CH1 绪论

一、算法与数据结构

1. 算法的定义

算法是对计算过程的描述，是为了解决某个问题而设计的有限长操作序列

1. 有穷性：一个算法必须可以用有穷条指令描述，每次操作都必须在有穷时间内完成。算法终止后必须给出处理问题的解或宣告问题无解。
2. 确定性：一个算法，对于相同的输入，无论运行多少次，总是得到相同的输出。
3. 可行性：算法中的指令含义明确无歧义，可以被机械化地自动执行
4. 输入/输出：算法可以不需要输入，但是没有输出的算法是没有意义的

2.程序或算法的时间复杂度：一个程序或算法的时间效率

时间复杂度是用算法运行过程中，某种时间固定的操作（找一句程序语言就可以）需要被执行的次数和n（问题规模）的关系来度量的。

计算复杂度的时候，只统计执行次数最多的那种固定操作（称为基本操作）的次数。我们只关心随n增长最快的那个函数

eg 有序的二分查找时间复杂度为O(logn),用in判断是否在集合/字典中，以及用len求列表、元组、集合、字典元素个数的时间复杂度均为O(1)

算法的时间复杂度，取决于问题的规模和待处理数据的初态

3.数据结构的定义

三要素：逻辑结构，存储结构，数据的运算

数据结构就是数据的组织和存储形式。描述一个数据结构，需要指出其逻辑结构、存储结构和可进行的操作。数据结构描述的就是数据单位（元素/结点）之间的关系

数据项是数据的最小单位，数据元素是数据的基本单位，数据结构是带有结构的各数据元素的集合

数据元素的逻辑结构与数据元素本身的形式、内容、相对位置和个数无关

抽象数据结构类型ADT = 逻辑结构 + 数据运算，不关心存储结构

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（广义分类：线性结构&非线性结构）

集合结构：结点之间没有什么关系，只是属于同一集合（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（线性表【链表，栈，队列】，字符串）

树结构：有且仅有一个结点称为根结点，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）；其他结点有唯一前驱（父结点），有1个或多个后继

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

数据的存储结构：数据在物理存储器上存储的方式，大部分情况下指的是数据在内存中存储的方式

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（顺序表）

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前驱节点和/或后继结点（链表【链式结构】，树）（注：链表所需空间与线性表长度成正比）

索引结构：将结点的关键字信息（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

数据的逻辑结构和存储结构无关，一种逻辑结构可以用不同的存储结构来存储

树结构、图结构、线性结构可以用链接结构存储，也可以用顺序结构存储

二、python知识巩固

（一）python的指针本质

1.python中的所有变量都是指针（赋值号=左边的东西）

指针代表内存单元的地址，所有变量都是指针（箭头），指向内存的某处。对变量进行赋值，本质就是让该变量指向某个地方

2.is和运算符==的区别

用一个变量对另一个变量赋值，意味着让两个变量指向同一个地方（a is b判断为True）

a is b：判断a和b是否指向同一个地方

a == b：判断a和b分别指向的地方，放的东西是否相同（但是a和b可以不指向相同的地方）

3.列表元素的指针本质

列表的元素（各索引）也可以赋值，因此也是指针（元组的元素虽然不可赋值，但也是指针）

乘号（*）表示生成列表的各个元素都指向了同一个位置

（二）python中的函数

1.python函数参数传递方式都是传值

形参只是实际参数的一个拷贝，函数参数也是指针。形参和实参指向同一个地方。

对形参赋值会让其指向别处，从而不会影响实参

然而，如果函数执行过程中，改变了形参所指向的地方的内容，则实参所指向的地方内容也会被改变

2. global变量和nonlocal变量

global关键字用于声明变量是在所有函数外部定义的变量。nonlocal函数用于声明变量实在函数的外层函数定义的变量

3.高阶函数

函数可以赋值给变量，也可以作为函数的参数和返回值。如果一个函数能接收函数作为参数，或者其返回值是函数，这样的函数就称为高阶函数

```
def combine(f,g,x):
    return f(g(x))
def combineFunctions(f,g):
    return lambda x:f(g(x))
# combineFunctions(square,inc)(x)
```

三、python中的类

1.类和对象的概念

类：概括了一种事物的特点，包括属性和方法（成员函数）

对象：通过类定义的变量，一个对象就是一个类的实例。python中所有的变量，小数、字符串、组合数据类型的常量，以及函数，都是对象

默认情况下，自定义类的对象可以作为集合元素或字典的键（集合和字典都是哈希表，可哈希的类的对象才可以作为集合的元素或字典的键【一个类有__hash__方法】），但被作为集合元素或字典键的是对象的id（内存地址，自定义类默认hash方法根据对象id算哈希值，哈希值是个整数，x.--hash--()），因此意义不大

2.对象的比较

默认情况下，自定义类的对象a和b只能用==比较，且a==b等价于a is b

默认情况下，自定义类的lt，gt，le，ge方法都被设置成了None，通过重写eq和这些成员函数，可以让==含义变化，以及对象可以用<,>,<=,>=进行比较

注：一些其他的魔术方法(不需要显示调用就可以执行的方法)：

len(x):x.len__();str(x):x.__str(x);y in x:x.__contains(y)(判断y在不在x里面)；x.__getitem(index):x[index] 根据下标访问x的元素；x.__setitem(index,item):x[index] = item

3.继承和派生

python所有类，包括自定义类，均派生自object类，因而自动继承object类方法

4.静态属性和物理方法

静态属性被类中所有对象共享，不是每个对象各自一份。当我们改变静态属性的值时，所有对象的静态属性都会发生变化

```
class employee:
    total = 0
    def __init__(self,name):
        self.name = name
    def pay(self,salary):
        employee.total += salary
a = employee("a")
b = employee("b")
a.pay(3)
b.pay(4)
print(a.total) #7
```

5.迭代器

如果一个类实现了__next__()方法和__iter__()方法，并且iter方法返回对象自身，则该类的对象就成为“迭代器”。迭代器对象通常用于存储一些元素，一般__next__()方法会用来返回对象中的下一个元素

```
class MyRange:
    def __init__(self,n):
        self.idx = 0
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.idx < self.n:
            val = self.idx
            self.idx += 1
            return val
        else:
            raise StopIteration() #引发异常
# for循环本质通过while循环进行构建:
it = iter(a)
while True:
    try:
        i = next(it)
        #语句组
    except StopIteration:
        break
```

CH2 线性表

线性表是一个元素构成的序列，该序列有唯一的头元素和尾元素。除了头元素外，每个元素都有唯一的前驱元素；除了尾元素外，每个元素都有唯一的后继元素

线性表分为顺序表和链表两种。顺序表中间插入太慢，链表访问第i个元素太慢。尽量使用顺序表，基本只有在找到一个位置后要反复在该位置进行增删，才适合使用链表

线性表是n个具有相同特性的数据元素的有限序列。线性表中的元素属于相同的数据类型，即每个元素所占的空间必须相同（不仅数据元素所包含的数据项的个数要相同，而且对应数据项的类型要一致）

一、顺序表：python的列表，以及其他语言中的数组

元素在内存中连续存放；每个元素都有唯一序号（下标），且根据序号访问（包括读取和修改）元素的时间复杂度是O（1）（随机访问：访问时间不变，实现访问时间不变的存储方式）

顺序表append的O（1）复杂度实现：在内存不够时，总是分配多于实际元素个数的空间，当元素个数等于容量时，append重新分配空间，且要拷贝原有元素到新空间，此时复杂度为O（n）

二、链表概述

1.基本性质

元素在内存中并非连续存放，元素之间通过指针链接起来；每个结点除了元素，还有next指针，指向后继；不支持随机访问。

访问/查找第i个元素，复杂度为O(n)；**已经找到**插入或删除位置的情况下，插入和删除元素的复杂度为O（1），且不需要复制或移动结点

2.形式1：单链表

```
class LinkList:
    class Node: #表结点
        def __init__(self,data,next=None):
            self.data,self.next = data,next
    def __init__(self):
        self.head = self.tail = None
        self.size = 0
    def printList(self): #打印全部结点
        ptr = self.head
        while ptr is not None:
            print(ptr.data,end = ",")
            ptr = ptr.next
    def insert(self,p,data): #在结点p后面插入元素
        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): #删除p后面的结点
        if self.tail is p.next:
            self.tail = p
        p.next = p.next.next
        self.size -= 1
    def popFront(self): #删除前端元素
```

```
        if self.head is None:
            return
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.head = self.tail = None
    def pushBack(self,data): #在尾部添加元素data
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail,data)
    def pushFront(self,data): #在链表前端插入一个元素data
        nd = LinkList.Node(data,self.head)
        self.head = nd
        self.size += 1
        if self.tail is None:
            self.tail = nd
```

3.形式2：循环链表

```
class LinkList:
    class Node:
        def __init__(self,data,next=None):
            self.data,self.next = data,next
    def __init__(self):
        self.tail,self.size = None,0
    def pushFront(self,data):
        nd = self.Node(data)
        if self.tail is None:
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
        self.size += 1
    def pushBack(self,data):
        nd = self.Node(data)
        if self.tail is None:
            self.tail = nd
            nd.next = self.tail
        else:
            self.tail.next = nd
            self.tail = nd
    def popFront(self):
        if self.size == 0:
            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
```

4.形式3：双向链表

每个结点有next指针指向后继，有prev指针指向前驱

```
class DoubleLinkList:
    class Node:
        def __init__(self,data,prev=None,next=None):
            self.data,self.prev,self.next = data,prev,next
    def __init__(self):
        self.head = DoubleLinkList.Node(None,None,None)
        self.tail = DoubleLinkList.Node(None,None,None)
        self.size = 0
    def insert(self,p,data):
        nd = DoubleLinkList.Node(data)
        if p is self.tail:
            self.tail = nd
            nd.prev = p
            p.next = nd
        else:
            nd.prev,nd.next = p,p.next
            p.next.prev = nd
            p.next = nd
        self.size += 1
    def delete(self,p): #删除节点p
        if p is self.head:
            if p is self.tail:
                self.head = self.tail = None
            else:
                self.head = p.next
        else:
            if p is self.tail:
                p.prev = self.tail
                p.prev.next = None
            else:
                p.prev.next = p.next
        self.size -= 1
```

三、其他问题

1.循环队列：队列的一种实现方法

如果不想浪费空间开足够大的列表，而是想根据实际情况分配空间，则可以使用列表+头尾循环法实现队列

预先开始一个capacity个空元素的列表queue，head = tail = 0

列表没有装满的情况下：把tail的位置放置新添元素，tail进一/循环回来。pop直接将head进一/循环回来就可以

判断队列为空/满：维护size，size == 0为空，size == capacity为满

若一个push操作后导致列表满：新建大容量列表，拷贝至新列表，重设新列表的head和tail

注：要么维护size，要么浪费一个存储空间，tail所指向的位置为空，在(tail+1)%capacity = head情况下表示满

```
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)
```

2.栈和递归的关系

递归的每一层都由一个栈来维护。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（返回地址表示了一个子问题解决后接下来应该做什么）

函数调用返回时，就会退一层栈

CH3 二叉树

一、基本定义

1.二叉树的相关概念

空集合是一个二叉树，称为空二叉树。

一个元素（根，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

结点的度：结点的非空子树数目，也可以说是结点的子结点数目

叶结点：度为0的结点

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

兄弟结点：父结点相同的两个结点，互为兄弟结点

结点的层次&深度：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（或者算从根结点到叶结点经过路径的数量）

二叉树的高度：结点的最大层次数（只有一个结点的二叉树高度是0）(求树高空间复杂度O(h))

2. 二叉树的分类

完美二叉树：每一层结点数目都达到最大。高为h的完美二叉树有2^(h+1)-1个结点

满二叉树：没有1度结点的二叉树（内部结点的度都为2）

完全二叉树：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，则缺的结点一定是在最右边的连续若干个

3.二叉树的性质

1. 第i层最多2^i个结点
2. 高为h的二叉树结点总数最多2^(h+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层数/高度至少有log_2(n+1)（取上界）-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. 非空二叉树中的空子树数目等于其结点数目加1
7. 二叉树在三中不同的周游序列中，叶子结点的相对次序不会发生改变
8. 总结点数 = 总(入)度数 + 1

4.完全二叉树的性质

1. 完全二叉树中的1度结点数目为0个或1个
2. **有n个结点的完全二叉树有(n+1)/2（取下界）个叶结点**

3. 有n个叶结点的完全二叉树有2n或2n-1个结点
4. 有n个结点的非空完全二叉树的高度/深度/最大层级为log_2(n+1)（向上取整）-1
5. 完全二叉树可以用列表存放结点。下标为i的结点的左右子结点下标是2i+1，2i+2。下标为i的元素，父结点的下标为（i-1）//2

5.k叉树的性质

1. 结点度数最多为k的树，第i层最多有k^i个结点（从0开始）
2. 结点度数最多为k的树，高为h时最多有（k^(h+1)-1）/(k-1)个结点
3. n个结点的K度完全树，高度h是log_k(n(k-1)+1)（向上取整）-1
4. n个结点的树有n-1条边

二、堆

1.堆的基本定义

堆（二叉堆）是完全二叉树，堆中任何结点优先级都高于或等于其两个子结点（eg：大根/顶堆，小根/顶堆）

用列表存放堆，堆顶下标元素为0的情况下，左右子结点下标分别为2i+1，2i+2

堆的作用：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.堆的性质

1. 堆顶元素是优先级最高的
2. 堆中的任何一棵子树都是堆
3. 往堆中添加一个元素，并维持堆的性质，复杂度O(log(n))
4. 删除堆顶元素，调整剩余元素使其依然维持堆性质，复杂度O(logn)
5. 在无序列表中原地建堆，**复杂度O(n)**（在从0-n-1中，建堆从第n-1-int((n+1)/2))开始
6. 堆排序：复杂度O(nlogn)，且非递归写法只需要O(1)额外空间，递归写法需要O(logn)额外空间

3.heapq不常用算法备注

```
import heapq
heapq.heapreplace(s,item) #取出并返回堆顶元素，并将元素item入堆s
heapq.heappushpop(s,item) #将item入堆，然后弹出最小元素
heapq.nsmallest(n,s,key) #返回序列s中最小的n个元素构成的列表，key是关键字函数
heapq.nlargest(n,s,key) #返回序列s中最大的n个元素构成的列表，key是关键字函数
```

三、森林

1.森林的概念

不相交的树的集合，就是森林。森林有序，有第1棵树、第2棵树、第3棵树之分

2.森林的遍历

森林广度优先遍历：将树从左到右按顺序排好，按照层级、从左到右进行遍历

森林前序遍历：先进行第一棵树的前序遍历，再进行第二棵树的前序遍历，以此类推

森林后序遍历：先进行第一棵树的后序遍历，再进行第二棵树的后序遍历，以此类推

3.森林的二叉树表示法（儿子兄弟树）

森林中第一棵树的根，就是二叉树的根s1，s1及其左子树，是森林的第1棵树的二叉树表示形式

s1的右子节s2，以及s2的左子树，是森林的第二棵树的二叉树表示形式

s2的右子节s3，以及s3的左子树，是森林的第三棵树的二叉树表示形式。以此类推

森林的前序遍历序列，和其儿子兄弟树的前序遍历序列一致；森林的后序遍历序列，和其儿子兄弟树的中序遍历序列一致

性质：若森林中有n个非终端结点，则二叉树中右指针域为空的结点有n+1个（最后一棵树根结点+每个非终端的最后一个儿子都为右空）

4.并查集

路径压缩后，find()函数的时间复杂度为O(logn)

四、二叉排序树/二叉搜索树

1.二叉排序树的概念

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

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

2.二叉排序树删除结点的方式

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：有两种做法

…………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）

注：要支持重复关键字，可以改成val部分是可以包含多个值的列表

3.二叉排序树复杂度

建树复杂度：O(nlogn)

不能保证查询、插入、查找、删除的log(n)复杂度（如果树退化成一根杆，复杂度就是O(n)）

AVL树加入平衡因子后，可以实现log(n)的查询、插入、删除复杂度

CH4 图

一、图的基本定义与相关概念

1.图的定义

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>

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

2.图的相关概念

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**

3.图的性质

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.图遍历的时间复杂度

基础dfs：

邻接表形式-O(E+V)(每个顶点看过一遍，每条边看过一遍(无向图两遍));

邻接矩阵形式-O(V^2)(每个顶点做dfs，每次查看和所有顶点的关系)

基础bfs（要注意由于图可能不连通/强连通，队列为空时，可能还有顶点未曾访问过）：

邻接表形式-每条边看过一遍（无向图两遍），时间复杂度O(E+V)

邻接矩阵形式-每个顶点出队一次，查看其和所有顶点的关系，时间复杂度O(V^2)

二、其他图算法相关概念与注意事项

1.拓扑排序(针对有向图，存在拓扑排序说明为有向无环图)

Kahn算法时间复杂度：每个顶点都出入队列一次，每条边都要看一次，复杂度O(E+V)

2.AOE网络（Activity on Edge network）带权有向无环图

顶点表示事件，事件不需要花时间；有向边表示活动，边权值表示活动需要花的时间

当前仅当一个顶点的入边代表的活动都已经完成，该顶点表示的事件会发生，即其出边代表的活动就都可以开始。先后顺序无关的活动可以同时进行

每个活动的最早开始时刻 = 起点事件的最早发生时刻；最晚开始时刻 = 终点事件的最晚发生时刻

关键活动：最早开始时刻和最晚开始时刻一致的活动

关键路径：AOE网络上权值之和最大的路径。关键路径一定由关键活动构成，T就是关键路径权值之和

求最早开始时间的方法：初始化入度为0的最早开始时间为0。按照拓扑排序的顺序递推每个事件的最早开始时间

earliestTime[j] = max(earliestTime[j],earliestTime[i] + Wij)

求最晚开始时间的方法：求出全部活动都完成的最早时刻T（max（earliestTime）），初始化每个出度为0的顶点为T。按照拓扑序列的逆序递推每个时间的最晚开始时间：

latestTime[i] = min(latestTime[i],latestTime[j]-Wij)

3.最小生成树算法

（1）生成树的概念

在一个**无向连通图**G中，取它的全部顶点和一部分边构成一个子图G’。如果子图的边集将图中的所有顶点连通，又不构成回路，则称子图G‘是原图G的一棵生成树

一个含有n个点的生成树，必含有n-1条边。无向图的极小连通子图（去掉一条边就不连通的子图）就是生成树

最小生成树的性质：最小生成树可能不止一棵。但一个图的两棵最小生成树，边的权值序列排序后，结果相同

当连通无向图G中所有边的权值均不同时，G具有唯一的最小生成树

（2）dfs求生成树

深度优先遍历一个图，记录每个顶点的父亲结点。遍历结束后，将每个顶点及其父亲节点之间的边输出，即得到生成树

（3）无向连通带权图中，最小生成树的概念

具有最小权值的生成树称为最小生成树

（4）prim算法求最小生成树的时间复杂度

时间复杂度：

邻接矩阵-加入Vi后，更新所有与Vi相连且不再T中的点Vj的Dist，然后再选出下一个入树的点。时间复杂度O(V^2)

邻接表-采用堆来选取。考察了所有的边，在考察一条边时，可能执行入堆操作，故总时间复杂度O(ElogV)

（5）Kruskal算法求最小生成树：选最小的边，并查集检验

时间复杂度：O(ElogE)（主要耗费时间在对边的排序上），适用于稀疏图

```
class DisjointSet:
    def __init__(self, num_vertices):#表示点的数量
        #这里的点标号为0至n-1
        self.parent = list(range(num_vertices))
        self.rank = [0] * 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
def kruskal(num_vertices,edges):
    # 按照权重排序
    edges.sort(key=lambda x: x[2])
    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)
    # 构建最小生成树的边集
    minimum_spanning_tree = []
```

```
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
    return minimum_spanning_tree
```

4.最短路径问题

（1）Dijkstra算法

思想：解决**无负权边的带权有向图或无向图**的单源最短路问题，内嵌贪心思想

时间复杂度：堆优化*后O(ElogV)*（邻接表O(V^2+E)）

(2) Floyd算法

用于求对**每一对**顶点之间的最短路径。有向图，无向图均可。无向图不可以带负权边，有向图可以负权边，但是不能有负权回路

思想：若vi-vk和vk-vj是中间顶点序号不大于k的最短路径，则将其和已经得到的从vi到vj且中间顶点的序号不大于k-1的最短路径相比较，其长度较短者便是从vi到vj的中间顶点的序号不大于k的最短路径

在经过n次比较后，最后求得的就是vi到vj的最短路径。此方法可以同时求得各对顶点间的最短路径

复杂度：O(N^3)

```
def floyd(G):
    n = len(G)
    INF = 10**9
    prev = [[None for i in range(n)] for j in range(n)]
    dist = [[INF for i in range(n)] for j in range(n)]
    #dist：i到j的最短路；prev：i到j最短路上，j的前驱
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            else:
                if G[i][j] != INF:
                    dist[i][j] = G[i][j]
                    prev[i][j] = i
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    prev[i][j] = prev[k][j]
    return dist,prev
```

5.求解强连通分量算法

1. 2DFS(Kosaraju算法)

```python
#注意：graph为字典
#思想：先找到拓扑排序序列，然后对图做转置，找scc
#注意：visited的设法要注意（目前是序号0至n-1）
#注意：visited中包含全部节点，因此graph也要对应包含全部
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)):#目前采用0至n-1标号
        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)):#目前采用0至n-1标号
        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)#目前采用0至n-1标号
    sccs = []
    while stack:
        #栈顶是最后完成的，按照完成时间的逆序（即拓扑排序的正序）
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

2. Tarjan算法

```python
def tarjan(graph):
    def dfs(node):
        nonlocal index, stack, indices, low_link, on_stack, sccs
        index += 1
        indices[node] = index#分配搜索次序和最低链接值入栈
        low_link[node] = index
        stack.append(node)
        on_stack[node] = True
        for neighbor in graph[node]:
```

```
            if indices[neighbor] == 0:  # Neighbor not visited yet
                dfs(neighbor)
                #回溯过程中，更新当前顶点的最低链接值
                low_link[node] = min(low_link[node], low_link[neighbor])
            elif on_stack[neighbor]:  # Neighbor is in the current SCC
                low_link[node] = min(low_link[node], indices[neighbor])
        if indices[node] == low_link[node]:
            scc = []#如果最低链接值=搜索次序：弹出从当前顶点开始的栈中顶点，构成scc
            while True:
                top = stack.pop()
                on_stack[top] = False
                scc.append(top)
                if top == node:
                    break
            sccs.append(scc)
    index = 0
    stack = []
    indices = [0] * len(graph)
    low_link = [0] * len(graph)
    on_stack = [False] * len(graph)
    sccs = []
    for node in range(len(graph)):
    #从图中选择一个未访问的顶点开始dfs
        if indices[node] == 0:
            dfs(node)
    return sccs
```

CH5 内排序算法

在内存中进行的排序，叫做内排序，简称排序，复杂度不可能优于O(nlogn)；对外存（硬盘）上的数据进行排序，叫外排序

对于排序算法的评价：时间复杂度，空间复杂度（额外空间），是否稳定（同样大小的元素，排序前和排序后是否先后次序不变）

排序总结：

各种排序方法总结

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

python内置排序：蒂姆排序Timsort，混合了归并和插入排序，稳定，最好情况O(n)，最坏时间复杂度O(nlogn)，额外空间最坏O(n)，但通常较少。是目前为止最快的排序算法

一、插入排序

（一）直接插入排序

1.基本思想：

将序列分成有序和无序的部分。有序部分在左边，无序的部分在右边。开始时，有序部分只有一个元素

每次找到无序部分的最左元素i，将其插入到有序部分的合适位置k（原下标为k到i-1的元素都右移一位），有序部分元素个数+1

```
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
```

2.评述

最坏情况倒序：总复杂度n^2;平均情况（1）执行i/2次，总复杂度O(n^2);最好情况有序，语句(1)不执行，总复杂度O(n)

额外空间：O(1)

规模很小的排序可以优先选用；特别适合元素基本有序的情况

（二）改进的插入排序（希尔排序Shell）

1.基本思想

选取增量（间隔）为D，根据增量将列表分为多组，每组分别插入排序。

初始可选取D = n//2，后续D = D//2

```
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
```

2.评述

不稳定，最好情况可以到O(n)，最坏情况O(n^2)，但平均可以到O(n^1.5)

希尔排序中，对确定规模的这些小序列进行插入排序时，要访问序列中的所有记录

二、选择排序

（一）直接选择排序

1.基本思想

将序列分为有序和无序的部分。有序部分在左边，无序部分在右边。开始有序部分**没有**元素

每次找到无序部分的**最小**元素i，和无序部分的最左边元素j交换。有序部分元素个数+1。做n-1次，排序即完成

```
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]
```

2.评述：整体弱于插入排序

最好、最坏、平均时间复杂度均为O(N^2)，丢失了在最好情况下的时间便捷性（这是由于比较必须执行1+2+...+(n-1)次）

稳定性：基于交换，不稳定

额外空间：O(1)

（二）堆排序

1.基本思想

将待排序列表a变成一个堆。将a[0]和a[n-1]做交换，然后对a[0]做下移，维持前n-1个元素依然是堆。以此类推，直到堆的长度变为1，列表就按照优先级从低到高排好序了

2.评述

最好、最坏、平均时间复杂度均为O(nlogn),排序不稳定

额外所需空间为O(1)（perc元素交换）；若用递归实现，需要O(logN)的额外栈空间

三、交换排序

（一）冒泡排序

1.基本思想

将序列分成有序和无序的部分。有序的部分在**右**边，无序的部分在**左**边。开始时，有序部分**没有**元素

每次从左到右，依次比较无序部分相邻的两个元素，如果右边的小于左边的，则交换它们。做完一次后，无序部分最大元素即被换到无序部分的最右边，有序部分元素个数+1.经过n-1次，排序完成

```
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.评述

改进前，时间复杂度为O(n^2)，改进后，最好情况下时间复杂度为O(n)。是稳定排序，需要的额外空间为O(1)

（二）快速排序（分治的典型应用）

注：分支可以递归，也可以不递归

1.基本思想

设k = a[0],将k挪到适当位置，使得比k小的元素都在k左边，比k大的元素都在k右边，和k相等的，不关心在k左右出现即可（O(n)）。然后把k左边的部分快速排序，把k右边的部分快速排序

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

2.评述

最好/平均情况时间复杂度为O(nlogn)，最坏情况（**已经基本有序或者倒序**）情况下，时间复杂度为O(n^2)

不稳定。存在递归情况下，所需要的额外空间复杂度为O(logn)

快排中，递归树上根据深度划分的每个层次都要访问序列的所有记录

三、归并排序、基数排序、桶排序

（一）归并排序

1.基本思想

把前一半排序，把后一半排序，再把两半归并到一个新的有序数组，然后再拷贝回原数组，排序完成

```
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
```

2.评述

无所谓最坏、最好或平均情况，复杂度都是O(nlogn)。是稳定排序，只要归并两个区间时，碰到相同元素，总是取左区间元素即可

额外空间：O(n)（归并用额外空间，栈空间O(logn)是小量）

归并排序过程中，递归树上每个层次的归并操作不需要访问序列中的所有记录

（二）桶排序（分配排序）

1.基本思想

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

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

```
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
```

2.评述

稳定排序，时间复杂度O(n+m)，额外空间O(n+m)

（三）基数排序（多轮分配排序）

1.基本思想：

将待排序元素看作由相同个数的原子构成的元组。长度不足的元素，用最小原子补齐左边空缺的部分。有n种原子，就设立n个桶

先按照最右边的原子将所有元素分配到各个桶里，然后从小桶到大桶收集所有元素。再按照右边第二个元素分配，直到完成分配，收集得到序列

```
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
```

2.评述

基数排序的复杂度O(d*(n+radix)) (radix：桶的个数，即组成元素的原子的种类数)

空间复杂度O(n+radix)

CH6 其他算法

一、单调栈

```
# 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
```

二、KMP算法（字符串匹配）复杂度O(m+n)

```python
# 计算pattern字符串的next数组
def compute_next(pattern):
    m = len(pattern)
    next = [0]*m
    length = 0
    for i in range(1,m):
        while length > 0 and pattern[i] != pattern[length]:
            length = next[length-1]
        if pattern[i] == pattern[length]:
            length += 1
        next[i] = length
        #做下一个比对时，直接去比对索引为length即可
        #length也是可以跳过的字符个数
    return next
def kmp(text,pattern):
    n,m = len(text),len(pattern)
    if m == 0:
        return 0
    next = compute_next(pattern)
    matches = []
    j = 0 #pattern索引
    for i in range(n):
        while j>0 and text[i] != pattern[j]:
            j = next[j-1] #看前一个next数组的值
            #j表示可以跳过匹配的字符个数
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i-j+1)
            j = next[j-1]
    return matches
```

三、二分查找算法

```python
#从有序列表a中查找元素p,复杂度O(logn)
def BinarySearch(a,p,key = lambda x:x):
    l,r = 0,len(a)-1
    while l <= r:
        mid = l + (r-l)//2
        if key(p) < key(a[mid]):
            r = mid - 1
        elif key(a[mid]) < key(p):
            l = mid + 1
        else:
            return mid
    return None
#写一个函数lowerBound，在从小到大的列表a里查找比给定元素p小，下标最大的元素
def lowerBound(a,p,key = lambda x:x):
    l,r = 0,len(a)-1
    result = None
    while l<=r:
        mid = l+(r-l)//2
        if key(a[mid]) < key(p):
```

```python
            l = mid + 1
            result = p
        else:
            r = mid - 1
    return result
```

四、其他

```python
#继承类的写法
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(f"{self.name} makes a sound.")
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
    def speak(self):
        print(f"{self.name} the {self.breed} dog barks.")
```