笔试准备的一些随记

2024.4.16 Complied by zxy

前言：笔试涉及了很多编程题之外的数据结构的基础知识和算法思路考察，学习起来是很收获的，下面是一些不全面的整理（因为手写整理了一部分没有加进来。）

Part One：时间复杂度&空间复杂度

1、时间复杂度：

对于累加算法，计算总和所用的赋值语句的数目就是一个很好的基本计算单位。在 `sumOfN` 函数中，赋值语句数是 1（`theSum = 0`）加上 n（`theSum = theSum + i` 的运行次数）。可以将其定义成函数 T，令 $T(n)=1+n$。参数 n 常被称作**问题规模**，可以将函数解读为“当问题规模为 n 时，解决问题所需的时间是 $T(n)$，即需要 $1+n$ 步”。

在前面给出的累加函数中，用累加次数定义问题规模是合理的。这样一来，就可以说处理前 100 000 个整数的问题规模比处理前 1000 个整数的大。鉴于此，前者花的时间要比后者长。接下来的目标就是揭示算法的执行时间如何随问题规模而变化。

计算机科学家将分析向前推进了一步。精确的步骤数并没有 $T(n)$ 函数中起决定性作用的部分重要。也就是说，随着问题规模的增长，$T(n)$ 函数的某一部分会比其余部分增长得更快。最后比较的其实就是这一起决定性作用的部分。数量级函数描述的就是，当 n 增长时，$T(n)$ 增长最快的部分。**数量级**（order of magnitude）常被称作**大 O 记法**（O 指 order），记作 $O(f(n))$。它提供了步骤数的一个有用的近似方法。$f(n)$ 函数为 $T(n)$ 函数中起决定性作用的部分提供了简单的表示。

对于 $T(n)=1+n$，随着 n 越来越大，常数 1 对最终结果的影响越来越小。如果要给出 $T(n)$ 的近似值，可以舍去 1，直接说执行时间是 $O(n)$。注意，1 对于 $T(n)$ 来说是重要的。但是随着 n 的增长，没有 1 也不会太影响近似值。

再举个例子，假设某算法的步骤数是 $T(n)=5n^2+27n+1005$。当 n 很小时，比如说 1 或 2，常数 1005 看起来是这个函数中起决定性作用的部分。然而，随着 n 增长，n^2 变得更重要。实际上，当 n 很大时，另两项的作用对于最终结果来说就不显著了，因此可以忽略这两项，只关注 $5n^2$。另外，当 n 变大时，系数 5 的作用也不显著了。因此可以说，函数 $T(n)$ 的数量级是 $f(n)=n^2$，或者直接说是 $O(n^2)$。

累加的例子没有体现的一点是，算法的性能有时不仅依赖于问题规模，还依赖于数据值。对于这种算法，要用**最坏情况**、**最好情况**和**普通情况**来描述性能。最坏情况指的是某一个数据集会让算法的性能极差；另一个数据集可能会让同一个算法的性能极好（最好情况）。大部分情况下，算法的性能介于两个极端之间（普通情况）。计算机科学家要理解这些区别，以免被某个特例误导。

2、空间复杂度：

空间复杂度是指一个算法在运行过程中临时占用存储空间大小的度量。它描述了算法使用的内存随输入数据量增加的变化情况。具体来说，空间复杂度不仅包括了输入数据本身所占的空间，还包括了算法在执行过程中需要的额外空间，如临时变量、递归调用栈等。

空间复杂度的度量：

空间复杂度通常用大O符号来表示。常见的空间复杂度有：

- **O(1)**: 常数空间，只需要固定的额外空间，与输入规模无关。
- **O(n)**: 线性空间，需要的额外空间与输入规模成正比。
- **O(n^2)**: 二次方空间，需要的额外空间与输入规模的平方成正比。
- **O(log n)**: 对数空间，需要的额外空间与输入规模的对数成正比。

Part Two：Sorting

1、stability

排序算法的稳定性是指在排序过程中，如果两个元素的关键字相同，排序前后的相对位置不变。换句话说，如果在排序过程中两个相等的元素的位置发生了变化，这种排序算法就是不稳定的；如果位置没有变化，则这种排序算法是稳定的。

稳定排序的意义

稳定排序在某些情况下非常有用。例如，当数据包含多个字段，并且需要先按一个字段排序，再按另一个字段排序时，稳定排序可以保证前一次排序的结果不会被后一次排序打乱。

举例说明

假设有一组数据，包括人的姓名和年龄，需要按年龄排序：

```
输入数据：
[("Alice", 25), ("Bob", 20), ("Charlie", 25), ("David", 20)]
```

排序时，只考虑年龄：

- 使用稳定排序（如归并排序或冒泡排序），结果可能是：

  ```
  [("Bob", 20), ("David", 20), ("Alice", 25), ("Charlie", 25)]
  ```

  这里，年龄相同的"Bob"和"David"、"Alice"和"Charlie"的相对位置与原始数据中的顺序一致。

- 使用不稳定排序（如快速排序或选择排序），结果可能是：

  ```
  [("David", 20), ("Bob", 20), ("Charlie", 25), ("Alice", 25)]
  ```

  这里，年龄相同的"Bob"和"David"、"Alice"和"Charlie"的相对位置可能发生了变化。

常见排序算法的稳定性

- **稳定排序算法：**
  - 冒泡排序（Bubble Sort）
  - 归并排序（Merge Sort）
  - 插入排序（Insertion Sort）
  - 基数排序（Radix Sort）
- **不稳定排序算法：**
  - 选择排序（Selection Sort）
  - 快速排序（Quick Sort）
  - 希尔排序（Shell Sort）
  - 堆排序（Heap Sort）

理解排序算法的稳定性对选择合适的排序算法非常重要，特别是在需要多级排序或对稳定性有特定要求的应用场景下。

2、排序时间复杂度

！！一个非常好的能看各种排序算法动态实现的网站：https://visualgo.net/en/sorting

每种排序算法的最差时间复杂度和最优时间复杂度取决于其具体的工作原理和处理不同输入情况下的性能。以下是每种排序算法的最差时间复杂度和最优时间复杂度的解释：

1. 插入排序（Insertion Sort）

- **最差时间复杂度**：O(n^2)
  - 原因：当输入数组是逆序时，每次插入都需要在已经排序的部分中进行n次比较和交换。
- **最优时间复杂度**：O(n)
  - 原因：当输入数组已经有序时，每次插入只需一次比较。

2. 选择排序（Selection Sort）

- **最差时间复杂度**：O(n^2)
  - 原因：无论数组的初始顺序如何，每次选择最小（或最大）元素都需要扫描整个数组的未排序部分。
- **最优时间复杂度**：O(n^2)
  - 原因：选择排序在任何情况下都需要扫描整个未排序部分，因此时间复杂度总是O(n^2)。

3. 冒泡排序（Bubble Sort）

- **最差时间复杂度**：O(n^2)
  - 原因：当输入数组是逆序时，每次比较和交换都需要遍历数组的每个元素。
- **最优时间复杂度**：O(n)
  - 原因：当输入数组已经有序时，只需进行n次比较，无需交换。

4. 希尔排序（Shell Sort）

- **最差时间复杂度**：O(n^2)（取决于增量序列）
  - 原因：最差时间复杂度取决于所选择的增量序列。对于一些增量序列，最差情况下可能达到O(n^2)。
- **最优时间复杂度**：O(n log n)
  - 原因：最佳情况下，使用合适的增量序列（如Sedgewick增量序列）可以达到O(n log n)。

5. 归并排序（Merge Sort）

- **最差时间复杂度**：O(n log n)
  - 原因：归并排序总是将数组分成两部分递归处理，并在合并阶段进行线性时间的合并操作。
- **最优时间复杂度**：O(n log n)
  - 原因：归并排序的分治策略在任何情况下都需要分割和合并，时间复杂度稳定为O(n log n)。

6. 快速排序（Quick Sort）

- **最差时间复杂度**：O(n^2)
  - 原因：当每次选择的基准元素都是最小或最大元素时，递归深度达到n，导致时间复杂度为O(n^2)。
- **最优时间复杂度**：O(n log n)
  - 原因：当每次选择的基准元素将数组均匀分割时，递归深度为log n，每层的比较次数为n。

7. 堆排序（Heap Sort）

- **最差时间复杂度**：O(n log n)
  - 原因：堆排序构建堆的过程需要O(n)时间，之后的每次堆调整需要O(log n)时间，总共n个元素。
- **最优时间复杂度**：O(n log n)
  - 原因：无论输入如何，堆排序的构建和调整过程都需要稳定的O(n log n)时间。

8. 计数排序（Counting Sort）

- **最差时间复杂度**：O(n + k)
  - 原因：计数排序的时间复杂度取决于输入数组的大小n和计数数组的范围k。
- **最优时间复杂度**：O(n + k)
  - 原因：计数排序的时间复杂度在任何情况下都是O(n + k)。

9. 桶排序（Bucket Sort）

- **最差时间复杂度**：O(n^2)
  - 原因：当所有元素都被分配到同一个桶中时，退化为插入排序，时间复杂度为O(n^2)。
- **最优时间复杂度**：O(n + k)
  - 原因：当元素均匀分布到各个桶中，每个桶进行线性时间的排序，总时间复杂度为O(n + k)。

Part Three：存储结构&逻辑结构

当然可以！理解逻辑结构和存储结构是学习数据结构和算法的基础。以下是详细的解释以及相关的例子：

1、逻辑结构：

逻辑结构描述了数据元素之间的逻辑关系，即数据元素如何相互连接和组织。这是一个抽象的概念，不涉及具体实现。逻辑结构主要分为以下几类：

1. **集合结构**：数据元素之间没有特定的关系，例如集合。
2. **线性结构**：数据元素之间存在一对一的关系，例如线性表、栈、队列。
3. **树形结构**：数据元素之间存在一对多的关系，例如树。
4. **图形结构**：数据元素之间存在多对多的关系，例如图。

2、存储结构：

存储结构描述了数据在计算机内存中的实际存储方式。不同的存储结构会直接影响数据处理的效率。存储结构主要分为以下几类：

1. **顺序存储结构**：数据元素按顺序存放在连续的存储位置上。例如，数组。
2. **链式存储结构**：数据元素通过指针或引用链接在一起。例如，链表。
3. **索引存储结构**：在存储数据元素的同时，附加一些索引信息以加快查找速度。例如，B树。
4. **散列存储结构**：通过散列函数将数据元素映射到存储位置上。例如，哈希表。

3、例子：

线性表

- **逻辑结构**：线性表是一种线性结构，数据元素之间存在一对一的关系。
- **存储结构**：
  - **顺序存储**：线性表可以用数组实现，数据元素按顺序存放在连续的内存位置上。
  - **链式存储**：线性表也可以用链表实现，数据元素通过指针链接在一起。

例子：

```
线性表（逻辑结构）：[1, 2, 3, 4, 5]

顺序存储（数组）：[1, 2, 3, 4, 5]  （内存地址连续）

链式存储（链表）：
1 -> 2 -> 3 -> 4 -> 5
```

树

- **逻辑结构**：树是一种树形结构，数据元素之间存在层次关系。
- **存储结构**：
  - **顺序存储**：树可以用数组实现，通常用于完全二叉树。
  - **链式存储**：树也可以用链表实现，每个节点包含数据和指向子节点的指针。

图

- **逻辑结构**：图是一种图形结构，数据元素之间可以存在多对多的关系。
- **存储结构**：
  - **邻接矩阵**：用二维数组表示图，行和列表示顶点，元素表示边的权重。
  - **邻接表**：每个顶点有一个链表，链表包含所有与该顶点相邻的顶点。

Part Four：散列表——一种存储结构

散列表是一种用于存储键值对的数据结构，其中每个键经过哈希函数计算后都会映射到数组中的一个位置，然后将对应的值存储在该位置上。这个哈希函数通常会将键转换成一个数字，然后使用取余等方法将其映射到数组的索引上。

散列表的优点在于它提供了快速的查找、插入和删除操作。通过哈希函数，我们可以快速计算出键对应的位置，从而直接访问到值，这样的时间复杂度通常是O(1)。但是，散列表也有一些缺点，最主要的是可能会发生哈希冲突，即不同的键经过哈希函数计算后映射到了数组的同一个位置。

为了解决碰撞，散列表通常使用以下两种方法之一：

1. **开放寻址法（Open Addressing）：** 当发生碰撞时，使用一定的方法在散列表中寻找下一个空槽来存放关键字。常见的方法包括线性探测（Linear Probing）、二次探测（Quadratic Probing）、双重散列（Double Hashing）等。
   - **线性探测：**当发生碰撞时，依次查找下一个位置，直到找到一个空槽或者遍历整个散列表。
   - **二次探测：**当发生碰撞时，使用一个固定的二次探测序列来寻找下一个位置，直到找到一个空槽或者遍历整个散列表。
   - **双重散列：**当发生碰撞时，使用第二个散列函数来计算一个步长，并以此步长依次查找下一个位置，直到找到一个空槽或者遍历整个散列表。
2. **链表法（Chaining）：** 当发生碰撞时，将多个关键字存储在同一个位置上的链表中。这样，每个槽可以存储多个关键字。

无论是开放寻址法还是链表法，都可以有效地解决碰撞问题。选择哪种方法取决于实际情况，包括散列表的大小、负载因子、操作的复杂性等。

后记：这个很不全，是初期学习的一些概念笔记，在后期复习计算题时树和图的一些知识有些放在了数算学习随记里面，有些手写在了打印出来的资料上。算是数算学习随记的一个概念补充吧。