一、十大经典排序算法:

(一)冒泡排序:

1.算法步骤:

每次比较相邻两个元素，若前者比后者大，则交换。每一对元素重复上述操作，第一趟循环结束后，最后的元素会是最大的数。

2.代码实现:

```
def BubbleSort(arr):
    for i in range(1,len(arr)):
        for j in range(0,len(arr)-i):
            if arr[j]>arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
    return arr
```

3、时间复杂度、额外空间复杂度、稳定性:

时间复杂度平均O(n^2),最好O(n)[正序],最坏O(n^2)[反序];空间复杂度O(1);稳定

(二)选择排序:

1.算法步骤:

首先在序列中找到最小的元素放在排序序列的起始位置，然后每次在未排序序列中选择最小的元素，放在已排序序列的末尾。

2.代码实现:

```
def SelectionSort(arr):
    for i in range(len(arr)-1):
        minIndex=i
        for j in range(i+1,len(arr)):
            if arr[j]<arr[minIndex]:
                minIndex=j
        if i!=minIndex:
            arr[i],arr[minIndex]=arr[minIndex],arr[i]
    return arr
```

3、时间复杂度、额外空间复杂度、稳定性:

时间复杂度平均、最好、最坏都是O(n^2);空间复杂度O(1);不稳定

(三)插入排序:

1.算法步骤:

将待排序序列的第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

2.代码实现:

```
def InsertionSort(arr):
    for i in range(len(arr)):
        preIndex=i-1
        current=arr[i]
        while preIndex>=0 and arr[preIndex]>current:
            arr[preIndex+1]=arr[preIndex]
            preIndex-=1
        arr[preIndex+1]=current
    return arr
```

3.时间复杂度、额外空间复杂度、稳定性:

时间复杂度平均O(n^2),最好O(n),最坏O(n^2);空间复杂度O(1);稳定

(四)希尔排序:

1.算法步骤：
选择一个递减的步长序列 t1,t2,…,tk,其中 tk=1;按步长序列个数 k 对序列进行 k 趟排序；每趟排序中，根据对应的步长 ti 将待排序序列分割成若干长度为 m 的子序列，分别对各子序列进行直接插入排序。
2.代码实现：
```
def ShellSort(arr):
    gap=int(len(arr)/3)
    while gap>0:
        for i in range(gap,len(arr)):
            temp=arr[i]
            j=i-gap
            while j>=0 and arr[j]>temp:
                arr[j+gap]=arr[j]
                j-=gap
            arr[j+gap]=temp
        gap=int(gap/3)
    return arr
```
3.时间复杂度、额外空间复杂度，稳定性：
时间复杂度与步长有关，最好 O(n^1.3),最坏 O(n^2);空间复杂度 O(1);不稳定

(五)归并排序&求逆序数/交换次数：

1.算法步骤：
将原序列划分为两个子序列，然后将每个子序列继续划分，直到每个子序列只含 1 个元素；申请用于储存结果的空间；定义两个指针，分别指向两个子序列的第一个元素;依次取出指针值进行比较，将较小值加入合并空间，同时将较小值的指针后移一位;如果其中一个指针直到最后一位，则将另一个指针剩下的序列加入合并空间;重复上述操作。(求逆序数/交换次数时增加一步：如果右子序列指针的值小于左子序列的，需要累加左子序列长度与左指针索引的差)
2.代码实现：
```
def MergeSort(arr):
    n=len(arr)
    if n<=1:
        return arr,0
    middle=n//2
    left,left_swap=MergeSort(arr[:middle])
    right,right_swap=MergeSort(arr[middle:])
    swaps=left_swap+right_swap
    result=[]
    i=j=0
    while i<len(left) and j<len(right):
        if left[i]<right[j]:
            result.append(left[i])
            i+=1
        else:
            result.append(right[j])
            j+=1
            swaps+=(middle-i)
    if i==len(left):
        result.extend(right[j:])
    else:
        result.extend(left[i:])
    return result,swaps
```

3.时间复杂度、额外空间复杂度、稳定性：
时间复杂度平均、最好、最坏都是O(nlogn);空间复杂度O(n);稳定

(六)快速排序:

1.算法步骤：
从序列中选取一个元素作为基准；重新排序，所有比基准值小的元素放在基准前面，比基准值大的放在基准后面，在这个分区退出之后，该基准就处于序列的中间位置;递归地把小于基准值的子序列和大于基准值的子序列排序。
2.代码实现：(双指针法)

```
def QuickSort(arr,start,end):
    if start>=end:
        return
    middle,left,right=arr[start],start,end
    while left<right:
        while arr[right]>=middle and left<right:
            right-=1
        arr[left]=arr[right]
        while arr[left]<middle and left<right:
            left+=1
        arr[right]=arr[left]
    arr[left]=middle
    QuickSort(arr,start,left-1)
    QuickSort(arr,left+1,end)
QuickSort(arr,0,len(arr)-1)
```

3.时间复杂度、额外空间复杂度、稳定性：
时间复杂度平均、最好都是O(nlogn),最坏O(n^2);空间复杂度O(logn);不稳定

(七)堆排序&堆的构建:

1.算法步骤：
(1)将待排序列表中的数据按从上到下、从左到右的顺序构造成一棵完全二叉树;
(2)将完全二叉树中每个节点(叶节点除外)的值与其子节点(1个或2个)中较大的值进行比较,若该节点的值小于子节点的值，则交换它们的位置(大根堆，小根堆反之)；
(3)将节点与子节点进行交换后，要继续比较子节点与孙节点的值，直到不需要交换子节点或子节点时叶节点时停止；比较所有的非叶节点后即可构建堆结构;
(4)将堆顶与堆底交换，然后将堆底从堆中取出，添加到已排序序列中;
(5)重复步骤(2)(3)(4)直至堆中数据全部取出。
2.代码实现：

```
def Big_Heap(arr,start,end): #构建大根堆
    root=start
    child=root*2+1
    while child<=end:
        if child+1<=end and arr[child]<arr[child+1]:
            child+=1
        if arr[root]<arr[child]:
            arr[root],arr[child]=arr[child],arr[root]
            root=child
            child=root*2+1
        else:
            break
def HeapSort(arr): #堆排序
    first=len(arr)//2-1
    for start in range(first,-1,-1):
```

```
        Big_Heap(arr,start,len(arr)-1)
    for end in range(len(arr)-1,0,-1):
        arr[0],arr[end]=arr[end],arr[0]
        Big_Heap(arr,0,end-1)
    return arr
```

3.时间复杂度、额外空间复杂度、稳定性：

时间复杂度平均、最好、最坏都是O(nlogn);空间复杂度O(1);不稳定

(八)计数排序:

1.算法步骤：

(1)求待排序数组A中最大和最小的元素;

(2)统计数组中每个值为i的元素出现次数，存入数组B的第i项;

(3)将每个元素i放入数组C,每放一个元素就将B[i]减1。

2.代码实现：

```
def CountSort(arr):
    max_num=max(arr)
    count_arr=[0]*(max_num+1)
    for value in arr:
        count_arr[value]+=1
    new_arr=[]
    for i in range(len(count_arr)):
        while count_arr[i]!=0:
            new_arr.append(i)
            count_arr[i]-=1
    return new_arr
```

3.时间复杂度、额外空间复杂度、稳定性：

时间复杂度平均、最好都是O(n+k)(k表示序列中的最大值)，最坏O(n^2);空间复杂度O(k);稳定

(九)桶排序:

1.算法步骤：

(1)根据待排序列的数据范围，将序列划分为k个相同大小的子区间，每个区间为一个桶;

(2)将每个元素装入对应区间的桶中;

(3)对每个非空桶内的元素单独排序(使用插入排序/归并排序/快速排序等);

(4)按照区间顺序将桶内元素合并。

2.代码实现：

```
def BucketSort(arr,bucket_size):
    arr_min,arr_max=min(arr),max(arr)
    bucket_count=(arr_max-arr_min)//bucket_size+1
    buckets=[[] for _ in range(bucket_count)]
    for num in arr:
        buckets[(num-arr_min)//bucket_size].append(num)
    new_arr=[]
    for bucket in buckets:
        bucket.sort()
        （也可换成其他排序算法）
        new_arr.extend(bucket)
    return new_arr
```

3.时间复杂度、额外空间复杂度、稳定性：

空间复杂度O(n+k)(k为桶的个数);时间复杂度和是否稳定与(3)的排序方法有关

(十)基数排序:

1.算法步骤：

(1)求出待排序列中的最大值，并求出其位数，有多少位就需要进行多少轮分桶和合并;
(2)创建 0~9 共 10 个桶;
(3)从个位(或最高位)开始，按当前位数对元素进行分桶;
(4)分桶完成后，将所有桶中的数据进行合并，合并时按照先进先出的原则取出桶中的元素;
(5)重复(3)(4),直至对每一位数据都进行分桶与合并。
2.代码实现：

```
def RadixSort(arr):
    max_num=max(arr)
    place=1
    while max_num>=10**place:
        place+=1
    for i in range(place):
        buckets=[[] for _ in range(10)]
        for num in arr:
            radix=int(num/(10**i)%10)
            buckets[radix].append(num)
         j=0
         for k in range(10):
            for num in buckets[k]:
                arr[j]=num
                j+=1
    return arr
```

3.时间复杂度、额外空间复杂度、稳定性：
时间复杂度 O(d*(n+k))(d 为最大位数，k 为桶的个数);空间复杂度 O(n+k);稳定

二、栈：

(一)定义：

只允许在一端进行插入或删除的线性表，满足先进后出的原则。相关代码：

```
stack=[] #创建栈
stack.append(element) #入栈
top_element=stack.pop() #出栈
top_element=stack[-1] #获取栈顶元素但不删除
```

(二)应用场景：

1.合法出栈序列：

```
def is_possible_out_stack(orig,test):
    stack=[]
    index=0
    if len(orig)!=len(test):
        return False
    for char in test:
        if char not in orig:
            return False
        while not stack or stack[-1]!=char:
             if index==len(orig):
                 return False
             stack.append(orig[index])
            index+=1
        stack.pop()
```

```
    return True
```

2.出栈序列统计(卡特兰数)：

```
from collections import comb
n=int(input())
ans=int(comb(2*n,n)/(n+1))
```

3.括号匹配：

```
def bracket_matching(string):
    stack=[]
    leftbkt= '{[('
    rightbkt= '}])'
    for char in string:
        if char in leftbkt:
            stack.append(char)
        elif char in rightbkt:
            if not stack:
                return False
            if rightbkt.index(char)!=leftbkt.index(stack.pop()):
                return False
    return not stack
```

4.进制转换(10 进制转为 k 进制)：

除 k 取余法。待处理数字除以 k，余数入栈，商作为新的待处理数字，重复上述过程直至商为 0.将栈中元素依次输出即可。

```
num=int(input())
stack=[]
if num==0:
    print(num)
else:
    while num>0:
        rem=num%k
        stack.append(rem)
        num=num//k
    print( '' .join(map(str,stack)))
```

5.前、中、后序表达式互相转化：

前序表达式也叫波兰表达式，运算符在操作数之前；中序表达式是我们平时常用的算式，运算符在操作数中间；后序表达式也叫逆波兰表达式，运算符在操作数之后。例：中序表达式(1+4)*3-10/5 对应的前序、后序表达式分别为 - * + 1 4 3 / 10 5 ; 1 4 + 3 * 10 5 / -.

(1)中序表达式转后序表达式(调度场算法)：

①初始化操作符栈 operator 和用于储存结果的列表 postfix;

②**从左至右**扫描中缀表达式，遇到操作数时，将其加入 postfix；

遇到操作符时,比较其与 operator 栈顶操作符的优先级：

a.operator 为空或栈顶操作符为**左括号(** ,则将此操作符压入栈；

b.否则，若**优先级比栈顶操作符高**，也将此操作符压入栈；

c.否则，将 operator 栈顶元素弹出并加入到 postfix 中，再次转到 a 与 operator 中新的栈顶操作符相比较；

遇到括号时：

a.若为**左括号(** ,则直接压入 operator；

b.若为**右括号)**,则依次弹出 operator 栈顶的操作符直至遇到左括号，然后将这一对括号丢弃；

③重复步骤②直至表达式**最右边**，然后将 operator 中剩余操作符依次弹出并加入 postfix 即可.

```
def infix_to_postfix(infix):
    prec={ '+' :1, '-' :1, '*' :2, '/' :2}
```

```
    stack=[]
    postfix=[]
    for token in infix:
        if token.isdigit():
            postfix.append(token)
        elif token== '(' :
            stack.append(token)
        elif token== ')' :
            while stack[-1]!= '(' :
                postfix.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1]!= '(' and prec[token]<=prec[stack[-1]]:
                postfix.append(stack.pop())
            stack.append(token)
    while stack:
        postfix.append(stack.pop())
    return postfix
```

(2)中序表达式转前序表达式：

与(1)类似，但有几处变动：

①**从右至左**扫描中缀表达式；

②**右括号改为左括号，左括号改为右括号；**

③(1)②b.处判断条件改为**优先级不低于栈顶操作符；**

④重复直至表达式**最左边**，最后需**将 prefix 翻转.**

```
def infix_to_prefix(infix):
    prec={ '+' :1, '-' :1, '*' :2, '/' :2}
    stack=[]
    prefix=[]
    for token in infix[::-1]:
        if token.isdigit():
            prefix.append(token)
        elif token== ')' :
            stack.append(token)
        elif token== '(' :
            while stack[-1]!= ')' :
                prefix.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1]!= ')' and prec[token]<prec[stack[-1]]:
                prefix.append(stack.pop())
            stack.append(token)
    while stack:
        prefix.append(stack.pop())
    return prefix[::-1]
```

• **Tips:**方便起见可对输入数据进行如下处理

```
infix=input().replace( '+', ' + ').replace( '-', ' - ').replace( '*', ' * ').replace( '/', ' /
').replace( '(',  ' ( ').replace( ')', ' ) ' ).split() #增加空格,便于分离
print(*infix_to_…fix(infix))
```

(3)计算前序表达式：

方法一：从右向左遍历表达式，遇到操作数则入栈；遇到操作符则出栈两次获得操作数，其中第一次出栈的数作为被操作数，第二次出栈的数作为操作数，计算这一次的子表达式的值，然后将结果入栈

```
def calculate(prefix):
    stack=[]
    for token in prefix[::-1]:
        if token.isdigit():
            stack.append(token)
        else:
            a=int(stack.pop())
            b=int(stack.pop())
            if token== '+' :
                stack.append(a+b)
            elif token== '-' :
                stack.append(a-b)
            elif token== '*' :
                stack.append(a*b)
            elif token== '/' :
                stack.append(a/b)
    return stack[0]
```

方法二：用函数写递归

```
index=-1
def exp():
    global index
    index+=1
    a=string[index]
    if a=='+':
        return exp()+exp()
    if a=='-':
        return exp()-exp()
    if a=='*':
        return exp()*exp()
    if a=='/':
        return exp()/exp()
    else:
        return float(a)
```

(4)计算后序表达式：

从左向右遍历表达式，遇到操作数则入栈；遇到操作符则出栈两次获得操作数，其中第一次出栈的数作为操作数，第二次出栈的数作为被操作数，计算这一次的子表达式的值，然后将结果入栈

```
def calculate(postfix):
    stack=[]
    for token in postfix:
        if token.isdigit():
            stack.append(token)
        else:
            a=int(stack.pop())
            b=int(stack.pop())
            if token== '+' :
                stack.append(b+a)
            elif token== '-' :
```

```
                stack.append(b-a)
            elif token== '*' :
                stack.append(b*a)
            elif token== '/' :
                stack.append(b/a)
    return stack[0]
```

6.单调栈(monotone stack):

单调栈是一种特殊的栈，在栈的先进后出基础上，要求从栈顶到栈底的元素是单调递增/减的。

单调递增(减)栈：只有比栈顶元素小(大)的元素才能直接进栈，否则需要先将栈中比当前元素小(大)的元素出栈，再将当前元素入栈。以此保证栈中保留的都是比当前入栈元素大(小)的值，且从栈顶到栈底的元素值是单调递增(减)的。

(1)寻找**左侧**第一个**比当前元素大**的元素(或其索引)：

从左到右遍历元素，构造**单调递增栈，**一个元素左侧第一个比它大的元素就是**将其压入栈**时的栈顶元素，如果栈为空则说明左侧不存在比该元素大的元素；

```
def monotone_increasing_stack(nums):
    ans=[None]*len(nums)
    stack=[]
    for i in range(len(nums)):
        while stack and nums[stack[-1]]<=nums[i]:
            stack.pop()
        if stack:
            ans[i]=nums[stack[-1]]或 stack[-1]
        stack.append(i)
```

(2)寻找**左侧**第一个**比当前元素小**的元素(或其索引)：

从左到右遍历元素，构造**单调递减栈，**一个元素左侧第一个比它小的元素就是**将其压入栈**时的栈顶元素，如果栈为空则说明左侧不存在比该元素小的元素；

```
def monotone_decreasing_stack(nums):
    ans=[None]*len(nums)
    stack=[]
    for i in range(len(nums)):
        while stack and nums[stack[-1]]>=nums[i]:
            stack.pop()
        if stack:
            ans[i]=nums[stack[-1]]或 stack[-1]
    stack.append(i)
```

(3)寻找**右侧**第一个**比当前元素大**的元素(或其索引)：

从右到左遍历元素，构造**单调递增栈，**一个元素右侧第一个比它大的元素就是**将其压入栈**时的栈顶元素，如果栈为空则说明右侧不存在比该元素大的元素；

```
def monotone_increasing_stack(nums):
    ans=[None]*len(nums)
    stack=[]
    for i in range(len(nums)-1,-1,-1):
        while stack and nums[stack[-1]]<=nums[i]:
            stack.pop()
        if stack:
            ans[i]=nums[stack[-1]]或 stack[-1]
    stack.append(i)
```

(4)寻找**右侧**第一个**比当前元素小**的元素(或其索引)：

从右到左遍历元素，构造**单调递减栈，**一个元素右侧第一个比它小的元素就是**将其压入栈时**的栈顶元素，如果栈为

空则说明右侧不存在比该元素小的元素；

```
def monotone_decreasing_stack(nums):
    ans=[None]*len(nums)
    stack=[]
    for i in range(len(nums)-1,-1,-1):
        while stack and nums[stack[-1]]>=nums[i]:
            stack.pop()
        if stack:
            ans[i]=nums[stack[-1]]或 stack[-1]
    stack.append(i)
```

(5)相关题目：

①oj28190 奶牛排队：寻找最长连续子序列，满足最左端的元素 a 值最小，最右端的 b 值最大，中间的元素位于(a,b)内。

思路：找左侧第一个不小于当前元素的索引 l 和右侧第一个不大于当前元素的索引 r；对当前元素(其索引为 i)，索引为 l+1 到 i-1 的元素都比当前元素小，可能成为 a；从左到右遍历这些元素，首个满足右侧第一个不大于该元素的索引大于 i 的元素(其索引为 j)使索引为 i 的元素成为 b，从而 i-j+1 可能是答案，不断取 max 即可 细节处理的代码如下：

```
for i in range(N):
    for j in range(left_bound[i]+1,i):
        if right_bound[j]>i:
            ans=max(ans,i-j+1)
            break
```

②oj26977 接雨水：数字代表每个位置的柱子高度，求能容纳的水量。

思路：求左右两侧大于当前元素的最大元素，此时只需把索引记录由 stack[-1]改为 stack[0]即可

7.dfs，回溯(详见树、图部分)

三、队列&双端队列：

(一)定义：

只允许在一端进行插入或删除的线性表，满足先进先出的原则。相关代码：

```
from collections import deque
queue=deque([])#创建(双端)队列
queue.append(element) #入队
top_element=stack.popleft() #出队
top_element=queue[0] #获取队首元素但不删除
```

(二)应用场景：

1.bfs(详见树、图部分)

2.oj02746 约瑟夫问题&oj03253 约瑟夫问题 NO.2:可用双端队列模拟环

3.其他补充：

(1)链式队列：

```
class Node:
    def __init__(self,val):
        self.val=val
        self.next=None
class Queue:
    def __init__(self):
        self.head=None
        self.tail=None
```

```
    def is_empty(self):
        return self.head is None
    def enqueue(self,val):
        new_node=Node(val)
        if self.head is None:
            self.head=self.tail=new_node
        else:
            self.tail.next=new_node
            self.tail=new_node
    def dequeue(self):
         if self.is_empty():
             raise Exception(“Queue is empty”)
         val=self.head.val
         if self.head=self.tail:
             self.head=self.tail=None
         else:
             self.head=self.head.next
         return val
```

(2)环形队列:

```
class CircleQueue:
    def __init__(self,size):
        self.queue=[0 for i in range(size)]
        self.size=size
        self.front=0 #队首指针
        self.rear=0 #队尾指针
    def enqueue(self,val): #入队
        if not self.is_full():
            self.rear=(self.rear+1)%self.size
            self.queue[self.rear]=val
        else:
            print(“队列已满”)
    def dequeue(self): #出队
        if not self.is_empty():
            self.front=(self.front+1)%self.size
            return self.queue[self.front]
        else:
            print(“队列为空”)
    def is_empty(self): #判断队列是否为空
        return self.front==self.rear
    def is_full(self): #判断队列是否已满
        return (self.rear+1)%self.size==self.front
```

四、线性表：

（一）定义：

线性表是 n 个数据元素的有限序列。是一种常见的线性结构。它具有以下特点：第一个元素无前驱；最后一个元素无后继；除第一个元素和最后一个元素外，所有的元素都有前驱和后继。

（二）顺序存储——顺序表：

通过一组地址连续的存储单元对线性表中的数据进行存储，逻辑上相邻的两个元素在物理位置上也是相邻的。根据

值查找元素时，若为无序表，时间复杂度为O(n)；若为有序表，可进行折半查找，时间复杂度可优化为O(logn)。根据位置查找元素时，时间复杂度为O(1)。插入、删除元素时，时间复杂度为O(n).

```
class SeqList(object):
    def __init__(self,max): #初始化顺序表数组
        self.max=max #顺序表最大容量
        self.num=0
        self.data=[None]*self.max
    def is_empty(self): #判断线性表是否为空
        return self.num is 0
    def is_full(self): #判断线性表是否全满
        return self.num is self.max
    def __getitem__(self,index): #获取线性表中某一位置的值
        if not isinstance(index,int): #判断 index 是否是 int 型数据
            raise TypeError
        if 0<=index<self.max:
            return self.data[index]
        else: #索引越界
            raise IndexError
    def __setitem__(self,index,value): #修改线性表中的某一位置的值
        if not isinstance(index,int):
            raise TypeError
        if 0<=index<self.max:
            self.data[index]=value
        else:
            raise IndexError
    def locate_item(self,value): #按值查找第一个等于该值的位置
        for i in range(self.num):
            if self.data[i]==value:
                return i
        return -1
    def count(self): #返回线性表中元素的个数
        return self.num
    def insert(self,index,value): #在表中某一位置插入元素
        if self.num>=self.max:
            print("list is full")
        if not isinstance(index,int):
            raise TypeError
        if index<0 or index>self.num:
            raise IndexError
        for i in range(self.num,index,-1):
            self.data[i]=self.data[i-1]
        self.data[index]=value
        self.num+=1
    def remove(self,index): #删除表中某一位置的值
        if not isinstance(index,int):
            raise TypeError
        if index<0 or index>=self.num:
            raise IndexError
        for i in range(index,self.num):
```

```
            self.data[i]=self.data[i+1]
        self.num-=1
```

（三）链式存储：

1.单链表：

以结点来表示，每个结点包含两个域，一个数据域和一个指针域。数据域存储结点信息，指针域指向链表中的下一个结点，最后一个结点的指针域指向一个空值。基本构成：结点；head（头结点），head 结点永远指向第一个结点；tail（尾节点），tail 结点永远指向最后一个结点；null，链表中最后一个节点的指针域为 None 值。根据值查找元素时，时间复杂度为 O(n);根据位置查找元素时，不能随意访问，只能从头结点开始；插入、删除元素时，操作较为方便。

```
class Node: #定义链表结点类
    def __init__(self,value):
        self.value=value #数据域
        self.next=None #指针域
class LinkedList(object): #单链表类
    def __init__(self):
        self.head=None #创建头结点
        self.length=0 #初始化链表长度
    def is_empty(self): #判断链表是否为空
        return self.head is None
    def find_by_index(self,position): #获取链表中某一位置的值
        p=self.head
        index=0
        while p and index!=position:
            p=p.next
            index+=1
        return p.value
    def replace(self,position,new_value): #修改链表中的某一位置的值
        p=self.head
        index=0
        while p and index!=position:
            p=p.next
            index+=1
        if p:
            p.value=new_value
    def find_by_value(self,value): #根据值查找节点并返回位置
        index=0
        p=self.head
        while p and p.value!=value:
            p=p.next
            index+=1
            return p,index
    def __len__(self): #返回链表中元素的个数
        return self.length
    def insert_node_to_head(self, node): #头部插入
        if node:
            node.next=self.head
            self.head=node
    def insert_value_to_head(self,value):
        node=Node(value)
```

```
        self.insert_node_to_head(node)
    def insert_node_after(self,node,new_node): #结点后插入
        if not node or not new_node:
            return
        new_node.next=node.next
        node.next=new_node
    def insert_value_after(self,node,value):
        new_node=Node(value)
        self.insert_node_after(node,new_node)
    def insert_node_before(self,node,new_node): #结点前插入
        if not self.head or not node or not new_node:
            return
        if node==self.head:
            self.insert_node_to_head(new_node)
            return
        p=self.head
        while p and p.next!=node:
            p=p.next
        if not p:
            return
        new_node.next=node
        p.next=new_node
    def insert_value_before(self,value,node):
        new_node=Node(value)
        self.insert_node_before(node,new_node)
    def delete_by_node(self,node): #删除某个节点
        if not self.head or not node:
            return
        if node.next:
            node.value=node.next.value
            node.next=node.next.next
        p=self.head
        while p and p.next!=node:
            p=p.next
        if not p:
            return
        p.next=node.next
    def delete_by_value(self,value): #删除某个值对应的节点
        node,position=self.find_by_value(self,value)
        self.delete_by_node(node)
```

2.双向链表：

每个结点再增加一个指向链表中的上一个结点的指针域，使插入、删除元素的操作更简便。

3.循环链表：

将尾结点的下一个结点设置为头结点，从而形成环状结构；为方便查找尾结点，可改成只设置尾指针。

五、树

(一)树的相关概念：

1.树是 n(n≥0)个结点的有限集。当 n=0 时，称为空树。在任意一棵非空树中应满足：有且仅有一个特定的结点称

为根的结点；
当 n>1 时,其余结点可分为 m（m>0）个互不相交的有限集 T1,T2,…,Tm,其中每个集合本身又是一棵树,并且称为根的子树。
树具有以下两个特点：树的根结点没有前驱，除根结点外所有结点有且只有一个前驱；树中所有结点可以有零个或多个后继。故 **n 个结点的树中有 n-1 条边。**
2.考虑结点 N,根 R 到结点 N 的唯一路径上的任意结点,称为结点 N 的**祖先**。如结点 R 就是结点 N 的祖先,而结点 N 是结点 R 的**子孙；**路径上最接近结点 N 的结点 P(也即 N 的唯一前驱)称为 N 的**双亲,**而 N 为结点 P 的**孩子；根 R 是树中唯一没有双亲的结点；**有相同双亲的结点称为**兄弟。**
3.树中一个结点的孩子个数称为该**结点的度,**树中结点的最大度数称为**树的度；**度为 0(没有子女结点)的结点称为**叶子结点。**
4.**结点的层次**从树根开始定义,根结点为第 1 层,它的子结点为第 2 层,以此类推；双亲在同一层的结点互为**堂兄弟；树的深度**是树中结点的最大层数,**树的高度**通常是树的深度减 1(关于树的高度和深度的定义,不同地方有不同解释,需具体情况具体分析)
5.**森林**是 m (m>0)棵互不相交的树的集合。森林的概念与树的概念十分相近，因为只要把树的根结点删去就成了森林。反之，只要给 m 棵独立的树加上一个结点，并把这 m 棵树作为该结点的子树，则森林就变成了树。

(二)树的性质：

1.树中的结点数等于所有结点的度数加 1;(度对应子结点，而根结点不是任何一个结点的子节点)
2.度为 m 的树中第 i 层上至多有 m^(i-1)个结点(i≥1)
3.深度为 h 的 m 叉树至多有(m^h-1)/(m-1)个结点
4.具有 n 个结点的 m 叉树的最小高度为$[\log_m(n(m-1)+1)]$

(三)树的存储结构：

1.双亲表示法：

以一组连续空间存储树的结点，同时在每个结点中，附设一个指示器指示其双亲结点到链表中的位置。

```
class Node:
    def __init__(self,val):
        self.val=val
        self.parent=None
```

2.孩子表示法：

将每个结点的孩子结点排列起来，以单链表作为存储结构，则 n 个结点有 n 个孩子链表，如果是叶子结点则此单链表为空。n 个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中。

```
class Node:
    def __init__(self,val):
        self.val=val
        self.first_child=None
class ChildNode:
    def __init__(self):
        self.index=-1
        self.next_sibling=None
```

3.(左)孩子(右)兄弟表示法：

又称为二叉树表示法，包括三部分：结点值、指向结点第一个孩子结点的指针、指向结点下一个兄弟结点的指针。也常用该方法将一棵树转化为二叉树。

```
class Node:
    def __init__(self,val):
        self.val=val
         self.first_child=None
         self.next_sibling=None
```

(四)二叉树的相关概念：

1.二叉树是另一种树形结构，其特点是每个结点至多只有左、右两棵子树(即二叉树中不存在度大于 2 的结点)；
2.二叉树是有序树，即互换左右子树会得到另一棵不同的二叉树，即使树中结点只有一棵子树，也需区分它是左子

树还是右子树。
3.斜树：所有结点都只有左子树的二叉树叫左斜树。所有结点都只有右子树的二叉树叫右斜树。二者统称为斜树。
4.满二叉树：深度为 h 且含有 2^h-1 个结点的二叉树称为满二叉树，即树中的每层都含有最多的结点。满二叉树的叶子结点都在二叉树的最后一层，且除叶子结点之外每个结点的度数均为 2。若对满二叉树从上至下、从左至右编号，则对于编号为 i 的结点，若有双亲，其双亲的编号为 i//2；若有左孩子，其左孩子的编号为 2*i;若有右孩子，其右孩子的编号为 2*i+1。
5.完全二叉树：深度为 h 且有 n 个结点的二叉树，若按上述方式编号后每个结点的编号与深度为 h 的满二叉树中编号为 1~n 的结点一一对应时，成为完全二叉树。其特点如下：若 i≤n//2，则结点 i 为双亲节点，否则为叶子结点；叶子结点只可能在层次最大的两层上出现，最大层次中的叶子结点都在该层的最左边，次大层次中的叶子结点都在该层的最右边；若有度为 1 的结点，则只能有一个，且该结点有左孩子而没有右孩子；若编号为 i 的结点只有左孩子或者是叶子结点，则编号大于 i 的结点全是叶子结点；若 n 为奇数，则每个双亲结点都有左孩子和右孩子;若 n 为偶数，则编号最大的双亲结点(编号为 n/2)只有左孩子，没有右孩子，其余双亲结点左、右孩子都有。

(五)二叉树的性质：

1.具有树的所有性质；
2.非空二叉树上的叶子结点数等于度为 2 的结点数加 1 (n_0+n_1+n_2=2*n_2+n_1+1,即 n_0=n_2+1)

(六)二叉树的存储结构：

1.顺序存储：适用于满/完全二叉树，一般二叉树需补全空结点
2.链式存储：与(三)中类似

```
class Node:
    def __init__(self,val):
        self.val=val
        self.left=None
        self.right=None
        self.parent=None
```

(七)二叉树的遍历：

1.先序遍历：

访问根结点→先序遍历左子树→先序遍历右子树

```
def preorder_travelsal(root):
    if not root:
        return
    res=[]
    res.append(root.val)
    res.append(preorder_traversal(root.left))
    res.append(preorder_traversal(root.right))
    return res
```

2.中序遍历：

中序遍历左子树→访问根结点→中序遍历右子树

```
def inorder_travelsal(root):
    if not root:
        return
    res=[]
    res.append(inorder_traversal(root.left))
    res.append(root.val)
    res.append(inorder_traversal(root.right))
    return res
```

3.后序遍历：

后序遍历左子树→后序遍历右子树→访问根结点

```
def postorder_travelsal(root):
    if not root:
```

```
        return
    res=[]
    res.append(postorder_traversal(root.left))
    res.append(postorder_traversal(root.right))
    res.append(root.val)
    return res
```

4.层次遍历(也称为广度优先遍历)：

按照从上到下的层次顺序，从左到右的结点顺序进行遍历

```
def level_traversal(root):
     res=[]
     if not root:
         return res
     queue=deque([root])
     while queue:
         node=queue.popleft()
         res.append(node.value)
         if node.left:
             queue.append(node.left)
         if node.right:
             queue.append(node.right)
     return res
```

5.根据前中遍历序列得后序遍历序列：

前序遍历序列的第一个一定是根结点，然后利用中序遍历序列可得到左右子树的中序遍历序列，进而得到左右子树的前序遍历序列，左右子树的前序遍历序列的第一个又是它们的根结点，如此递归下去即可。

```
def postorder(preorder,inorder):
    if not preorder or not inorder:
        return []
    root=preorder[0]
    root_index=inorder.index(root)
    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index+1:]
    left_preorder=preorder[1:len(left_inorder)+1]
    right_preorder=preorder[len(left_inorder)+1:]
    tree=[]
    tree.extend(postorder(left_preorder,left_inorder))
    tree.extend(postorder(right_preorder,right_inorder))
    tree.append(root)
    return tree
```

6.根据中后遍历序列得前序遍历序列：

与 5 类似，只是改成后序遍历序列的最后一个一定是根结点。

```
def preorder(inorder,postorder):
    if not inorder or not postorder:
        return []
    root=postorder[-1]
    root_index=inorder.index(root)
    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index+1:]
    left_postorder=postorder[:len(left_inorder)]
    right_postorder=postorder[len(left_inorder):-1]
```

```
    tree=[root]
    tree.extend(preorder(left_inorder,left_postorder))
    tree.extend(preoreder(right_inorder,right_postorder)
    return tree
```

注：无法根据前后遍历序列得到中序遍历序列，因为只能得到根结点的信息

（八）（二叉）树的应用：

1.二叉搜索树/二叉查找树/二叉排序树(BST)：

指满足以下性质的二叉树：若左子树非空，则左子树上所有结点的值均不大于它的根结点的值；若右子树非空，则右子树上所有结点的值不小于它的根结点的值；左右子树也分别为BST。

二叉树中的删除操作：

(1)如果待删除的结点是叶子结点，那么可以立即被删除；

(2)如果结点只有一个儿子，则将此结点的 parent 的孩子指针指向此结点的孩子，然后删除节点；

(3)如果结点有两个儿子，则将其右子树的最小数据代替此结点的数据，并将其右子树的最小数据删除。

```
class Node:
    def __init__(self,data):
        self.data=data
        self.lchild=None
        self.rchild=None
class BST:
    def insert(self,data): #插入
        flag,n,p=self.search(self.root,self.root,data)
        if not flag:
            new_node=Node(data)
            if data>p.data:
                p.rchild=new_node
            else:
                p.lchild=new_node
    def search(self,node,parent,data): #搜索
        if node is None:
            return False,node,parent
        if node.data==data:
            return True,node,parent
        if node.data>data:
            return self.search(node.lchild,node,data)
        else:
            return self.search(node.rchild,node,data)
    def delete(self, root, data): #删除
        flag,n,p=self.search(root,root,data)
        if flag is False:
            print "无该关键字，删除失败"
        else:
            if n.lchild is None:
                if n==p.lchild:
                    p.lchild=n.rchild
                else:
                    p.rchild=n.rchild
                del n
            elif n.rchild is None:
                if n==p.lchild:
```

```
                p.lchild=n.lchild
            else:
                p.rchild=n.lchild
            del n
        else:  #左右子树均不为空
            pre=n.rchild
            if pre.lchild is None:
                n.data=pre.data
                n.rchild=pre.rchild
                del pre
            else:
                next=pre.lchild
                while next.lchild is not None:
                    pre=next
                    next=next.lchild
                n.data=next.data
                pre.lchild=next.rchild
                del next
```

2.平衡二叉树(AVL)：

一种特殊的BST，每个结点应满足左子树与右子树高度差的绝对值不大于1。将二叉树上结点的左子树高度减去右子树高度的值称为结点的平衡因子(Blance Factor)，则平衡二叉树上所有结点的平衡因子只能为±1，0。

平衡二叉树插入结点导致失衡时恢复平衡的操作：

(1)LL型失衡(导致失衡的插入结点位于被破坏结点左孩子的左子树中)：

将被破坏结点A的左孩子B作为新根，将A作为B的右孩子，若B已有右孩子C，则将C作为A的左孩子。这个过程称为右旋；

(2)RR型失衡(导致失衡的插入结点位于被破坏结点右孩子的右子树中)：

将被破坏结点A的右孩子B作为新根，将A作为B的左孩子，若B已有左孩子C，则将C作为A的右孩子。这个过程称为左旋；

(3)LR型失衡(导致失衡的结点位于被破坏节点的左孩子的右子树中)：

以被破坏节点的左孩子为基础进行一次左旋，再以被破坏结点为基础进行一次右旋；

(4)RL型失衡(导致失衡的结点位于被破坏节点的右孩子的左子树中)：

以被破坏节点的右孩子为基础进行一次右旋，再以被破坏结点为基础进行一次左旋。

平衡二叉树删除节点导致失衡时恢复平衡的操作：

(1)删除右子树的结点且被破坏结点的左孩子的左子树高度大于或等于右子树：

相当于LL型失衡，右旋即可。

(2)删除右子树的结点且被破坏结点的左孩子的左子树高度小于右子树：

相当于LR型失衡，以左孩子为基础左旋后再以被破坏结点为基础右旋即可。

(3)删除左子树的结点且被破坏节点的右孩子的右子树高度大于或等于左子树：

相当于RR型失衡，左旋即可。

(2)删除左子树的结点且被破坏结点的右孩子的右子树高度小于左子树：

相当于RL型失衡，以右孩子为基础右旋后再以被破坏结点为基础左旋即可。

```
class Treenode:
    def __init__(self,val):
        self.val=val
        self.left=None
        self.right=None
        self.height=1
class AVL:
    def get_height(self,node): #获取树的高度
```

```
        if not node:
            return 0
        return node.height
    def get_balance_factor(self,node): #获取平衡因子
        if not node:
            return 0
        return self.get_height(node.left)- self.get_height(node.right)
    def right_rotate(self,lost_balance_node): #右旋操作
        new_node=lost_balance_node.left
        right_subtree=new_node.right
        new_node.right=lost_balance_node
        lost_balance_node.left=right_subtree
        lost_balance_node.height=1+max(self.get_height(lost_balance_node.left),self.get_height(lost
_balance_node.right))
        new_node.height=1+max(self.get_height(new_node.left),self.get_height(new_node.right))
        return new_node
    def left_rotate(self,lost_balance_node): #左旋操作
        new_node=lost_balance_node.right
        left_subtree=new_node.left
        new_node.left=lost_balance_node
        lost_balance_node.right=left_subtree
        lost_balance_node.height=1+max(self.get_height(lost_balance_node.left),self.get_height(lost
_balance_node.right))
        new_node.height=1+max(self.get_height(new_node.left),self.get_height(new_node.right))
        return new_node
    def insert(self,node,key): #插入操作
        if not node:
            return Treenode(key)
        if node.val>key:
            node.left=self.insert(node.left,key)
        if node.val<key:
            node.right=self.insert(node.right,key)
        node.height=1+max(self.get_height(node.left),self.get_height(node.right))
        balance_factor=self.get_balance_factor(node)
        if balance_factor>1 and key<node.left.val: #LL 型失衡，右旋一次
            return self.right_rotate(node)
        if balance_factor<-1 and key>node.right.val: #RR 型失衡，左旋一次
            return self.left_rotate(node)
        if balance_factor>1 and key>node.left.val: #LR 型失衡，失衡节点的左子树左旋一次，然后整个树右
旋一次
            node.left=self.left_rotate(node.left)
            return self.right_rotate(node)
        if balance_factor<-1 and key<node.right.val: #RL 型失衡，失衡节点右子树右旋一次，然后整个树左
旋一次
            node.right=self.right_rotate(node.right)
            return self.left_rotate(node)
        return node
avltree=AVL()
root=None
```

```
keys=list(map(int,input().split()))
for key in keys:
    root=avltree.insert(root,key)
```

3.Huffman 树与 Huffman 编码：

在许多应用中，树中结点常常被赋予一个表示某种意义的数值，称为该结点的权。该结点到根的路径长度(经过的边数)与它的权值的乘积称为**该结点的带权路径长度。**树中所有叶子结点的带权路径长度之和称为**该树的带权路径长度（WPL）。**在含有 n 个带权叶子结点的二叉树中，WPL 最小的二叉树**称为 Huffman 树，也称最优二叉树。**它的构造步骤如下：

(1)先把 n 个带权叶子结点按权值大小排序成一个有序序列；

(2)取权值最小的两个结点作为一个新结点的两个子结点，左孩子的权值相对较小；

(3)把两个子结点的权值和赋给新结点，将其插入(1)中的有序序列，并保持大小顺序；

(4)重复步骤(2)(3)直到出现根结点。

Huffman 编码是一种将字母串转化为二进制字符串的编码方式，把每个字母作为一个叶子结点，它的权是在字母串中出现的频率，按照上述步骤构造 Huffman 树后，从根到叶子结点按左 0 右 1 的方式对字母进行编码，这样得到的二进制字符串的平均长度最短。

```
class Node(object): #节点类
    def __init__(self,name=None,value=None):
        self._name=name
        self._value=value
        self._left=None
        self._right=None
class HuffmanTree(object): #哈夫曼树类
    def __init__(self,char_weights):
        self.a=[Node(part[0],part[1]) for part in char_ weights] #根据输入的字符及其频数生成叶子节点
        while len(self.a)!=1:
            self.a.sort(key=lambda node:node._value,reverse=True)
            c=Node(value=(self.a[-1]._value+self.a[-2]._value))
            c._left=self.a.pop()
            c._right=self.a.pop()
            self.a.append(c)
        self.root=self.a[0]
        self.b=range(*) #self.b 用于保存每个叶子节点的 Haffuman 编码,range 的值不小于树的深度即可
    def pre(self,tree,length):#用递归的思想生成编码
        node=tree
        if (not node):
            return
        elif node._name:
            return node._name,self.b[:length]
        self.b[length]=0
        self.pre(node._left,length+1)
        self.b[length]=1
        self.pre(node._right,length+1)
    def get_code(self): #生成哈夫曼编码
        self.pre(self.root,0)
```

4.字典树/前缀树/Trie 树：

据给定的字符串生成具有以下特点的树：根结点为空，把每个字符串的第一个字符作为根结点的子结点(相同的字母共用一个结点)，每个结点的子结点都指向字符串中的下一个字母，这样从根结点到每个叶子结点的路径都对应一个字符串，与字典类似。

```
class Node:
```

```
    def __init__(self):
        self.children={} #当前节点的子节点字典
        self.is_leaf=False
class Trie:
    def __init__(self):
        self.root=Node() #Trie 的根节点为空
    def insert(self,string): #添加新子树
        node=self.root
        for char in string:
            if char not in node.children:
                node.children[char]=Node()
            node=node.children[char]
        node.is_leaf=True
    def search(self,string):
        node=self.root
        for char in string:
            if char not in node.children:
                return False
            node=node.children[char]
        return node.is_leaf
```

5.并查集：

并查集是一种树型的数据结构，用于处理一些不相交集合的相关问题，有如下两个基本操作：

(1)查询：查询元素所属的集合，通常使用一个结点来代表整个集合，即一个元素的根结点/集合的代表元

(2)合并：将两个代表元不同的集合进行合并，并更新合并后集合的代表元。

```
class UnionFindSet(object):
    def __init__(self,n):
        self.p=list(range(n))
        self.h=[0]*n
    def find(self,x): #路径压缩
        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
```

六、图：

(一)图的相关概念：

1.图由顶点的有穷非空集合和顶点之间的边的集合组成，通常表示为 G(V,E)，其中 G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

2.一个图 G 若满足：不存在重复的边、顶点到自身的边，则称图 G 为简单图。数据结构中仅讨论简单图。
3.设有两个图 G=(V,E)和 G′ =(V′ ,E′),若 V′ 是 V 的子集，E′ 是 E 的子集，则称 G′ 是 G 的子图。
4.在一个图中，每条边都可以标上具有某种含义的数值，称为该边的**权值**。边上带有权值的图称为**带权图，**也称**网。**
5.若 E 是无向边(简称边)的有限集合，则图 G 是**无向图**。边是顶点的无序对，记为(v,w)或(w,v)，其中 v,w 是顶点，因为(v,w)=(w,v)，所以它们表示同一条边。
6.若 E 是有向边(也称弧)的有限集合，则图 G 为**有向图**。弧是顶点的有序对，记为<v,w>，其中 v,w 是顶点，v 称为弧尾，w 称为弧头，注意<v,w>≠<w,v>。
7.有 n 个顶点、n(n-1)/2 条边的无向图称为**完全图**，即任意两个顶点之间都存在边；有 n 个顶点、n(n-1)条弧的有向图称为**有向完全图，**在有向完全图中任意两个顶点之间都存在方向相反的两条弧。
8.边数很少的图称为**稀疏图**，反之称为**稠密图**。一般认为图 G 满足|E|<|V|log|V|时，可将其视为稀疏图。
9.图中每个顶点的**度**定义为以该顶点为一个端点的边的数目，记为 TD(v)。对于无向图，全部顶点的度等于边数的 2 倍。对于有向图，顶点的度分为**入度**和**出度，**入度是以顶点为终点的有向边的数目，记为 ID(V)；出度是以顶点为起点的有向边的数目，记为 OD(v)；顶点的度等于其入度和出度之和，即 TD(v)=ID(v)+OD(v)。
10.顶点 v_p到顶点 v_q之间的一条**路径**是指顶点序列 v_p，v_1，v_2，…，v_m，v_q，关联的边可理解为路径的构成要素。路径上边的数目称为**路径长度，**顶点 v_p到顶点 v_q路径长度的最小值称为它们之间的距离。第一个顶点和最后一个顶点相同的路径称为**环。**若一个无向图有 n 个顶点和大于 n-1 条边，则此图一定有环。
11.在无向图中，若从顶点 v 到顶点 w 有路径存在，则称 v 和 w 是**连通**的，若图 G 中任意两个顶点都是连通的，则称图 G 为**连通图，**否则称为**非连通图。**无向图中的极大连通子图称为**连通分量。**
12.在有向图中，若从顶点 v 到顶点 w 和顶点 w 到顶点 v 之间都有路径，则称 v 和 w 是**强连通**的。若图 G 中任意两个顶点之间都是强连通的，则称图 G 为**强连通图**。有向图中的极大强连通子图称为**强连通分量。**
13.连通图的生成树是包含图中全部顶点的一个极小连通子图。若图中顶点数为 n，则它的生成树含有 n-1 条边。对于生成树，删去其任一条边，都会变成非连通图。

(二)图的存储结构：

1.邻接矩阵：

用一个矩阵储存图中的边或弧的信息，设图 G 有 n 个顶点，则邻接矩阵 A 是一个 n 级方阵，若顶点 v_i与顶点 v_j之间有边/弧，则 A[i][j]=1,否则为 0。无向图的邻接矩阵一定是个主对角元全为 0 的对称矩阵，每个顶点的度就是顶点对应的行或列的非零元素个数；有向图的邻接矩阵主对角元全为 0，但不一定是对称矩阵，每个顶点的入度是顶点对应的列的非零元素个数，出度是对应的行的非零元素个数。对于带权图，若顶点 v_i与顶点 v_j之间有边/弧，则 A[i][j]=该边/弧的权值，若顶点 v_i与顶点 v_j是同一个顶点，则 A[i][j]=0，若顶点 v_i与顶点 v_j之间没有边/弧，则 A[i][j]=∞。

2.邻接表&逆邻接表：

若一个图为稀疏图，使用邻接矩阵会浪费存储空间，此时用邻接表存储更合适。邻接表是指对图 G 的每个顶点建立一个单链表，第 i 个单链表中的顶点表示依附于顶点 v_i的边(对于有向图则是以顶点 v_i为尾的弧)，这个单链表就称为顶点 v_i的边表(对于有向图则称为出边表)。边表的头指针和顶点的数据采用顺序存储(称为顶点表)
逆邻接表用于存储有向图，与邻接表的不同只是改为第 i 个单链表中的顶点表示以顶点 v_i为头的弧，得到顶点 v_i的入边表。

(三)图的遍历：

1.深度优先遍历(DFS):

(1)算法简介：

DFS 是一种系统地访问图中所有顶点的算法。它起始于一个初始顶点，然后沿着一条路径持续走到尽可能深的顶点，直至到达一个没有未访问邻接顶点的顶点为止，然后回溯并继续访问其他分支。基本步骤：
①选择一个起始顶点，将其标记为已访问。
②访问与起始顶点之间有边的顶点，根据要求处理当前节点。
③递归地对当前节点的所有未访问邻接节点进行深度优先遍历或者用栈来模拟递归的过程，将当前节点的所有未访问邻接节点压入栈中，然后从栈顶节点开始继续遍历。

(2)代码实现：

```
def dfs_recursive(graph,node,visited=None):
    if visited is None:
        visited=set()
```

```
    if node not in visited: #如果该节点未被访问过
        print(node) #处理当前节点
        visited.add(node) #标记节点为已访问
        for neighbor in graph[node]:
            dfs_recursive(graph,neighbor,visited) #递归访问邻接节点
def dfs_stack(graph, start):
    visited=set()
    stack=[start]
    while stack:
        node=stack.pop()
        if node not in visited:
            print(node) #处理当前节点
            visited.add(node) #标记节点为已访问
            for neighbor in reversed(graph[node]): #将未访问的邻接节点压入栈
                if neighbor not in visited:
                    stack.append(neighbor)
```

(3)应用：

①连通性问题，如最大连通域面积

②棋盘问题，如八皇后，骑士周游(优化：Warnsdoff's rule，每次访问具有最少未访问邻居的顶点)

2.广度优先遍历(BFS)：

(1)算法简介：

与 DFS 不同，BFS 从一个起始顶点开始，访问完当前顶点的所有邻接点之后，再访问下一层的所有顶点，整个过程需要维护队列。基本步骤：

①选择一个起始顶点，将其标记为已访问并放入队列。

②从队列中取出一个节点，根据要求处理当前节点后访问所有与之相邻且未被访问的节点，将这些节点标记为已访问并加入队列。

③重复以上步骤直到队列为空。

(2)代码实现：

```
from collections import deque
def bfs(graph,start):
    visited=set() #用于存储已访问的节点
    queue = deque([start]) #初始化队列并将起始节点放入队列
    while queue:
        node=queue.popleft() #从队列左侧取出一个节点
        if node not in visited:
            print(node) #处理当前节点，例如打印节点值
            visited.add(node) #标记节点为已访问
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor) #将未访问的邻居节点加入队列
```

(3)应用：求解最短路径

(四)带权图的最短路径(指边上的权值和最小)问题：

1.Dijkstra 算法：

从起始顶点开始，采用贪心算法的策略，每次访问与当前顶点之间的边的权值最小且未被访问的邻接顶点，直至扩展到终点位置。可用于有向图，但是不能存在负权值。

```
#使用 vis 集合
import heapq
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]))
#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
```

2.Floyd 算法：

定义一个 n 阶方阵序列 A_0，A_1，…，A_{n-1}。其中 A_0 是图 G 的邻接矩阵，$A_k[i][j]=\min\{A_{k-1}[i][j],A_{k-1}[i][k]+A_{k-1}[k][j]\}$。这是一个递归迭代的过程，经过 n 次迭代后得到的 $A_{n-1}[i][j]$ 就是 v_i 到 v_j 的最短路径长度。可用于带负权值的图但不能有环。

```
def floyd():
    n=len(graph)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if graph[i][k]+graph[k][j]<graph[i][j]:
                    graph[i][j]=graph[i][k]+graph[k][j]
```

(五)判断图是否连通&有环：

1.判断连通：

若 DFS/BFS 可以访问到所有顶点点，或者并查集进行合并后所有顶点的祖先是同一个，则说明连通。

2.无向图判环：

若 DFS/BFS 访问某顶点的邻接点时显示该邻接点已被访问且不是当前顶点的父顶点，或者并查集进行合并过程中发现两个顶点在合并之前已经属于同一个集合，则说明有环。

3.有向图判环：

若 DFS 过程中有顶点被第二次访问到，则说明有环。也可用拓扑排序(见下文)

4.拓扑排序：

(1)相关概念：

在一个表示工程的有向无环图(DAG)中，用顶点表示活动，用弧表示活动之间的优先关系，即<v，w>表示活动 v 必须先于活动 w，把 v 称为 w 的直接前驱，w 称为 v 的直接后继。这样的有向无环图称为 AOV 网(Activity On VertexNetwork)。

设 G=(V,E)是一个具有 n 个顶点的有向图，V 中的顶点序列 v_1，v_2，…，v_m，若满足每一个顶点都是它前一个顶点的直接后继，后一个顶点的直接前驱，则称为一个拓扑序列。拓扑排序就是对一个有向图构造拓扑序列的过程。

(2)算法步骤：

①从 AOV 网中选择一个没有前驱的顶点，也即入度为 0 的顶点并输出；

②从网中删除该顶点和所有以它为尾的弧；

③重复上述操作直至 AOV 网为空或者当前网中不存在没有前驱的顶点为止。**若输出的顶点数少于初始 AOV 网的顶点数，则说明有环。**

(3)代码实现：

```
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
```

(六)最小生成树(MST)：

对于一个带权无向图，边的权值和最小的那棵生成树称为 G 的最小生成树。

1.Prim 算法：

建立最小生成树时，将顶点按是否已包含在树中分为 A,B 两类。初始状态所有点都属于 B 类，然后任取一个点作为起始点，将它移至 A 类，在 B 类中查找与起始点相连且权值最小的点，再将该点移至 A 类。每次都从 B 类中查找与 A 类中的点相连且权值最小的点直到 B 类为空为止。

```
def prim(start,Graph):
    visited=set(start)
    total_weight=0
    while len(visited)<len(Graph):
        min_weight=float('inf')
        min_edge=None
        for node in visited:
            for edge in Graph[node]:
                if edge not in visited:
                    if Graph[node][edge]<min_weight:
                        min_weight=Graph[node][edge]
                        min_edge=edge
        if min_edge:
            total_weight+=min_weight
            visited.add(min_edge)
```

2.Kruskal 算法：

将所有权值按升序排列，每次对最小权值进行判断，如果不形成环就添加；否则不添加。是否形成环要用到并查集。

```
edges=[(顶点 1,顶点 2,对应边的权值),……]
vertices=list(……)
edges.sort(key=lambda x:x[2])
UnionFindSet=dict()
for i in vertices:
    UnionFindSet[i]=i
def find_node(x):#寻找根节点
    if UnionFindSet[x]!=x:
        UnionFindSet[x]=find_node(UnionFindSet[x])
    return UnionFindSet[x]
mst=[] #定义最小生成树
n=len(vertices)-1 #定义循环次数，n 为需要添加的边数=顶点数-1
for edge in edges:
    v1,v2,_=edge
    if find_node(v1)!=find_node(v2):
        UnionFindSet[find_node(v2)]=find_node(v1)
        mst.append(edge)
        n-=1
        if n==0:
            break
```

七、笔试补充：

(一)数据结构的相关概念：

1.数据：

数据是描述客观事物的符号，是计算机可以操作的对象，是能被计算机识别，并输入到计算机处理的符号集合。（数据不仅仅包括整型、实型等数值型，还有字符、声音、图像、视频等非数值类型）

2.数据元素：

数据元素是组成数据的、有一定意义的**基本单位，**在计算机中通常作为整体处理，也称为记录

3.数据项：

一个数据元素可以由若干个数据项组成，数据项是数据不可分割的**最小单位**

4.数据对象：

数据对象是性质相同的数据元素的集合，是数据的子集

5.数据结构：

数据结构是相互之间存在一种或多种特定关系的数据元素的集合，**包括逻辑结构、存储结构和数据的运算**

6.逻辑结构：

(1)定义：

逻辑结构是指数据对象中数据元素之间的相互关系（逻辑关系），即从逻辑关系上描述数据。它与数据的存储无关，是独立于计算机存储器的。

(2)分类（线性结构和非线性结构）：

根据数据元素之间关系的不同特征，通常有下列 4 类基本结构，复杂程度依次递进。

①集合：数据元素之间除了同属于一个集合外，没有其他的关系

②线性结构：数据元素之间是一对一的关系

③树形结构：数据元素之间是一对多的关系

④图状结构或网状结构：数据元素之间是多对多的关系

7.存储结构（物理结构）：

(1)定义：

数据的存储结构是指数据的逻辑结构在计算机中的存储方式，又称物理结构

(2)分类：

①顺序存储：利用数据元素在存储器中的相对位置来表示数据元素之间的逻辑顺序，把数据元素放在地址连续的存储单元中，程序设计中使用数组类型来实现（**逻辑相邻物理相邻**）

②链式存储：利用结点中指针来表示数据元素之间的关系，把数据元素存储在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的，，程序设计中使用指针类型来实现（**逻辑相邻物理不一定相邻**）

③散列存储：通过关键字直接计算出元素的物理地址

(二)散列表：

1.散列法、散列、散列表：

在查找数据对象时，由函数 h 对给定值 key 计算出地址，将 key 与该地址单元中数据对象关键字进行比较，确定查找是否成功。因此，散列法又称为“关键字-地址转换法”。散列方法中使用的计算函数称为散列函数（也称哈希函数），按这个思想构造的表称为散列表，所以它是一种存储方法。

2.装填因子：

一般情况下，设散列表空间大小为 m，填入表中的元素个数是 n,则称 α=n/m 为散列表的装填因子

3.同义词：

映射到同一散列地址上的关键字称为同义词

4.散列函数的构造方法：

•一个“好”的散列函数一般应考虑下列两个因素：计算简单，以便提高转换速度；关键词对应的地址空间分布均匀，以尽量减少冲突。即对于关键词集合中的任何一个关键字，经散列函数映射到地址集合中任何一个地址的概率是基本相等的。

(1)直接定址法：

h(key)=a*key+b(a，b 为常数)，这类函数计算简单，分布均匀，不会产生冲突，但要求地址集合与关键词集合大小相同，因此，对于较大的关键词集合不适用。

(2)除留取余法：

假设散列表长为 TableSize(TableSize 的选取，通常由关键词集合的大小 n 和允许最大装填因子 α 决定，一般将 TableSize 取为 n/α),选择一个正整数 p<=TableSize（最好选成不大于 TableSize 的最大素数）,取散列函数为：h(key)=key%p。

5.处理冲突的方法：

(1)开放定址法：

若发生了第 i 次冲突，试探的下一个地址将增加 d_i，基本公式是 $h_i(key)=(key+d_i)\%TableSize$，而 d_i 的选取又有如下几种方案：

①线性探测：$d_i=i$

②平方探测法：$d_i=\pm i^2$，以增量序列 1，-1，4，-4，……，$q^2,-q^2$ 且 $q\leq[TableSize/2]$循环试探下一个存储地址

③双散列探测法：d_i 为 $i*h_2(key)$,$h_2(key)$是另一个散列函数，注意对任意的 key，$h_2(key)\neq0$

(三)串：

1.相关概念：

(1)串(string)是由零个或多个字符组成的有限序列，又叫字符串，一般记为 S=‘$a_1a_2\cdots a_n$’ ($n\geq0$)。

(2)空串：n=0 时的串称为空串

(3)空格串：只包含空格的串。注意它与空串的区别!

(4)子串与主串：串中任意个数的连续字符组成的子序列称为该串的子串，相应地，包含子串的串称为主串。

(5)子串在主串中的位置就是子串的第一个字符在主串中的序号

2.串的模式匹配——KMP 算法：

•KMP 算法是一种字符串匹配算法，用于在一个文本串 S 中查找一个模式串 P 的出现位置。它的核心思想是利用已经匹配过的部分字符信息，避免不必要的回溯。主要步骤如下：

(1)预处理模式串 P：构建模式串的最长公共前后缀数组 next[]，用于记录每个位置之前的最长公共前后缀长度。

(2)在文本串 S 中匹配模式串 P：使用两个指针 i 和 j 分别指向文本串和模式串的当前位置，

①如果 S[i]等于 P[j],表示当前字符匹配成功，同时向后移动两个指针 i 和 j；

②如果 S[i]不等于 P[j]，则根据 next 数组进行回退操作：

a.如果 j 大于 0，则将 j 更新为 next[j-1]，并保持 i 不变。
b.如果 j 等于 0，则将 i 向后移动一位。
③重复步骤②直到找到匹配或者遍历完整个文本串。
•KMP 算法的关键在于如何构建最长公共前后缀数组 next[]。下面给出构建 next 数组的具体步骤：
(1)初始化 next[0]=0，j=0。
(2)从位置 i=1 开始遍历模式串 P：
①如果 P[i]等于 P[j]，表示找到了长度为 j+1 的最长公共前后缀，因此将 next[i]设置为 j+1,并同时向后移动两个指针 i 和 j；
②如果 P[i]不等于 P[j]，则需要回退 j，直到找到一个更短的最长公共前后缀或者 j 回退到 0。
a.如果 j 大于 0，则将 j 更新为 next[j-1]。
b.如果 j 等于 0，则将 next[i]设置为 0。
③重复步骤②直到遍历完整个模式串。
•KMP 算法的时间复杂度是 O(n+m)，其中 n 是文本串的长度，m 是模式串的长度。相比于朴素的字符串匹配算法（时间复杂度为 O(nm)），KMP 算法具有更高的效率。