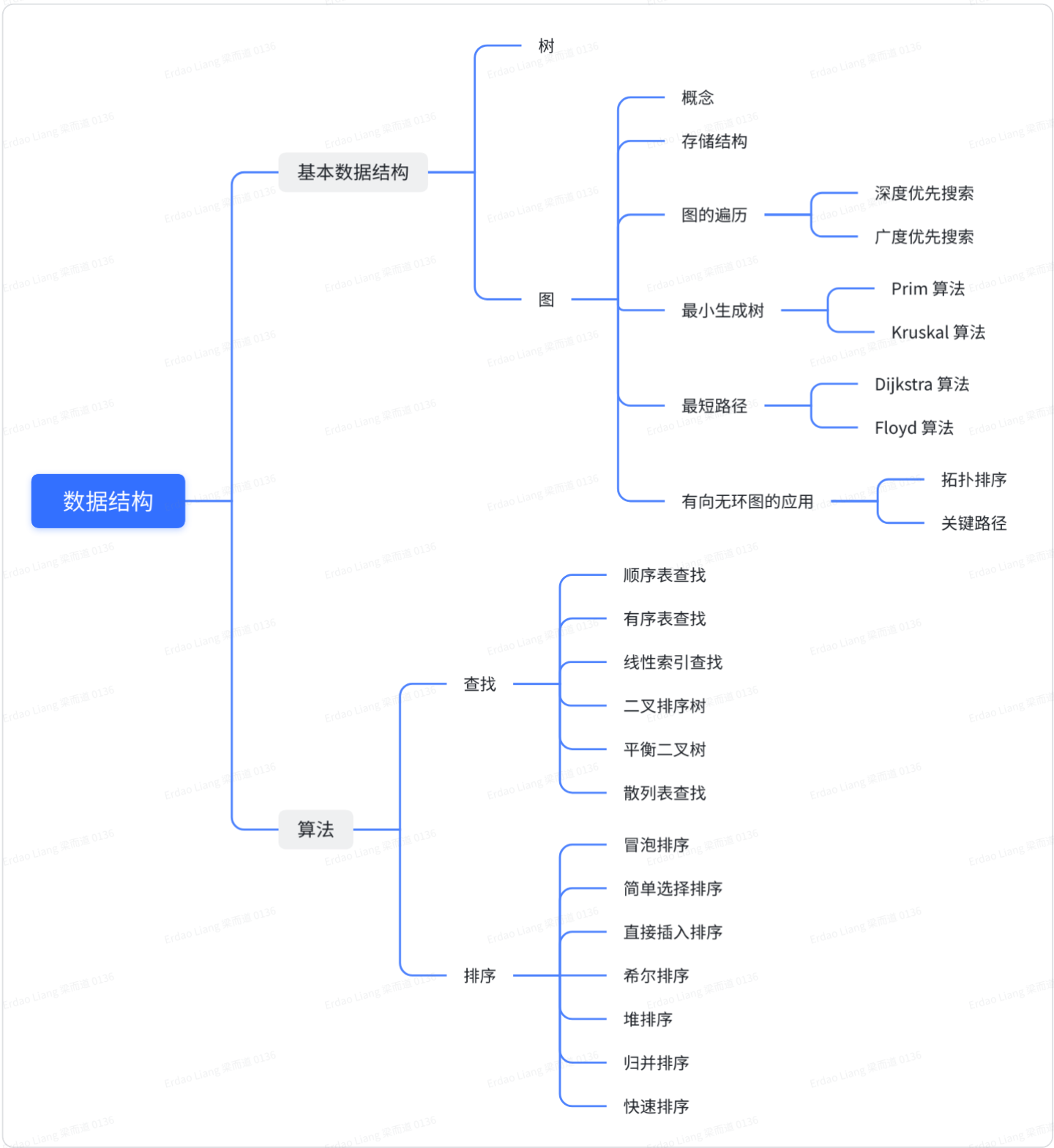
数据结构与算法



基本数据结构

二叉树 Binary Tree

1. 定义
2. 特殊二叉树：
    a. 斜树：所有结点都只有左（右）子树的二叉树叫左（右）斜树
    b. 满二叉树：所有分支节点有左子树和右子树，所有叶子都在同一层
    c. 完全二叉树：（直观理解）从上往下，从左往右，尽可能排满每一层

    满二叉树⊂完全二叉树
3. 二叉树的性质：
    a. 第i层至多有2^(i-1)个结点
    b. 高度为k的二叉树至多有(2^k)-1个结点
    c. 对任一二叉树，如果叶子数为n0，度为2的结点数为n2，则n0=n2+1
    d. 具有n个结点的完全二叉树的深度为 **floor(log2(n))+1**
    e. 对具有n个结点的完全二叉树的结点按层序编号，对任意结点i (1<=i<=n)
        i. 若2i>n，则结点i无左孩子，否则**左孩子是结点2i**
        ii. 若2i+1>n，则结点i无有孩子，否则**右孩子是结点2i+1**
4. 线索二叉树
5. 树、森林和二叉树的转换

图 Graph

概念

1. 图
2. 图的类型
    a. 图分为**无向图**和**有向图**。无向图由顶点和**边**构成，有向图由顶点和**弧**构成。弧有**弧尾**和**弧头**之分。
    b. 如果任意两个顶点之间都存在边叫**完全图**，有向的叫**有向完全图**。若无重复的边或顶点到自身的边则叫**简单图**。
3. 顶点与边的关系
    a. 图中顶点之间有**邻接点**、依附的概念。无向图顶点的边数叫做**度**，有向图顶点分为**入度**和**出度**。
    b. 图上的边或弧上带**权**则称为**网**。
4. 连通图

a. 图中顶点间存在**路径**，两顶点存在路径则说明是连通的，如果路径最终回到起始点则称为环，当中不重复叫**简单路径**。

b. 若任意两顶点都是连通的，则图就是连通图，有向则称**强连通图**。图中有子图，若子图极大连通则就是连通分量，有向的则称强连通分量。

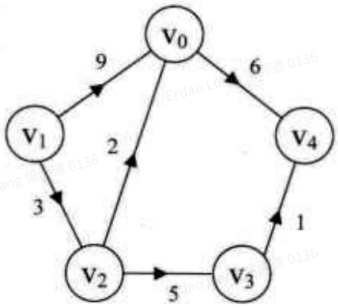
c. 无向图中连通且 n 个顶点 n-1 条边叫**生成树**。有向图中一顶点入度为0其余顶点入度为1的叫**有向树**。一个有向图由若干有向树构成**生成森林**。

图的存储结构

1. **邻接矩阵** Adjacency Matrix

   组成：顶点数组（一维，存储顶点信息）+边数组（二维）

   无向图的边数组是一个对称矩阵



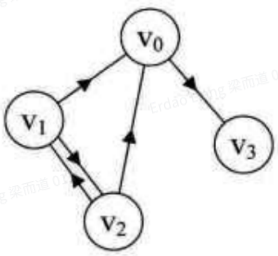
顶点数组：

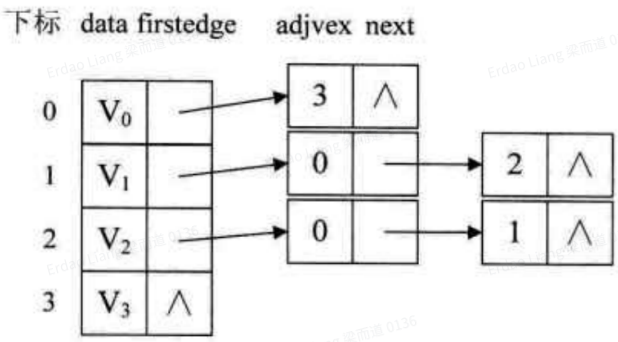
v_0	v_1	v_2	v_3	v_4

边数组：

	v_0	v_1	v_2	v_3	v_4
v_0	0	∞	∞	∞	6
v_1	9	0	3	∞	∞
v_2	2	∞	0	5	∞
v_3	∞	∞	∞	0	1
v_4	∞	∞	∞	∞	0

2. **邻接表** Adjacency List





   邻接表（链表是出边） vs 逆邻接表（链表是入边）

3. **十字链表** Orthogonal List（针对有向图）

   a. 本质：同一条边只用一个结点表示；结合邻接表和逆邻接表，方便同时访问出边和入边

   b. 组成

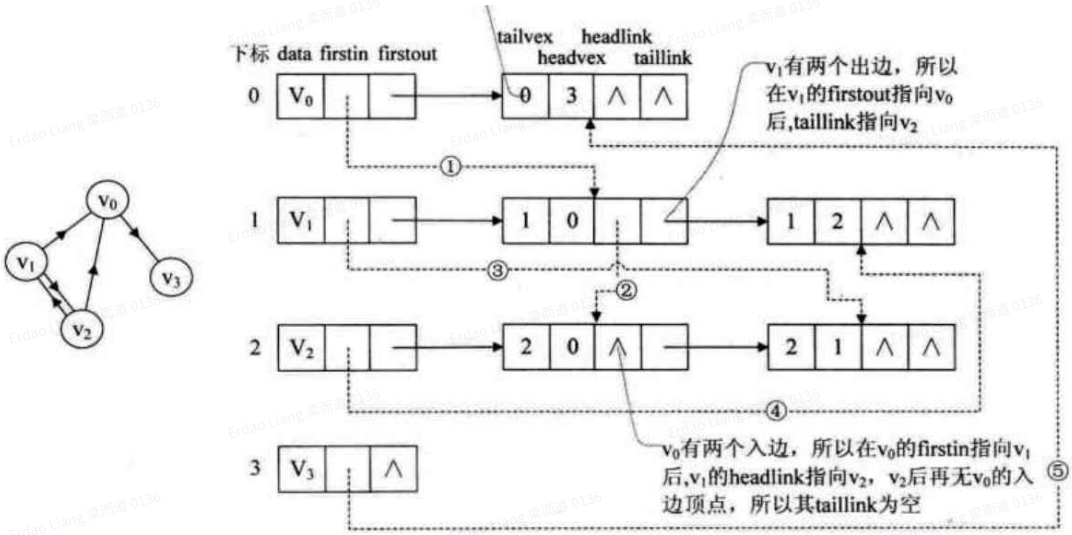
      i. 顶点表 data | firstin | firstout

      ii. 边表

tailvex	headvex	headlink	taillink（headlink/taillink指向一个边结点整体）
该边的弧尾	该边的弧头	终点相同的下一条边	起点相同的下一条边

c. 找入边：顶点表的顶点→firstin→headlink

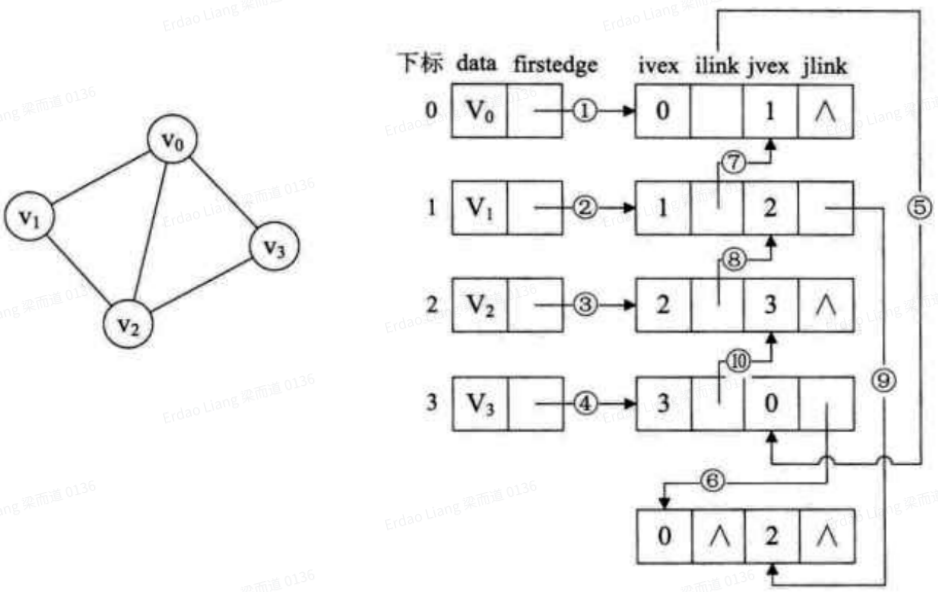
d. 找出边：顶点表的顶点→firstout→taillink



4. **邻接多重表**（针对无向图）

a. 查找一个顶点的所有邻边：data→firstedge→ivex：ilink / jvex：jlink

b. 注意：ilink/jlink指向一个边结点的ivex或jvex而非整体



5. **边集数组**

图的遍历

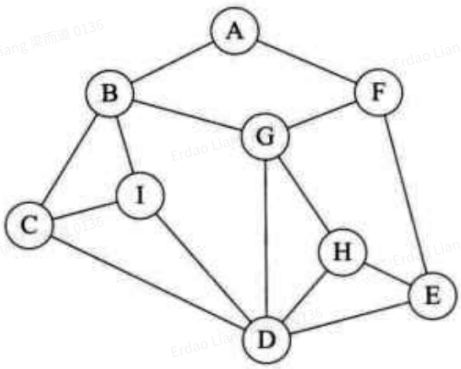
深度优先遍历 DFS

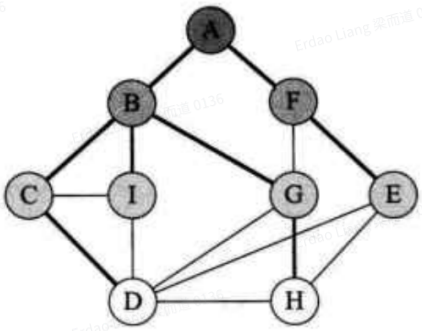
1. 逻辑：树的前序遍历；递归实现
2. 伪代码

```
def DFSTraverse():
    visited = [全部未访问]
    for 每一个顶点:
        if 未访问过这个顶点: DFS(从这个顶点开始) #连通图只会有一次

def DFS():
    更新visited
    对该顶点操作
    for/while 每个与该顶点邻接的顶点: #矩阵可用for，链表可用while
        if 未访问过这个顶点: DFS(这个邻接顶点)
```

广度优先遍历 BFS

1. 逻辑：树的层序遍历；用**队列**实现
   a. 队列存放的是待处理的顶点11.12 11.13
   b. 每次循环处理一个元素时做三件事：该元素出队列→操作该元素→探测其孩子顶点并入队列（它们是下一层的，还没轮到它们）





2. 伪代码

```
def BFSTraverse():
    queue = 一个队列
    for 每一个顶点:
        if 未访问过这个顶点: #连通图只会有一次
            enqueue(当前顶点) # 若为连通图，只针对第一个元素
            while 队列非空: # 这个循环不断在队列中拿元素出来处理
                一个顶点 = dequeue()
                对顶点操作
                for/while 每个与该顶点邻接的顶点: #矩阵可用for，链表可用while
                    if 未访问过这个顶点: enqueue(这个邻接顶点)
```

最小生成树 Minimum Cost Spanning Tree

目标：以最小的权值和构造一个图的连通网

Prim 算法

1. 基于邻接矩阵或邻接表
2. 逻辑：
   a. 从任意一个顶点出发
   b. 每轮选一个新的顶点纳入生成树，这个顶点是目前到当前整棵生成树（到生成树上的任意顶点）的距离最小的
   c. 直到所有顶点都被纳入生成树
3. 伪代码

```
def miniSpanTree_prim(图):
    lowcost = [第i个元素存放当前生成树和第i顶点之间已知的最小距离的列表]
    # 为0表示该顶点已在生成树里，为无限大表示该顶点与生成树还没有边相连
    adjvex = [第i个元素存放使其有上述所说的最小距离的边的邻接顶点]
    # 这个邻接顶点是已经在生成树里的

    #每轮将一个新的顶点纳入生成树
    for i = 1:n-1（n-1次循环，因为初始已经有一个顶点算在树里了）:
        k = index(min(lowcost)) #找到与当前生成树距离最小的顶点，k是其序号
        将该路径加入最小生成树，并标注已加入 #（输出这条路径OR存放到某个地方）
        for j = 1:n:
            修正lowcost和adjvex
            # 因为这个被纳入生成树的顶点k又“开发了一小片新大陆”
```

4. 时间复杂度：O(n^2)，n为顶点数

Kruskal 算法

1. 基于排好序的边集数组
2. 逻辑
   a. 每次选取列表内权值最小&不会使得生成树形成环路的边 加入生成树
   b. 关键：排除**形成环路**的情况 - 对待检验的边的两个顶点分别给它们“溯源”：
      i. 如果某两个点本身还没被连起来，则它们的“源顶点”不会相同，可以相连
      ii. 如果本身已经有一条路径连起这两个点，则它们会溯源到同一个顶点，不能再相连
3. 伪代码

```
def miniSpanTree_kruskal():
    for i = 1:n:  （循环每一条边）
        n = find_parent(边[i].begin)
        m = find_parent(边[i].end)
        if n != m: # n==m意味着这条边连起来会形成环路
            将该边加入最小生成树

def find_parent()
    return parent
```

4. 时间复杂度：O(eloge)，e为边数

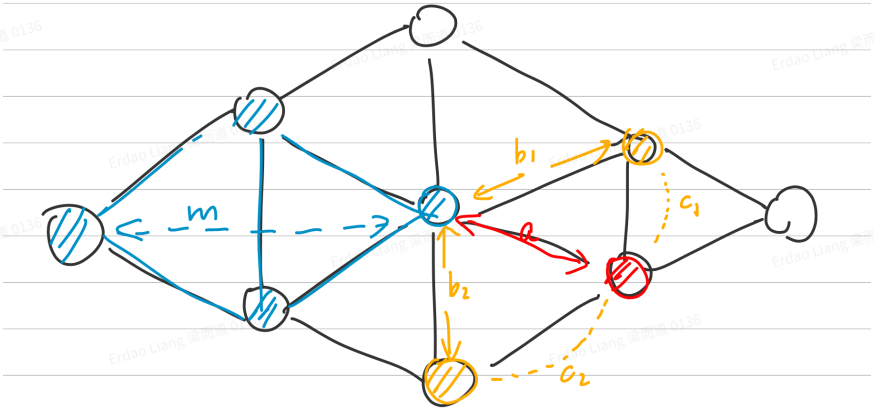
最短路径 Shortest Path

Dijsktra 算法

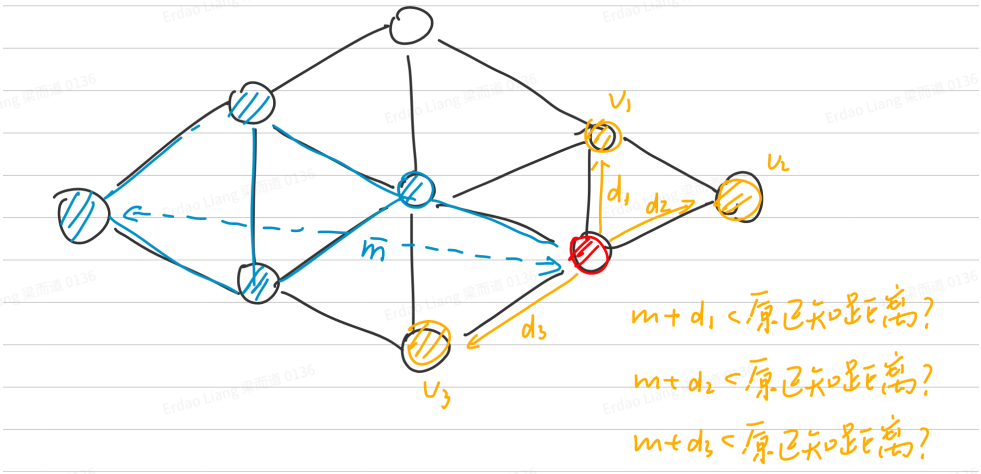
1. 逻辑
   a. 很重要的一个逻辑是：Q 为什么只要在每一轮循环开始时，在还没确定最短路径的那些顶点中选一个已发现的到它的路径比到其他顶点的路径都短的，那个到它的路径就一定是到它的最短路径？明明这些值还在不断被修正？
   b. A 要注意在最短路径列表shortpathtable中，对那些最短路径未确定的顶点，表格对应位置存放的是已发现的最短路径，而这个已发现的最短路径 = {v0到某个已确定最短路径的顶点v1的路径长度（下图的m）}+ {v1到该v的一条直接相连的边长（下图的a）};
   c. 要用最短路径走到v，肯定是要从v1开始的：要么从v1直接走到v，长度{m+a}；要么从v1先走到另外一个点vi，再有某种方式走到v。后者的长度为{m+b+c}，虽然c未知，但我们已经知道{m+a}<{m+bi}（对任何i），所以必有{m+b+c}>{m+a}，所以{m+a}无论如何都是最佳选择
2. 伪代码

```
def shortesetPath_Dijsktra():
    adjvex = []
    shortpathtable = []

    #每轮算出v0到某一个顶点的最短路径
    for i = 1:n-1  （n-1次循环）:
        k = index(min(shortpathtable))

        （此时shortpathtable在k处的值就已经是最短路径）
        for j = 1:n:
            修正shortpathtable和adjvex
```

a. 选顶点



b. 修正



3. 时间复杂度：O(n^2) （算出一个顶点和其余所有顶点间的最短距离）

Floyd 算法

1. 核心：如果经过下标为k的顶点的路径比原两点v,w间路径更短，则将当前两点间权值设为更小的一个

   a. 抛开传统的点到点的思维

2. 伪代码

```
def shortesetPath_Floyd():
    for k:
        for a:
            for b:
                if 路径[a→k]+路径[k→b] < 路径[a→b]:
                    修正a→b的最短路径
```

3. 时间复杂度：O(n^3)（一次过算出了所有顶点两两之间的最短距离）

有向无环图的应用

拓扑排序 Topological Sorting

1. AOV网：用顶点表示活动，边表示活动之间的制约关系
2. 目标：对一个有向（无环）AOV网构造拓扑序列
   a. 拓扑序列是一个顶点序列，若存在从vi到vj的路径，则序列中vi要在vj之前（描述顶点间的先后发生的关系）；拓扑序列不唯一
3. 核心：不断从图中选取并删除**入度为0**的顶点及以这个顶点为弧尾的弧
4. 逻辑：使用队列/栈（类似广度优先搜索中的方式）

关键路径 Critical Path

1. AOE网：用顶点表示事件，边表示活动

目标

核心：

查找 Search

1. 查找（搜索）的定义：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
2. 查找表：由同一类型的数据元素构成的集合
3. 关键字：数据元素中某个数据项的值
   a. 主关键字：能唯一表示一个元素的关键字
   b. 次关键字：可以识别多个元素的关键字

名称	代码	涨跌幅	最新价	涨跌额	买入/卖出价	成交量(手)
中国石油	sh601857	-0.47%	12.68	-0.06	12.68/12.69	391306
工商银行	sh601398	-2.31%	4.66	-0.11	4.65/4.66	442737
中国银行	sh601988	-1.43%	3.45	-0.05	3.45/3.46	194203
招商银行	sh600036	-1.63%	14.52	-0.24	14.52/14.54	385271
交通银行	sh601328	-1.29%	6.10	-0.08	6.09/6.10	347937
中信证券	sh600030	-2.69%	15.22	-0.42	15.22/15.23	597025
中国石化	sh600028	-1.16%	9.38	-0.11	9.37/9.38	538695

4. 查找方式
   a. **静态查找表**：只做查找操作的查找表
   b. **动态查找表**：查找过程同时插入表中不存在的数据元素，或删除某个已存在的数据元素

符号表

顺序查找 Sequential Search

1. 从第一个记录开始逐个比对
2. 对查找表没有要求，但效率低下O(n)
3. 优化：在查找方向的尽头设置**哨兵**，免去每次比较判断查找是否越界

有序查找

1. 前提：查找表中的元素依关键码有序
2. 二分查找、插值查找和斐波那契查找本质上是分隔点的选择不同
   a. 二分查找 Binary Search（折半查找）

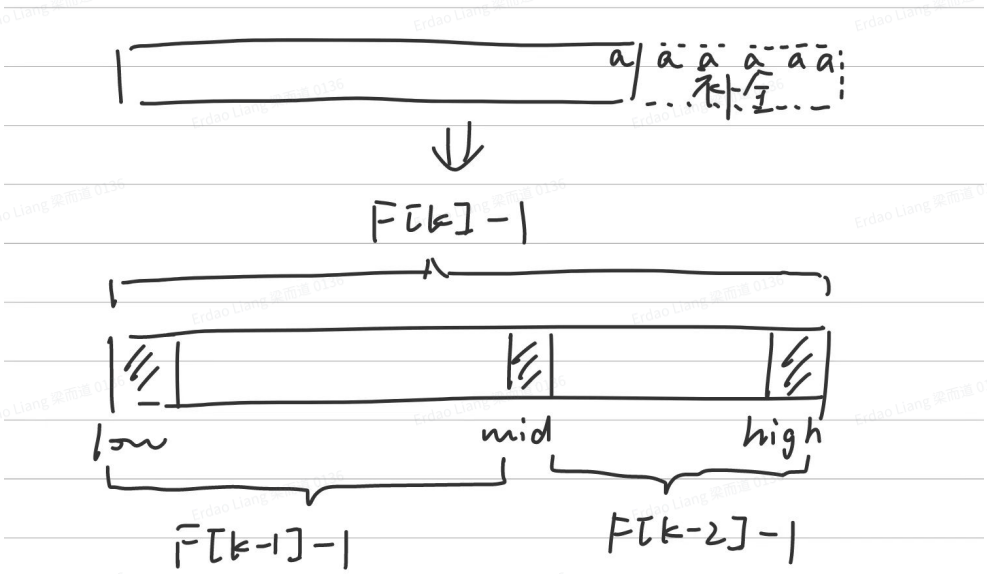
$$mid = \frac{low + high}{2} = low + 0.5(high - low)$$

   b. 插值查找 Interpolation Search
      i. $mid = low + \frac{key - a[low]}{a[high] - a[low]}(high - low)$
      ii. key是要查找的关键码
      iii. key越小（大），$\frac{key - a[low]}{a[high] - a[low]}$ 就越小（大），mid位置就越靠左（右），循环次数就有几率越少

   c. 斐波那契查找 Fibonacci Search
      i. $mid = low + F[k-1] - 1$
      ii. 先将序列的长度补全为F[k]-1（F[k]是一个斐波那契数），以防止索引值溢出
      iii. 在low和high围成的子序列中，mid左侧部分长度为F[k-1]-1，右侧长度为F[k-2]-1
      iv. 伪代码

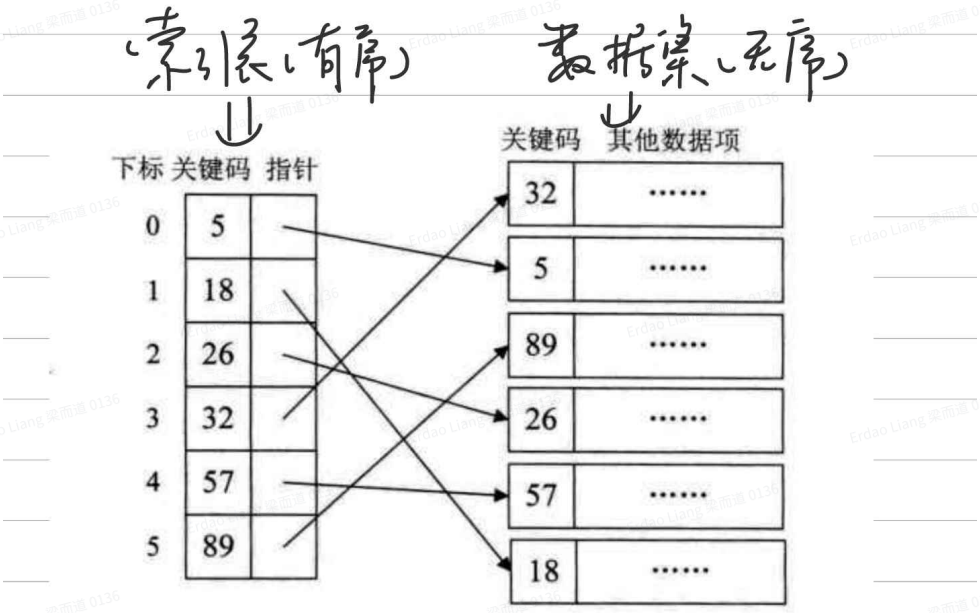
```
def fibonacciSearch():
    补全列表为F[k]-1
    high, low = 初始化
    while low < high:
        mid = low + F[k-1]-1
        if key < 序列[mid]
            调整high, low
            k = k-1
        elif key > 序列[mid]:
            调整high, low
            k = k-2
        elif ==:
```

```
        return
```



线性索引查找

1. 索引：建立一个索引表，索引表的每一项存储着数据集中一条记录的**关键码**和指向具体记录位置的**指针**，数据集无序，但索引表有序
2. 索引的逻辑：对整个数据集排序耗时耗力，但给索引表排序十分方便；查找时只需从索引表中查找关键码，然后由指针访问具体数据即可。



3. 三种索引方式：
   a. 稠密索引：将数据集的每个记录对应一个索引项
   b. 分块索引：对数据集分块，每一块建立一个索引项
      i. 块间有序，块内无序；
      ii. 先通过有序的索引表查找数据在哪一块→再在块内顺序查找数据

c. 倒排索引：索引表的每一项由次关键码+记录号表组成，
    i. 记录号表存储具有相同关键字的所有记录的记录号（主关键字，或指向位置的指针）
    ii. 逻辑：这种索引表结构为“查找具有相同次关键字的记录”的需求服务

二叉排序树 Binary Sort Tree

二叉排序

1. 递归定义：一棵二叉树，左子树所有结点的值均小于根结点的值，右子树所有结点的值均大于根结点的值（若不空）
2. 查找、插入与删除

平衡二叉树

多路查找树 Multi-way Search Tree

每个结点的孩子可以多于两个，或/且每个结点可存储多个元素

1. 2-3树：每个结点具有两个OR三个孩子
2. 2-3-4树：
3. B树：平衡的多路查找树。
    a. 结点最大的孩子数目称为B的阶
4. B+树：

散列表查找 Hash Table

1. 散列：在元素的**存储位置**（e.g.下标）和**关键字**之间建立一个确定的**映射f**，使得每个关键字key对应一个存储位置f(key)。f称为散列函数（哈希函数）。Location = f(key)，直接由函数输出位置而无需比较
2. 散列表：采用散列法将元素存储在的一块存储空间称为**散列表**（哈希表）
3. 散列函数的选择
    a. 直接定址：f(key) = a key + b

b. 除留余数：f(key) = key mod p (p<=总空间数)

c. 随机数法……

4. 处理散列表冲突

a. 冲突：key1 ≠ key2 但 f(key1) = f(key2)。key1和key2称同义词

b. 处理散列表冲突

i. 开放定址法：找**紧邻**的下一个散列地址 $f_i(key) = (f(key) + d_i) \ mod \ m$

线性探测法	二次探测法	随机探测法
$d_i = 1, 2, 3, 4, \cdots$	双向寻找空位置 $d_i = 1^2, -1^2, 2^2, -2^2, \cdots$	d_i = 随机数列

ii. 再散列函数法：准备多几个散列函数

iii. 链地址法：将关键词为同义词的记录存储在一个单链表中

iv. 公共溢出区法

5. 散列表查找性能分析

a. 散列函数是否均匀

b. 处理冲突的方法

c. 散列表的装填因子：要填入表中的记录个数 / 散列表长度

排序 Sort

1. 排序的定义：假设含有n个**元素**的序列{r1,r2,...,rn}，对应**关键字**分别为{k1,k2,...,kn}，需确定一种排列p1,p2,...,pn，使得对应**关键字**满足kp1<=kp2<=...<=kpn关系。
2. 排序的稳定性：关键字相同的元素顺序是否有可能在排序后交换前后顺序
3. 内排序和外排序

   内排序：待排序的所有元素被放置在内存中

   外排序：排序过程需要在内外存之间多次交换数据
4. 排序的性能衡量

a. 时间性能（关键字的比较+元素移动）

b. 辅助空间大小

c. 算法本身的复杂性

5. 内排序算法的种类

	交换排序	插入排序	选择排序	归并排序
简单算法	冒泡排序	直接插入排序	选择排序	
改进算法	快速排序	希尔排序	堆排序	归并排序

6. 排序算法对比

排序方法	平均时间复杂度	辅助空间	稳定性
冒泡排序	O(n^2)	O(1)	稳定：两两交换时，关键字相同的元素不会被左右交换
选择排序	O(n^2)	O(1)	稳定：在选择待排序列表的最小值元素时是从左到右查找的
直接插入排序	O(n^2)	O(1)	稳定：元素在插入左边的有序列表时，遇到相等值的元素就不会再移动
希尔排序	O(nlogn)~O(n^2)	O(1)	不稳定：元素跳跃式移动
堆排序	O(nlogn)	O(1)	不稳定：堆的定义导致的
归并排序	O(nlogn)	O(n+logn)	稳定：merge时可以优先挑选左子数组的元素
快速排序	O(nlogn)	O(logn)	不稳定：partition时元素跳跃式交换

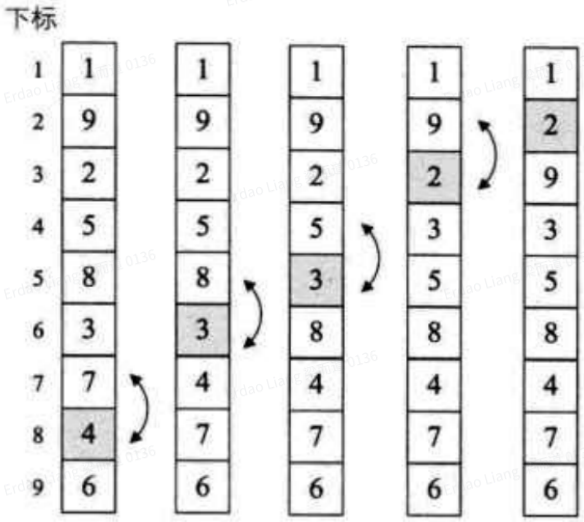
冒泡排序 Bubble Sort

1. 思路：两两比较**相邻元素**的关键字，如果反序则交换，直到没有反序的元素为止
2. 伪代码

```
def BubbleSort():
    for i = 1:n（循环操作n-1次，不针对哪个元素）：
        for j = n:i（从后往前遍历）：
            if front>back: exchange(front, back) #把已发现的最小元素抓到最前面
```

3. 逻辑
   a. 列表左侧是已经排好的有序的列表，右侧是待处理的

b. 每一轮迭代都是对待处理部分进行操作：从后往前（从右往左）两两比较，**每次都抓取已发现的最小元素**，将其带到有序部分的末端（也就是待处理部分的左端）



当i=2时将次小值2冒泡到第二位置

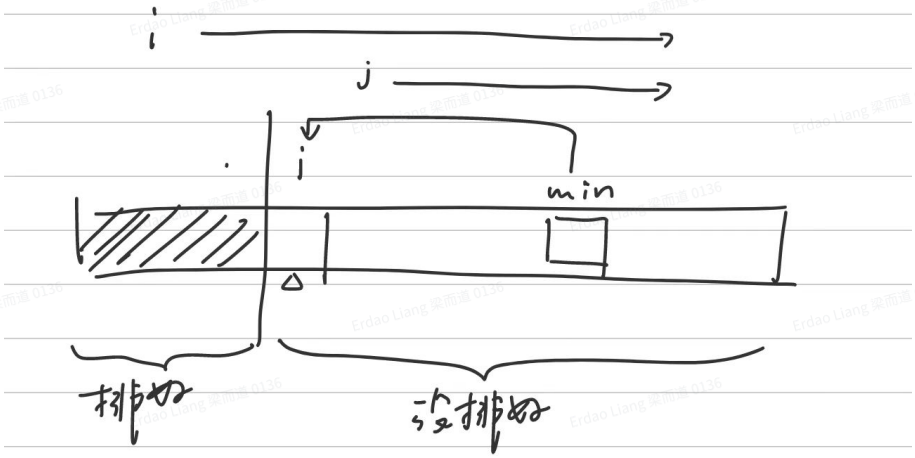
4. 复杂度分析：O(n^2)

选择排序 Selection Sort

1. 思路：在第i轮迭代中，通过n-i次关键字的比较，从n-i+1个元素中选出关键字最小的元素，并和第i个元素交换
2. 伪代码

```
def SelectSort():
    for i = 1:n-1 （循环n-1次，每次第i个元素要给找出的最小值腾出空位）:
        for:
            find_min(第i个后面的元素中)
        exchange(i,min) #将剩下未排序的元素中的最小那个排上来
```

3. 逻辑
   a. 列表左侧是已经排好的有序的列表，右侧是待处理的
   b. 每一轮迭代的任务是在右侧剩下的元素中找到最小的，并放到左侧有序部分的最右边（相当于对于每次的下一轮迭代，要找的都是次小的）；



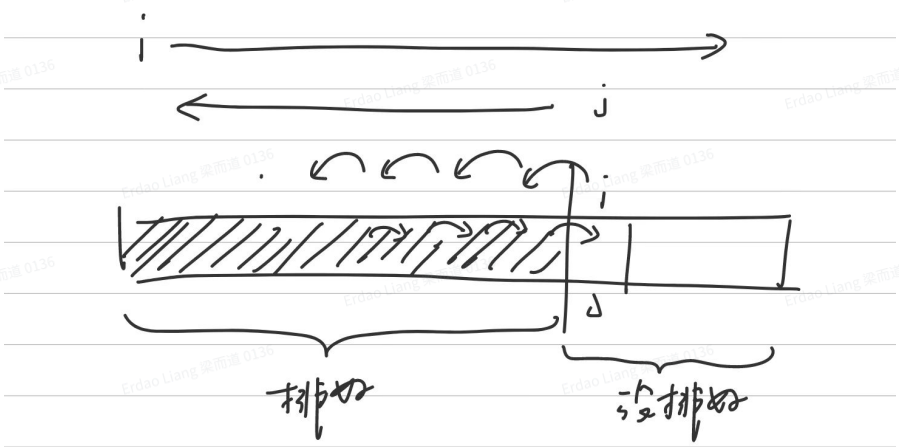
4. 复杂度分析：O(n^2)
    a. 第i轮要比较n-i次，总比较次数n(n-1)/2次
    b. 交换次数n-1次

直接插入排序 Straight Insertion Sort

1. 思路：将一个元素插入到已经排好序的列表，从而得到一个新的、元素数增1的有序表
2. 伪代码

```
def InsertionSort():
    for i = 2:n（循环n-1次）: #每轮循环中是第i个元素被执行插入排序
        for j=i:1 （从后往前遍历已排序的列表）:
            if front>back: exchange(front, back) #将原来的第i个元素插到正确的位置
```

3. 逻辑
    a. 列表左侧是已经排好的有序的列表，右侧是待处理的
    b. 每一轮迭代的任务是直接取待处理列表最靠近左侧有序列表的元素（最左边的一个），并将该元素通过逐次交换的方式向左移动到有序表里的合适位置



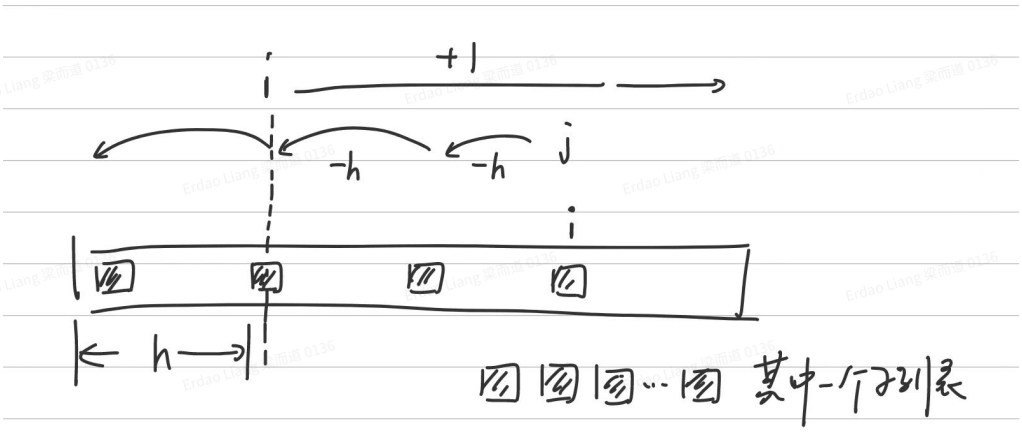
4. 复杂度分析：O(n^2)
    a. 平均比较和移动次数是n^2/4，比简单选择排序略好

希尔排序 Shell Sort

1. 理念：使列表向基本有序的方向发展
2. 思路：直接插入排序的改进版——将相距某个增量的元素组成一个子序列，每次在这些子序列内进行直接插入排序。通过进行局部插入排序，且间隔越来越小，最终完成整体排序
3. 伪代码

```
def shellSort():
    h = 初始化间隔
    while h >= 1: #进行多次间隔不断缩小的插入排序，直至间隔为1
        for:
            for:
                if front>back: exchange(front, back) #两层循环，对子列表做插入排序
        h = 缩小间隔
```

1. 逻辑
    a. 在第一轮外循环中，将增量h定为一个初始值（常为列表长度的1/4~1/3），这样原列表可被视为h个子列表，这些子列表的长度为n//h（或n//h + 1）。
    b. 对这h个子列表进行直接插入排序（但不是分开进行，而是同步进行。在内迭代中，从第h+1个元素遍历到第n个元素时，慢慢地完成了这h个子列表的插入排序）
    c. 减少增量h（通常取上次的1/4~1/3），列表又被分为了子列表（子列表个数更多，长度更短）；重复①②两步，直到**增量取h=1时也被进行过同样的操作**



2. 复杂度分析：希尔排序的时间复杂度没有明确值

堆排序 Heap Sort

优先队列

1. **堆**是具有这样性质的**完全二叉树**：大顶堆：每个结点的值都>=左右孩子结点的值（小顶堆相反）

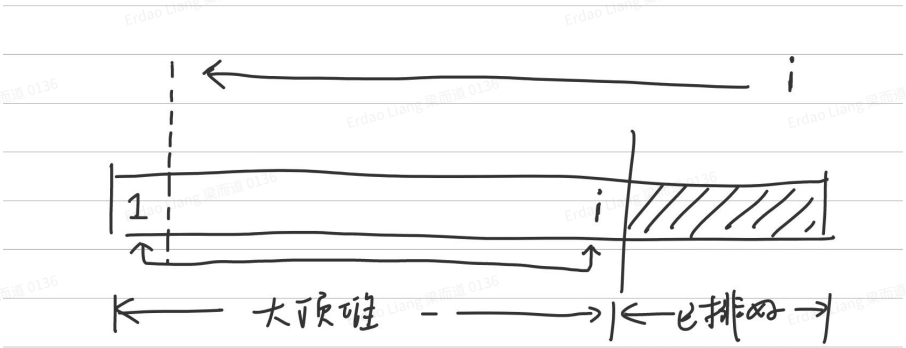
利用堆进行排序

1. 逻辑
   a. 总体逻辑：序列的最大值是堆顶的根结点。移走根结点（i.e.与堆数组的末尾交换），则堆数组末尾元素就是最大值；然后将剩余的n-1个元素**重新调整为堆**，移走根结点得到次大值；重复执行得到有序序列。
   b. 将n-1元素调整为新的堆（HeapAdjust）：

   此时除了被换上来的新堆顶元素以外，其余元素均满足堆的定义。基本思路是沿关键字较大的孩子结点向下筛选，把堆顶元素和下层的元素逐步交换到合适的位置

   c. 由无序序列构建一个堆：

   从下往上，从右往左，对每个**非叶节点**当作根结点，进行HeapAdjust调整



2. 伪代码

```
def headSort():
    for 倒序遍历所有非叶节点:
        heapAdjust() #构建大顶堆
    for i = n:2 （倒序遍历n-1次）:
        exchange(1,i)
        heapAdjust(剩下的1~i-1个元素) #将剩下的元素重新排成大顶堆
```

3. 复杂度分析：O(nlogn)

归并排序 Merge Sort

递归式归并排序

1. 思路：先递归地将数组分成两半排序，然后将结果归并起来（2路归并排序）
2. 伪代码

a. 主程序 - MergeSort

```
def mergeSort(序列, low, high):
    if 列表长度为1: return
    elif 列表长度为2: if >: exchange() return
    else
        mergeSort(左侧) #左侧子序列往里递归排序
        mergeSort(右侧) #右侧子序列往里递归排序
        merge(当前整个列表)
```

b. 核心 - Merge（处理整个序列的部分）

```
def merge():
    sorted = 定义一个辅助数组
    for i = low:high: #对
        #在左右两侧每次选一个，挑出来放
        if 左侧元素选完了: 选右侧的
        elif 右侧元素选完了: 选左侧的
        elif 左侧可选的元素更小: 选左侧的
        elif 右侧可选的元素更小: 选右侧的
    return sorted
```

3. 逻辑

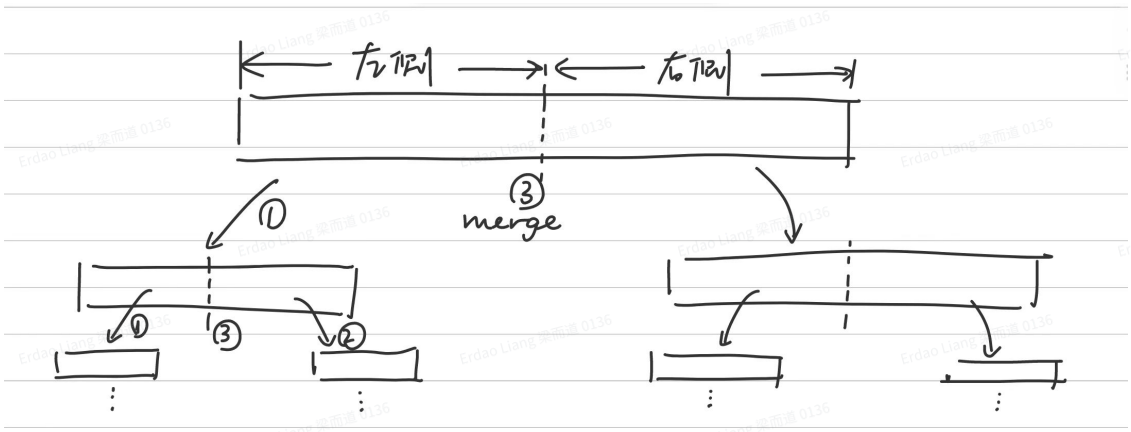
a. 在主程序MergeSort中，

i. 先将数组分成两半递归→使得两半各自已经有序→再处理整个数组→使得整个数组完全有序

ii. 递归时数组不断分半，直到遇到两种情况之一时不再递归：

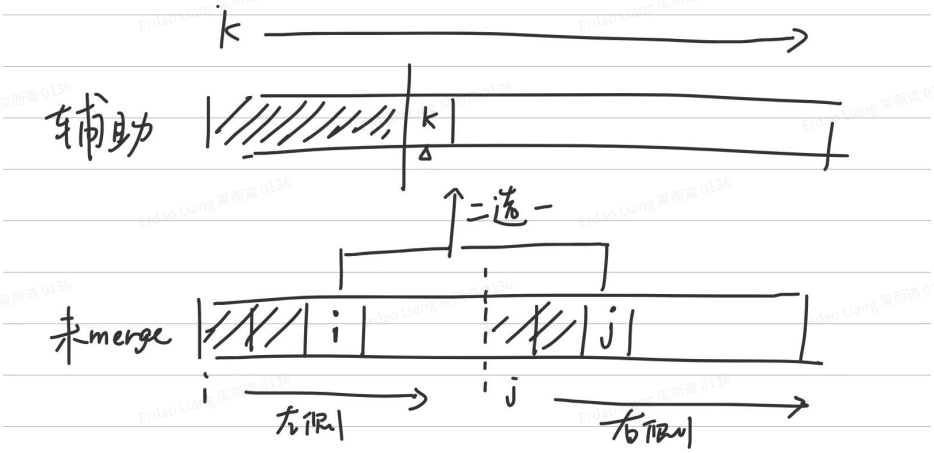
1）子数组仅有一个元素，什么也不做

2）子数组仅有两个元素，进行必要的交换



b. 在Merge中，

i. 从两边已经排好序的子数组中逐个从小到大地抓取元素，从左到右放入一个新的辅助数组里，直到全部抓取完（要考虑其中一侧提前被抓取完的情况）



4. 复杂度分析：

a. 时间复杂度O(nlogn)

b. 空间复杂度O(n+logn)

i. O(n)：要创建等长度的辅助数组

ii. O(logn)：递归时深度为log2n的栈空间

非递归实现归并排序

快速排序 Quick Sort

快速排序

1. 思路：将序列分为独立的两部分，其中一部分所有关键字均比另一部分的关键字小，然后递归地对两部分继续排序

2. 伪代码

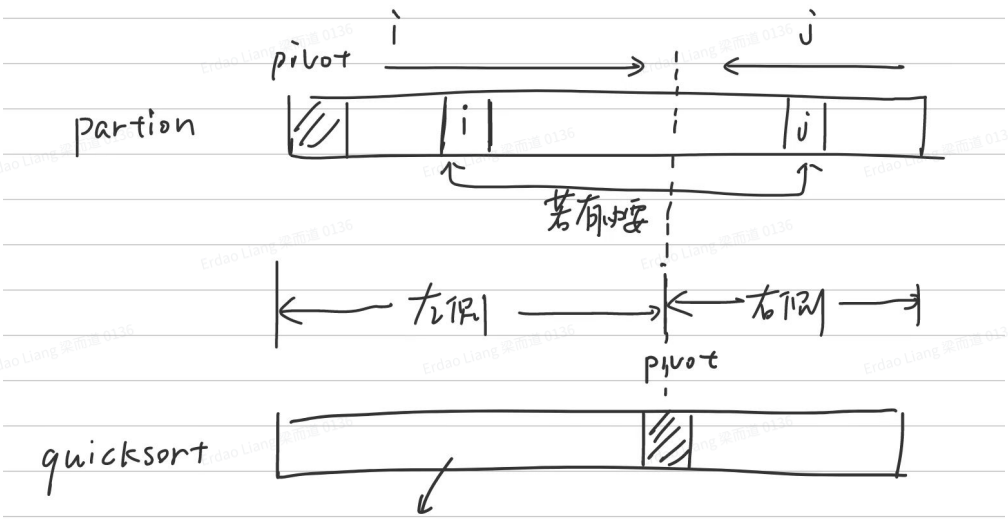
a. 主程序 - QuickSort

```
def quickSort(序列, low, high):
    if 序列长度大于1:
        分隔点=partition()
        quickSort(左侧)
        quickSort(右侧)
```

b. 核心 - Partition（处理整个序列的部分）

```
def partition(序列, low, high):
    pivot = 设定枢轴值
    while 从左向右遍历的low和从右向左遍历的high未交汇:
        if 序列[low] >pivot && 序列[high] < pivot:
            exchange()
        low++
        high--
```

3. 逻辑
    a. 在主程序QuickSort中，
        i. 先将整个数组分成两边→大致分出规律→再递归处理两半→处理完两半后整个数组完全有序
        ii. 递归时，直到子序列长度为0或1时不再递归
    b. 在Partition
        i. 先选取一个关键字（一般是第一个）（称为枢轴）
        ii. 其余部分列表从两侧同时开始往中间遍历，通过在必要时交换，实现一侧比枢轴值小，另一侧比枢轴值大（两侧未必等长）



4. 复杂度分析
    a. 时间复杂度O(nlogn)
    b. 空间复杂度O(logn)：log2n大小的栈空间

快速排序的优化

优化枢轴的选取

出现的问题：

快速排序和归并排序的对比

	归并排序	快速排序
什么时候进行递归	递归调用发生在处理整个数组之前	递归调用发生在处理整个数组之后
切分后子数组的长度	数组被等分（或长度差1）； 两个子数组相邻	切分方法取决于数组的内容； 两个子数组中间有一个元素隔开不参与后续递归

差分算法

算法设计技巧

分治算法 Divide and Conquer

贪心算法 Greedy Algorithm

动态规划 Dynamic Programming

- 背包问题
  - 0-1背包
  - 完全背包
- 树形DP https://zhuanlan.zhihu.com/p/633448505
- 数位DP

状压DP

回溯算法

高级数据结构

线段树

并查集

https://zhuanlan.zhihu.com/p/93647900