0.补充代码

·埃氏筛求最大公因数：

```python
def gcd(m,n):
    while m%n !=0:
        oldm=m
        oldn=n
        m=oldn
        n=oldm%oldn
    return n
```

1.W02 排序方法

·冒泡排序：

不断交换相邻元素，时间复杂度$O(n^2)$，空间复杂度$O(1)$，冒泡排序是原地排序算法，无需额外空间，稳定

```python
def bubble_sort(arr):
    n=len(arr)
    for i in range(n):
        swapped=False
        for j in range(n-i-1):
            if arr[j]>arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
                swapped=True
        if not swapped:
            break
    return arr
```

·选择排序：

从列表中未排序的部分反复选择最大或最小的而元素移动到列表已排序的部分，时间复杂度$O(n^2)$，空间复杂度$O(1)$，原地排序，无需额外空间，适合小的数据集，极端情况下时间复杂度高，不稳定

```python
def selection_sort(arr):
    n=len(arr)
    for i in range(n-1):
        min_num=arr[i]
        position=i
        for j in range(i+1,n):
            if arr[j]<min_num:
                position,min_num=j,arr[j]
        arr[i],arr[position]=arr[position],arr[i]
    return arr

alst=[int(x) for x in input()]
print(selection_sort(alst))
```

·快速排序:

基于分治算法的排序，选择一个元素作为基准，围绕基准分区，时间复杂的$O(nlogn)$，最差$O(n^2)$，空间复杂度考虑递归堆栈为$O(n)$，不考虑为$O(1)$，适合大数据集，极端情况会显示交叉的复杂度，不稳定

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

·归并排序:

将数组划分为更小的子数组，对每个子数组进行排序，将排序后的数组合并，形成最终的子数组，时间复杂度$O(nlogn)$，空间复杂度$O(n)$，天然可并行化算法，稳定，适合大数据集，但是需要额外空间

```python
def merge(left,right):
    merged=[]
    inv_count=0
    i=j=0
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            merged.append(left[i])
            i+=1
        else:
            merged.apppend(right[i])
            j+=1
            inv_count+=len(left)-i
    merged+=left[i:]
    merged+=right[j:]
    return merged,inv_count
def merge_sort(lst):
    if len(lst)<=1:
        return lst,0
    middle=len(lst)//2
    left,inv_left=merge_sort(lst[:middle])
    right,inv_right=merge_sort(lst[middle:])
    merged,inv_merged=merge(left,right)
    return merged,inv_left+inv_right+inv_merged
```

·插入排序：

通过先前已经排好序的数组得到目标插入元素的位置从而不断排序

时间$O(n^2)$空间$O(1)$原地排序

·希尔排序：

插入排序的变种，可以交换远项

时间复杂度最差$O(n)$空间$O(1)$

·堆排序：

基于完全二叉树，每个节点的值大于等于子节点的值

（一般使用heapq的最小堆，最大堆给每个元素加负号）

时间$O(nlogn)$，空间$O(1)$适合大数据集，原地排序，不稳定，可能交换相同元素

2.W03 基本数据结构

单调栈——找第一个比当前节点高的节点位置

```
n=int(input())
a=list(map(int,input().split()))
def monotonic_stack(nums):
    stack=[]
    result=[str(0)]*len((nums))
    for i in range(len(nums)):
        while stack and nums[i]>nums[stack[-1]]:
            result[stack.pop()]=str(i+1)
        stack.append(i)
    return result
```

T快速堆猪——通过栈维护最大值，只记录上行极大值：

```
N=int(input())
data=[]
for _ in range(N):
    ipt=[int(x) for x in input().split()]
    if len(ipt)==2:
        if data:
            data.append(max(data[-1],ipt[1]))
        else:
            data.append(ipt[-1])
    elif ipt[0]==1:
        if data:
            data.pop()
    else:
        if data:
            print(data[-1])
        else:
            print(0)
```

T波兰表达式——逆波兰，后序表达式的求值方式：

数字入栈，操作符从栈顶弹出两个数字进行运算，再入栈，直到最后剩下一个元素为结果

T合法出栈序列——模拟入栈出栈过程，一个数据栈一个处理栈

T括号匹配——准备一个括号栈，只放括号，左括号入栈，右括号出栈，如果没有元素说明右括号不匹配，如果最后剩下左括号说明左括号不匹配

调度场算法——中序表达式转后序表达式

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

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

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

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

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

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

八皇后

```
def is_safe(board, row, col):
    # 检查当前位置是否安全
    # 检查同一列是否有皇后
    for i in range(row):
        if board[i] == col:
            return False
    # 检查左上方是否有皇后
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:
            return False
        i -= 1
        j -= 1
    # 检查右上方是否有皇后
    i = row - 1
    j = col + 1
    while i >= 0 and j < 8:
        if board[i] == j:
            return False
        i -= 1
        j += 1
    return True

def queen_dfs(board, row):
    if row == 8:
        # 找到第b个解，将解存储到result列表中
        ans.append(''.join([str(x+1) for x in board]))
        return
    for col in range(8):
        if is_safe(board, row, col):
            # 当前位置安全，放置皇后
            board[row] = col
            # 继续递归放置下一行的皇后
            queen_dfs(board, row + 1)
            # 回溯，撤销当前位置的皇后
            board[row] = 0

ans = []
queen_dfs([None]*8, 0)
for _ in range(int(input())):
    print(ans[int(input()) - 1])
```

约瑟夫问题

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

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.pop(0))   # O(N)
        queue.pop(0)                                        # O(N)
    return queue.pop(0)                                     # O(N)


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

3.树

括号嵌套树

```
class treenode:
    def __init__(self,s):
        self.key=s
        self.child=[]
ipt=input()
baselst=[]
for i in ipt:
    baselst.append(i)

def buildtree(alst):
    stack=[]
    node=None
    for char in alst:
        if char not in ['(',',',')']:
            node=treenode(char)#构建节点
            if stack:
                stack[-1].child.append(node)#如果此时栈内有节点，就把这个节点放入最后一个节点的child中
        elif char=='(':
            if node:
                stack.append(node)#如果这时候有node，说明这个括号是这个node的child的开端，把node压入栈中，此时栈中最后一个元素是当前括号对应的node
                node=None#重新分析节点，将旧的节点重置
        elif char==')':
            if stack:
                node=stack.pop()#当前节点编辑结束，重新返回编译好的父节点
    return node

def pre(node):
    ans=[node.key]
    for chd in node.child:
        ans+=pre(chd)
    return ans
def post(node):
    ans=[]
    for chd in node.child:
        ans+=post(chd)
    ans.append(node.key)
    return ans

nd=buildtree(baselst)
print(''.join(pre(nd)))
print(''.join(post(nd)))
```

二叉搜索树BST——中间的比左边的大，比右边的小

Huffman算法

```
import heapq
class HuffmanTreeNode:
    def __init__(self,weight,char=None):
        self.weight=weight
        self.char=char
        self.left=None
        self.right=None

    def __lt__(self,other):
        return self.weight<other.weight

def BuildHuffmanTree(characters):
    heap=[HuffmanTreeNode(weight,char) for char,weight in characters.items()]
    heapq.heapify(heap)
    while len(heap)>1:
        left=heapq.heappop(heap)
        right=heapq.heappop(heap)
        merged=HuffmanTreeNode(left.weight+right.weight,None)
        merged.left=left
        merged.right=right
        heapq.heappush(heap,merged)
    root=heapq.heappop(heap)
    return root

def enpaths_huffman_tree(root):
    # 字典形如(idx,weight):path
    paths={}
    def traverse(node,path):
        if node.char:
            paths[(node.char,node.weight)]=path
        else:
            traverse(node.left,path+1)
            traverse(node.right,path+1)
    traverse(root,0)
    return paths

def min_weighted_path(paths):
    return sum(tup[1]*path for tup,path in paths.items())

n,characters=int(input()),{}
raw=list(map(int,input().split()))
for char,weight in enumerate(raw):
    characters[str(char)]=weight
root=BuildHuffmanTree(characters)
paths=enpaths_huffman_tree(root)
```

```
print(min_weighted_path(paths))
```

4.图

连通区域问题

```
import sys
sys.setrecursionlimit(20000) # 防止递归爆栈
dx = [-1,-1,-1,0,0,1,1,1]
dy = [-1,0,1,-1,1,-1,0,1]
def dfs(x,y):
    field[x][y] = '.' # 标记，避免再次访问
    for i in range(8):
        nx,ny = x+dx[i],y+dy[i]
        if 0<=nx<n and 0<=ny<m and field[nx][ny]=='W': # 注意判断是否越界
            dfs(nx,ny) # DFS需递归
n,m = map(int,input().split())
field = [list(input()) for _ in range(n)]
cnt = 0
for i in range(n):
    for j in range(m):
        if field[i][j] == 'W':
            dfs(i,j)
            cnt += 1
print(cnt)
```

dijk

```
class Vertex:
    def __init__(self,v):
        self.value=v
        self.connectedto={}

class Graph:
    def __init__(self):
        self.vertexes={}

    def add_vertex(self,s):
        self.vertexes[s]=Vertex(s)

    def add_edge(self,s1,s2,w):
        if s1 not in self.vertexes:
            self.vertexes[s1]=Vertex(s1)
        if s2 not in self.vertexes:
            self.vertexes[s2]=Vertex(s2)
        V1=self.vertexes[s1]
        V2=self.vertexes[s2]
        V1.connectedto[s2]=w
        V2.connectedto[s1]=w
        self.vertexes[s1]=V1
        self.vertexes[s2]=V2

def dijkstra(v1,v2):
    sheet={}
    not_visited=set(map_graph.vertexes.keys())
    path={}
    for ver in map_graph.vertexes:
        sheet[ver]=99999999
    sheet[v1]=0
    nearest=v1
    while not_visited:
        nearest=None
        for vtx in not_visited:
            if not nearest or sheet[vtx]<sheet[nearest]:
                nearest=vtx
        not_visited.discard(nearest)
        neighbours=map_graph.vertexes[nearest].connectedto.keys()
        for neighbour in neighbours:
            nl=sheet[nearest]+map_graph.vertexes[nearest].connectedto[neighbour]
            if nl<sheet[neighbour]:
                sheet[neighbour]=nl
                path[neighbour]=
(nearest,map_graph.vertexes[nearest].connectedto[neighbour])
    way=[v2]
    while v2 in path:
        lastver,w=path[v2]
        way.append(f'({w})')
        way.append(lastver)
        v2=lastver
    way.reverse()
    return '->'.join(way)
```

带有开销的dijk

```
import heapq

K=int(input())
N=int(input())
R=int(input())
g=[[]for i in range(N)]
for i in range(R):
    f,t,l,c=map(int,input().split())
    g[f-1].append([t-1,l,c])

que=[(0,0,0)]
dist=[[9999999999]*(K+1) for _ in range(N)]
dist[0][0]=0

while que:
    current_l,current_node,current_cost=heapq.heappop(que)
    if current_l>dist[current_node][current_cost]:
        continue

    for new_ver,length,cost in g[current_node]:
        if current_cost+cost<=K and current_l+length<dist[new_ver]
[current_cost+cost]:
            dist[new_ver][current_cost+cost]=current_l+length
            heapq.heappush(que,(current_l+length,new_ver,current_cost+cost))
```

```
min_dist=min(dist[N-1])
if min_dist!=9999999999:
    print(min_dist)
else:
    print(-1)
```

并查集

```
def find(x):
    if bottles[x]!=x:
        bottles[x]=find(bottles[x])
    return bottles[x]
def union(x,y):
    rx=find(x)
    ry=find(y)
    if rx!=ry:
        bottles[ry]=rx
while True :
    try:
        n,m=map(int,input().split())
        bottles=list(range(n+1))
        for _ in range(m):
            a,b=map(int,input().split())
            if find(a)==find(b):
                print('Yes')
            else :
                print('No')
                union(a,b)

        not_empty=set(find(x) for x in range(1,n+1))
        ans=sorted(not_empty)
        print(len(ans))
        print(*ans)
    except EOFError:
        break
```

并查集+Kruskal

```
class DisjointsetUnion:#定义并查集类
    def __init__(self,n):
        self.parent=list(range(n))
        self.rank=[0]*n

    def find(self,x):#路径压缩
        if self.parent[x]!=x:
            self.parent[x]=self.find(self.parent[x])
        return self.parent[x]

    def union(self,x,y):#按秩合并
        xr=self.find(x)
        yr=self.find(y)
        if xr==yr:
            return False
        elif self.rank[xr]<self.rank[yr]:
            self.parent[xr]=yr
        elif self.rank[xr]>self.rank[yr]:
            self.parent[yr]=xr
        else:
            self.parent[yr]=xr
            self.rank[xr]+=1
        return True

def Kruskal(n,edges):
    dsu=DisjointsetUnion(n)
    mst_weight=0
    for weight,u,v in sorted(edges):
        if dsu.union(u,v):
            mst_weight+=weight
    return mst_weight
```

拓扑排序（邻接表）——有向图是否有环

```
from collections import deque, defaultdict
#实际应用中可能需要import heapq
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:
            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:
                queue.append(v)
    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def hasCycle(graph, v, visited, recStack):
    visited.add(v)
    recStack.add(v)
    for nei in graph[v]:
        if nei not in visited:
            if hasCycle(graph, nei, visited, recStack):
                return True
        elif nei in recStack:
            return True
```

```
    recStack.remove(v)
    return False

T = int(input())
anslst = []
for _ in range(T):
    N, M = map(int, input().split())
    graph = {i: set() for i in range(1, N+1)}
    for _ in range(M):
        x, y = map(int, input().split())
        graph[x].add(y)

    ans = 'No'
    visited = set()
    for i in range(1, N+1):
        if hasCycle(graph, i, visited, set()):
            ans = 'Yes'
            break
    anslst.append(ans)

for ans in anslst:
    print(ans)
```

Bellman-Ford算法——单源最短路径

Floyd-Warshall算法——多源最短路径，在有向图或无向图中找到任意两点最短路径

```
def floyd_warshall(graph):#graph邻接表
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]
    for k in range(n):
        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
```

prim算法——最小生成树

```
#返回最小生成树的所有权重之和
class Node:
    def __init__(self,v):
        self.value=v
        self.joint=set()#存储相邻节点
        self.way=dict()#存储到相邻节点的边的权重

def prim(x,num,al,ans,farm):#num是总节点数量，al是已经遍历的节点数，x是初始节点，ans是权重之和，farm是包含所有节点的列表
```

```
    visited=set()
    visited.add(x)
    min_way=[100000]*num#设置inf的初始值
    while al!=num:#当还没有遍历完每个节点的时候
        for a in range(numm):
            if min_way[a]!=200000:
                for b in visited:
                    if farm[a] in b.joint:

                        min_way[a]=min(min_way[a],b.way[farm[a]])#更新最小距离
        shortest=min(min_way)
        y=min_way.index(shortest)
        ans+=shortest
        visited.add(farm[y])
        min_way[y]=200000#表示该节点已经访问过
        al+=1
    if al==num:
        return ans
```

判断无向图是否联通/回路

```
class Vertex:
    def __init__(self,v):
        self.value=v
        self.connectedto={}

class Graph:
    def __init__(self):
        self.vertexes={}

    def add_vertex(self,s):
        self.vertexes[s]=Vertex(s)

    def add_edge(self,s1,s2,w):
        if s1 not in self.vertexes:
            self.vertexes[s1]=Vertex(s1)
        if s2 not in self.vertexes:
            self.vertexes[s2]=Vertex(s2)
        V1=self.vertexes[s1]
        V2=self.vertexes[s2]
        V1.connectedto[s2]=w
        V2.connectedto[s1]=w
        self.vertexes[s1]=V1
        self.vertexes[s2]=V2

g=Graph()
n,m=map(int,input().split())
for _ in range(n):
    g.add_vertex(_)

for _ in range(m):
    u,v=map(int,input().split())
    g.add_edge(u,v,1)

def connected(graph):
    if n==1:
        return 'connected:yes'
    visited=[False for i in range(n)]
    queue=[0]
    while queue:
        ver=queue.pop(0)
        neighbours=graph.vertexes[ver].connectedto.keys()
        for neighbour in neighbours:
            if not visited[neighbour]:
                visited[neighbour]=True
                queue.append(neighbour)
    for x in visited:
        if not x:
            return 'connected:no'
    return 'connected:yes'

def dfs(graph,begin,ver,visited,last):
#    print(f'dealing with {ver}, current visited:{visited}')
    if visited[ver] and ver==begin:
#        print('have loop')
        return True
    elif visited[ver] and ver!=begin:
        return False
    else:
        visited[ver]=True
        for  neighbour in graph.vertexes[ver].connectedto.keys():
            if neighbour!=last and dfs(graph,begin,neighbour,visited,ver):
                return True
        return False

def loop(graph):
    for i in range(n):
        visited=[False for i in range(n)]
        if dfs(graph,i,i,visited,i):
            return 'loop:yes'
    return 'loop:no'

print(connected(g))
print(loop(g))
```