数据结构基础

数组

定义：

- 根据顺序索引（最常见的是基于0的索引）存储数据元素。
- 基于集合论的元组。
- 它们是最古老，最常用的数据结构之一。

您需要了解的内容：

- 最适合索引；搜索，插入和删除效果不佳（末尾除外）。
- **线性数组**或一维数组是最基本的。
  - 大小是静态的，意味着它们以固定大小声明。
- **动态数组**就像一维数组，但为其他元素保留了空间。
  - 如果动态阵列已满，则会将其内容复制到更大的阵列。
- **多维数组**嵌套的数组允许多个维度，例如通过x，y坐标提供二维空间表示的数组的数组。

时间复杂度：

- 索引：线性数组：O（1），动态数组：O（1）
- 搜索：线性数组：O（n），动态数组：O（n）
- 优化搜索：线性数组：O（log n），动态数组：O（log n）
- 插入：线性数组：n / a动态数组：O（n）

链表

定义：

- 与存储数据**的节点**指向其他节点。
  - 节点在最基本的情况下具有一个基准和一个参考（另一个节点）。
  - 链表*链*通过指向一个节点对另一节点的参考节点一起。

您需要了解的内容：

- 专为优化插入和删除而设计，索引和搜索速度较慢。
- **双链列表**具有也引用前一个节点的节点。
- **循环链表**是简单链表，其**尾部**（最后一个节点）引用**头**（第一个节点）。
- **Stack**通常由链表实现，但也可以由数组组成。
  - 堆栈是后进**先出**（LIFO）数据结构。
  - 通过使头部成为唯一的插入和移除位置，可以使用链表制成。
- **队列**也可以用链表或数组来实现。
  - 队列是一个**先入先出**（FIFO）的数据结构。

- 用双向链表制成，该链表仅从头部移开而在尾部增加。

时间复杂度：

- 索引：链接列表：O（n）
- 搜索：链接列表：O（n）
- 优化搜索：链接列表：O（n）
- 插入：链表：O（1）

哈希表或哈希图

定义：

- 存储具有键值对的数据。
- **哈希函数**接受一个键，并返回仅对该特定键唯一的输出。
  - 这称为**散列**，即输入和输出与地图信息具有一一对应的概念。
  - 哈希函数返回该数据在内存中的唯一地址。

您需要了解的内容：

- 旨在优化搜索，插入和删除。
- **哈希冲突**是指哈希函数为两个不同的输入返回相同的输出。
  - 所有哈希函数都有此问题。
  - 通常通过使哈希表非常大来解决此问题。
- 哈希对于关联数组和数据库索引很重要。

时间复杂度：

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

二叉树

定义：

- 是一个像树一样的数据结构，其中每个节点最多有两个孩子。
  - 有一个左右子节点。

您需要了解的内容：

- 旨在优化搜索和排序。
- 甲**退化树**是一个不平衡的树，其中，如果完全片面是一个基本上是一个链表。
- 它们比其他数据结构更容易实现。
- 用于制作**二叉搜索树**。
  - 使用可比较键来指定孩子的方向的二叉树。
  - 左子节点的键小于其父节点。

- 右子项的键大于其父节点。
- 不能有重复的节点。
- 由于上述原因，它比二叉树更有可能用作数据结构。

时间复杂度：

- 索引：二进制搜索树：O（log n）
- 搜索：二进制搜索树：O（log n）
- 插入：二进制搜索树：O（log n）

搜索基础

广度优先搜索

定义：

- 通过从根开始首先搜索树的级别来搜索树（或图）的算法。
  - 它找到每个级别相同的节点，最经常从左到右移动。
  - 在执行此操作时，它将跟踪当前级别上节点的子节点。
  - 检查完一个级别后，它将移至下一级别的最左侧节点。
  - 最右下角的节点最后被评估（最深且距离其级别最远的节点）。

您需要了解的内容：

- 最适合搜索宽于深的树。
- 当队列遍历树时，使用队列来存储有关树的信息。
  - 因为它使用队列，所以比**深度优先搜索**要占用更多的内存。
  - 队列使用更多的内存，因为它需要存储指针

时间复杂度：

- 搜索：广度优先搜索：O（V + E）
- E是边数
- V是顶点数

深度优先搜索

定义：

- 一种算法，该算法通过从根开始首先搜索树的深度来搜索树（或图）。
  - 它沿着一棵树向左走，直到不能走远为止。
  - 一旦到达分支的末尾，它将遍历该分支的右子节点，如果可能的话，再返回右子节点。
  - 检查完分支后，它移到根的右侧节点，然后尝试在所有子节点上向左移动，直到到达底部。
  - 最右边的节点最后被评估（所有祖先右边的节点）。

您需要了解的内容：

- 最适合搜索比宽更深的树。
- 使用堆栈将节点压入。
  - 由于堆栈是LIFO，因此它不需要跟踪节点指针，因此与广度优先搜索相比，其内存密集度较低。
  - 一旦无法继续前进，它将开始评估堆栈。

时间复杂度：

- 搜索：深度优先搜索：O（| E | + | V |）
- E是边数
- V是顶点数

广度优先搜索与 深度优先搜索

- 这个问题的简单答案是，它取决于树的大小和形状。
  - 对于宽而浅的树木，请使用广度优先搜索
  - 对于深而窄的树木，请使用深度优先搜索

细微差别：

- 因为BFS使用队列来存储有关节点及其子节点的信息，所以它使用的内存可能比计算机上可用的内存更多。（但是您可能不必为此担心。）
- 如果在非常深的树上使用DFS，则可能会不必要地深入搜索。有关更多信息，请参见xkcd。
- 广度优先搜索往往是一种循环算法。
- 深度优先搜索往往是一种递归算法。

高效分类基础

合并排序

定义：

- 基于比较的排序算法
  - 将整个数据集分成最多两个的组。
  - 一次比较每个数字，将最小的数字向左移动。
  - 对所有对进行排序后，便会比较两个最左对中的最左元素，从而创建一个四个一组的排序组，其中最小的数字在左边，最大的数字在右边。
  - 重复此过程，直到只有一套为止。

您需要了解的内容：

- 这是最基本的排序算法之一。
- 知道它将所有数据分成尽可能小的集合，然后进行比较。

时间复杂度：

- 最佳情况排序：合并排序：O（n）
- 平均个案排序：合并排序：O（n log n）
- 最坏情况排序：合并排序：O（n log n）

快速排序

定义：

- 基于比较的排序算法
  - 通过选择中间元素并将所有较小的元素放在元素的左边，将较大的元素放在右边，将整个数据集分成两半。
  - 它在左侧重复此过程，直到只比较左侧排序的两个元素。
  - 左侧完成排序后，它将在右侧执行相同的操作。
- 计算机体系结构支持快速排序过程。

您需要了解的内容：

- 尽管它的Big O与许多其他排序算法相同（或在某些情况下更糟），但实际上它通常比许多其他排序算法（例如合并排序）更快。
- 知道它将所有数据按平均值连续减半，直到对所有信息进行排序。

时间复杂度：

- 最佳情况排序：快速排序：O（n）
- 平均案例排序：快速排序：O（n log n）
- 最坏情况排序：快速排序：O（n ^ 2）

气泡排序

定义：

- 基于比较的排序算法
  - 比较每个对联，从左到右进行迭代，将较小的元素向左移动。
  - 重复此过程，直到不再将元素移到左侧。

您需要了解的内容：

- 尽管实现起来非常简单，但在这三种排序方法中效率最低。
- 知道它向右移动一个空间，一次比较两个元素，然后向左移动较小的空间。

时间复杂度：

- 最佳情况排序：冒泡排序：O（n）
- 平均案例排序：冒泡排序：O（n ^ 2）
- 最坏情况排序：冒泡排序：O（n ^ 2）

合并排序与 快速排序

- Quicksort在实践中可能会更快。
- 合并排序将集合立即划分为最小的组，然后在对分组进行排序时逐步重建增量。
- Quicksort连续将集合除以平均值，直到对集合进行递归排序。

基本算法类型

递归算法

定义：

- 在其定义中调用自身的算法。
  - **递归情况**是用于触发递归的条件语句。
  - **基本情况**是用于中断递归的条件语句。

您需要了解的内容：

- **堆栈级别太深，堆栈溢出**。
  - 如果您从递归算法中看到了这两种方法，那么您就搞砸了。
  - 这意味着您的基本案例永远不会被触发，因为它是有缺陷的，或者问题是如此之大，以至于耗尽了已分配的内存。
  - 知道您是否会遇到基本情况是正确使用递归的必要条件。
  - 经常在深度优先搜索中使用

迭代算法

定义：

- 一种算法，该算法被重复调用但次数有限，每次都是一次迭代。
  - 通常用于增量移动数据集。

您需要了解的内容：

- 通常，您将迭代视为for，while和直到语句的循环。
- 将迭代视为一次移动一个集合。
- 通常用于遍历数组。

递归与 迭代

- 递归和迭代之间的区别可能会造成混淆，因为两者都可以用来实现彼此。但是要知道
  - 递归通常更具表现力，更易于实现。
  - 迭代使用较少的内存。
- **功能语言**倾向于使用递归。（即Haskell）
- **命令式语言**倾向于使用迭代。（即Ruby）
- 请查看此Stack Overflow帖子以获取更多信息。

遍历数组的伪代码（这就是为此使用迭代的原因）

```
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：

- 一种在执行时仅选择满足特定条件的信息的算法。
- 一般的五个组成部分，取自维基百科：
  - 从中创建解决方案的候选集。
  - 选择功能，选择要添加到解决方案中的最佳候选者。
  - 可行性函数，用于确定候选人是否可以用于解决方案。
  - 目标函数，为解决方案或部分解决方案分配值。
  - 解决方案功能，它将指示我们何时发现了完整的解决方案。

您需要了解的内容：

- 用于找到给定问题的权宜之计，尽管不是最优的。
- 通常用于仅评估信息的一小部分就能达到所需结果的数据集。
- 贪婪算法通常可以帮助减少算法的BigO。

贪婪算法的伪代码，用于查找数组中任何两个数字的最大差值。

```
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
```

这种算法永远不需要将所有差异相互比较，从而节省了整个迭代时间。