l].isalpha():
                for r in range(l,len(s)):
                    if not s[r].isalpha():break
                lst.append(s[l:r])
                l=r
            #遇到运算符，如果后一个是运算符，那么如果下一个不是~+-
            #则认为是与前一个符号共同构成运算符，
            elif s[l+1] not in '(. )~+-' and\
            not s[l+1].replace(".",'').isalnum():
                lst.append(s[l:l+2])
                l+=2
            #下一个是其他则认为是一元运算符
            else:
                lst.append(s[l])
                l+=1
        #处理一元运算符,not
        for i in range(len(lst)-1):
            if lst[i]=='not':lst[i]='#not'
            if not lst[i].replace('.','').isalnum() and \
                not lst[i+1].replace('.','').isalnum() and\
                   lst[i]  not in '()' and lst[i+1] not in '()':
                lst[i+1]='#'+lst[i+1]
        return lst
```



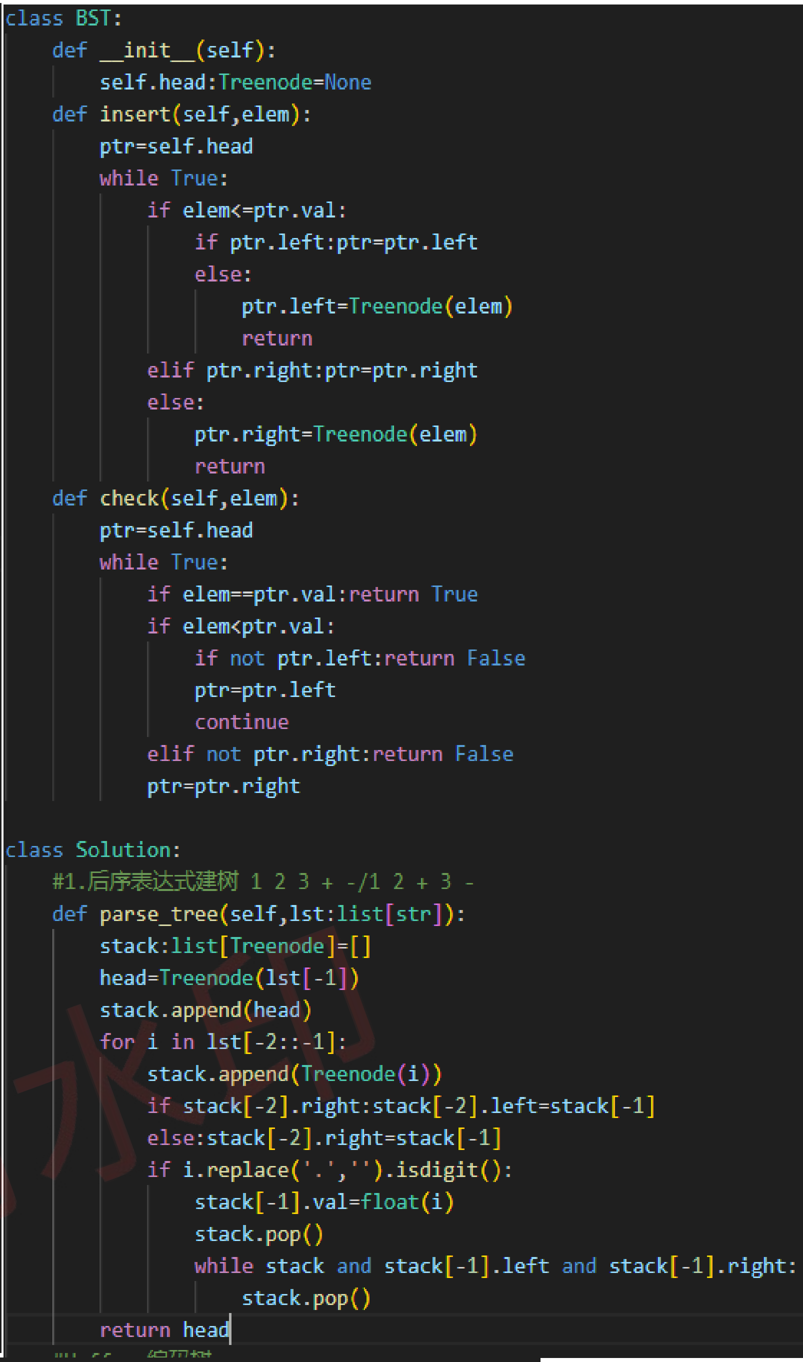
```
class Solution:
    def parse_exp(self,s:str):
                lst[i+1]='#'+lst[i+1]
        return lst
    def inorder_to_postorder(self,s:str):
        #我们考虑的后序表达式，把一元运算符紧邻地写在变量之前
        #运算符能打掉同级运算符和比他高级的运算符，
        #同时，右括号能打掉左括号之前的所有运算符
        lst=Solution().parse_exp(s)
        pred={'**':1,"#~":2,"#+":2,'#-':2,"*":3,'/':3,'%':3
              ,'//':3,'+':4,'-':4,'>>':5,'<<':5,'&':6,
              '^':7,'|':7,'<=':8,'<':8,'>':8,'>=':8,'==':9,
              '!=':9,'#not':10,'and':11,'or':12}
        ops=[]
        res=[]
        for i in lst:
            if i==')':#括号
                while ops[-1]!='(':res.append(ops.pop())
                ops.pop()
                continue
            if i=='(':
                ops.append(i)
                continue
            if i not in pred and i!='(':#操作数
                res.append(i)
                continue
            tmp=pred[i]
            while ops and ops[-1]!='(' and pred[ops[-1]]<tmp:
                #其余运算符
                res.append(ops.pop())
            ops.append(i)
        newlst=[]
        for i in res:
            if i[0]!='#':
                newlst.append(i)
                continue
            newlst.append(i[1:]+newlst.pop())
        return ' '.join(newlst)
```



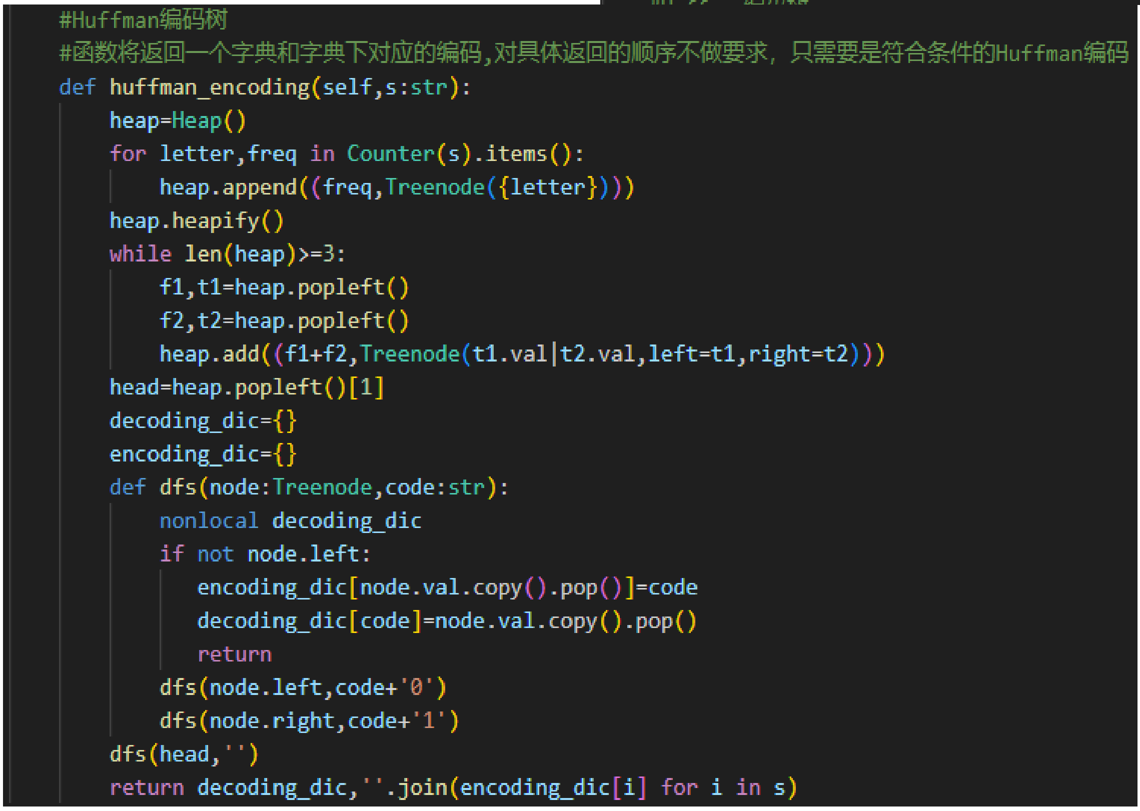
```
from collections import Counter

#二叉树的基本概念
class Treenode:
    def __init__(self,val,left=None,right=None):
        self.val,self.left,self.right=val,left,right
    def traverse(self,method:int):
        #method=0,1,2分别表示前序、中序和后序
        if method==0:print(self.val,end=' ')
        if self.left:self.left.traverse(method)
        if method==1:print(self.val,end=' ')
        if self.right:self.right.traverse(method)
        if method==2:print(self.val,end=' ')
    def level_order(self):
        pass
    def __lt__(self,another):
        return self.val<another.val
    def __str__(self):
        return str(self.val)
```



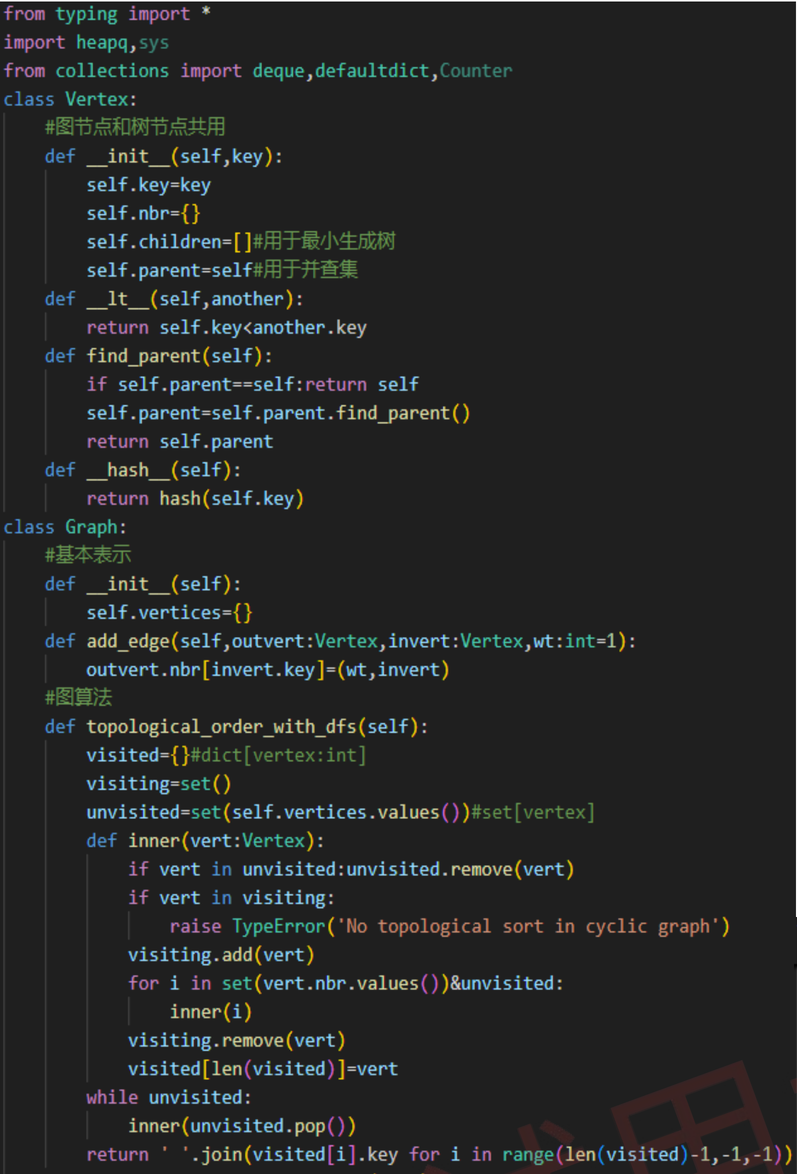
```
#所有有关树的概念和表示方法的题目都可以使用递归，没有必要单独说明
#二叉树的应用
class Heap(list):
    def __init__(self):
        super().__init__([0])
    def heapify(self):
        for i in range((len(self)-1)>>1,0,-1):
            self.moveup(i)
    def moveup(self,pos):
        x=self[pos]
        while True:
            newpos=pos<<1
            if newpos|1<len(self) and \
                self[newpos+1]<self[newpos]:newpos|=1
            if newpos>=len(self) or x<self[newpos]:
                self[pos]=x
                break
            self[pos]=self[newpos]
            pos=newpos
    def movedown(self,pos):
        x=self[pos]
        while pos>1:
            newpos=pos>>1
            if self[newpos]<x:break
            self[pos]=self[newpos]
            pos=newpos
        self[pos]=x
    def popleft(self):
        res=self[1]
        if len(self)>2:
            self[1]=self.pop()
            self.moveup(1)
        else:self.pop()
        return res
    def add(self,elem):
        self.append(elem)
        self.movedown(len(self)-1)
```



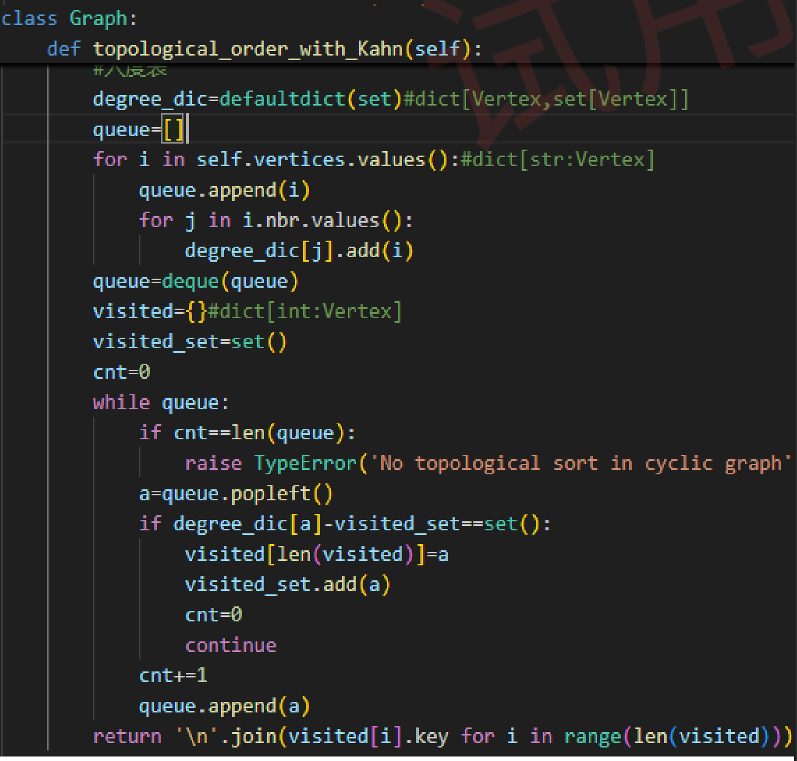
```
class BST:
    def __init__(self):
        self.head:Treenode=None
    def insert(self,elem):
        ptr=self.head
        while True:
            if elem<=ptr.val:
                if ptr.left:ptr=ptr.left
                else:
                    ptr.left=Treenode(elem)
                    return
            elif ptr.right:ptr=ptr.right
            else:
                ptr.right=Treenode(elem)
                return
    def check(self,elem):
        ptr=self.head
        while True:
            if elem==ptr.val:return True
            if elem<ptr.val:
                if not ptr.left:return False
                ptr=ptr.left
                continue
            elif not ptr.right:return False
            ptr=ptr.right

class Solution:
    #1.后序表达式建树 1 2 3 + -/1 2 + 3 -
    def parse_tree(self,lst:list[str]):
        stack:list[Treenode]=[]
        head=Treenode(lst[-1])
        stack.append(head)
        for i in lst[-2::-1]:
            stack.append(Treenode(i))
            if stack[-2].right:stack[-2].left=stack[-1]
            else:stack[-2].right=stack[-1]
            if i.replace('.','').isdigit():
                stack[-1].val=float(i)
                stack.pop()
                while stack and stack[-1].left and stack[-1].right:
                    stack.pop()
        return head
```



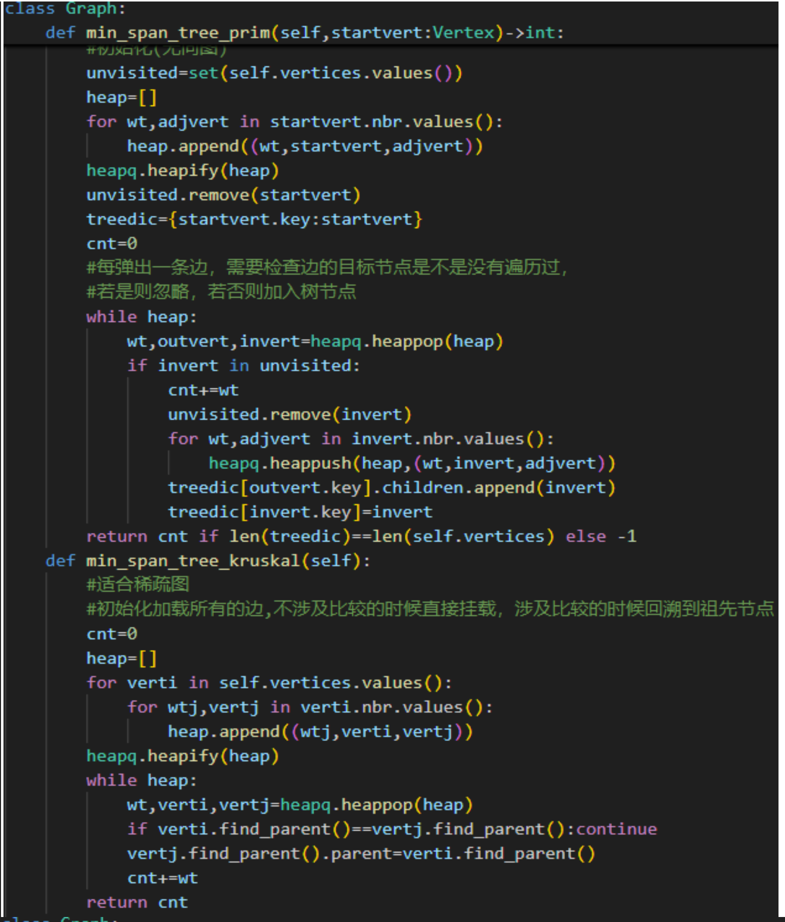
```
    #Huffman编码树
    #函数将返回一个字典和字典下对应的编码,对具体返回的顺序不做要求，只需要是符合条件的Huffman编码
    def huffman_encoding(self,s:str):
        heap=Heap()
        for letter,freq in Counter(s).items():
            heap.append((freq,Treenode({letter})))
        heap.heapify()
        while len(heap)>=3:
            f1,t1=heap.popleft()
            f2,t2=heap.popleft()
            heap.add((f1+f2,Treenode(t1.val|t2.val,left=t1,right=t2)))
        head=heap.popleft()[1]
        decoding_dic={}
        encoding_dic={}
        def dfs(node:Treenode,code:str):
            nonlocal decoding_dic
            if not node.left:
                encoding_dic[node.val.copy().pop()]=code
                decoding_dic[code]=node.val.copy().pop()
                return
            dfs(node.left,code+'0')
            dfs(node.right,code+'1')
        dfs(head,'')
        return decoding_dic,''.join(encoding_dic[i] for i in s)
```



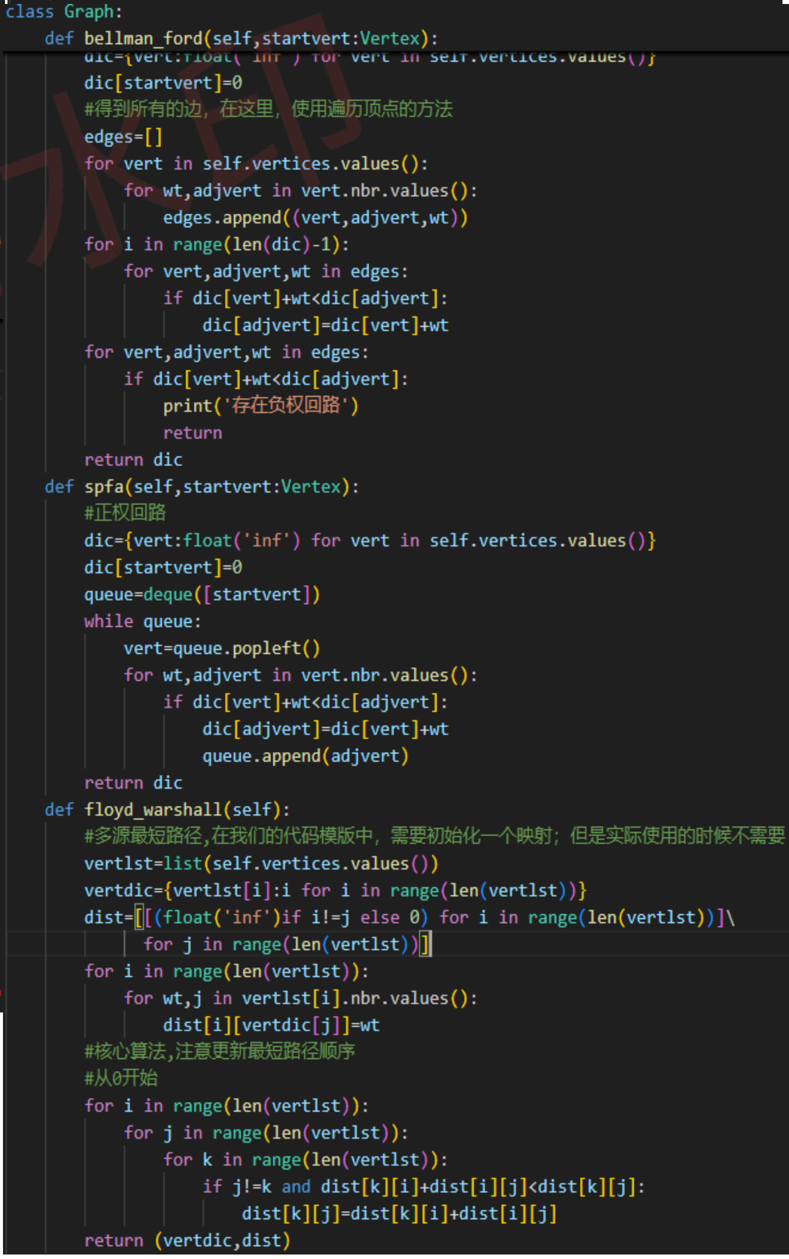
```python
from typing import *
import heapq,sys
from collections import deque,defaultdict,Counter
class Vertex:
    #图节点和树节点共用
    def __init__(self,key):
        self.key=key
        self.nbr={}
        self.children=[]#用于最小生成树
        self.parent=self#用于并查集
    def __lt__(self,another):
        return self.key<another.key
    def find_parent(self):
        if self.parent==self:return self
        self.parent=self.parent.find_parent()
        return self.parent
    def __hash__(self):
        return hash(self.key)
class Graph:
    #基本表示
    def __init__(self):
        self.vertices={}
    def add_edge(self,outvert:Vertex,invert:Vertex,wt:int=1):
        outvert.nbr[invert.key]=(wt,invert)
    #图算法
    def topological_order_with_dfs(self):
        visited={}#dict[vertex:int]
        visiting=set()
        unvisited=set(self.vertices.values())#set[vertex]
        def inner(vert:Vertex):
            if vert in unvisited:unvisited.remove(vert)
            if vert in visiting:
                raise TypeError('No topological sort in cyclic graph')
            visiting.add(vert)
            for i in set(vert.nbr.values())&unvisited:
                inner(i)
            visiting.remove(vert)
            visited[len(visited)]=vert
        while unvisited:
            inner(unvisited.pop())
        return ' '.join(visited[i].key for i in range(len(visited)-1,-1,-1))
```



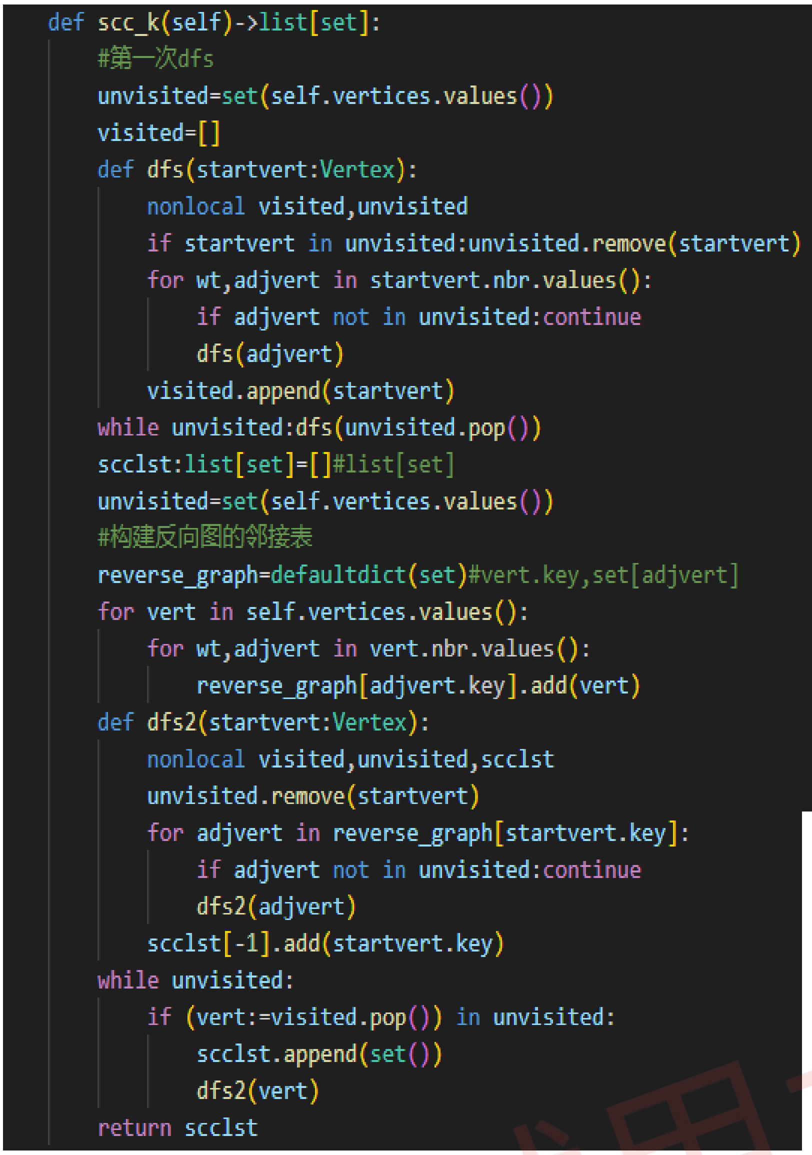
```python
class Graph:
    def topological_order_with_Kahn(self):
        #入度表
        degree_dic=defaultdict(set)#dict[Vertex,set[Vertex]]
        queue=[]
        for i in self.vertices.values():#dict[str:Vertex]
            queue.append(i)
            for j in i.nbr.values():
                degree_dic[j].add(i)
        queue=deque(queue)
        visited={}#dict[int:Vertex]
        visited_set=set()
        cnt=0
        while queue:
            if cnt==len(queue):
                raise TypeError('No topological sort in cyclic graph'
            a=queue.popleft()
            if degree_dic[a]-visited_set==set():
                visited[len(visited)]=a
                visited_set.add(a)
                cnt=0
                continue
            cnt+=1
            queue.append(a)
        return '\n'.join(visited[i].key for i in range(len(visited)))
```



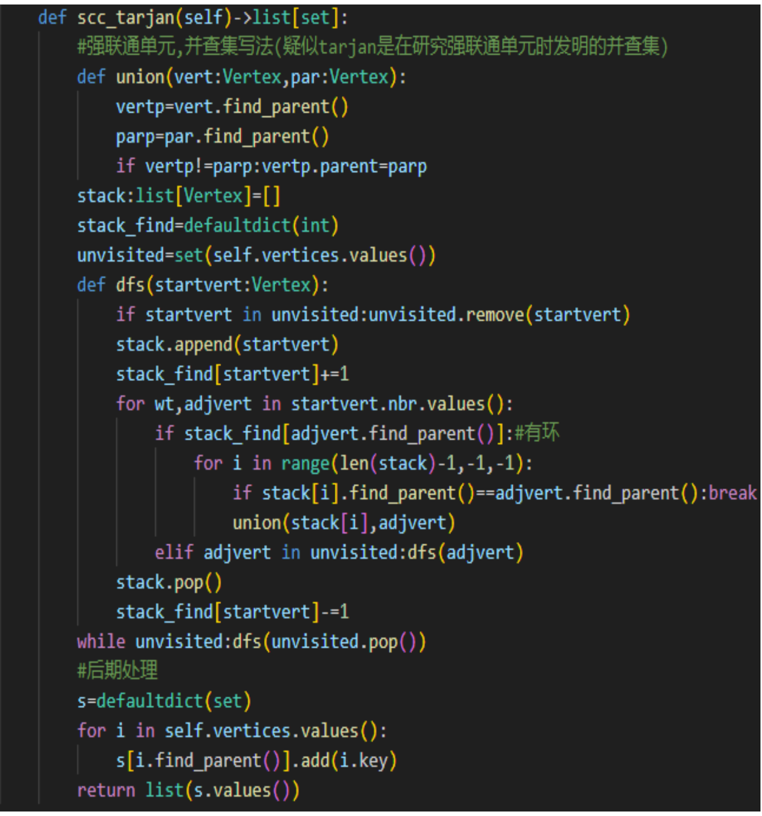
```python
class Graph:
    def min_span_tree_prim(self,startvert:Vertex)->int:
        #初始化(无向图)
        unvisited=set(self.vertices.values())
        heap=[]
        for wt,adjvert in startvert.nbr.values():
            heap.append((wt,startvert,adjvert))
        heapq.heapify(heap)
        unvisited.remove(startvert)
        treedic={startvert.key:startvert}
        cnt=0
        #每弹出一条边，需要检查边的目标节点是不是没有遍历过，
        #若是则忽略，若否则加入树节点
        while heap:
            wt,outvert,invert=heapq.heappop(heap)
            if invert in unvisited:
                cnt+=wt
                unvisited.remove(invert)
                for wt,adjvert in invert.nbr.values():
                    heapq.heappush(heap,(wt,invert,adjvert))
                treedic[outvert.key].children.append(invert)
                treedic[invert.key]=invert
        return cnt if len(treedic)==len(self.vertices) else -1
    def min_span_tree_kruskal(self):
        #适合稀疏图
        #初始化加载所有的边,不涉及比较的时候直接挂载，涉及比较的时候回溯到祖先节点
        cnt=0
        heap=[]
        for verti in self.vertices.values():
            for wtj,vertj in verti.nbr.values():
                heap.append((wtj,verti,vertj))
        heapq.heapify(heap)
        while heap:
            wt,verti,vertj=heapq.heappop(heap)
            if verti.find_parent()==vertj.find_parent():continue
            vertj.find_parent().parent=verti.find_parent()
            cnt+=wt
        return cnt
```



```python
class Graph:
    def bellman_ford(self,startvert:Vertex):
        dic={vert:float('inf') for vert in self.vertices.values()}
        dic[startvert]=0
        #得到所有的边，在这里，使用遍历顶点的方法
        edges=[]
        for vert in self.vertices.values():
            for wt,adjvert in vert.nbr.values():
                edges.append((vert,adjvert,wt))
        for i in range(len(dic)-1):
            for vert,adjvert,wt in edges:
                if dic[vert]+wt<dic[adjvert]:
                    dic[adjvert]=dic[vert]+wt
        for vert,adjvert,wt in edges:
            if dic[vert]+wt<dic[adjvert]:
                print('存在负权回路')
                return
        return dic
    def spfa(self,startvert:Vertex):
        #正权回路
        dic={vert:float('inf') for vert in self.vertices.values()}
        dic[startvert]=0
        queue=deque([startvert])
        while queue:
            vert=queue.popleft()
            for wt,adjvert in vert.nbr.values():
                if dic[vert]+wt<dic[adjvert]:
                    dic[adjvert]=dic[vert]+wt
                    queue.append(adjvert)
        return dic
    def floyd_warshall(self):
        #多源最短路径,在我们的代码模版中，需要初始化一个映射；但是实际使用的时候不需要
        vertlst=list(self.vertices.values())
        vertdic={vertlst[i]:i for i in range(len(vertlst))}
        dist=[[(float('inf')if i!=j else 0) for i in range(len(vertlst))]\
            for j in range(len(vertlst))]
        for i in range(len(vertlst)):
            for wt,j in vertlst[i].nbr.values():
                dist[i][vertdic[j]]=wt
        #核心算法,注意更新最短路径顺序
        #从0开始
        for i in range(len(vertlst)):
            for j in range(len(vertlst)):
                for k in range(len(vertlst)):
                    if j!=k and dist[k][i]+dist[i][j]<dist[k][j]:
                        dist[k][j]=dist[k][i]+dist[i][j]
        return (vertdic,dist)
```



```
    def scc_k(self)->list[set]:
        #第一次dfs
        unvisited=set(self.vertices.values())
        visited=[]
        def dfs(startvert:Vertex):
            nonlocal visited,unvisited
            if startvert in unvisited:unvisited.remove(startvert)
            for wt,adjvert in startvert.nbr.values():
                if adjvert not in unvisited:continue
                dfs(adjvert)
            visited.append(startvert)
        while unvisited:dfs(unvisited.pop())
        scclst:list[set]=[]#list[set]
        unvisited=set(self.vertices.values())
        #构建反向图的邻接表
        reverse_graph=defaultdict(set)#vert.key,set[adjvert]
        for vert in self.vertices.values():
            for wt,adjvert in vert.nbr.values():
                reverse_graph[adjvert.key].add(vert)
        def dfs2(startvert:Vertex):
            nonlocal visited,unvisited,scclst
            unvisited.remove(startvert)
            for adjvert in reverse_graph[startvert.key]:
                if adjvert not in unvisited:continue
                dfs2(adjvert)
            scclst[-1].add(startvert.key)
        while unvisited:
            if (vert:=visited.pop()) in unvisited:
                scclst.append(set())
                dfs2(vert)
        return scclst
```



```
    def scc_tarjan(self)->list[set]:
        #强联通单元,并查集写法(疑似tarjan是在研究强联通单元时发明的并查集)
        def union(vert:Vertex,par:Vertex):
            vertp=vert.find_parent()
            parp=par.find_parent()
            if vertp!=parp:vertp.parent=parp
        stack:list[Vertex]=[]
        stack_find=defaultdict(int)
        unvisited=set(self.vertices.values())
        def dfs(startvert:Vertex):
            if startvert in unvisited:unvisited.remove(startvert)
            stack.append(startvert)
            stack_find[startvert]+=1
            for wt,adjvert in startvert.nbr.values():
                if stack_find[adjvert.find_parent()]:#有环
                    for i in range(len(stack)-1,-1,-1):
                        if stack[i].find_parent()==adjvert.find_parent():break
                        union(stack[i],adjvert)
                elif adjvert in unvisited:dfs(adjvert)
            stack.pop()
            stack_find[startvert]-=1
        while unvisited:dfs(unvisited.pop())
        #后期处理
        s=defaultdict(set)
        for i in self.vertices.values():
            s[i.find_parent()].add(i.key)
        return list(s.values())
```