机考

排序、栈、队列

逆波兰表达式求值

```
stack=[]
for t in s:
    if t in '+-*/':
        b,a=stack.pop(),stack.pop()
        stack.append(str(eval(a+t+b)))
    else:
        stack.append(t)
print(f'{float(stack[0]):.6f}')
```

中序表达式转后序表达式

```
pre={'+':1,'-':1,'*':2,'/':2}
for _ in range(int(input())):
    expr=input()
    ans=[]; ops=[]
    for char in expr:
        if char.isdigit() or char=='.':
            ans.append(char)
        elif char=='(':
            ops.append(char)
        elif char==')':
            while ops and ops[-1]!='(':
                ans.append(ops.pop())
            ops.pop()
        else:
            while ops and ops[-1]!='(' and pre[ops[-1]]>=pre[char]:
                ans.append(ops.pop())
            ops.append(char)
    while ops:
        ans.append(ops.pop())
    print(''.join(ans))
```

最大全0子矩阵

```
for row in ma:
    stack=[]
    for i in range(n):
        h[i]=h[i]+1 if row[i]==0 else 0
        while stack and h[stack[-1]]>h[i]:
            y=h[stack.pop()]
```

```
            w=i if not stack else i-stack[-1]-1
            ans=max(ans,y*w)
        stack.append(i)
    while stack:
        y=h[stack.pop()]
        w=n if not stack else n-stack[-1]-1
        ans=max(ans,y*w)
print(ans)
```

求逆序对数

```
from bisect import *
a=[]
rev=0
for _ in range(n):
    num=int(input())
    rev+=bisect_left(a,num)
    insort_left(a,num)
ans=n*(n-1)//2-rev
```

```
def merge_sort(a):
    if len(a)<=1:
        return a,0
    mid=len(a)//2
    l,l_cnt=merge_sort(a[:mid])
    r,r_cnt=merge_sort(a[mid:])
    merged,merge_cnt=merge(l,r)
    return merged,l_cnt+r_cnt+merge_cnt
def merge(l,r):
    merged=[]
    l_idx,r_idx=0,0
    inverse_cnt=0
    while l_idx<len(l) and r_idx<len(r):
        if l[l_idx]<=r[r_idx]:
            merged.append(l[l_idx])
            l_idx+=1
        else:
            merged.append(r[r_idx])
            r_idx+=1
            inverse_cnt+=len(l)-l_idx
    merged.extend(l[l_idx:])
    merged.extend(r[r_idx:])
    return merged,inverse_cnt
```

树

根据前中序得后序、根据中后序得前序

```
def postorder(preorder,inorder):
    if not preorder:
        return ''
    root=preorder[0]
    idx=inorder.index(root)
    left=postorder(preorder[1:idx+1],inorder[:idx])
    right=postorder(preorder[idx+1:],inorder[idx+1:])
    return left+right+root
```

```
def preorder(inorder,postorder):
    if not inorder:
        return ''
    root=postorder[-1]
    idx=inorder.index(root)
    left=preorder(inorder[:idx],postorder[:idx])
    right=preorder(inorder[idx+1:],postorder[idx:-1])
    return root+left+right
```

层次遍历

```
from collections import deque
def levelorder(root):
    if not root:
        return ""
    q=deque([root])
    res=""
    while q:
        node=q.popleft()
        res+=node.val
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return res
```

解析括号嵌套表达式

```
def parse(s):
    node=Node(s[0])
    if len(s)==1:
        return node
    s=s[2:-1]; t=0; last=-1
    for i in range(len(s)):
        if s[i]=='(': t+=1
        elif s[i]==')': t-=1
```

```
        elif s[i]==',' and t==0:
            node.children.append(parse(s[last+1:i]))
            last=i
    node.children.append(parse(s[last+1:]))
    return node
```

二叉搜索树的构建

```
def insert(root,num):
    if not root:
        return Node(num)
    if num<root.val:
        root.left=insert(root.left,num)
    else:
        root.right=insert(root.right,num)
    return root
```

并查集

```
class UnionFind:
    def __init__(self,n):
        self.p=list(range(n))
        self.h=[0]*n
    def find(self,x):
        if self.p[x]!=x:
            self.p[x]=self.find(self.p[x])
        return self.p[x]
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx!=rooty:
            if self.h[rootx]<self.h[rooty]:
                self.p[rootx]=rooty
            elif self.h[rootx]>self.h[rooty]:
                self.p[rooty]=rootx
            else:
                self.p[rooty]=rootx
                self.h[rootx]+=1
```

字典树的构建

```
def insert(root,num):
    node=root
    for digit in num:
        if digit not in node.children:
            node.children[digit]=TrieNode()
```

```
        node=node.children[digit]
        node.cnt+=1
```

图

bfs

```
from collections import deque
def bfs(graph, start_node):
    queue = deque([start_node])
    visited = set()
    visited.add(start_node)
    while queue:
        current_node = queue.popleft()
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

棋盘问题（回溯法）

```
def dfs(row, k):
    if k == 0:
        return 1
    if row == n:
        return 0
    count = 0
    for col in range(n):
        if board[row][col] == '#' and not col_occupied[col]:
            col_occupied[col] = True
            count += dfs(row + 1, k - 1)
            col_occupied[col] = False
    count += dfs(row + 1, k)
    return count
col_occupied = [False] * n
print(dfs(0, k))
```

dijkstra

```
# 1.使用vis集合
def dijkstra(start,end):
    heap=[(0,start,[start])]
    vis=set()
    while heap:
        (cost,u,path)=heappop(heap)
        if u in vis: continue
```

```
        vis.add(u)
        if u==end: return (cost,path)
        for v in graph[u]:
            if v not in vis:
                heappush(heap,(cost+graph[u][v],v,path+[v]))
# 2.使用dist数组
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

kruskal

```
uf=UnionFind(n)
edges.sort()
ans=0
for w,u,v in edges:
    if uf.union(u,v):
        ans+=w
print(ans)
```

prim

```
vis=[0]*n
q=[(0,0)]
ans=0
while q:
    w,u=heappop(q)
    if vis[u]:
        continue
    ans+=w
    vis[u]=1
    for v in range(n):
        if not vis[v] and graph[u][v]!=-1:
            heappush(q,(graph[u][v],v))
print(ans)
```

拓扑排序

```
from collections import deque
def topo_sort(graph):
    in_degree={u:0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v]+=1
    q=deque([u for u in in_degree if in_degree[u]==0])
    topo_order=[]
    while q:
        u=q.popleft()
        topo_order.append(u)
        for v in graph[u]:
            in_degree[v]-=1
            if in_degree[v]==0:
                q.append(v)
    if len(topo_order)!=len(graph):
        return []
    return topo_order
```

笔试

Sort algorithm:

1. 十大经典排序算法及动图演示

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

排序：是计算机程序设计中的一种重要操作，其功能是对一个数据元素集合或序列重新排列成一个按数据元素某个值有序的序列。

选择排序、堆排序和归并排序是最好和最坏复杂度相同的。 空间归并最大，但是是外部排序。

2. （1）quicksort双指针

①最好情况： 时间复杂度为：O(nlogn)：即每次划分产生的两个区间大小都为区间总长（设为n）的一半； 空间复杂度：o（lgn），也是一般情况（上面标错了） ②最坏情况： 时间复杂度为：O(n^2)，即每次划分产生的两个区间大小分别包含n-1个元素和1个元素； 空间复杂度为：O(n) 注：快排使用的空间为：O（1），真正消耗空间的是递归调用。 （2）mergesort从len=1的子表开始合并，递归先左后右 （3）冒泡排序、选择排序每次提取最大的到右边，冒泡是不断比较冒上来（同时交换），选择是直接找到最大的 （4）Shellsort：注意gapsize=前后2个元素索引差值，不是纯间隔的长度 （5）堆排序是选择排序，堆是完全二叉树，堆排序最适用于partial-sort.

堆排序： 基本步骤： 1.首先将待排序的数组构造成一个大根堆，此时，整个数组的最大值就是堆结构的顶端

2.将顶端的数与末尾的数交换，此时，末尾的数为最大值，剩余待排序数组个数为n-1

3.将剩余的n-1个数再构造成大根堆，再将顶端数与n-1位置的数交换，如此反复执行，便能得到有序数组

在数据结构中，从逻辑上可以把数据结构分成线性结构和非线性结构。 数据的基本单位是数据元素，数据的最小单位是数据项。 数据对象：是性质相同的数据元素的集合，是数据的子集 数据结构是数据对象，以及存在于该对象的实例合组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。

存储结构： 存储结构描述了数据在计算机内部的实际存储方式。它主要关注数据在计算机内存中的组织形式，以及如何在内存中存储和访问数据。存储结构通常包括以下几种： 顺序存储结构：数据元素在内存中按照一定的顺序依次存储，相邻的元素在内存中也是相邻的，如顺序表、数组，可以随机存取。 链式存储结构：数据元素通过指针相互连接，每个元素在内存中可以不连续存储，如链表。 链接存储的存储结构所占存储空间分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针,单链表数据密度<1 索引存储结构：除了数据元素本身外，还维护一个索引表，索引表中的每个元素指向相应数据元素的位置。 散列存储结构：通过散列函数将数据元素直接映射到内存地址，实现快速的查找。 存储结构与具体的数据结构有关，不同的数据结构可以选择不同的存储结构来实现。例如，数组通常使用顺序存储结构，链表则使用链式存储结构。

逻辑结构：描述了数据元素之间的逻辑关系，以及对这些关系的操作。它不涉及具体的存储细节，而是关注数据元素之间的关联和组织方式。通常要求同一逻辑结构中的所有数据元素具有相同的特性，这意味着不仅数据元素所包含的数据项的个数要相同，而且对应数据项的类型要一致

逻辑结构通常包括以下几种： 线性结构：数据元素之间存在一对一的关系，可以通过一维的方式排列，例如线性表、栈、队列等。 树形结构：数据元素之间存在一对多的关系，可以用树形方式组织，例如二叉树、堆、哈夫曼树等。 图形结构：数据元素之间存在多对多的关系，例如图、网等。 有序表也属于逻辑结构

逻辑结构与数据的抽象关系更为密切，它们描述了数据元素之间的逻辑联系和操作规则，而不关注数据元素的存储方式。例如，队列的逻辑结构就是先进先出（FIFO）的关系，而队列的存储方式可以是数组或链表。

单链表是一种链式存取的数据结构，用一组地址任意的存储单元存放线性表中的数据元素。链表中的数据是以结点来表示的，每个结点的构成：元素(数据元素的映象) + 指针(指示后继元素存储位置)，元素就是存储数据的存储单元，指针就是连接每个结点的地址数据。 线性表中的元素属于相同的数据类型，即每个元素所占的空间必须相同。

有向图的邻接矩阵、邻接表和逆邻接表_逆邻接表： 在邻接表中，对图中每个顶点建立一个单链表（区分邻接表和邻接矩阵，邻接矩阵是n**2的）

散列表： 散列函数和散列地址：在记录的存储位置p和其关键字key之间建立一个确定的对应关系H，使得p=H(key)，称这个对应关系H为散列函数，p为散列地址 散列表：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。 冲突和同义词：对不同的关键字可能得到同一散列地址，即key1 != key2，但是H(key1) == H(key2)，这种现象称为冲突。具有相同函数值的关键字对该散列表函数来说称为同义词，key1与key2互称为同义词。 eg：A称为关键字key，则A经过关系H后，在数组中的位置称为存储位置p；而不同的关键字8，4，经过关系（关键字%2）得到的存储地址都是0，称为冲突，而8，4则为同义词 1.开放地址法 开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 key 的初始散列地址 H0 = H(key)发生冲突时，以 H0 为基础，采取合适方法计算得到另一个地址 H1，如果 H1 仍然发生冲突，以 为基础再求下一个地址 H2，若 H2 仍然冲突，再求得 H3。依次类推，直至 Hk 不发生冲突为止，则 Hk 为该记录在表中的散列地址。 这种方法在寻找 “下一个” 空的散列地址时，原来的数组空间对所有的元素都是开放的所以称为开放地址法。通常把寻找 “下一个” 空位的过程称为探测，上述方法可用如下公式表示： H{i}=(H(key) +d{i})%m ， i=1,2,…,k(k≤m-l) 其中，H(key)为散列函数，m 为散列表表长，d为增量序列。根据d取值的不同，可以分为以下3种探测方法。 (1) 线性探测法 ： 使用线性探测法处理散列表碰撞问题，若表中仍有空槽（空单元），插入操作一定成功(T)这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

可以看出，上述三种处理方法各有优缺点。线性探测法的优点是：只要散列表未填满，总能找到一个不发生冲突的地址。缺点是：会产生“二次聚集”现象。而二次探测法和伪随机探测法的优点是：可以避免“二次聚集”现象。缺点也很显然：不能保证一定找到不发生冲突的地址。 （二次聚集：指在处理冲突过程中发生的两个第一个哈希地址不同的记录争夺同一个后继哈希地址的现象。）

链地址法(拉链法，开散列)：每个同义词建立一个链表

有 n（n>=2）个顶点的有向强连通图最少有 _ _ _ _ _ 条边。n 循环队列：
https://zhuanlan.zhihu.com/p/663842362?utm_id=0

1. 一个递归算法必须包括终止条件和递归部分
2. 回溯算法的核心思想是:从根结点出发,沿着某条路径向前搜索,当搜索到某一结点发现不满足要求时,就返回(回溯)到上一结点,继续尝试其他路径。回溯算法不需要使用队列来保存路径,而是通过递归调用、返回上一层等方式来实现状态的保存和恢复。 3.串是一种特殊的线性表，其特殊性体现在数据元素是一个字符,串是字符的有限序列,模式匹配是串的一种重要运算,串既可以采用顺序存储，也可以采用链式存储
3.
4.
5. prim适合稠密图，kruskal不能有负权边 树可以等价转化二叉树，树的先序遍历序列与其相应的二叉树的前序遍历序列相同 二叉树： 高度=层数-1，无向图节点度数=相连边数 度的定义：节点所拥有的子树的数目称为该节点的度 注意： 叶子节点的度为0 其中n0=n2+1 森林到二叉树的转换：森林是一组不相交的树。将森林转换为二叉树的方法通常称为“孩子-兄弟表示法”：每个节点的第一个孩子作为该节点的左孩子。每个节点的兄弟作为该节点的右孩子。二叉树中右指针域为空的节点，我们需要确定在这种转换中，二叉树中右指针域为空的节点的数量。 分析过程 非终端结点的定义：非终端节点是指在原始森林中有孩子的节点。 转换后的二叉树：在二叉树中，只有那些没有右孩子的节点，其右指针域为空。 如果一个节点没有兄弟节点，那么在二叉树中这个节点的右指针域为空。 在孩子-兄弟表示法中，以下几类节点会导致右指针为空： 每棵树的根节点（因为它们没有兄弟）。每个非终端节点的最右侧的孩子（因为它们没有右兄弟）。右指针为空的节点数量。每棵树的根节点的右指针域为空。每个非终端节点的最后一

个孩子的右指针域为空。考虑所有叶子节点，这些节点也没有右孩子。 由于根节点的右指针域为空，而森林的根节点数就是树的数量，每个树至少有一个根节点。如果我们设 n 是非终端节点的数量，那么根节点（第一层）的数量加上叶子节点的数量会影响右指针为空的节点总数。 关键点 对于森林转换成的二叉树： 如果 F 中有 n 个非终端节点，每个非终端节点的每个最后一个孩子节点的右指针域为空。 每棵树的根节点的右指针域为空，这些根节点数通常为 n+1（根节点数 = 非终端节点数 + 1） 因此，二叉树中右指针域为空的节点数为 n+1 结论 如果 F 中有 n个非终端结点，则由 F变换得的二叉树 B中右指针域为空的节点数为： n+1

DFS和BFS树：BFS遍历树和DFS遍历树_bfs树和dfs树-CSDN博客
https://blog.csdn.net/z13192905903/article/details/103306846

图！： 如果各边权值各不相同，存在唯一最小生成树 对任意一个连通的无向图，如果存在一个环，且这个环中的一条边的权值大于该环中 任意一个其它的边的权值，那么这条边一定不会是该无向图的最小生成树中的边。

拓扑排序只适用于DAG有向无环图

Kmp: Next: void Getnext(int next[],String t) { int j=0,k=-1; next[0]=-1; while(j<t.length-1) { if(k == -1 || t[j] == t[k]) { j++;k++; if(t[j]==t[k])//当两个字符相同时，就跳过 next[j] = next[k]; else next[j] = k; } else k = next[k]; } }

将栈顶第1个元素记为top1，栈顶的第2的元素记为top2， 注意：计算方式为 top2 运算符 top1 = 值