大O运行时

O(log n)：对数时间，算法包括二分查找

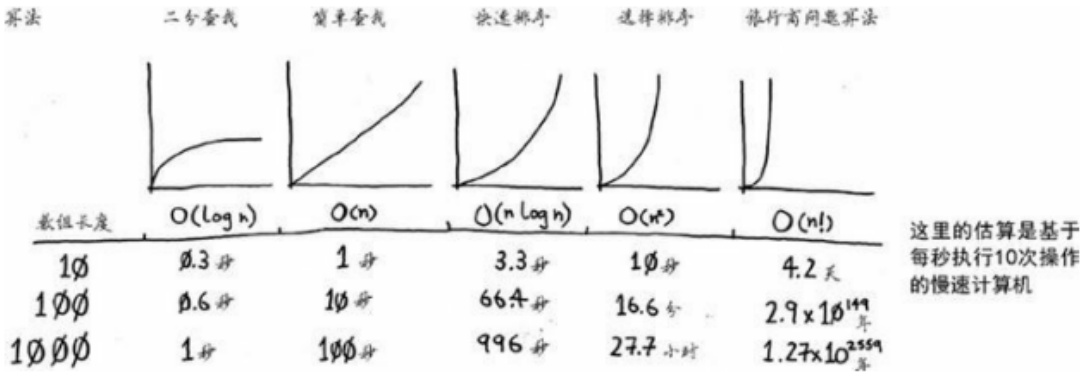
O(n)：线性时间，算法包括简单查找

O(n* log n)：算法包括快速排序

O(n^2)：算法包括选择排序

O(n!)：算法包括旅行商问题的解决方案

O(1)：常量时间不意味着马上，而是表示不管数据多大，从中获取一个元素所需的时间都是相同的，例如散列表

- 算法运行时间从其增速的角度度量，而不是以秒为单位
- 

数组和链表

数组的读取速度快，插入速度慢；（一起坐）

链表的读取速度慢，插入速度快。（分开坐）

	数组	链表
读取	O(1)	O(n)
插入	O(n)	O(1)
删除	O(n)	O(1)

- 计算机内部犹如一大堆抽屉
- 需要存储多个元素时，可使用数组或链表
- 数组的元素都在一起
- 链表的元素是分开的，其中每个元素都存储了下一个元素的地址
- 数组的读取速度快
- 链表的插入和删除速度快
- 在同一个数组中，所有元素的类型都必须相同（都为int、double等）

递归

递归只是令解决方案更清晰，并没有性能上的优势。在有些情况下，使用循环的性能更好。

编写递归必须有停止递归的条件。

递归条件：函数调用自己

基线条件：函数不再调用自己

- 尾递归

栈

先进后出

进栈：push（压入）

出栈：pop（弹出）

计算机内部的函数调用都要进入调用栈，每个函数调用都要占用一定的内存。

分而治之（D&C）

D&C算法是递归的。

- 步骤：
    1. 找出基线条件，且条件必尽可能简单
    2. 不断将问题分解（缩小规模），直到符合基线条件
- 提示：

  编写设计数组的递归函数时，基线条件通常是数组为空或只包含一个元素

散列表（hash table）

又被称为：散列映射、映射、字典、关联数组

- 散列函数必须满足的条件
    1. 必须是一致的（相同的输入映射到相同的索引）
    2. 不同的输入映射到不同的索引
- 散列表是包含额外逻辑的数据结构，它使用散列函数来确定元素的存储位置，同时也使用数组来存储数据，获取速度与数组一样快。同时，插入和删除速度也非常快。
- 数组和链表都被直接映射到内存。
- 应用场景：
    1. 查找：DNS解析、电话簿
    2. 防止重复：投票
    3. 用于缓存：Facebook网页缓存
- 冲突：

  理想情况是散列函数将键均匀地映射到散列表的不同位置，如果散列表存储的链表很长将严重性能（如果两个键映射到同一个位置，就在这个位置存储一个链表）

  避免冲突需要有：较低的填装因子、良好的散列函数。一旦填装因子超过0.7，就该调整散列表的长度。
- 性能：平均情况下，散列表的速度很快——O(1)；最糟糕情况下，所有操作都很慢——O(n)

队列（queue）

队列：先进先出(FIFO)

栈：后进先出(LIFO)

NP完全问题

以难解著称的问题，如旅行商问题和集合覆盖问题

- 识别可能是NP完全问题的方法

  1.元素较少时算法的运行速度非常快，但随着元素数量的增加，速度会变得非常慢

  2.涉及“所有组合”的问题通常是NP完全问题

  3.不能将问题分成小问题，必须考虑各种可能的情况

  4.如果问题涉及序列且难以解决

  5.如果问题涉及集合且难以解决

二分查找

- 二分查找的速度比简单查找快
- O(log n)比O(n)快。需要搜索的元素越多，前者比后者就快的越多

快速排序

速度取决于选择的基准值，每次随机地选择一个数组元素作为基准值。

- 步骤：

1. 选择基准值
2. 将数组分成两个子数组：小于基准值的元素和大于基准值的元素
3. 对这两个子数组进行快速排序

- 大O表示法

  平均情况下：O(n log n)

  最糟情况下：O(n^2)

广度优先搜索（BFS）

图由节点和边组成。一个节点可能于众多节点直接相连，这些节点被称为邻居。

有向图：图中边为箭头，箭头指定了关系的方向+

无向图：边不带箭头，关系是双向的。（也就是环）

广度优先搜索是一种用于图的查找算法。可以帮助解决两种问题：1.从节点A出发，有前往节点B的路径吗？2.从节点A出发，前往节点B的最短路径是哪条？

广度优先搜索需要按添加顺序查找，否则找到的就不是最短路径，因此搜索列表必须是队列。对于检查过得节点务必不再检查，否则可能导致无限循环。

广度优先搜索运行时间：O(V+E)。V为顶点，E为边数。

狄克斯特拉算法

找出最快的路径。即找出总权重最小的路径。只适用于有向无环图,且不包含负权边。

*贝尔曼-福德算法

加权图：带权重——>狄克斯特拉算法
非加权图：不带权重——>广度优先搜索

- 步骤：

  1. 找出“最便宜”的节点，即可在最短时间内达到的节点
  2. 更新该节点的邻居的开销，检查是否有前往它们的更短路径，有的话就更新其开销
  3. 重复这个过程，直到对图中的每个节点都这样做了

4. 计算最终路径

找出图中最便宜的节点，并确保没有到该节点的更便宜的路径

贪婪算法

每步都采取最优的做法，即每步都选择局部最优解，最终得到的就是全局最优解。

使用场景：寻找一个大致的而不是完美的解决方案。

优势：实现容易，结果与正确结果相当接近（近似算法）

运行时间：O(n^2)

广度优先搜索和狄克斯特拉算法都是贪婪算法

动态规划

先解决子问题，再逐步解决大问题。但仅当每个子问题都是离散的，即不依赖于其他子问题时，动态规划才有用。

动态规划可以在给定约束条件下找到最优解，如背包问题、最长公共子串、最长公共子序列

- 设计动态规划：
  - 每种动态规划解决方案都涉及网格
  - 单元格中的值通常就是要优化的值，如背包问题中商品的价值
  - 每个单元格都是一个子问题，因此要考虑如何将问题分成子问题
- 应用场景：
  - 生物学家根据最长公共序列来确定DNA链的相似性
  - git diff
  - 编辑距离算法，如拼写检查
  - Microsoft Word的断字功能，确定在什么地方断字来确保行长一样

K最近邻算法

KNN用于分类和回归，需要考虑最近的邻居。分类就是编组，回归就是预测结果。

特征抽取就是意味着将物品转换为一系列可比较的数字。

- 应用场景：
  - 创建推荐系统
  - OCR
  - 垃圾邮件过滤器

余弦相似度

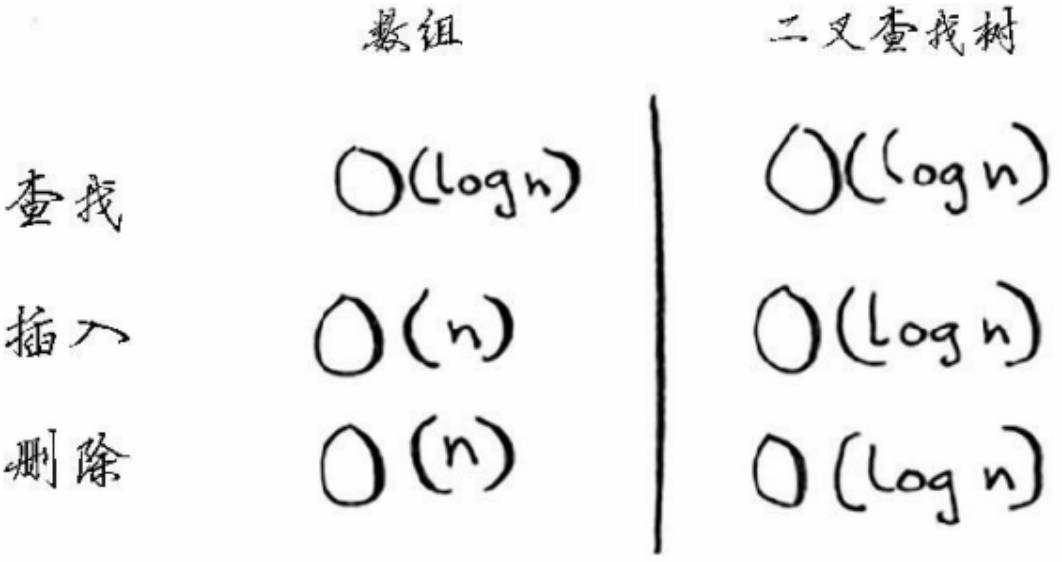
拓展内容

一.树

二叉查找树　（*B*树、红黑树、堆、伸展树）

平均运行时间：O(log n)

最糟糕运行时间：O(n)



缺点：不能随机访问

二.反向索引

一个散列表，将单词映射到包含它的页面。常用于创建搜索引擎

三.傅里叶变换

适用于处理信号，可以使用它来压缩音乐、地震预测、DNA分析等

四.并行算法

并行算法设计起来很难，要确保它们能够正确的工作并实现期望的速度提升也很难。

有一点确定的是，速度的提升并非线性的。原因：（1）并行性管理开销，即合并的开销；（2）负载均衡，即让内核一样忙

五.MapReduce

分布式算法，让算法在多台计算机上运行。

基于两个基本理念：映射（map）函数和归并（reduce）函数

六.布隆过滤器和HyperLogLog

布隆过滤器是一种概率型数据结构，它提供的答案有可能不对，但很可能正确。优点：占用的存储空间很少。

HyperLogLog近似地计算集合中不同的元素数，与布隆过滤器一样，不能给出准确答案，但也八九不离十，占用内存空间少。

七.SHA算法

安全散列算法（secure hash algorithm）函数。它是一个散列函数，生成一个散列值——一个较短的字符串。用于创建散列表的散列函数根据字符串生成数组索引。而SHA根据字符串生成另一个字符串。

可以用来比较两个文件是否相同；在不知道原始字符串的情况下检查密码

八.局部敏感的散列算法

如果对字符串做细微的修改，Simhash生成的散列值也只存在细微的差别，可以用来检查两项内容的相似程度。

九.Diffie-Hellman密钥交换

Diffie-Hellman解决了两个问题：（1）双方无须知道加密算法；（2）要破解加密的消息很难

使用两个密钥：公钥和私钥

十.线性规划

线性规划用于在给定约束条件下最大限度地改善指定的指标。线性规划使用Simplex算法

所有的图算法都可使用线性规划来实现

js排序代码

应用场景，数组关联排序

- 堆排序
- 归并排序

https://www.cnblogs.com/emojio/p/12408193.html

https://github.com/wy-ei/notebook/issues/34

https://segmentfault.com/a/1190000004375263

读后感小结

想要学习算法的起因是看到前端知识图谱，

学习算法对于编写代码的好处，

这本书深入浅出，非常适合入门。

重要的不是这些算法在前端领域当中是否真的用到，而是算法能提高自身对于问题的理解能力，以及解决问题的能力（思考能力、分析能力）