一、数据结构

1.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。

（1）逻辑结构：是指数据对象中数据元素之间的相互关系。

集合结构、线性结构（一对一）、树形结构（一对多）、图形结构（多对多）

（2）物理结构（存储结构）：是指数据的逻辑结构在计算机中的存储形式。

顺序存储结构：是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。

链式存储结构：是把数据元素存放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。数据元素的存储关系并不能反映其逻辑关系，因此需要用一个指针存放数据元素的地址，这样通过地址就可以找到相关联数据元素的位置。

2.数据类型：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

在 C 语言中，按照取值的不同，数据类型可以分为两类：

原子类型：是不可以再分解的基本类型，包括整型、实型、字符型等。

结构类型：由若干个类型组合而成，是可以再分解的。例如，整型数组是由若干整型数据组成的。

3.抽象数据类型（Abstract Data Type，ADT）：是指一个数学模型及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关。

一个抽象数据类型定义了：一个数据对象、数据对象中各数据元素之间的关系及对数据元素的操作。

事实上，抽象数据类型体现了程序设计中问题分解、抽象和信息隐藏的特性。抽象数据类型把实际生活中的问题分解为多个规模小且容易处理的问题，然后建立一个计算机能处理的数据模型，并把每个功能模块的实现细节作为一个独立的单元，从而使具体实现过程隐藏起来。

二、算法

1.算法：算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

算法具有五个基本特性：输入、输出、有穷性、确定性和可行性

算法设计的要求：正确性、可读性、健壮性、时间效率高和存储量低

健壮性：当输入数据不合法时，算法也能做出相关处理，而不是产生异常或莫名其妙的结果。

2.算法的时间复杂度：在进行算法分析时，语句总的执行次数 T(n)是关于问题规模 n 的函数，进而分析 T(n)随 n 的变化情况并确定 T(n)的数量级。算法的时间复杂度，也就是算法的时间量度，记作：T(n)=O(f(n))。它表示随问题规模 n 的增大，算法执行时间的增长率和 f(n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中 f(n)是问题规模 n 的某个函数。

这样用大写 O()来体现算法时间复杂度的记法，我们称之为大O记法。

一般情况下，随着 n 的增大，T(n)增长最慢的算法为最优算法。

O(n)常数阶 O(n)线性阶 O(n^2)平方阶 O(logn)对数阶

3.常用的时间复杂度所耗费的时间从小到大依次是:

o(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)

执行次数函数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	O(n^2)	平方阶
$5\log_2 n+20$	O($\log n$)	对数阶
$2n+3n\log_2 n+19$	O($n\log n$)	nlogn 阶
$6n^3+2n^2+3n+4$	O(n^3)	立方阶
2^n	O(2^n)	指数阶

4.最坏运行情况和平均运行情况

一般来说，时间复杂度都是指最坏时间复杂度，即最坏情况下的时间复杂度。

5.算法空间复杂度

一般来说，算法的复杂度都是指时间复杂度。我们可以通过空间来换取时间。

三、线性表
线性表（List）：零个或多个数据元素的有限序列。
线性表的长度：线性表中元素的个数 n（n≥0）。当 n=0 时，称为空表。
在较复杂的线性表中，一个数据元素可以由若干个数据项组成。
1.线性表的顺序存储结构
优点：
无须为表示表中元素之间的逻辑关系而增加额外的存储空间
可以快速地存取表中任位置的元素
缺点：
插入和删除操作需要移动大量元素
当线性表长度变化较大时,难以确定存储空间的容量
造成存储空间的“碎片”
2 线性表的链式存储结构
为了表示每个数据元素a_i与其直接后继数据元素a_{i+1}之间的逻辑关系，对数据元素a_i来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的信息(即直接后继的存储位置)。我们把存储数据元素信息的域称为数据域，把存储直接后继位置的域称为指针域。指针域中存储的信息称做指针或链。这两部分信息组成数据元素a_i的存储映像，称为结点(Node)。
结点=存放数据元素的数据域+存放后继结点地址的指针域
链表中第一个节点的存储位置称为头指针，最后一个节点的指针为空(NULL)。
头结点：在单链表中第一个结点前附设一个结点，可存储线性表长度等公共信息
3 对于插入或删除数据越频繁的操作，单链表的效率优势就越明显。
若线性表需要频繁查找，很少进行插入和删除操作时，宜采用顺序存储经构。若需要频繁插入和删除时，宜采用单链表结构。
4 静态链表：用数组描述的链表，是为了给没有指针的高级语言设计的一种实现单链表能力的方法。
首先我们让数组的元素都是由两个数据域组成，data 和 cur。也就是说，数组的每个下标都对应一个 data 和一个 cur。数据域 data，用来存放数据元素，也就是通常我们要处理的数据；而游标 cur 相当于单链表中的 next 指针，存放该元素的后继在数组中的下标。
5 循环列表：头尾相接的单链表
6 双向链表

四、栈与队列
栈：限定仅在表尾进行插入和删除操作的线性表。
队列：只允许在一端进行插入操作、而在另一端进行删除操作的线性表。
1 栈
栈顶 top：允许插入和删除的一端　栈底 bottom：另一端
LIFO 结构：后进先出 Last in First out
优点：只准栈顶进出元素，不存在线性表插入和删除时需要移动元素的问题（操作栈顶指针即可）
缺点：必须事先确定数据存储空间大小（可使用一个数组存储两个相同类型的栈）
2 栈的应用：递归
递归：一个直接调用自己或通过一系列调用语句间接调用自己的函数

递归和迭代的区别：迭代使用的是循环结构，递归使用的是选择结构。递归可以使程序的结构更清晰、更简洁。

3 队列

FIFO 结构：First in First out

加入头尾两个指针（front、rear），删除元素时不需要移动元素

循环队列：解决“假溢出”问题

五、串

串（string）是由 0 个或者=多个字符组成的有限序列，又称字符串

六、树

树（Tree）是 n （n≥0）个结点的有限集。n=0 时称为空树。在任意一棵非空树中：（1）有且仅有一个特定的称为根（Root）的结点；（2）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、……、Tn，其中每一个集合本身又是一棵树，并且称为根的子树（SubTree）。

特点：根结点是唯一的，子树互不相交

1 度 Degree：结点拥有的子树数目

叶结点 Leaf（终端结点）：度为 0　　非终端结点（分支结点）：度不为 0

树的度是树内各结点度的最大值

2 结点之间的关系

结点的子树的根称为该结点的孩子（Child），相应地，该结点称为孩子的双亲（Parent）。同一个双亲的孩子之间互称兄弟（Sibling）。结点的祖先是从根到该结点所经分支上的所有结点。反之，以某结点为根的子树中的任一结点都称为该结点的子孙。

3 树的其他相关概念

结点的层次（Level）从根开始定义起，根为第一层，根的孩子为第二层。

树中结点的最大层次称为树的深度（Depth）或高度。

如果将树中结点的各子树看成从左至右是有次序的，不能互换的，则称该树为有序树，否则称为无序树。

森林（Forest）是 m (m≥0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。对于树而言，它的两棵子树其实就可以理解为森林。

4 二叉树

二叉树（Binary Tree）是 n（n≥0）个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

二叉树的特点有：

（1）每个结点最多有两棵子树，所以二叉树中不存在度大于 2 的结点。注意不是只有两棵子树，而是最多有。没有子树或者有一棵子树都是可以的。

（2）左子树和右子树是有顺序的，次序不能任意颠倒。

即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

5 特殊二叉树

斜树：所有结点都只有左子树的二叉树称为左斜树，都只有右子树的二叉树称为右斜树。这两者统称斜树。

满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

满二叉树的特点：（1）叶子只能出现在最下一层。出现在其他层就不可能达成平衡；（2）

非叶子结点的度一定是 2；（3）在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多。

完全二叉树：对一棵具有 n 个结点的二叉树按层序编号，如果编号为 i（1≤i≤n）的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

完全二叉树的特点：（1）叶子结点只能出现在最下两层；（2）最下层的叶子一定集中在左部连续位置；（3）倒数二层，若有叶子结点，一定都在右部连续位置；（4）如果结点度为 1，则该结点只有左子树，即不存在只有右子树的情况；（5）同样结点数的二叉树，完全二叉树的深度最小。

6 二叉树的性质

（1）在二叉树的第 i 层上至多有 2^{i-1} 个结点（i≥1）

（2）深度为 k 的二叉树至多有 2^k-1 个结点（k≥1）

（3）对任何一棵二叉树 T，如果其终端结点数（叶子结点数）为 n_0，度为 2 的结点数为 n_2，度为 1 的结点数为 n_1，则结点总数 $n=n_0+n_2+n_1$，$n_0=n_2+1$

（4）具有 n 个结点的完全二叉树的深度为$[\log_2 n]+1$（[x]表示不大于 x 的最大整数）

（5）如果对一棵有 n 个结点的完全二叉树（其深度为$[\log_2 n]+1$）的结点按层序编号（从第 1 层到第$[\log_2 n]+1$ 层，每层从左到右），对任一结点 i（1≤i≤n）有：

1）如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i > 1，则其双亲是结点[1/2]

2）如果 2i > n，则结点 i 无左孩子（结点 i 为叶子结点）；否则其左孩子是结点 2i

3）如果 2i+1 > n，则结点 i 无右孩子（结点 i 为叶子结点）；否则其右孩子是结点 2i+1

7 二叉树的遍历

二叉树的遍历(traversing binary tree）是指从根结点出发,按照某种次序依次访问二叉树中所有结点,使得每个结点被访问一次且仅被访问一次。

（1）前序遍历

规则是若二叉树为空，则空操作返回，否则先访问根结点，然后前序遍历左子树，再前序遍历右子树。

（2）中序遍历

规则是若二叉树为空，则空操作返回，否则先从根结点开始，然后中序遍历左子树，然后访问根结点，最后中序遍历右子树。

（3）后序遍历

规则是若二叉树为空，则空操作返回，否则，先后序遍历左子树，再后序遍历右子树，最后访问根结点

（4）层序遍历

规则是若树为空，则空操作返回，否则从树的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

8 赫夫曼树（最优二叉树）

可用于优化算法效率（if-else 问题等）

实现方式：从树中一个结点到另一个结点之间的分支构成两个结点之间的路径，路径上的分支数目称为路径长度。树的路径长度就是从树根到每一结点的路径长度之和。考虑带权的结点，结点的带权路径长度为树中所有叶子结点的带权路径长度之和。其中带权路径长度 WPL 最小的二叉树称做赫夫曼树。

七、图

图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，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。
图是一种较线性表和树更加复杂的数据结构。在图形结构中，结点之间的关系可以是任意的。

八、查找
查找（Searching）是根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。
1 查找概论
查找表（Search Table）是由同一类型的数据元素（或记录）构成的集合
关键字（Key）是数据元素中某个数据项的值，又称为键值，用它可以标识一个数据元素。也可以标识一个记录的某个数据项（字段），我们称为关键码。
主关键字（Primary Key）：可以唯一地表示一个记录的关键字。主关键字所在的数据项称为主关键码。
次关键字（Secondary Key）：可以识别多个数据元素（或记录）的关键字。次关键字所对应的数据项就是次关键码。
2 查找表
静态查找表（Static Search Table）：只作查询操作的查找表
动态查找表（Dynamic Search Table）：在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已经存在的某个数据元素
3 顺序查找　时间复杂度 O(n)
顺序查找（Sequential Search）又叫线性查找，是最基本的查找技术，它的查找过程是：从表中第一个（或最后一个）记录开始，逐个进行记录的关键字和给定值比较，若某个记录的关键字和给定值相等，则查找成功，找到所查的记录；如果直到最后一个（或第一个）记录，其关键字和给定值比较都不等时，则表中没有所查的记录，查找不成功。
4 有序查找
（1）折半查找（二分查找）　时间复杂度 O(logn)
折半查找 (Binary Search) 的前提是线性表中的记录必须是关键码有序 (通常从小到大有序)，线性表必须采用顺序存储。
折半查找的基本思想是：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键字相等，则查找成功；若给定值小于中间记录的关键字，则在中间记录的左半区继续查找；若给定值大于中间记录的关键字，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所有查找区域无记录，查找失败为止。
（2）插值查找　时间复杂度 O(logn)
插值查找（Interpolation Search）是根据要查找的关键字 key 与查找表中最大最小记录的关键字比较后的查找方法，是基于折半查找的改进。
对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好得多。
（3）斐波那契查找　时间复杂度 O(logn)
中间结点的位置（mid）在黄金分割点附近，而不是中间或者插值得到
5 线性索引查找
索引：把一个关键字与它对应的记录相关联的过程
线性索引：将索引项集合组织为线性结构，也称为索引表
1 三种线性索引
（1）稠密索引：在线性索引中，将数据集中的每个记录对应一个索引项。索引项一定是按照关键码有序的排列。

优点：可以利用有序查找算法
缺点：对于非常大的数据集，查找性能不佳
（2）分块索引
把数据集的记录分成若干块，且满足两个条件：
块内无序，即每一块内的记录不要求有序。当然，你如果能够让块内有序对查找来说更理想，不过这就要付出大量时间和空间的代价，因此通常我们不要求块内有序。
块间有序，例如，要求第二块所有记录的关键字均要大于第一块中所有记录的关键字，第三块的所有记录的关键字均要大于第二块的所有记录关键字……因为只有块间有序，才有可能在查找时带来效率。
对于分块有序的数据集，将每块对应一个索引项，这种索引方法叫做分块索引。
我们定义的分块索引的索引项结构分三个数据项：
①最大关键码，它存储每一块中的最大关键字，这样的好处就是可以使得在它之后的下一块中的最小关键字也能比这一块最大的关键字要大；
②存储了块中的记录个数，以便于循环时使用；
③用于指向块首数据元素的指针，便于开始对这一块中记录进行遍历。
分块索引查找的步骤：
①在分块索引表中查找要查关键字所在的块。由于分块索引表是块间有序的，因此很容易利用折半、插值等算法得到结果。
②根据块首指针找到相应的块，并在块中顺序查找关键码。因为块中可以是无序的，因此只能顺序查找。
（3）倒排索引　用于浏览器搜索等
索引项的通用结构是：次关键码，记录号表
其中记录号表存储具有相同次关键字的所有记录的记录号（可以是指向记录的指针或者是该记录的主关键字）。这样的索引方法就是倒排索引（inverted index）。倒排索引源于实际应用中需要根据属性（或字段、次关键码）的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引。
6 二叉排序树
二叉排序树（Binary Sort Tree），又称为二叉查找树。它或者是一棵空树，或者是具有下列性质的二叉树：
（1）若它的左子树不空，则左子树上所有结点的值均小于它的根结构的值
（2）若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值
（3）它的左右子树也分别为二叉排序树
从二叉排序树的定义也可以知道，它前提是二叉树，然后它采用了递归的定义方法，再者，它的结点间满足一定的次序关系，左子树结点一定比其双亲结点小，右子树结点一定比其双亲结点大。
构造一棵二叉排序树的目的，其实并不是为了排序，而是为了提高查找和插入删除关键字的速度。不管怎么说，在一个有序数据集上的查找，速度总是要快于无序的数据集的，而二叉排序树这种非线性的结构，也有利于插入和删除的实现。
7 平衡二叉树（AVL 树）
平衡二叉树（Self-Balancing Binary Search Tree 或 Height-Balanced Binary SearchTree），是一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于 1。
我们将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子 BF (Balance Factor)，那么平衡二叉树上所有结点的平衡因子只可能是一 1、0 和 1。只要二叉树上有一个结点的

平衡因子的绝对值大于 1，则该二叉树就是不平衡的。

8 多路查找树（B 树）

多路查找树（muitl-way search tree），其每一个结点的孩子数可以多于两个，且每一个结点处可以存储多个元素。由于它是查找树，所有元素之间存在某种特定的排序关系。

9 散列表查找（哈希表）概述

通过查找关键字，不需要比较就可获得需要的记录的存储位置

散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使得每个关键字 key 对应一个存储位置 f (key)。查找时，根据这个确定的对应关系找到给定值 key 的映射 f (key)，若查找集合中存在这个记录，则必定在 f (key)的位置上。

这里我们把这种对应关系 f 称为散列函数，又称为哈希(Hash)函数。按这个思想，采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表(Hash table)。那么关键字对应的记录存储位置我们称为散列地址。

散列技术的记录之间不存在逻辑关系，它只与关键字有关，是主要面向查找的存储结构。散列技术最适合的求解问题是查找与给定值相等的记录。

九、排序

排序：假设含有 n 个记录的序列为{r_1，r_2……，r_n]，其相应的关键字分别为{k_1，k_2……,k_n}，需确定 1，2，……，n 的一种排列 p_1，p_2……,p_n，使其相应的关键字满足 $k_{p1}\leqslant k_{p2}\leqslant\cdots\cdots\leqslant k_{pn}$（非递减或非递增）关系，即使得序列成为一个按关键字有序的序列{r_{p1},r_{p2}，……，r_{pn}}，这样的操作就称为排序。

1 排序的基本概念与分类

内排序和外排序：内排序是在排序整个过程中，待排序的所有记录全部被放置在内存中。外排序是由于排序的记录个数太多，不能同时放置在内存，整个排序过程需要在内外存之间多次交换数据才能进行。我们这里主要就介绍内排序的多种方法。

对于内排序来说，排序算法的性能主要是受 3 个方面影响：

（1）时间性能

（2）辅助空间

（3）算法的复杂性

根据排序过程中借助的主要操作，我们把内排序分为：插入排序、交换排序、选择排序和归并排序。

本章一共要讲解七种排序的算法，按照算法的复杂度分为两大类，冒泡排序、简单选择排序和直接插入排序属于简单算法，而希尔排序、堆排序、归并排序、快速排序属于改进算法。后面我们将依次讲解。

2 冒泡排序 $O(n^2)$

冒泡排序（Bubble Sort）一种交换排序，它的基本思想是：两两比较相邻记录的关键字，如果反序则交换，直到没有反序的记录为止。在冒泡排序的过程中，较小的数字会如同气泡般慢慢浮到上面，因此就将此算法命名为冒泡算法。

3 简单选择排序 $O(n^2)$，但总体性能要略优于冒泡排序

简单选择排序法（Simple selection Sort）就是通过 n-i 次关键字间的比较，从 n-i+1 个记录中选出关键字最小的记录，并和第 i（1≤i≤n）个记录交换之。

4 直接插入排序 $O(n^2)$，需要一个记录的辅助空间，性能略优于冒泡排序和简单选择排序

直接插入排序（Straight Insertion Sort）的基本操作是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增 1 的有序表。

*适用于有部分数据已经排好序的情况，并且排好序的部分越大越好（但数据量也不能太大）

5 希尔排序 时间复杂度好于 O(n^2)，但并不是稳定的排序算法
希尔排序（Shell Sort）的关键并不是随便分组后各自排序，而是将相隔某个“增量”的记录组成一个子序列，实现跳跃式的移动，使得排序的效率提高。
6 堆排序 O(nlogn)，不稳定，不适合排序序列个数较少的情况
堆排序（Heap Sort）是对简单选择排序的一种改进，利用堆数据结构（类似完全二叉树），每次在选择最小记录的同时并根据比较结果对其他记录做出相应的调整，提高排序的总体效率
7 归并排序 O(nlogn) 比较占用内存，但是效率高且稳定
归并排序（Merging Sort）是利用归并的思想实现的排序方法，同样涉及到完全二叉树。
8 快速排序 O(logn) 不稳定，有优化版本，整体性能强，实际应用较多
快速排序（Quick Sort）的基本思想是:通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序的目的。
*希尔排序相当于直接插入排序的升级，它们同属于插入排序类，堆排序相当于简单选择排序的升级，它们同属于选择排序类。而快速排序其实就是我们前面认为最慢的冒泡排序的升级，它们都属于交换排序类。即它也是通过不断比较和移动交换来实现排序的，只不过它的实现，增大了记录的比较和移动的距离，将关键字较大的记录从前面直接移动到后面，关键字较小的记录从后面直接移动到前面，从而减少了总的比较次数和移动交换次数。

十、算法
1.算法的两种基本设计方法:
枚举和选择（选优）：根据问题，设法枚举出所有可能的情况，首先筛选出问题的解（为此需要判断枚举生成的结果是否为问题的解），而后从得到的解中找出最优解（这一步需要能评价解的优劣）。
贪心法：根据当时已知的局部信息，完成尽可能多的工作。这样做通常可以得到正确的解，但可能并非最优。对于一个复杂的问题，全面考虑的工作代价可能太高，为得到实际可用的算法，常常需要在最优方面做出妥协。
2.python 集合操作
判断一个集合 vs 是空集：not vs
设置一个集合为空：vs = set()
从集合中去掉一个元素：vs.remove(v)
向集合里增加一个元素：vs.add(v)

【代码】
一、基础部分
1、算法
1.1 常用:
（1）二分
（2）递归

```
def countdown(i):
    print i
    if i <= 0:   基线条件
        return
    else:       递归条件
        countdown(i-1)
```

```
#递归计算列表所有数字之和
def add(list1):
    if len(list1)<2:
        return list1[0]
    else:
        return list1[0]+add(list1[1:])
```

（3）快速排序

```
def quicksort(array):
    if len(array) < 2:  基线条件
        return array
    else:
        pivot = array[0]  递归条件
        less = [i for i in array[1:] if i <= pivot]  由所有小于基准值的
        元素组成的子数组
        greater = [i for i in array[1:] if i > pivot]  由所有大于基准值的
        元素组成的子数组
        return quicksort(less) + [pivot] + quicksort(greater)
```

（9）广度优先搜索（breadth-first search，BFS）

广度优先搜索：可找出两样东西之间的最短距离（最短路径问题）

狄克斯特拉算法：可找出加权图中前往 X 的最短路径（有向无环图）

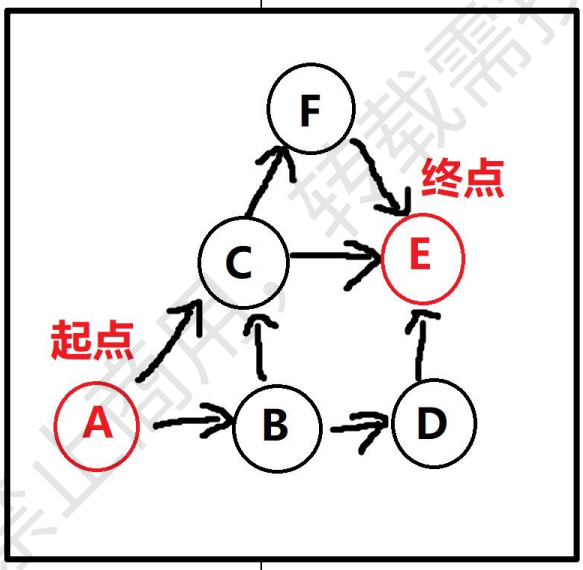
*在 Python 中，可以用字典来表示图，列表表示队列

```
graph1 = {"A":["B","C"],"B":["C","D"],"C":["E","F"],"D":["E"],"E":[],"F":["E"]}

def get_path(graph,list1,s1,s2):
    t = list1[-1]
    path = [t]
    list1 = list1[:-1]
    while list1:
        for i in list1:
            if t in graph[i]:
                t = i
                path.append(i)
                list1 = list1[:-1]
            else:
                list1 = list1[:-1]
    path = path[::-1]
    path.append(s2)
    return path

def BFS(graph,start,end):
    search = [start]
    searched = []
    while search:
        s = search[0]
        search = search[1:]
        if s not in searched:
            if end not in graph[s]:
                search += graph[s]
                searched.append(s)
            else:
                searched.append(s)
                print('True')
                return searched

s1 = BFS(graph1,'A','E')
p1 = get_path(graph1,s1,'A','E')
print(p1)
```



（3）深度优先搜索（DFS）

按照一定的顺序，使用回溯法进行搜索，其采用了一种“一直向下走，走不通就掉头”的思想

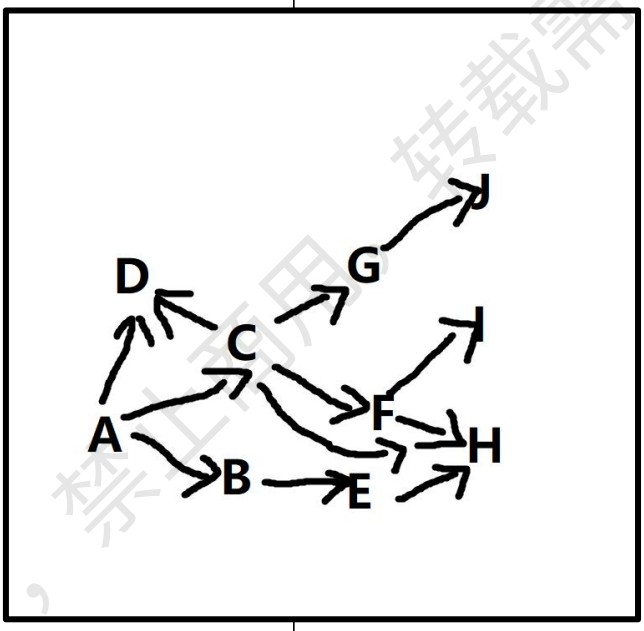
（广度优先搜索则是一层一层的搜索）

*深度优先搜索和广度优先搜索的区别：把队列换成栈

```
graph1 = {
      "A":["B","C","D"],
      "B":["E"],
      "C":["D","G","F","H"],
      "D":[],
      "E":["H"],
      "F":["H","I"],
      "G":["J"],
      "H":[],
      "I":[],
      "J":[]
}

def DFS(graph,start,end):
      search = [start]
      searched = []
      while search:
            s = search[-1]
            search = search[:-1]
            if s not in searched:
                  if end not in graph[s]:
                        search += graph[s]
                        searched.append(s)
                  else:
                        searched.append(s)
                        print('True')
                        return searched

s1 = DFS(graph1,'A','I')
print(s1)
```



（4）动态规划
动态规划先解决子问题，再逐步解决大问题
适用条件：需要在给定约束条件下优化某种指标，问题可分解为离散子问题时
（6）贪心（贪婪算法）
选择局部最优解，企图以这种方式获得全局最优解
贪婪算法易于实现、运行速度快，是不错的近似算法
1.2 进阶：两点、动态规划、最短路径、最小生成树 等
2、数据结构
1.1 常用：数组、队列、堆栈、表
1.2 进阶：单调栈、树状数组、线段树、并查集