程序员必备：技术面试准备手册

2015-12-08 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：Tsiege
>
> 译文：伯乐在线 - 胡西瓜
>
> 链接：http://android.jobbole.com/82193/

这份清单，既是一份有助于对这些题目做深入研究的快速指南和参考，也算是计算机科学课程中不能忘记的基础知识总结，因此并不可能全面覆盖所有内容。它也可以作为 gist 在 Github 上公开，人人都可以编辑和补充。

一、数据结构基础

数组

定义

- 按顺序连续存储数据元素，通常索引从0开始
- 以集合论中的元组为基础
- 数组是最古老，最常用的数据结构

知识要点

- 索引最优；不利于查找、插入和删除（除非在数组最末进行）
- 最基础的是线性数组或一维数组
  - 数组长度固定，意味着声明数组时应指明长度
- 动态数组与一维数组类似，但为额外添加的元素预留了空间
  - 如果动态数组已满，则把每一元素复制到更大的数组中
- 类似网格或嵌套数组，二维数组有 x 和 y 索引

时间复杂度

- O(1)索引：一维数组：O(1)，动态数组：O(1)
- O(n)查找：一维数组：O(n)，动态数组：O(n)
- O(log n)最优查找：一维数组：O(log n)，动态数组：O(log n)

- O(n)插入：一维数组：n/a，动态数组：O(n)

链表

定义

- 结点存储数据，并指向下一结点
  - 最基础的结点包含一个数据和一个指针（指向另一结点）
- 链表靠结点中指向下一结点的指针连接成链

要点

- 为优化插入和删除而设计，但不利于索引和查找
- 双向链表包含指向前一结点的指针
- 循环链表是一种终端结点指针域指向头结点的简单链表
- 堆栈通常由链表实现，不过也可以利用数组实现
  - 堆栈是“后进先出”（LIFO）的数据结构
    - 由链表实现时，只有头结点处可以进行插入或删除操作
- 同样地，队列也可以通过链表或数组实现
  - 队列是“先进先出”（FIFO）的数据结构
    - 由双向链表实现时，只能在头部删除，在末端插入

时间复杂度

- O(n)索引：链表：O(n)
- O(n)查找：链表：O(n)
- Linked Lists: O(n)最优查找：链表：O(n)
- O(1)插入：链表：O(1)

哈希表或哈希图

定义

- 通过键值对进行储存
- 哈希函数接受一个关键字，并返回该关键字唯一对应的输出值
  - 这一过程称为散列（hashing），是输入与输出一一对应的概念
    - 哈希函数为该数据返回在内存中唯一的存储地址

要点

- 为查找、插入和删除而设计
- 哈希冲突是指哈希函数对两个不同的数据项产生了相同的输出值
  - 所有的哈希函数都存在这个问题
    - 用一个非常大的哈希表，可以有效缓解这一问题
    - 哈希表对于关联数组和数据库检索十分重要

时间复杂度

- O(1)索引：哈希表：O(1)
- O(1)查找：哈希表：O(1)
- O(1)插入：哈希表：O(1)

二叉树

定义

- 一种树形的数据结构，每一结点最多有两个子树
- 子结点又分为左子结点和右子结点

要点

- 为优化查找和排序而设计
- 退化树是一种不平衡的树，如果完全只有一边，其本质就是一个链表
- 相比于其他数据结构，二叉树较为容易实现
- 可用于实现二叉查找树
  - 二叉树利用可比较的键值来确定子结点的方向
  - 左子树有比双亲结点更小的键值
  - 右子树有比双亲结点更大的键值
  - 重复的结点可省略
  - 由于上述原因，二叉查找树通常被用作一种数据结构，而不是二叉树

时间复杂度

- 索引：二叉查找树：O(log n)
- 查找：二叉查找树：O(log n)
- 插入：二叉查找树：O(log n)

二、搜索基础

广度优先搜索

定义

- 一种在树（或图）中进行搜索的算法，从根结点开始，优先按照树的层次进行搜索
- 搜索同一层中的各结点，通常从左往右进行
- 进行搜索时，同时追踪当前层中结点的子结点
- 当前一层搜索完毕后，转入遍历下一层中最左边的结点
- 最下层最右端是最末结点（即该结点深度最大，且在当前层次的最右端）

要点

- 当树的宽度大于深度时，该搜索算法较优
- 进行树的遍历时，使用队列存储树的信息
  - 原因是：使用队列比深度优先搜索更为内存密集
  - 由于需要存储指针，队列需要占用更多内存

时间复杂度

- O(|E| + |V|)查找：广度优先搜索：O(|E| + |V|)
- E 是边的数目
- V 是顶点的数目

深度优先搜索

定义

- 一种在树（或图）中进行搜索的算法，从根结点开始，优先按照树的深度进行搜索
  - 从左边开始一直往下遍历树的结点，直到不能继续这一操作
  - 一旦到达某一分支的最末端，将返回上一结点并遍历该分支的右子结点，如果可以将从左往右遍历子结点
  - 当前这一分支搜索完毕后，转入根节点的右子结点，然后不断遍历左子节点，直到到达最底端
  - 最右的结点是最末结点（即所有祖先中最右的结点）

要点

- 当树的深度大于宽度时，该搜索算法较优

- 利用堆栈将结点压栈
  - 因为堆栈是“后进先出”的数据结构，所以无需跟踪结点的指针。与广度优先搜索相比，它对内存的要求不高。
  - 一旦不能向左继续遍历，则对栈进行操作

时间复杂度

- O(|E| + |V|)查找：深度优先搜索：O(|E| + |V|)
- E 是边的数目
- V 是结点的数目

广度优先搜索 VS. 深度优先搜索

- 这一问题最简单的回答就是，选取何种算法取决于树的大小和形态
  - 就宽度而言，较浅的树适用广度优先搜索
  - 就深度而言，较窄的树适用深度优先搜索

细微的区别

- 由于广度优先搜索（BFS）使用队列来存储结点的信息和它的子结点，所以需要用到的内存可能超过当前计算机可提供的内存（不过其实你不必担心这一点）
- 如果要在某一深度很大的树中使用深度优先搜索（DFS），其实在搜索中大可不必走完全部深度。可在 xkcd 上查看更多相关信息。
- 广度优先搜索趋于一种循环算法。
- 深度优先搜索趋于一种递归算法

三、高效排序基础

归并排序

定义

- 一种基于比较的排序算法
  - 将整个数据集划分成至多有两个数的分组
  - 依次比较每个数字，将最小的数移动到每对数的左边
  - 一旦所有的数对都完成排序，则开始比较最左两个数对中的最左元素，形成一个含有四个数的有序集合，其中最小数在最左边，最大数在最右边
  - 重复上述过程，直到归并成只有一个数据集

要点

- 这是最基础的排序算法之一
- 必须理解：首先将所有数据划分成尽可能小的集合，再作比较

时间复杂度

- O(n)最好的情况：归并排序：O(n)
- 平均情况：归并排序：O(n log n)
- 最坏的情况：归并排序：O(n log n)

快速排序

定义

- 一种基于比较的排序算法
  - 通过选取平均数将整个数据集划分成两部分，并把所有小于平均数的元素移动到平均数左边
  - 在左半部分重复上述操作，直到左边部分的排序完成后，对右边部分执行相同的操作
- 计算机体系结构支持快速排序过程

要点

- 尽管快速排序与许多其他排序算法有相同的时间复杂度（有时会更差），但通常比其他排序算法执行得更快，例如归并排序。
- 必须理解：不断通过平均数将数据集对半划分，直到所有的数据都完成排序

时间复杂度

- O(n)最好的情况：归并排序：O(n)
- O(n log n)平均情况：归并排序：O(n log n)
- 最坏的情况：归并排序：O(n2)

冒泡排序

定义

- 一种基于比较的排序算法

- 从左往右重复对数字进行两两比较，把较小的数移到左边
- 重复上述步骤，直到不再把元素左移

要点

- 尽管这一算法很容易实现，却是这三种排序方法中效率最低的
- 必须理解：每次向右移动一位，比较两个元素，并把较小的数左移

时间复杂度

- O(n)最好的情况：归并排序：O(n)
- O(n2)平均情况：归并排序： O(n2)
- O(n2)最坏的情况：归并排序： O(n2)

归并排序 VS. 快速排序

- 在实践中，快速排序执行速率更快
- 归并排序首先将集合划分成最小的分组，在对分组进行排序的同时，递增地对分组进行合并
- 快速排序不断地通过平均数划分集合，直到集合递归地有序

伯乐在线推荐阅读：

- 《视觉直观感受 7 种常用的排序算法》
- 《匈牙利 Sapientia 大学的 6 种排序算法舞蹈视频》
- 《视频：6 分钟演示 15 种排序算法》
- 《SORTING：可视化展示排序算法的原理，支持单步查看》
- 《VisuAlgo：通过动画学习算法和数据结构》

四、算法类型基础

递归算法

定义

- 在定义过程中调用其本身的算法
  - 递归事件：用于触发递归的条件语句
  - 基本事件：用于结束递归的条件语句

要点

- 堆栈级过深和栈溢出
  - 如果在递归算法中见到上述两种情况中的任一个，那就糟糕了
  - 那就意味着因为算法错误，或者问题规模太过庞大导致问题解决前 RAM 已耗尽，从而基本事件从未被触发
  - 必须理解：不论基本事件是否被触发，它在递归中都不可或缺
  - 通常用于深度优先搜索

迭代算法

定义

- 一种被重复调用有限次数的算法，每次调用都是一次迭代
  - 通常用于数据集中递增移动

要点

- 通常迭代的形式为循环、for、while和until语句
- 把迭代看作是在集合中依次遍历每个元素
- 通常用于数组的遍历

递归 VS. 迭代

由于递归和迭代可以相互实现，两者之间的区别很难清晰地界定。但必须知道：

- 通常递归的表意性更强，更易于实现
  - 迭代占用的内存更少
  - (i.e. Haskell)函数式语言趋向于使用递归（如 Haskell 语言）
- 命令式语言趋向于使用迭代（如 Ruby 语言）
- 点击 Stack Overflow post 了解更多详情
- 遍历数组的伪代码（这就是为什么使用迭代的原因）

```
Recursion | Iteration
----------------------------------|---------------------------------
recursive method (array, n) | iterative method (array)
if array[n] is not nil | for n from 0 to size of array
print array[n] | print(array[n])
```

```
recursive method(array, n+1) |
else |
exit loop
```

贪婪算法

定义

- 一种算法，在执行的同时只选择满足某一条件的信息
- 通常包含5个部分，摘自维基百科：
  - 候选集，从该集合中可得出解决方案
  - 选择函数，该函数选取要加入解决方案中的最优候选项
  - 可行性函数，该函数用于决策某一候选项是否有助于解决方案
  - 目标函数，该函数为解决方案或部分解赋值
  - 解决方案函数，该函数将指明完整的解决方案

要点

- 用于找到预定问题的最优解
- 通常用于只有少部分元素能满足预期结果的数据集合
- 通常贪婪算法可帮助一个算法降低时间 复杂度

伪代码：用贪婪算法找到数组中任意两个数字间的最大差值

```
greedy algorithm (array)
 var largest difference = 0
 var new difference = find next difference (array[n], array[n+1])
 largest difference = new difference if new difference is > largest difference
 repeat above two steps until all differences have been found
 return largest difference
```

这一算法无需比较所有数字两两之间的差值，省略了一次完整迭代。

阅读原文

微信扫一扫
关注该公众号