基础知识

```
字符串：
    建立（转为）：
                数字转字符串：str()
                列表转字符串：“（连接符）”.join(list)

    处理:
        分割：str.split(“（分隔符）”)
        大写：str.upper()
        小写：str.lower()
        首字母：str.title()
        合并：+

    特征：长度：len()



列表：
    建立：list()
        列表推导式：[声明变量 for 声明变量 in 某集合 if 共同满足的条件]

    处理：
        切片：list[起点：终点+1]（不填默认起点为list[0],终点为列表末尾）
        遍历：for （临时变量名） in list：
        在末尾添加元素：list.append(元素)
        添加多个元素：list.extend([a,b,c,...])
        插入元素：list.insert(index,元素)；bisect库
        删除已知元素：list.remove(元素)
        删除已知索引的元素：del list[index]
        弹出元素：list.pop(index)
        顺序排序：list.sort()（ASCII码顺序）
        倒序排序：list.sort(reverse=True)
        指定顺序排序：list.sort(key= lambda s:排序指标（与s相关））
        拼接：list1+list2

     特征：
        长度：len()
        寻找索引：list.index(元素)
        正数第n个元素：list[n-1]
        倒数第n个元素：list[-n]
        元素个数：list.count(元素)
        itertools 库
        判断共有特征：all(特征 for 元素 in 列表)
        索引，元素元组：enumerate()函数（遍历方法：for index，元素代称 in enumerate(列
表))


字典：
    建立：
        {}
```

```
        dict(元组)
        半有序：Ordereddict()
    添加/修改键值对：dict[key]=value
    遍历字典的键：for 元素 in dict() ； for 元素 in dict.keys()（注：一定要加s!）
    遍历字典的值：for 元素 in dict.values()（一定要加s!）
    删除键值对：del dict[键]
    遍历键值对：for key,value in dict.items():
    按顺序遍历：for key in sorted(dict.keys()):



元组：
    建立：
        直接定义：(...,...,...,...)
        含元组的列表：zip(a,b,c,...)
        访问：元组[index]



集合：
    建立：
        set()
        向集合中添加元素：set.add()
        添加多个元素：set.update()
        删除元素：set.remove() 或set.discard()（前者有KeyError风险，后者没有）
        随机删除：set.pop()
        并集：set1 | set2（竖杠“|”在回车键上方）
        交集：set1 & set2
        差集（补集）：set1 - set2
        对称差集（补集之交）：set1^set2
        元素个数：len()
        不可变集合：frozenset()



堆：heapq库



库：
    注意：库中函数要先import之后才能使用。
    math库：
        向上取整：math.ceil()
        向下取整：math.floor()
        阶乘：math.factoria()
        数学常数：math.pi（圆周率），math.e（自然对数的底）
        开平方：math.sqrt(x)
        x的y次幂：math.pow(x,y)
        e的x次幂：math.exp(x)
        数函数：math.log(真数，底数)（不填底数默认为自然对数）
        三角：math.sin(),math.cos(),math.tan()
        反三角：math.asin(),math.acos(),math.atan()



heapq库：
    列表转堆：最小值在上层：heapq.heapify(list)；最大值在上层：
heapq._heapify_max(list)
    插入元素：heapq.heappush(堆名，被插元素)
```

```
    弹出元素：heapq.heappop(堆名)（可被命名为其他变量临时调用）
            （应用：堆排序：a=[heapq.heappop(b) for _ in range(len(b))],返回排序后的b）
    插入元素的同时弹出顶部元素：heapq.heappushpop(堆名，被插元素)
                              （或heapq.heapreplace(堆名，被插元素)）
    以上操作在最大堆中应换为“_a_max”（a是它们中的任意一个）

    建堆时，先定义一个空列表，然后一个一个往里面压入元素。


itertools库：
    整数集：itertools.count(x,y)（从x开始往大数的整数，间隔为y）
    循环地复制一组变量：itertools.cycle(list)
    所有排列：itertools.permutations(集合，选取个数)
    所有组合：itertools.combinations
    拼接列表的另一种方式：itertools.chain(list1,list2)
    已排序列表去重：[i for i,_ in itertools.groupby(list)]（每种元素只能保留一个）
                    或者list(group)[:n]（group被定义为分组，保留每组的n个元素）


 collections库：
    双端队列：创建：a=deque(list)
    从末尾添加元素：a.append(x)
    从开头添加元素：a.appendleft(x)
    从末尾删除元素：b=a.pop()
    从开头删除元素：b=a.popleft()
                  （其中b用于接收a弹出的元素）
    有序字典：Ordereddict()
    默认值字典：a=defaultdict(默认值)，如果键不在字典中，会自动添加值为默认值的键值对，而不报KeyError。
    计数器：Counter(str)，返回以字符种类为键，出现个数为值的字典


sys库：
    sys.exit()用于及时退出程序
    sys.setrecursionlimit()用于调整递归限制（尽量少用，递归层数过多会引起MLE）


statistics库：
    statistics 是 Python 标准库中用于统计学计算的模块，提供了各种用于处理统计数据的函数。
    常用函数：
        mean(data)：计算数据的平均值（均值）。
        harmonic_mean(data)：计算数据的调和平均数。
        median(data)：计算数据的中位数。
        median_low(data)：计算数据的低中位数。
        median_high(data)：计算数据的高中位数。
        median_grouped(data, interval=1)：计算分组数据的估计中位数。
        mode(data)：计算数据的众数。
        pstdev(data)：计算数据的总体标准差。
        pvariance(data)：计算数据的总体方差。
        stdev(data)：计算数据的样本标准差。
        variance(data)：计算数据的样本方差。
```

```
    数据处理：
        二进制：bin()
        八进制：oct()
        十六进制：hex()
        整型：int()
        浮点型：float()
        保留n位小数：round(原数字，保留位数)（如不写保留位数，则默认保留到整
数）；’%.nf’%原数字；’{:.nf}’.format(原数字)；
        n位有效数字：’%.ng’%原数字；’{:.ng}’.format(原数字)
        最大值max(),最小值min()
        ASCII转字符：chr();字符转ASCII：ord()
        判断数据类型：isinstance(object,class)

    其他：
        if， while循环；try， except 某error；
        类的创建：
            class type(father):
            def __init__(self,specific_level):
            self.character=specific_level
```

广度优先搜索的辅助：队列；伸缩：栈

算法的定义和特性

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表，元素值随机	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	length()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	find(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	remove(i)	删除下标为i的元素	O(n)

常用的算法思想

- **枚举法：** 对所有可能的解进行逐个验证，直到发现真正的解。
- **二分法：** 对于有些问题，将所有可能解排序，通过对位于解的查找区间中点的解进行一次验证，就可以找到解或缩小查找区间到原来的一半，这样就能很快找到解或宣告无解。
- **贪心法：** 在寻找解的过程中，每一步都只选取眼前最优的做法，不考虑后续影响。并不适用于所有需要求最优解的问题。
- **递归法和分治法：** 为解决问题，可以先采取一步行动，剩下的问题就变成和原问题形式相同、但是规模更小的问题，这样就可以用递归解决。或者，将原问题分解为几个和原问题形式相同、但是规模更小的子问题，子问题都解决，原问题也就解决，这就叫分治。分治往往用递归实现
- **深度优先搜索、回溯和分支限界法：** 在许多问题中，搜索解的过程，可以抽象为在迷宫中找出口。走迷宫的一个策略就是能往前走就往前走，这就叫深度优先；走不动了就回退到上一个岔路口选没走过的岔道继续走，这就叫回溯。有的情况下有办法预判一个岔道走下去肯定没前途，于是就不会走它，这就叫分支限界法。回溯和分支限界都是深度优先搜索过程中使用的手段
- **广度优先搜索法：** 解决问题，可能需要采取多步行动，每步行动都有不同选择。先把第一步能采取的所有选择都试一遍，看看问题有没有解决。如果没有，再把采取两步行动的所有方案都试一遍，看看问题有没有解决......这样当问题解决时，采取的步数一定是最少的

时间复杂度

表 13.7.1 各种排序算法的复杂度

算法分类	算法名称	时间复杂度			额外空间复杂度	是否稳定
		最好	平均	最坏		
插入排序	直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
	Shell 排序	$O(n)$	$O(n^{1.25})$	$O(n^2)$	$O(1)$	否
选择排序	简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
	堆排序	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$	否
交换排序	冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
	快速排序	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(\log(n))$	否
分配排序	桶排序	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$	是
	计数排序	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$	是
	基数排序	$O(d\times(n+m))$	$O(d\times(n+m))$	$O(d\times(n+m))$	$O(n+m)$	是
归并排序		$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$	是

- 常数复杂度：$O(1)$ 时间(操作次数)和问题的规模无关
- 对数复杂度：$O(\log(n))$
- 线性复杂度：$O(n)$
- 多项式复杂度：$O(n^k)$
- 指数复杂度：$O(a^n)$
- 阶乘复杂度：$O(n!)$

➢ O(1) 复杂度的常见操作

1) 根据下标访问列表、字符串、元组中的元素

2) 在集合、字典中增删元素

3) 调用列表的append函数在列表末尾添加元素，以及用pop()函数删除列表末尾元素

4) 用in判断元素是否在集合中或某关键字是否在字典中

5) 以关键字为下标访问字典中的元素的值

6) 用len函数求列表、元组、集合、字典的元素个数

➢ O(n) 复杂度的常见操作

1) 用in 判断元素是否在字符串、元组、列表中

2) 用insert在列表中插入元素

3) 用remove或del删除列表中的元素

4) 用字符串、元组或列表的find、rfind、index等函数做顺序查找

5) 用字符串、元组或列表的count函数计算元素出现次数

6) 用max,min函数求列表、元组的最大值，最小值

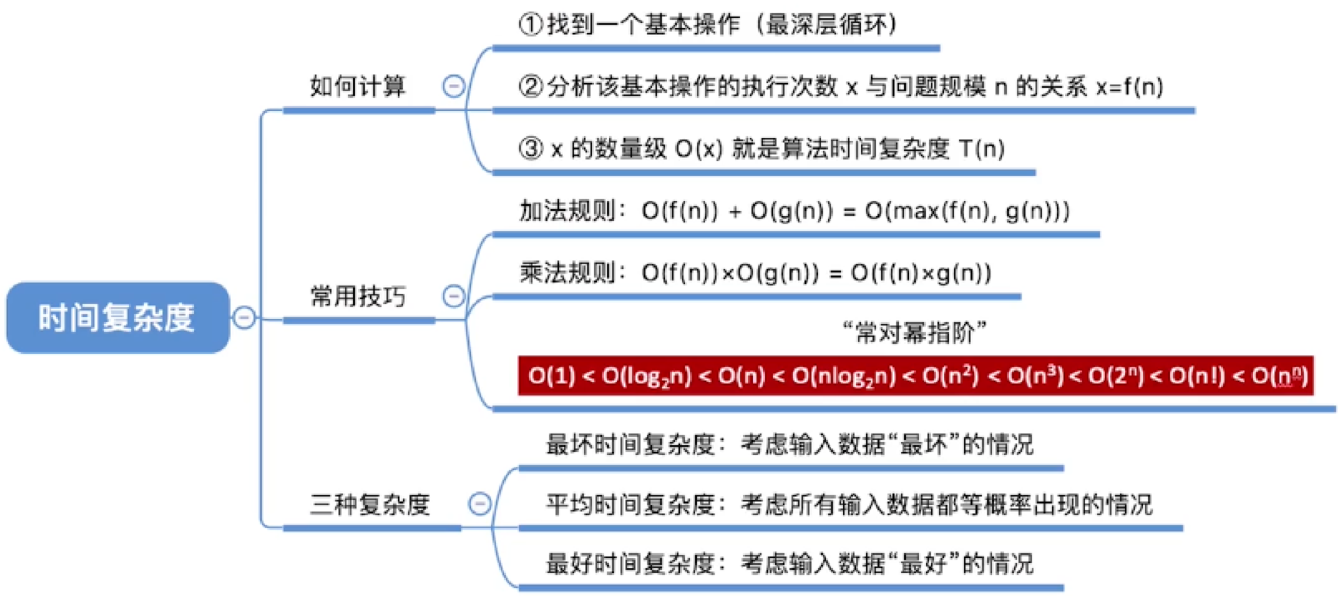
7) 列表和元组加法

➢ O(nlog(n)) 复杂度的常见操作

Python自带排序 sort,sorted

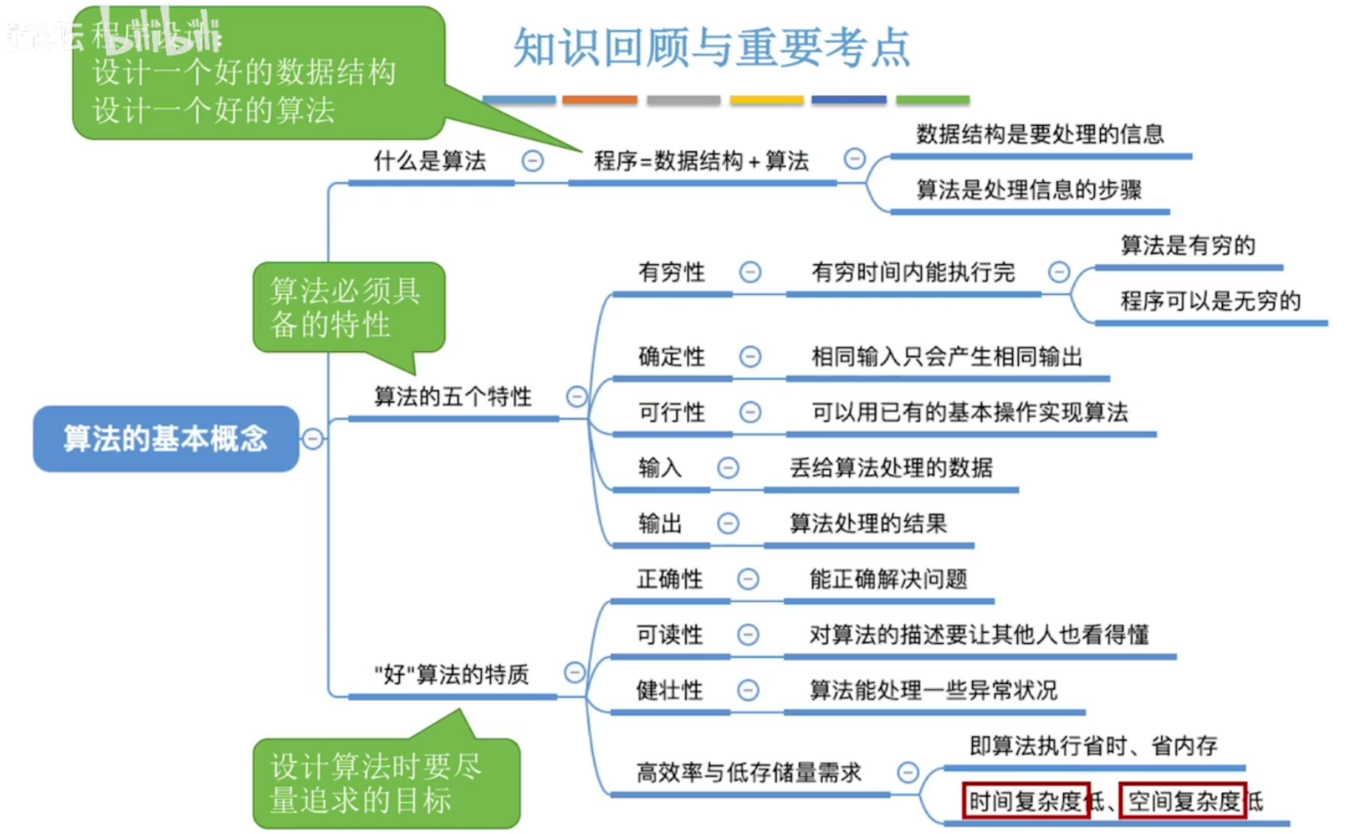
➢ O(log(n)) 复杂度的常见操作

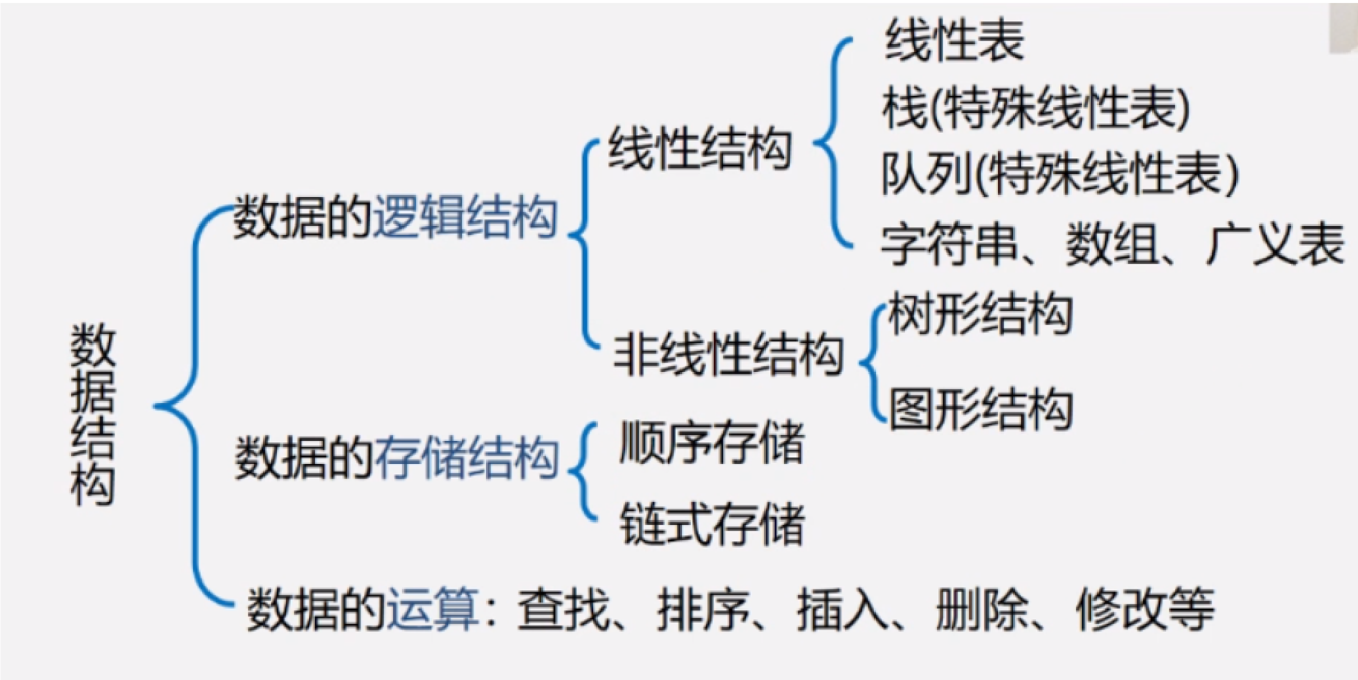
在排好序的列表或元组上进行二分查找（初始的查找区间是整个元组或列表，每次和查找区间中点比较大小，并缩小查找区间到原来的一半。类似于查英语词典）有序就会找得快！ Pyhon并不自带二分查找函数



若是函数之和，只看增长最快的。

数据结构


知识回顾与重要考点
设计一个好的数据结构
设计一个好的算法
算法的基本概念
什么是算法
程序=数据结构+算法
数据结构是要处理的信息
算法是处理信息的步骤
算法必须具备的特性
算法的五个特性
有穷性
有穷时间内能执行完
算法是有穷的
程序可以是无穷的
确定性
相同输入只会产生相同输出
可行性
可以用已有的基本操作实现算法
输入
丢给算法处理的数据
输出
算法处理的结果
"好"算法的特质
正确性
能正确解决问题
可读性
对算法的描述要让其他人也看得懂
健壮性
算法能处理一些异常状况
高效率与低存储量需求
即算法执行省时、省内存
时间复杂度低、空间复杂度低
设计算法时要尽量追求的目标


数据结构
数据的逻辑结构
线性结构
线性表
栈(特殊线性表)
队列(特殊线性表)
字符串、数组、广义表
非线性结构
树形结构
图形结构
数据的存储结构
顺序存储
链式存储
数据的运算：查找、排序、插入、删除、修改等

数据的逻辑结构

从逻辑上描述结点之间的关系，和数据的存储方式无关。

- 集合结构：结点之间没有什么关系，只是属于同一集合。如set。
- 线性结构：除了最靠前的结点，每个结点有唯一前驱结点；除了最靠后的结点，每个结点有唯一后继结点。如list。
- 树结构：有且仅有一个结点称为”根结点”，其没有前驱(父结点)；有若干个结点称为 “叶结点”，没有后继(子结点)；其它结点有唯一前驱，有1个或多个后继。如家谱
- 图结构：每个结点都可以有任意多个前驱和后继，两个结点还可以互为前驱后继。如铁路网，车站是结点。

数据的存储结构

- 顺序结构：结点在内存中连续存放，所有结点占据一片连续的内存空间。如list。
- 链接结构：结点在内存中可不连续存放，每个结点中存有指针指向其前驱结点和/或后继结点。如链表，树。
- 索引结构：将结点的关键字信息（比如学生的学号）拿出来单独存储，并且为每个关键字x配一个指针指向关键字为x的结点，这样便于按照关键字查找到相应的结点。
- 散列结构：设置散列函数，散列函数以结点的关键字为参数，算出一个结点的存储位置。

```
1. 数据的逻辑结构和存储结构无关

2. 一种逻辑结构的数据可用不同的存储结构来存储。

3. 树结构、图结构、线性结构 可用 链接结构/顺序结构存储
```

线性表

```
定义：
    由n(n≥0)个数据元素构成的有限序列，表中由且只有一个根结点和一个终端结点，除根元素外的
其它元素有且只有一个前件，除终端元素外的其它元素有且只有一个后件（如：春→夏→秋→冬）

分类：
    线性表的顺序存储结构叫做顺序表（随机存取）
```

```
    线性表的链式存储结构叫做线性链表（顺序存取）

顺序表和链表的选择：
    顺序表：
        中间插入太慢

    链表：
        访问第i个元素太慢
        顺序访问也慢(现代计算机有cache，访问连续内存域比跳着访问内存区域快很多)
        还多费空间

    结论：
        尽量选用顺序表（比如栈和队列，都没必要用链表实现）
        基本只有在找到一个位置后反复要在该位置周围进行增删，才适合用链表
```

1. 顺序表

```
1.线性表中所有元素所占的存储空间是连续的

2.线性表中数据元素在存储空间中是按逻辑顺序依次存放的

3.可以随机访问数据元素

4.做插入、删除时需移动大量元素，因此线性表不便于插入和删除元素

5.其存储空间连续，各个元素所占字节数相同，元素的存储顺序与逻辑顺序一致
```

顺序表支持的操作

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表，元素值随机	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	length()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	find(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	remove(i)	删除下标为i的元素	O(n)

2. 链表

```
1.各数据结点的存储空间可以不连续

2.各数据元素的存储顺序与逻辑顺序可以不一致，可任意

3.所占存储空间大于顺序存储结构（每节点多出至少一个指针域）

4.查找结点时要比顺序存储慢

5.插入删除元素比顺序存储灵活
```

2a. 单链表

```python
class LinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self):
        self.head = self.tail = LinkList.Node(None, None)
        self.size = 0

    def print(self):
        ptr = self.head
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next

    def insert(self, p, data):
        nd = LinkList.Node(data, None)
        if self.tail is p:
            self.tail = nd
        nd.next = p.next
        p.next = nd
        self.size += 1

    def delete(self, p):
        if self.tail is p.next:
            self.tail = p
        p.next = p.next.next
        self.size -= 1

    def popFront(self):
        if self.head is None:
            raise Exception("Popping front for Empty link list.")
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.head = self.tail = None
```

```python
    def pushFront(self, data):
        nd = LinkList.Node(data, self.head)
        self.head = nd
        self.size += 1
        if self.size == 1:
            self.tail = nd

    def pushBack(self, data):
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail, data)

    def clear(self):
        self.head = self.tail = None
        self.size = 0

    def __iter__(self):
        self.ptr = self.head
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
```

2b. 双链表

```python
class DoubleLinkList:
    class Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next

    class Iterator:
        def __init__(self, p):
            self.ptr = p

        def get(self):
            return self.ptr.data

        def set(self, data):
            self.ptr.data = data

        def __iter__(self):
            self.ptr = self.ptr.next
            if self.ptr is None:
                return None
            else:
```

```python
                return DoubleLinkList.Iterator(self.ptr)

            def prev(self):
                self.ptr = self.ptr.prev
                return DoubleLinkList.Iterator(self.ptr)

    def __init__(self):
        self.head = self.tail = DoubleLinkList.Node(None, None, None)
        self.size = 0

    def _insert(self, p, data):
        nd = DoubleLinkList.Node(data, p, p.next)
        if self.tail is p:
            self.tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self.size += 1

    def _delete(self, p):
        if self.size == 0 or p is self.head:
            return Exception("Illegal deleting.")
        else:
            p.prev.next = p.next
            if p.next:
                p.next.prev = p.prev
            if self.tail is p:
                self.tail = p.prev
            self.size -= 1

    def clear(self):
        self.tail = self.head
        self.head.next = self.head.prev = None
        self.size = 0

    def begin(self):
        return DoubleLinkList.Iterator(self.head.next)

    def end(self):
        return None

    def insert(self, i, data):
        self._insert(i.ptr, data)

    def delete(self, i):
        self._delete(i.ptr)

    def pushFront(self, data):
        self._insert(self.head, data)

    def pushBack(self, data):
        self._insert(self.tail, data)

    def popFront(self):
```

```python
            self._delete(self.head.next)

    def popBack(self):
        self._delete(self.tail)

    def __iter__(self):
        self.ptr = self.head.next
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data

    def find(self, val):
        ptr = self.head.next
        while ptr is not None:
            if ptr.data == val:
                return DoubleLinkList.Iterator(ptr)
            ptr = ptr.next
        return self.end()

    def printList(self):
        ptr = self.head.next
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next
```

2c. 循环链表

```python
class CircleLinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self):
        self.tail = None
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def pushFront(self, data):
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail
        else:
```

```python
                nd.next = self.tail.next
                self.tail.next = nd
            self.size += 1

    def pushBack(self, data):
        self.pushFront(data)
        self.tail = self.tail.next

    def popFront(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data

    def popBack(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            while nd.next != self.tail:
                nd = nd.next
            data = self.tail.data
            nd.next = self.tail.next
            self.tail = nd
            return data

    def printList(self):
        if self.is_empty():
            print('Empty!')
        else:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=',')
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print()
```

3. 链表+顺序表

```
collections.deque：
    结合链表和顺序表的特点
    是一张双向链表，每个结点是一个64个元素的顺序表
```

```
class Node:
    def __init__(self ,prev=None,next = None):
        self.data = [0 for i in range(64)]
        self.data[0],self.data[-1] = prev,next
```

二分查找函数

```
1. 前提：单调性

2. 写一个函数BinarySeach，在从小到大排序的列表a里查找元素p,如果找到，则返回元素下标，如果找不到，则返回None。

3. 复杂度O(log(n))
```

```
def binarySearch(a,p,key = lambda x : x):
    L,R = 0,len(a)-1  #查找区间的左右端点，区间含右端点
    while L <= R: #如果查找区间不为空就继续查找
        mid = L+(R-L)//2 #取查找区间正中元素的下标
        if key(p) < key(a[mid]):
            R = mid - 1  #设置新的查找区间的右端点
        elif key(a[mid]) < key(p):
            L = mid + 1  # 设置新的查找区间的左端点
        else:
            return mid
    return None
```

栈

```
1. 栈的入口和出口是同一个口，只能在栈顶进行插入和删除

2. 栈的修改原则是“先进后出”或“后进先出”

3. 栈的栈底指针bottom和栈顶指针top，从入栈到栈满再到退栈，栈低指针bottom不变，栈中元素随栈顶指针的变化而动态变化（指针存放的是地址而非数据）

4. 栈能临时保存数据，具有记忆功能

5. 栈支持子程序调用

6. 可用列表可实现栈
```

支持的操作	操作的功能	操作的实现（stack为一个列表）
top()	返回栈顶元素	stack[-1]
push(x)	将x压入栈中	stack.append(x)
pop()	弹出并返回栈顶元素	stack.pop()
isEmpty()	看栈是否为空	len(stack) == 0

要求上面操作复杂度都是O（1）

单调栈

```
#模版（找右边第一个大于ai的下标，i=1..n)
def monotonic_stack(arr,n): #n = len(arr)
    i = 0
    stack = []
    ans = [0 for _ in range(n)] #不存在用0表示
    while i < n:
        while stack and arr[i]>arr[stack[-1]]:
            ans[stack.pop()] = i+1
        stack.append(i)
        i += 1
    return ans


# 找左边第一个比自己(严格)小的元素/右边第一个比自己小的元素,结果都以标号形式输出
# 维护(严格)递增栈
def first_min(arr):
    stack = [-1]
    left_first_min = [-1 for i in range(len(arr))] #最终结果-1表示没有比自己小的
    right_first_min = [len(arr) for i in range(len(arr))] #最终结果len(arr)表示没有比自己小的
    for i in range(len(arr)):
        while len(stack) > 1 and arr[i] <= arr[stack[-1]]: #加等号，左侧严格递增，右侧非严格
            right_first_min[stack[-1]] = i
            left_first_min[stack[-1]] = stack[-2]
            stack.pop()
        stack.append(i)
    for i in range(1,len(stack)):
        left_first_min[stack[i]] = stack[i-1]
    return left_first_min,right_first_min #求总的（不比之小的）范围，相减-1即可


# 找左边第一个比自己(严格)大的元素/右边第一个比自己大的元素,结果都以标号形式输出
# 维护(严格)递减栈
def first_max(arr):
    stack = [-1]
    left_first_max = [-1 for i in range(len(arr))] #最终结果-1表示没有比自己大的
    right_first_max = [len(arr) for i in range(len(arr))] #最终结果len(arr)表示没有比自己大的
```

```
    for i in range(len(arr)):
        while len(stack) > 1 and arr[i] >= arr[stack[-1]]: #加等号，左侧严格递减，右侧非严格
            right_first_max[stack[-1]] = i
            left_first_max[stack[-1]] = stack[-2]
            stack.pop()
        stack.append(i)
    for i in range(1,len(stack)):
        left_first_max[stack[i]] = stack[i-1]
    return left_first_max,right_first_max
#求总的（不比之大的）范围，相减-1即可
```

例题1:字符串中的括号配对

```
def match(s):  #复杂度O(n)
    stack = []
    pairs = {")":"(","]":"[","}":"{" }
    for x in s:
        if x in "([{":
            stack.append(x)
        elif x in ")]}":
            if len(stack) == 0 or stack[-1] != pairs[x]:
                return False
            stack.pop()
    return len(stack) == 0

print(match(input()))
```

例题2：后序表达式求值

```
def countSuffix(s):  #计算后序表达式s的值，复杂度O(n)
    s = s.split()
    stack = []
    for x in s:
        if x in "+-*/":
            a,b = stack.pop(),stack.pop()
            stack.append(eval(str(b) + x + str(a)))
        else:
            stack.append(float(x))
    return stack[0]
```

队列

```
1、队列中队头指针front指向对头元素的前一位置，队尾指针rear指向最末元素，从入队到出队
2、队列的入口和出口非同一个口，只允许在队尾插入，而在队头删除
```

3、队列的修改原则是“先进先出”或“后进后出”（先到先服务的作业调度）
4、队列中元素随front和rear的变化而动态变化，并非固定

循环队列

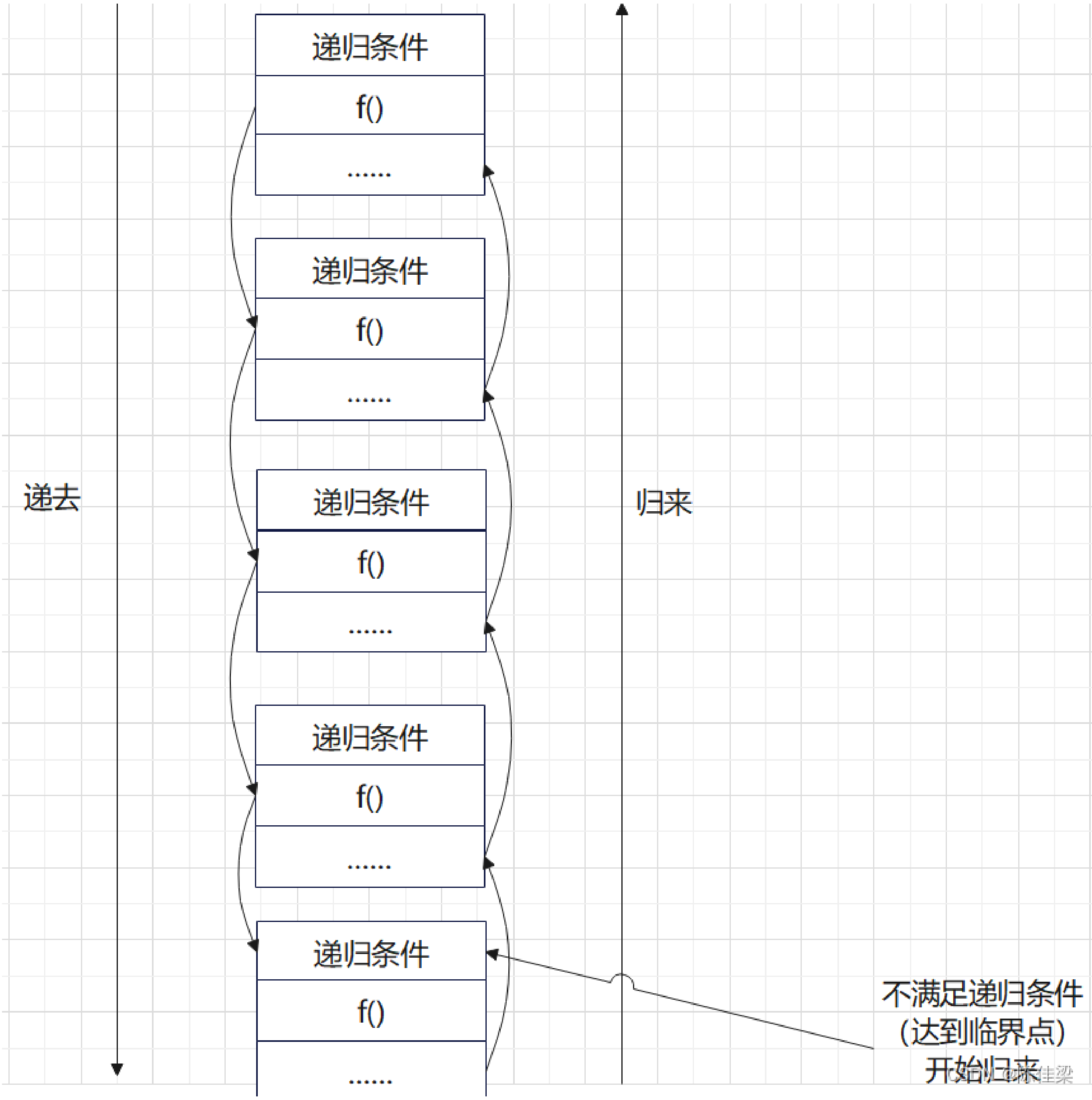
将队列存储空间的最后一个位置绕到第一个位置，形成逻辑上的环状空间，供队列循环使用

```
class Queue:
    _initC = 8 #存放队列的列表初始容量
    _expandFactor = 1.5 #扩充容量时容量增加的倍数
    def __init__(self):
        self._q = [None for i in range(Queue._initC)]
        self._size = 0 #队列元素个数
        self._capacity = Queue._initC #队列最大容量
        self._head = self._rear = 0
    def front(self): #看队头元素
        if self._size == 0:
            return None
        return self._q[self._head]
    def back(self): #看队尾元素
        if self._size == 0:
            return None
        if self._rear > 0: #rear表示下一个要加的元素位置
            return self._q[self._rear - 1]
        else:
            return self._q[-1]
    def push(self,x):
        if self._size == self._capacity:
            tmp = [None for i in range(int(self._capacity*Queue._expandFactor))]
            k = 0
            while k < self._size:
                tmp[k] = self._q[self._head]
                self._head = (self._head+1)%self._capacity
                k += 1
            self._q = tmp
            self._q[k] = x
            self._head,self._rear = 0,k+1
            self._capacity = int(self._capacity*Queue._expandFactor)
        else:
            self._q[self._rear] = x
            self._rear = (self._rear+1) % self._capacity
        self._size += 1
    def pop(self):
        if self._size == 0:
            return None
        self._size -= 1
        self._head = (self._head+1) % len(self._q)
```

动态规划

```
解题思路：
    1. 将原问题分解为子问题
    2. 确定状态
    3. 确定一些初始状态（边界状态）的值
    4. 确定状态转移方程

问题的特点：
    1. 问题具有最优子结构性质
    2. 无后效性

常用的形式：
    1. 递归型
        优点：直观，容易编写
        缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。无法使用滚动数组节省空间。总体来说，比递推型慢。
    2. 递推型
        效率高，有可能使用滚动数组节省空间
```

1. 递归



1. 替代多重循环进行枚举
2. 解决本来就是用递归形式定义的问题
3. 将问题分解为规模更小的子问题进行求解
4. 可用栈实现递归
   a. 编译器生成的代码自动维护一个栈，栈的每一层代表一个子问题
   b. 在进入下一层函数调用前，会将本层所有参数和局部变量，以及返回地址入栈中
   c. 返回地址表示了一个子问题解决后接下来应该做什么
   d. 函数调用返回时，就会退一层栈

N皇后问题

```
#如果只要找一组解

def queen(N,m):#解决N皇后问题，现在第0行到第m-1行的m个的皇后已经摆放好了
                #要摆放第m行的皇后，
    if m == N: #已经摆好了N个皇后，说明问题已经解决，输出结果即可
        for k in range(N):
            print(result[k], end=" ")
        print("")
        return True
    succeed = False
    for i in range(N):    #枚举所有位置
        if isOk(m,i):      #看可否将第m行皇后摆在第i列
            result[m] = i  #可以摆在第i列，就摆上
            if queen(N,m+1): return True   #接着去摆放第i+1行的皇后
    return succeed
```

2. 递推

```
n = int(input())
D = []
maxSum = [[-1 for j in range(i+1)] for i in range(n)]
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    for i in range(n):
        maxSum[n-1][i] = D[n-1][i]
    for i in range(n-2,-1,-1):
        for j in range(0,i+1):
            maxSum[i][j] = max(maxSum[i+1][j],maxSum[i+1][j+1]) + D[i][j]
    print(maxSum[0][0])

main()
```

空间优化后的程序

```
n = int(input())
D = []
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    maxSum = D[n-1]
    for i in range(n-2,-1,-1):
        for j in range(0,i+1):
```

```
            maxSum[j] = max(maxSum[j],maxSum[j+1]) + D[i][j]
    print(maxSum[0])

main()
```

堆

定义

```
1. 堆(二叉堆)是一个完全二叉树

2. 堆中任何结点优先级都高于或等于其两个子结点（什么叫优先级高可以自己定义）

3. 一般将堆顶元素最大的堆称为大根堆（大顶堆），堆顶元素最小的堆称为小根堆（小顶堆）
```

性质

```
1. 堆顶元素是优先级最高的(啥叫优先级高可自定义）

2. 堆中的任何一棵子树都是堆

3. 往堆中添加一个元素，并维持堆性质，复杂度O(log(n))

4. 删除堆顶元素，剩余元素依然维持堆性质，复杂度O(log(n))

5. 在无序列表中原地建堆，复杂度O(n)
```

作用

```
1. 堆用于需要经常从一个集合中取走(即删除)优先级最高元素，而且还要经常往集合中添加元素的场
合(堆可以用来实现优先队列）

2. 可以用堆进行排序，复杂度O(nlog(n))，且只需要O(1)的额外空间,称为“堆排序”。递归写法需要
o(log(n))额外空间，非递归写法需要O(1)额外空间。
```

树

概念

```
1. 每个结点可以有任意多棵不相交的子树
```

```
2. 子树有序，从左到右依次是子树1,子树2......

3. 二叉树的结点在只有一棵子树的情况下，要区分是左子树还是右子树。树的结点在只有一棵子树的情况下，都算其是第1棵子树 (所以二叉树不是树)

4. 支持广度优先遍历、前序遍历(先处理根结点，再依次处理各个子树）和后序遍历（先依次处理各个子树，再处理根结点），中序遍历无明确定义
```

性质

```
1. 结点度数最多为K的树，第i层最多Ki个结点(i从0开始）。

2. 结点度数最多为K的树，高为h时最多有(Kh+1 -1 )/(k-1) 个结点。

3. n个结点的K度完全树，高度h是logk (n)向下取整

4. n个结点的树有n-1条边
```

建树

```python
class TreeNode:
    def __init__(self,val):
        self.val=val
        self.left=None
        self.right=None
```

森林

概念

```
1. 不相交的树的集合，就是森林

2. 森林有序，有第1棵树、第2棵树、第3棵树之分

3. 森林可以表示为树的列表，也可以表示为一棵二叉树
```

森林转二叉树

```python
def woodsToBinaryTree(woods):
    #woods是个列表，每个元素都是一棵二叉树形式的树

    biTree = woods[0]
```

```
    p = biTree
    for i in range(1,len(woods)):
        p.addRight(woods[i])
        p = p.right
    return biTree
#biTree和woods共用结点,执行完后woods的元素不再是原儿子兄弟树
```

二叉树转森林

```
def binaryTreeToWoods(tree):
#tree是以二叉树形式表示的森林
    p = tree
    q = p.right
    p.right = None
    woods = [p]
    if q:
        woods += binaryTreeToWoods(q)
    return woods

#woods是兄弟-儿子树的列表,woods和tree共用结点
#执行完后tree的元素不再原儿子兄弟树
```

二叉树

定义

```
1. 二叉树是有限个元素的集合。

2. 空集合是一个二叉树，称为空二叉树。

3. 一个元素(称其为“根”或“根结点”)，加上一个被称为“左子树”的二叉树，和一个被称为“右子
树”的二叉树，就能形成一个新的二叉树。要求根、左子树和右子树三者没有公共元素。
```

概念

```
1. 二叉树的的元素称为“结点”。结点由三部分组成：数据、左子结点指针、右子结点指针。

2. 结点的度(degree)：结点的非空子树数目。也可以说是结点的子结点数目。

3. 叶结点(leaf node)：度为0的结点。

4. 分支结点：度不为0的结点。即除叶子以外的其他结点。也叫内部结点。

5. 兄弟结点(sibling)：父结点相同的两个结点，互为兄弟结点。
```

```
6. 结点的层次(level)：树根是第0层的。如果一个结点是第n层的，则其子结点就是第n+1层的。

7. 结点的深度(depth)：即结点的层次。

8. 祖先(ancestor):
    a. 父结点是子结点的祖先

    b. 若a是b的祖先，b是c的祖先，则a是c的祖先。

9. 子孙(descendant)：也叫后代。若结点a是结点b的祖先，则结点b就是结点a的后代。

10. 边：若a是b的父结点，则对子<a,b>就是a到b的边。在图上表现为连接父结点和子结点之间的线段。

11. 二叉树的高度(height)：二叉树的高度就是结点的最大层次数。只有一个结点的二叉树，高度是0。结点一共有n层，高度就是n-1。

12. 完美二叉树(perfect binary tree): 每一层结点数目都达到最大。即第i层有2i个结点。高为h的完美二叉树，有2h+1 -1个结点。

13. 满二叉树（full binary tree）: 没有1度结点的二叉树

14. 完全二叉树(complete binary tree)：除最后一层外，其余层的结点数目均达到最大。而且，最后一层结点若不满，则缺的结点定是在最右边的连续若干个。
    a. 完全二叉树中的1度结点数目为0个或1个

    b. 有n个结点的完全二叉树有⌊(n+1)/2⌋个叶结点。

    c. 有n个叶结点的完全二叉树有2n或2n-1个结点(两种都可以构建)

    d. 有n个结点的非空完全二叉树的高度为⌈log2(n+1)⌉-1。即：有n个结点的非空完全二叉树共有⌈log2(n+1)⌉层结点。
```

性质

```
1. 第i层最个多2i个结点

2. 高为h的二叉树结点总数最多2h+1-1

3. 结点数为n的树，边的数目为n-1

4. n个结点的非空二叉树至少有⌈log2(n+1)⌉层结点，即高度至少为⌈log2(n+1)⌉- 1

5. 在任意一棵二叉树中，若叶子结点的个数为n0，度为2的结点个数为n2，则n0=n2+1。

6. 非空满二叉树叶结点数目等于分支结点数目加1。

7. 非空二叉树中的空子树数目等于其结点数目加1。
```

实现方法

```python
class BinaryTree:
    def __init__(self,data,left = None,right = None):
        self.data,self.left,self.right = data,left,right
    def addLeft(self,tree): #tree是一个二叉树
        self.left = tree
    def addRight(self,tree): #tree是一个二叉树
        self.right = tree
```

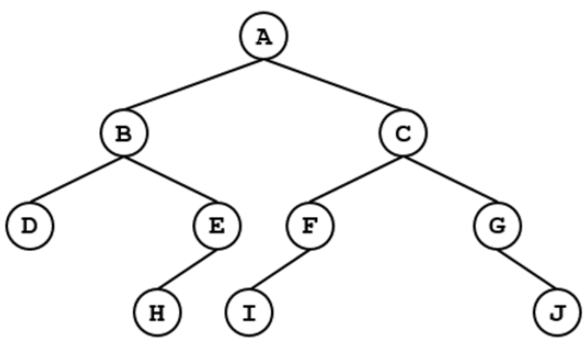
列表实现方法

```python
class BinaryTree:
    def __init__(self,data,left = [],right = []):
        self.treeList = [data,left,right]
    def addLeft(self,tree):
        self.treeList[1] = tree.treeList
    def addRight(self,tree):
        self.treeList[2] = tree.treeList
```

遍历

```
广度优先遍历：使用队列，按层遍历

深度优先遍历：编写递归函数
    前序遍历过程：1)访问根结点 2)前序遍历左子树 3)前序遍历右子树。

    中序遍历过程：1)中序遍历左子树 2)访问根结点 3)中序遍历右子树。

    后序遍历过程：1)后序遍历左子树 2)后序遍历右子树 3)访问根结点。

“访问”指的是对结点进行某种具体操作，比如输出其值、修改其值等。

遍历只需要访问每个结点一次，因此复杂度O(n)。n是总结点数目。
```

- 前序遍历访问序列：ABDEHCFIGJ
- 中序遍历访问序列：DBHEAIFCGJ
- 后续遍历访问序列：DHEBIFJGCA
- 按层遍历访问序列：ABCDEFGHIJ



```
class BinaryTree:
    def __init__(self,data,left = None,right = None):
        self.data,self.left,self.right = data,left,right
    def addLeft(self,tree): #tree是一个二叉树
        self.left = tree
    def addRight(self,tree): #tree是一个二叉树
        self.right = tree
    def preorderTraversal(self, op): #前序遍历,op是函数，表示操作
        op(self)    #访问根结点
        if self.left:   #左子树不为空
            self.left.preorderTraversal(op) #遍历左子树
        if self.right:
            self.right.preorderTraversal(op) #遍历右子树


def inorderTraversal(self, op): #中序遍历
        if self.left:
            self.left.inorderTraversal( op)
        op(self)
        if self.right:
            self.right.inorderTraversal(op)
    def postorderTraversal(self, op): #后序遍历
        if self.left:
            self.left.postorderTraversal(op)
        if self.right:
            self.right.postorderTraversal(op)
        op(self)


def bfsTraversal(self,op): #按层次遍历
        import collections
        dq = collections.deque()
        dq.append(self)
        while len(dq) > 0:
            nd = dq.popleft()
            op(nd)
            if nd.left:
                dq.append(nd.left)
            if nd.right:
                dq.append(nd.right)

#用法
tree.preorderTraversal(lambda x: print(x.data,end=""))
tree.preorderTraversal(lambda x: x.data+=100)
```

二叉搜索树

概念

```
1. 二叉树中的每个结点存储关键字（key）和值（value）两部分数据。对每个结点x，其左子树中的全部结点的key都小于x的key，且x的key小于其右子树中全部结点的key

2. 一个二叉树中的任意一棵子树都是二叉搜索树
```

性质

```
一个二叉树是二叉搜索树，当且仅当其中序遍历序列是递增序列

适合对动态查找表进行高效率查找的组织结构
```

删除结点的方式

```
1. 若x是叶子结点：直接删除，即x的父结点去掉x这个子结点

2. 若x只有左子结点: 则其左子结点取代x的地位（若x没有父亲，即为树根，则x的左儿子成为新的树根）

3. 若x只有右子结点：则其右子节点取代x的地位（若x没有父亲，即为树根，则x的右儿子成为新的树根）

4. 若x既有左子结点，又有右子节点：
    方法1：找到x中序遍历后继结点，即x右子树中最小的结点y（进入x的右子节点，不停往左走），用y的key和value覆盖x的key和value，然后递归删除y（即接下来进行y的删除）

    方法2：找到x的中序遍历前驱结点，即x左子树中最大的结点y（进入x的左子结点，不停往右走），用y的key和value覆盖x的key和value，然后递归删除y（即接下来进行y的删除）
```

哈夫曼树（最优二叉树）

```
1. 开始n个结点位于集合S

2. 从S中取走两个权值最小的结点n1和n2，构造一棵二叉树，树根为结点r, r的两个子结点是n1和n2，且Wr=Wn1+Wn2，并将r加入S

3. 重复(2.),直到S中只有一个结点，最优二叉树就构造完毕，根就是S中的唯一结点

不唯一
```

哈夫曼编码树

```python
import heapq
class HuffmanTreeNode:
```

```python
    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))
```

并查集

```
N 个不同的元素分布在若干个互不相交集合中，需要多次进行以下3个操作：

1. 合并a,b两个元素所在的集合 Merge(a,b)
```

```
2. 查询一个元素在哪个集合
3. 查询两个元素是否属于同一集合 Query(a,b)
```

发现它，抓住它

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

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

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

def solve():
    n, m = map(int, input().split())
    uf = UnionFind(2 * n)  # 初始化并查集，每个案件对应两个节点,一个是本身，另一个是其对立案件。
    for _ in range(m):
        operation, a, b = input().split()
        a, b = int(a) - 1, int(b) - 1
        if operation == "D":
            uf.union(a, b + n)  # a与b的对立案件合并
            uf.union(a + n, b)  # a的对立案件与b合并
        else:  # "A"
            if uf.find(a) == uf.find(b) or uf.find(a + n) == uf.find(b + n):
                print("In the same gang.")
            elif uf.find(a) == uf.find(b + n) or uf.find(a + n) == uf.find(b):
                print("In different gangs.")
            else:
                print("Not sure yet.")

T = int(input())
for _ in range(T):
    solve()
```

食物链

```python
class DisjointSet:
    def __init__(self, n):
        #设[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物。
        self.parent = [i for i in range(3 * n + 1)] # 每个动物有三种可能的类型，用 3 * n 来表示每种类型的并查集
        self.rank = [0] * (3 * n + 1)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv:
            return False
        if self.rank[pu] > self.rank[pv]:
            self.parent[pv] = pu
        elif self.rank[pu] < self.rank[pv]:
            self.parent[pu] = pv
        else:
            self.parent[pv] = pu
            self.rank[pu] += 1
        return True


def is_valid(n,statements):
    dsu = DisjointSet(n)

    false_count = 0
    for d, x, y in statements:
        if x>n or y>n:
            false_count += 1
            continue

        if d == 1:  # 同类
            if dsu.find(x)==dsu.find(y+n) or dsu.find(x)==dsu.find(y+2*n):  # 不是同类
                false_count += 1
            else:
                dsu.union(x, y)
                dsu.union(x + n, y + n)
                dsu.union(x + 2 * n, y + 2 * n)

        else:  # X吃Y
            if dsu.find(x) == dsu.find(y) or dsu.find(x + 2*n) == dsu.find(y):
                false_count += 1
            else: #[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物
                dsu.union(x + n, y)
                dsu.union(x, y + 2 * n)
                dsu.union(x + 2 * n, y + n)
```

```python
    return false_count


if __name__ == "__main__":
    N, K = map(int, input().split())
    statements = []
    for _ in range(K):
        D, X, Y = map(int, input().split())
        statements.append((D, X, Y))
    result = is_valid(N,statements)
    print(result)
```

图

定义

1. 图由顶点集合和边集合组成，每条边连接两个不同顶点。无向图的边记为(u,v),有向图连接顶点的边，记为<u,v>

2. 无向图中边存在，称u,v相邻，u,v互为邻点；有向图中边<u,v>存在，称v是u的邻点

概念

1. 顶点的度数：和顶点相连的边的数目。有向图中 = 入度 + 出度

2. （有向图）顶点的入度/出度：以该顶点作为终点/起点的边的数目

3. （有向图）顶点的入边/出边：以该顶点为终点/起点的边

4. 路径的长度：路径上的边的数目

5. 回路（环）：起点和终点相同的路径

6. 简单路径：除了起点和终点可能相同外，其它顶点都不相同的路径

7. 完全图：任意两个顶点都有边（无向图）/有两条相反方向的边（有向图）相连

8. 连通：如果存在从顶点u到顶点v的路径，就称u到v连通

9. 连通无向图：图中任意两个顶点u和v互相可达（连通图一般指的是无向的）

10. 强连通有向图：图中任意两个顶点u和v互相可达

11. 子图：从图中抽取部分或全部边和点构成的图

12. 连通分量（极大连通子图）：无向图的一个子图，是连通的，且再添加任何一些原图中的顶点和

边，新子图都不再连通（连通图的连通分量就是其自身，非连通的无向图有多个连通分量）

13. 强连通分量：有向图的一个子图，是强连通的，且再添加任何一些原图中的顶点和边，新子图都不再强连通

14. 带权图：边被赋予一个权值的图

15. 网络：带权无向连通图

性质

1. （无向图&有向图）图的边数等于顶点度数之和的一半

2. （无向图）n个顶点的连通图至少有n-1条边

3. （无向图）n个顶点、无回路的连通图就是一棵树，有n-1条边

4. 有一种特别的图，称为带权图（weighted graph）。在带权图中，每条边都有一个权重（weight）

在内存中存储图这种数据结构的方法有：

（1）邻接矩阵存储方法：图最直观的一种存储方法就是，邻接矩阵（Adjacency Matrix）。邻接矩阵的底层依赖一个二维数组。对于无向图来说，如果顶点 i 与顶点 j 之间有边，我们就将 A[i][j]和 A[j][i]标记为 1；对于有向图来说，如果顶点 i 到顶点 j 之间，有一条箭头从顶点 i 指向顶点 j 的边，那我们就将 A[i][j]标记为 1。同理，如果有一条箭头从顶点 j 指向顶点 i 的边，我们就将 A[j][i]标记为 1。对于带权图，数组中就存储相应的权重。

优点：首先，邻接矩阵的存储方式简单、直接，因为基于数组，所以在获取两个顶点的关系时，就非常高效。其次，用邻接矩阵存储图的另外一个好处是方便计算。这是因为，用邻接矩阵的方式存储图，可以将很多图的运算转换成矩阵之间的运算。

缺点：如果存储的是稀疏图（Sparse Matrix），即顶点很多，但每个顶点的边并不多，那邻接矩阵的存储方法就更加浪费空间了。

（2）邻接表存储方法：每个顶点对应一条链表，链表中存储的是与这个顶点相连接的其他顶点。尽管邻接表的存储方式比较节省存储空间，但链表不方便查找，所以查询效率没有邻接矩阵存储方式高。

遍历

图的深度优先遍历

```
#领接表形式
def dfsTravel(G,op): #G是邻接表
    def dfs(v):
        visited[v] = True
        op(v)
        for u in G[v]:
```

```python
            if not visited[u]:
                dfs(u)
    n = len(G)  # 顶点数目
    visited = [False for i in range(n)]
    for i in range(n):  # 顶点编号0到n-1
        if not visited[i]:
            dfs(i)


#非递归写法
def dfsTravel3(G,op):   #顶点编号从0开始，G是邻接表
    n = len(G)  # 顶点数目
    visited = [False for i in range(n)]
    for x in range(n):
        if not visited[x]:
            stack = [[x,0]] #0表示只看了0个邻点
            visited[x] = True
            while len(stack) > 0:
                nd = stack[-1] #nd[1]表示已经看过nd[1]个邻点
                v = nd[0]
                if nd[1] == 0:
                    op(v)
                if nd[1] == len(G[v]): #最后一个邻点已经看过
                    stack.pop()
                else: #对应if nd[1] == len(G[v]):
                    for i in range(nd[1],len(G[v])):
                        u = G[v][i]
                        nd[1] += 1 #看过的邻点多了一个
                        if not visited[u]:
                            stack.append([u,0])
                            visited[u] = True
                            break
```

```python
#领接矩阵形式
def dfsTravel2(G,op): #G是邻接矩阵
    def dfs(v): #从顶点v开始进行深度优先遍历
        visited[v] = True
        op(v)
        for i in range(n):
            if G[v][i] and not visited[i]:
                dfs(i)
    n = len(G)  # 顶点数目
    visited = [False for i in range(n)]
    for i in range(n):  # 顶点编号0到n-1
        if not visited[i]:
            dfs(i)
```

图的广度优先遍历

```python
#领接表形式
def bfsTravel(G,op): #G是邻接表形式的图，op是访问操作
    import collections
    n = len(G) #顶点数目
    q = collections.deque()  #队列,即Open表
    visited = [False for i in range(n)]
    for i in range(n): #顶点编号0到n-1
        if not visited[i]:
            q.append(i)
            visited[i] = True
            while len(q) > 0:
                v = q.popleft() #弹出队头顶点
                op(v) #访问顶点v
                for e in G[v]: #G[v]是点v的边的列表,e是Edge对象
                    if not visited[e.v]: #e.v是边e的另一个顶点，还有一个是v
                        q.append(e.v)
                        visited[e.v] = True
class Edge:
        def __init__(self,v,w):
            self.v,self.w = v,w  #v是顶点，w是权值
```

```python
#领接矩阵形式
def bfsTravel2(G,op):
    import collections
    n = len(G) #顶点数目
    q = collections.deque()  #队列,即Open表
    visited = [False for i in range(n)]
    for x in range(n): #顶点编号0到n-1
        if not visited[x]:
            q.append(x)
            visited[x] = True
            while len(q) > 0:
                v = q.popleft()
                op(v) #访问顶点v
                for i in range(n):
                    if G[v][i]: #G[v][i]不为0说明有边(v,i)或<v,i>
                        if not visited[i]:
                            q.append(i)
                            visited[i] = True
```

排序

```
排序的稳定性

    稳定性定义：
        排序前后两个相等的数相对位置不变，则算法稳定。
```

```
    稳定性的好处：
        从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用

    各排序算法的稳定性：
        1、堆排序、快速排序、希尔排序、直接选择排序不是稳定的排序算法；
        2、基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序是稳定的排序算法。
```

1. 冒泡排序

```
1、小的元素往前调或者把大的元素往后调；

2、比较是相邻的两个元素比较，交换也发生在这两个元素之间；

3、稳定排序算法。
```

```python
def bubbleSort(a):
    n = len(a)
    for i in range(1,n):
        done = True #改进
        for j in range(n-i):
            if a[j+1] < a[j]:
                a[j+1],a[j] = a[j],a[j+1]
                done = False
        if done:
            break
```

2. 选择排序

```
1、每个位置选择当前元素最小的；

2、在一趟选择中，如果当前元素比一个元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了；

3、举个例子，序列5 8 5 2 9，  我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了；

4、不稳定的排序算法。
```

```python
def selectionSort(a):
    n = len(a)
    for i in range(n-1):
        minPos = i #最小元素位置
        for j in range(i+1,n):
            if a[j] < a[minPos]:
```

```
                minPos = j
            if minPos != i:
                a[minPos],a[i] = a[i],a[minPos]
```

3. 插入排序

```
1、已经有序的小序列的基础上，一次插入一个元素；

2、想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置；

3、如果碰见一个和插入元素相 等的，那么插入元素把想插入的元素放在相等元素的后面；

4、相等元素的前后顺序没有改变；

5、稳定排序算法。
```

```python
def insertionSort(a):
    for i in range(1,len(a)):
        e,j = a[i],i
        while j>0 and e<a[j-1]: #从右往左，方便交换，不用开空间;保证稳定
            a[j] = a[j-1] # (1)
            j -= 1
        a[j] = e
```

4. 快速排序

```
1、两个方向，左边的i下标一直往右走，当a[i] <= a[center_index]，其中center_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j] > a[center_index]；

2、如果i和j都走不动了，i <= j, 交换a[i]和a[j],重复上面的过程，直到i>j；

3、交换a[j]和a[center_index]，完成一趟快速排序；

4、在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11,  现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱；

5、不稳定发生在中枢元素和a[j] 交换的时刻；

6、不稳定的排序算法。
```

```python
#挖坑法快排
def quickSort(a,s,e): #将a[s:e+1]排序
```

```python
    if s>=e:
        return
    i,j = s,e
    while i != j:
        while i < j and a[i] <= a[j]:
            j -= 1
        a[i],a[j] = a[j],a[i]
        while i < j and a[i] <= a[j]:
            i += 1
        a[i],a[j] = a[j],a[i]
    quickSort(a,s,i-1)
    quickSort(a,i+i,e)

#霍尔法（双指针法）快排
def quick_sort(left,right,arr):
    if left<right:
        p = partition(left,right,arr)
        quick_sort(left,p-1,arr)
        quick_sort(p+1,right,arr)
def partition(arr,left,right):
    pivot = arr[right]
    i,j = left,right-1
    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
```

5. 归并排序

```
1、把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换),然后把各个有序的短序列合并成一个有序的长序列，不断合并直到原序列全部排好序；

2、合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结 果序列的前面，这样就保证了稳定性；

3、稳定排序算法。
```

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        l,r = arr[:mid],arr[mid:]
        merge_sort(l)
        merge_sort(r)
```

```
        i = j = k = 0
        while i<len(l) and j<len(r):
            if l[i] <= r[j]: #稳定
                arr[k] = l[i]
                i += 1
            else:
                arr[k] = r[j]
                j += 1
            k += 1
        while i < len(l):
            arr[k] = l[i]
            i += 1
            k += 1
        while j < len(r):
            arr[k] = r[j]
            j += 1
            k += 1
```

6. 希尔排序(shell)

```
1、按照不同步长对元素进行插入排序；

2、当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；

3、当元素基本有序了，步长很小， 插入排序对于有序的序列效率很高；

4、所以，希尔排序的时间复杂度会比o(n^2)好一些

由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元 素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱；

5、不稳定的排序算法。
```

```
def shell_sort(arr):
    n = len(arr)
    gap = n//2
    while gap > 0:
        j = gap
    while j<n:
        i = j-gap
        while i>=0 and arr[i]>arr[i+gap]:
            arr[i],arr[i+gap] = arr[i+gap],arr[i]
            i -= gap
        j += 1
    gap //= 2
```

7. 出基数排序 数：

```
1、按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位；

2、有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优 先级排序，最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前；

3、用于整数；

4、需要较多的存储空间；

5、基于分别排序，分别收集；

6、稳定排序算法。
```

```python
def radixSort(s,m,d,key): #d：元素由多少个原子组成
    for k in range(d):
        buckets = [[] for j in range(m)]
        for x in s:
            buckets[key(x,k)].append(x)
        i = 0
        for bkt in buckets:
            for e in bkt:
                s[i] = e
                i += 1
def getKey(x,i):
    tmp = None
    for k in range(i+1):
        tmp = x%10
        x //= 10
    return tmp
```

8. 堆排序

```
1、是选择排序的一种；

2、堆的结构是节点i的孩子为2*i和2*i+1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点，是完全二叉树；

3、在一个长为n 的序列，堆排序的过程是从第n/2开始和其子节点共3个值选择最大(大顶堆)或者最小(小顶堆),这3个元素之间的选择当然不会破坏稳定性。但当为n /2-1, n/2-2, …1这些个父节点选择元素时，就会破坏稳定性。有可能第n/2个父节点交换把后面一个元素交换过去了，而第n/2-1个父节点把后面一个相同的元素没 有交换，那么这2个相同的元素之间的稳定性就被破坏了；

4、不稳定的排序算法。
```

```

```

```python
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

9. 桶排序

```
1、如果待排序元素只有m种不同取值，且m很小，则可以采用桶排序；

2、设立m个桶，分别对应m种取值。桶和桶可以比大小，桶的大小就是其对应取值的大小，把元素依次放入其对应的桶，然后再按先小桶后大桶的顺序，将元素都收集起来，即完成排序。

3、稳定排序算法
```

```python
def bucketSort(s,m,key = lambda x:x):
    buckets = [[] for i in range(m)]
    for x in s:
        buckets[key(x)].append(x)
    i = 0
    for bkt in buckets:
        for e in bkt: #先进先出，保证稳定
            s[i] = e
            i += 1
```

其他算法

Shunting Yard算法

中缀转后缀

```python
operators=['+','-','*','/']
cals=['(',')']
# 预处理数据的部分已省略。
def pre_to_post(lst):
    s_op,s_out=[],[]
    while lst:
        tmp=lst.pop(0)
        if tmp not in operators and tmp not in cals:
            s_out.append(tmp)
            continue

        if tmp=="(":
            s_op.append(tmp)
            continue

        if tmp==")":
            while (a:=s_op.pop())!="(":
                s_out.append(a)

        if tmp in operators:
            if not s_op:
                s_op.append(tmp)
                continue
            if is_prior(tmp,s_op[-1]) or s_op[-1]=="(":
                s_op.append(tmp)
                continue
            while (not (is_prior(tmp,s_op[-1]) or s_op[-1]=="(")
                or not s_op):
                s_out.append(s_op.pop())
            s_op.append(tmp)
            continue

    while len(s_op)!=0:
        tmp=s_op.pop()
        if tmp in operators:
            s_out.append(tmp)

    return " ".join(s_out)

def is_prior(A,B):
    if (A=="*" or A=="/") and (B=="+" or B=="-"):
        return True
    return False

def input_to_lst(x):
    tmp=list(x)

for i in range(int(input())):
    print(pre_to_post(expProcessor(input())))
```

Prim算法

步骤：

1. 起点入堆。
2. 堆顶元素出堆（排序依据是到该元素的开销），如已访问过，continue；否则标记为visited。
3. 访问该节点相邻节点，（访问开销（排序依据），相邻节点）入堆。
4. 相邻节点前驱设置为当前节点（如需）。
5. 当前节点入树

全部精要在于：每次走出下一步的开销都是当前最小的。

Agri-net

题目：用邻接矩阵给出图，求最小生成树路径权值和。

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
        # 注意这一步continue很关键，因为一个节点会同时很多存在于pq中（这是由出队标记决定的）
        # 如果不设计这一步continue，则会重复加路径长。
```

```python
from heapq import heappop, heappush
def prim(matrix):
    ans=0
    pq,visited=[(0,0)],[False for _ in range(N)]
    while pq:
        c,cur=heappop(pq)
        if visited[cur]:continue
        visited[cur]=True
        ans+=c
        for i in range(N):
            if not visited[i] and matrix[cur][i]!=0:
                heappush(pq,(matrix[cur][i],i))
    return ans

while True:
    try:
        N=int(input())
        matrix=[list(map(int,input().split())) for _ in range(N)]
        print(prim(matrix))
    except:break
```

Kruskal算法（Prim优先）

Agri-net

```python
class DisJointSet:
    def __init__(self,num_vertices):
        self.parent=list(range(num_vertices))
        self.rank=[0 for _ in range(num_vertices)]

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

    def union(self,x,y):
        root_x=self.find(x)
        root_y=self.find(y)
        if root_x!=root_y:
            if self.rank[root_x]<self.rank[root_y]:
                self.parent[root_x]=root_y
            elif self.rank[root_x]>self.rank[root_y]:
                self.parent[root_y]=root_x
            else:
                self.parent[root_x]=root_y
                self.rank[root_y]+=1

# graph是邻接表
def kruskal(graph:list):
    res,edges,dsj=[],[],DisJointSet(len(graph))
    for i in range(len(graph)):
        for j in range(i+1,len(graph)):
            if graph[i][j]!=0:
                edges.append((i,j,graph[i][j]))

    for i in sorted(edges,key=lambda x:x[2]):
        u,v,weight=i
        if dsj.find(u)!=dsj.find(v):
            dsj.union(u,v)
            res.append((u,v,weight))
    return res

while True:
    try:
        n=int(input())
        graph=[list(map(int,input().split())) for _ in range(n)]
        res=kruskal(graph)
        print(sum(i[2] for i in res))
    except EOFError:break
```

Kahn算法

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。

2. 遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。

3. 不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。

4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

拓扑排序

题目：给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。

题解中graph是邻接表，形如graph[1]=[2,3,4]，由于本题要求顺序，因此不用队列而用优先队列。

```python
from collections import defaultdict
from heapq import heappush,heappop
def Kahn(graph):
    q,ans=[],[]
    in_degree=defaultdict(int)
    for lst in graph.values():
        for vert in lst:
            in_degree[vert]+=1

    for vert in graph.keys():
        if vert not in in_degree or in_degree[vert]==0:
            heappush(q,vert)

    while q:
        vertex=heappop(q)
        ans.append('v'+str(vertex))
        for neighbor in graph[vertex]:
            in_degree[neighbor]-=1
            if in_degree[neighbor]==0:
                heappush(q,neighbor)
    return ans

v,a=map(int,input().split())
graph={}
for _ in range(a):
    f,t=map(int,input().split())
    if f not in graph:graph[f]=[]
    if t not in graph:graph[t]=[]
    graph[f].append(t)

for i in range(1,v+1):
    if i not in graph:graph[i]=[]

res=Kahn(graph)
print(*res)
```

Dijkstra算法

道路（更推荐第二种剪枝写法）

N个以 1 ... N 标号的城市通过单向的道路相连。每条道路包含两个参数：道路的长度和需要为该路付的通行费（以金币的数目来表示）。Bob从1到N。他希望能够尽可能快的到那，但是他囊中羞涩。我们希望能够帮助Bob找到从1到N最短的路径，前提是他能够付的起通行费。输出结果应该只包括一行，即从城市1到城市N所需要的最小的路径长度（花费不能超过K个金币）。如果这样的路径不存在，结果应该输出-1。

S：起点；D：终点；L：道路长；T：通行费。

```python
from heapq import heappop,heappush
from collections import defaultdict
K,N,R=int(input()),int(input()),int(input())
graph=defaultdict(list)
for i in range(R):
    S,D,L,T=map(int,input().split())
    graph[S].append((D,L,T))
def Dijkstra(graph):
    global K,N,R
    q,ans=[],[]
    heappush(q,(0,0,1,0))
    while q:
        l,cost,cur,step=heappop(q)
        if cur==N:return l
        for next,nl,nc in graph[cur]:
            # 剪枝：如果步数不少于N：意味着一定走了回头路，减掉。
            if cost+nc<=K and step+1<N:
                heappush(q,(l+nl,cost+nc,next,step+1))
    return -1
print(Dijkstra(graph))
```

```python
from heapq import heappop,heappush
from collections import defaultdict
K,N,R=int(input()),int(input()),int(input())
graph=defaultdict(list)
for i in range(R):
    S,D,L,T=map(int,input().split())
    graph[S].append((D,L,T))

def Dijkstra(graph):
    global K,N,R
    q,ans=[],[]
    min_cost={i:float('inf') for i in range(1,N+1)}
    heappush(q,(0,0,1))
    while q:
        l,cost,cur=heappop(q)
        min_cost[cur]=min(min_cost[cur],cost)
        if cur==N:return l
        for next,nl,nc in graph[cur]:
```

```
            # 剪枝1：只有花费小于等于K才能入堆。
            # 剪枝2：只有到达下一个节点的花费比上次更小时才能入堆（否则路程长花费大，无意义）。
            if cost+nc<=K and nc+cost<min_cost[next]:
                heappush(q,(l+nl,cost+nc,next))
    return -1
print(Dijkstra(graph))
```

Kosaraju算法

```python
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
```

```python
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```