1. 欧拉筛

①输出列表的

```
def euler_sieve(n):
    is_prime, primes = [True] * (n + 1), []
    for i in range(2, n + 1):
        if is_prime[i]: primes.append(i); (for j in range(i * i, n + 1, i):
is_prime[j] = False)
    return primes
```

②输出真值表的

```
def euler_sieve(n):
    is_prime = [False, False] + [True] * (n - 1)
    for i in range(2, n + 1):
        if is_prime[i]: (for j in range(i * i, n + 1, i): is_prime[j] = False)

    return is_prime
```

二、数据结构

1. 栈stack

```
class stack:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def push(self, item): self.items.append(item)#添加数据
    def pop(self): return self.items.pop()#弹出数据
    def peek(self): return self.items[len(self.items)-1]#查看数据
    def size(self): return len(self.items)#栈长度
```

2. 队列queue

```
class queue:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def enqueue(self, item): self.items.insert(0, item)#添加数据
    def dequeue(self): return self.items.pop()#弹出数据
    def size(self): return len(self.items)#队列长度
```

3. 双端队列deque

```
class deque:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def addFront(self, item): self.items.append(item)#添加数据
    def addRear(self, item): self.items.insert(0, item)#添加数据
    def removeFront(self): return self.items.pop()#弹出数据
    def removeRear(self): return self.items.pop(0)#弹出数据
    def size(self): return len(self.items)#双端队列长度
##from collections import deque
```

4. 链表linked_list

单向链表SinglyLinkedList

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def reverse(self):
        previous = None
        current = self.head
        while current:
            next_node = current.next
            current.next = previous
            previous = current
            current = next_node
        self.head = previous
```

2.树

(1)二叉树（基础)

根据每个节点左右子树建树

设共有n个节点，且节点的值分别为1~n，依次输入每个节点的左右子节点，若没有则输入-1

```
class Node: # 定义节点，用class实现
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None
n = int(input())
nodes = [node(_) for _ in range(1,n+1)]
for i in range(n):
    l,r = map(int,input().split())
    if l != -1: # 一定要先判断子节点是否存在
        nodes[i].left = nodes[l]
    if r != -1:
```

```
        nodes[i].right = nodes[r]
# 这一方法中，指针只能表示相邻两层之间的关系
```

根据前中/中后序序列建树

以前中序为例

- 前提是每个节点的值不同，否则不方便用index()

```
def build_tree(preorder, inorder):
    if not preorder or not inorder: # 先判断是否为空树
        return None
    root_value = preorder[0]
    root = Node(root_value)
    root_index = inorder.index(root_value)
    root.left = build_tree(preorder[1:root_index+1], inorder[:root_index]) #递归
    root.right = build_tree(preorder[root_index_inorder+1:], 
inorder[root_index_inorder+1:])
    return root
```

根据扩展前/后序序列建树

以前序为例，设preorder中空的子节点用'.'表示

```
def build_tree(preorder):
    if not preorder: # 先判断是否为空树
        return None
    value = preorder.pop() # 倒序处理（若给后序，则正序处理）
    if value == '.':
        return None
    root = Node(value)
    root.left = build_tree(preorder) # 递归是树部分的关键思想
    root.right = build_tree(preorder)
    return root
```

计算深度

- 高度=深度-1（空树深度为0，高度为-1）

```
def depth(root):
    if root is None: # 先判断是否为空树
        return 0 # 若计算高度，则return -1
    else:
        left_depth = depth(root.left) # 递归
        right_depth = depth(root.right)
        return max(left_depth, right_depth)+1
```

计算叶节点数目

```python
def count_leaves(root):
    if root is None: # 先判断是否为空树
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left)+count_leaves(root.right)
```

前/中/后序遍历

- DFS
- 特别地，BST的中序遍历就是从小到大排列
  以后序为例（前序：C→A→B，中序：A→C→B）

```python
def post_order_traversal(root):
    output = []
    if root.left: # part A
        # 先判断子节点是否存在
        output.extend(post_order_traversal(root.left))
        # 是extend而不是append
    if root.right: # part B
        output.extend(post_order_traversal(root.right))
    output.append(root.value) # part C
    return output
```

层次遍历

- BFS

```python
from collections import deque
def level_order_traversal(root):
    q = deque()
    q.append(root)
    output = []
    while q:
        node = q.popleft()
        output.append(node.value)
        if node.left: # 仍然是先判断子节点是否存在
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return output
```

一般树遍历

括号嵌套树

```python
class TreeNode:
    def __init__(self, value): #类似字典
        self.value = value
        self.children = []

def parse_tree(s):
    stack = []
    node = None
    for char in s:
```



```python
        if char.isalpha():  # 如果是字母，创建新节点
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点


def preorder(node):
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def postorder(node):
    output = []
    for child in node.children:
        output.extend(postorder(child))
    output.append(node.value)
    return ''.join(output)
```

表达式建树

```python
def build_tree(postfix):
    stack = []
    for char in postfix:
        node = TreeNode(char)
        if char.isupper():
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]
```

Huffman编码

```python
import heapq

class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

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

```

```python
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        #merged = Node(left.weight + right.weight) #note: 合并后，char 字段默认值是空
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

        #if node.char:
        if node.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded
```

(3)BST

根据数字列表建树

- 每次从列表中取出一个数字插入BST

```python
def insert(root,num):
    if root is None: # 先判断是否为空树
        return node(num)
    if num < root.value:
        root.left = insert(root.left,num)
    elif num > root.value:
        root.right = insert(root.right,num)
    return root
```

27928:遍历树（按大小的递归遍历）

遍历规则：遍历到每个节点（值为互不相同的正整数）时，按照该节点和所有子节点的值从小到大进行遍历。
输入的第一行为节点个数n，接下来的n行中第一个数是此节点的值，之后的数分别表示其所有子节点的值；分行输出遍历结果。

```python
class Node:
    def __init__(self,value):
        self.value = value
        self.children = []
        # self.parent = None（有些题中需要，便于确定节点归属）
def traverse_print(root,nodes):
    if root.children == []: # 同理，先判断子节点是否存在
        print(root.value)
        return
    to_sort = {root.value:root} # 此处利用value来查找Node，而不是用指针（因为多叉树的指针往往只能表示相邻两层之间的关系）
    for child in root.children:
        to_sort[child] = nodes[child]
    for value in sorted(to_sort.keys()):
        if value in root.children:
            traverse_print(to_sort[value], nodes) # 递归
        else:
            print(root.value)
n = int(input())
nodes = {}
children_list = [] # 用来找根节点
for i in range(n):
    l = list(map(int,input().split()))
    nodes[l[0]] = Node(l[0])
    for child_value in l[1:]:
        nodes[l[0]].children.append(child_value)
        children_list.append(child_value)
root = nodes[[value for value in nodes.keys() if value not in children_list][0]]
traverse_print(root,nodes)
```

(5)Trie

构建

```
class Node:
    def __init__(self,value):
        self.value = value
        self.children = []
def insert(root,num):
    node = root
    for digit in num:
        if digit not in node.children:
            node.children[digit] = Node()
        node=node.children[digit]
        node.cnt+=1
```

3.并查集

- 实质上也是树，元素的parent为其**父节点**，find所得元素为其所在集合（树）的**根节点**
- 有几个互不重合的集合，就有几棵独立的树

(1)列表实现parent

- 若parent[x] == y，则说明y是x所在集合的代表元素

```
parent = list(range(n+1))
# 将列表长度设为n+1是为了使元素本身与下标能够对应
```

(2)查询操作

- 目的是找到x所在集合的代表元素

```
def find(x):
    if parent[x] == x: # 如果x所在集合的parent就是x自身
        return x # 那么就用x代表这一集合
    else: # 递归，直到找到x所在集合的代表
        return find(parent[x])
```

(3)合并操作

- 目的是将y所在集合归入x所在集合

```
def union(self,x,y):
    x_rep,y_rep = find(x),find(y)
    if x_rep != y_rep:
        parent[y_rep] = x_rep
```

(4)rank优化

- rank表示代表某集合的树的深度
- 引入rank可保证合并后新树的深度最小

```
rank = [1]*n
# 以下是有rank时的合并操作
def union(self,x,y):
    x_rep,y_rep = find(x),find(y)
    if rank[x_rep] > rank[y_rep]:
        parent[y_rep] = x_rep
    elif rank[x_rep] < rank[y_rep]:
        parent[x_rep] = y_rep
    else:
        parent[y_rep] = x_rep
        rank[x_rep] += 1
```

4.图

(1)图的实现

- 通常用dict套list（有权值时为dict套dict）
- dict的键为各顶点，值为存储相应顶点所连顶点的list（或键为相应顶点所连顶点，值为相应边权值的dict）

(2)DFS

02386:Lake Counting（连通区域问题）

输入n行m列由'.'和'W'构成的矩阵，求'W'连通区域的个数

```
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)
```

01321:棋盘问题（回溯法）

每组数据的第一行n(n<=8)、k表示将在一个n*n的矩阵内描述棋盘，以及摆放k个棋子；随后的n行描述了棋盘的形状，'#'表示棋盘区域，'.'表示空白区域。要求任意两个棋子不能放在棋盘中的同一行或同一列，求所有可行的摆放方案数。

- 回溯法就是“走不通就退回再走”

```
chess = [['' for _ in range(8)] for _ in range(8)]
def dfs(now_row,cnt):
    global ans
    if cnt==k:
        ans += 1
        return
    if now_row==n:
        return # 走不通就退回
    for i in range(now_row,n): # 一行一行地找，当在某一行上找到一个可放入的'#'后，就开始找下一行的'#'，如果下一行没有，就从再下一行找
        for j in range(n):
            if chess[i][j]=='#' and not col_occupied[j]:
                col_occupied[j] = True
                dfs(i+1,cnt+1)
                col_occupied[j] = False # 若想在矩阵中寻找多条路径，访问完某点后要将其状态改回来
while True:
    n,k = map(int,input().split())
    if n==-1 and k==-1:
        break
    for i in range(n):
        chess[i] = list(input())
    col_occupied = [False]*8
    ans = 0
    dfs(0,0)
    print(ans)
```

(3)BFS

04115:鸣人和佐助（基于矩阵的BFS）

输入M行N列的地图（@代表鸣人，+代表佐助，*代表通路，#代表大蛇丸的手下）和鸣人初始的查克拉数量T（每一个查克拉可以打败一个大蛇丸的手下）。鸣人可以往上下左右四个方向移动，每移动一单位距离需要花费一单位时间。求鸣人追上佐助最少需要花费的时间（追不上则输出-1）。

- 本题的vis需要维护经过时的最大查克拉数t，只有t大于T值时候才能通过

```
from collections import deque
M,N,T = map(int,input().split())
graph = [list(input()) for i in range(M)]
dir = [(0,1),(1,0),(-1,0),(0,-1)]
for i in range(M): # 查找起点
    for j in range(N):
        if graph[i][j] == '@':
            start = (i,j)
def bfs(): # BFS也可以不定义函数直接写，此处是为了方便追不上时直接print(-1)
    q = deque([start+(T,0)])
```

```
    vis = [[-1]*N for i in range(M)] # 注意特殊的vis用法（维护t）
    vis[start[0]][start[1]] = T
    while q:
        x,y,t,time = q.popleft()
        time += 1
        for dx,dy in dir:
            if 0<=x+dx<M and 0<=y+dy<N: # 同样也要判断是否越界
                if graph[x+dx][y+dy]=='*' and t>vis[x+dx][y+dy]:
                    vis[x+dx][y+dy] = t
                    q.append((x+dx,y+dy,t,time))
                elif graph[x+dx][y+dy]=='#' and t>0 and t-1>vis[x+dx][y+dy]:
                    vis[x+dx][y+dy] = t-1
                    q.append((x+dx,y+dy,t-1,time))
                elif graph[x+dx][y+dy]=='+':
                    return time
    return -1
print(bfs())
```

(4)23163:判断无向图是否连通有无回路

- 注意是**无向图**
  输入第一行为顶点数n和边数m，接下来m行为u和v，表示顶点u和v之间有边。要求第一行输出“connected:yes/no"，第二行输出“loop:yes/no"。

```
n,m = map(int,input().split())
graph = [[] for _ in range(n)]
for _ in range(m):
    u,v = map(int,input().split())
    graph[u].append(v)
    graph[v].append(u)
def is_connected(graph):
    n = len(graph)
    vis = [False for _ in range(n)]
    cnt = 0
    def dfs(u):
        global cnt
        vis[u] = True
        cnt += 1
        for v in graph[u]:
            if not vis[v]:
                dfs(v)
    dfs(0)
    return cnt==n # 能从一个顶点出发搜索到其他顶点，说明连通
def has_loop(graph):
    n = len(graph)
    vis = [False for _ in range(n)]
    def dfs(u,x):
        vis[u] = True
        for v in graph[u]:
            if vis[v]==True: # 能从一个顶点出发搜索回到自身，说明有环
                if v!=x:
                    return True
            else:
                if dfs(v,u):
                    return True
```

```
        return False
    for i in range(n):
        if not vis[i]:
            if dfs(i,-1):
                return True
    return False
print('connected:yes' if is_connected(graph) else 'connected:no')
print('loop:yes' if has_loop(graph) else 'loop:no')
```

(5)拓扑排序

- 可判断**有向图**是否存在环
- 本质上是加了条件判断的BFS
  此处graph是dict套list的有向图

```
from collections import deque,defaultdict
def topological_sort(graph):
    indegree = defaultdict(int)
    order = []
    vis = set()
    for u in graph: # 统计各顶点入度
        for v in graph[u]:
            indegree[v] += 1
    q = deque()
    for u in graph:
        if indegree[u] == 0:
            q.append(u)
    while q:
        u = q.popleft()
        order.append(u)
        vis.add(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0 and v not in vis:
                q.append(v)
    if len(order) == len(graph):
        return order
    else: # 说明存在环
        return None
```

(6)最短路径（Dijkstra算法）

- 本质上是元素在队列中按**总距离**排序的BFS（一般的BFS按**步数**排序）
  此处graph用dict套list表示

```
import heapq
def dijkstra(start,end):
    q = [(0,start,[start])]
    vis = set()
    while q:
        (distance,u,path) = heappop(q) # q中元素自动按distance排序，先取出distance小的
        if u in vis:
            continue
```

```
        vis.add(u)
        if u == end:
            return (distance,path) # 可以记录并返回路径
        for v in graph[u]:
            if v not in vis:
                heappush(q,(distance+graph[u][v],v,path+[v]))
```

(7)最小生成树（Prim算法）

- 本质上是元素在队列中按**某一步**距离排序的BFS
  此处graph用dict套list表示

```
import heapq
vis = [False]*n # vis可用list（因为最小生成树有且仅有n个顶点），比set快
q = [(0,0)]
ans = 0
while q:
    distance,u = heappop(q) # 贪心思想，通过堆找到下一步可以走的边中权值最小的
    if vis[u]:
        continue
    ans += distance # 对于某一顶点，最先pop出来的distance一定是最小的
    vis[u] = True
    for v in graph[u]:
        if not vis[v]:
            heappush(q,(graph[u][v],v))
print(ans) # 返回最小生成树中所有边权值（距离）之和
```

二、笔试部分