《[数据结构与算法之美](https://time.geekbang.org/column/126)》是极客时间上的一个算法学习系列，在学习之后特在此做记录和总结。

数据结构和算法是相辅相成的，数据结构是为算法服务的，算法要作用在特定的数据结构之上。 \* 从广义上讲，数据结构就是指一组数据的存储结构。算法就是操作数据的一组方法。 \* 从狭义上讲，是指某些著名的数据结构和算法，比如队列、栈、堆、二分查找、动态规划等。

## 一、时间复杂度分析

  大 O 复杂度表示法，实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，所以，也叫作渐进时间复杂度（asymptotic time complexity），简称时间复杂度。

  分析一段代码的时间复杂度有三个比较实用的方法： 1. 只关注循环执行次数最多的一段代码。例如代码被执行了 n 次，所以总的时间复杂度就是 O(n)。 2. 加法法则：总复杂度等于量级最大的那段代码的复杂度。例如两段代码的复杂度分别为O(n) 和 O(n^2)，那么整段代码的时间复杂度就为 O(n^2)。 3. 乘法法则：嵌套代码的复杂度等于嵌套内外代码复杂度的乘积。假设 T1(n) = O(n)，T2(n) = O(n^2)，则 T1(n) \* T2(n) = O(n^3)

**1）多项式量级**

  非多项式量级只有两个：O(2^n) 和 O(n!)。接下来主要来看几种常见的多项式时间复杂度。 1. O(1)，并不是指只执行了一行代码，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是Ο(1)。 2. O(logn)、O(nlogn)，假设代码的执行次数是2x=n，那么x=x=log2n，因此这段代码的时间复杂度就是 O(log2^n)。 3. O(m+n)、O(m\*n)，因为无法事先评估 m 和 n 谁的量级大，所以在表示复杂度的时候，就不能简单地利用加法法则，省略掉其中一个。

**2）最好和最坏**

  最好情况时间复杂度就是，在最理想的情况下，执行这段代码的时间复杂度。例如在最理想的情况下，要查找的变量 x 正好是数组的第一个元素。

  最坏情况时间复杂度就是，在最糟糕的情况下，执行这段代码的时间复杂度。如果数组中没有要查找的变量 x，那么需要把整个数组都遍历一遍才行。

## 二、数据结构

**1）数组**

  数组（Array）是一种线性表数据结构。它用一组连续的内存空间，来存储一组具有相同类型的数据。 1. 线性表（Linear List）就是数据排成像一条线一样的结构。每个线性表上的数据最多只有前和后两个方向。比如数组、链表、队列、栈等。 2. 在非线性表中，数据之间并不是简单的前后关系。比如二叉树、堆、图等。

  在面试的时候，常常会问数组和链表的区别，很多人都回答说，“链表适合插入、删除，时间复杂度 O(1)；数组适合查找，查找时间复杂度为 O(1)”。

  实际上，这种表述是不准确的。数组是适合查找操作，但是查找的时间复杂度并不为 O(1)。即便是排好序的数组，你用二分查找，时间复杂度也是 O(logn)。

  所以，正确的表述应该是，数组支持随机访问，根据下标随机访问的时间复杂度为 O(1)。

**2）链表**

  数组和链表的区别如下： 1. 数组需要一块连续的内存空间来存储，对内存的要求比较高。链表恰恰相反，它并不需要一块连续的内存空间，它通过“指针”将一组零散的内存块串联起来使用。 2. 链表删除一个数据是非常快速的，只需考虑相邻结点的指针改变，所以对应的时间复杂度是 O(1)。而链表随机访问的性能没有数组好，需要 O(n) 的时间复杂度。

  三种最常见的链表结构，它们分别是：单链表、双向链表和循环链表。 1. 单链表通过指针将一组零散的内存块串联在一起。其中，把内存块称为链表的“结点”。为了将所有的结点串起来，每个链表的结点除了存储数据之外，还需要记录链上的下一个结点的地址。把这个记录下个结点地址的指针叫作后继指针 next。 2. 循环链表是一种特殊的单链表。其优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用循环链表。 3. 双向链表支持两个方向，每个结点不止有一个后继指针 next 指向后面的结点，还有一个前驱指针 prev 指向前面的结点。

  检查链表代码是否正确的边界条件有这样几个： 1. 如果链表为空时，代码是否能正常工作？ 2. 如果链表只包含一个结点时，代码是否能正常工作？ 3. 如果链表只包含两个结点时，代码是否能正常工作？ 4. 代码逻辑在处理头结点和尾结点的时候，是否能正常工作？

  5 个常见的链表操作。 1. 单链表反转 2. 链表中环的检测 3. 两个有序的链表合并 4. 删除链表倒数第 n 个结点 5. 求链表的中间结点

**3）栈**

  后进者先出，先进者后出，这就是典型的“栈”结构。从栈的操作特性上来看，栈是一种“操作受限”的线性表，只允许在一端插入和删除数据。

  栈既可以用数组来实现（顺序栈），也可以用链表来实现（链式栈）。

  比较经典的一个应用场景就是函数调用栈。另一个常见的应用场景，编译器如何利用栈来实现表达式求值。

  除了用栈来实现表达式求值，还可以借助栈来检查表达式中的括号是否匹配。比如，{[] ()[{}]}或[{()}([])]等都为合法格式，而{[}()]或[({)]为不合法的格式。

**4）队列**

  先进者先出，这就是典型的“队列”。队列跟栈非常相似，支持的操作也很有限

  对于栈来说，只需要一个栈顶指针就可以了。但是队列需要两个指针：一个是 head 指针，指向队头；一个是 tail 指针，指向队尾。

  作为一种非常基础的数据结构，队列的应用也非常广泛。 1. 循环队列长得像一个环。原本数组是有头有尾的，是一条直线。现在把首尾相连，扳成了一个环。 2. 阻塞队列是在队列为空的时候，从队头取数据会被阻塞。如果队列已经满了，那么插入数据的操作就会被阻塞，直到队列中有空闲位置后再插入数据。 3. 并发队列最简单直接的实现方式是直接在 enqueue()、dequeue() 方法上加锁，但是锁粒度大并发度会比较低，同一时刻仅允许一个存或者取操作。

**5）跳表**

  跳表（Skip List）是一种各方面性能都比较优秀的动态数据结构，可以支持快速地插入、删除、查找操作，可以替代红黑树（Red-black Tree）。

  在链表上加一层索引之后，查找一个结点需要遍历的结点个数减少了，也就是说查找效率提高了。这种链表加多级索引的结构，就是跳表。

  在一个单链表中查询某个数据的时间复杂度是 O(n)，在跳表中查询任意数据的时间复杂度就是 O(logn)，空间复杂度是 O(n)。

**6）散列表**

  散列表（Hash Table）平时也叫“哈希表”或者“Hash 表”。散列表用的是数组支持的按照下标随机访问数据的特性，时间复杂度是 O(1) ，所以散列表其实就是数组的一种扩展，由数组演化而来。

  散列函数定义成 hash(key)，其中 key 表示元素的键值，hash(key) 的值表示经过散列函数计算得到的散列值。其设计的基本要求： 1. 散列函数计算得到的散列值是一个非负整数； 2. 如果 key1 = key2，那 hash(key1) == hash(key2)； 3. 如果 key1 ≠ key2，那 hash(key1) ≠ hash(key2)。

  常用的散列冲突解决方法有两类，开放寻址法（open addressing）和链表法（chaining）。 1. 开放寻址法的核心思想是，如果出现了散列冲突，就重新探测一个空闲位置，将其插入。 2. 在散列表中，每个“桶（bucket）”或者“槽（slot）”会对应一条链表，所有散列值相同的元素我们都放到相同槽位对应的链表中。

**7）二叉树**

  树（Tree）有三个比较相似的概念：高度（Height）、深度（Depth）、层（Level）。

  除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫做满二叉树。

  子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大，这种二叉树叫做完全二叉树。

  二叉树的遍历有三种，前序遍历、中序遍历和后序遍历。遍历的时间复杂度是 O(n)。 1. 前序遍历是指，对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。 2. 中序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。 3. 后序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。

  二叉查找树（Binary Search Tree，BST）要求，在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值。

  中序遍历二叉查找树，可以输出有序的数据序列，时间复杂度是 O(n)，非常高效。

**8）红黑树**

  平衡二叉树的严格定义是这样的：二叉树中任意一个节点的左右子树的高度相差不能大于 1。

  红黑树的英文是“Red-Black Tree”，简称 R-B Tree。它是一种不严格的平衡二叉查找树。

  红黑树中的节点，一类被标记为黑色，一类被标记为红色。除此之外，一棵红黑树还需要满足这样几个要求： 1. 根节点是黑色的； 2. 每个叶子节点都是黑色的空节点（NIL），也就是说，叶子节点不存储数据； 3. 任何相邻的节点都不能同时为红色，也就是说，红色节点是被黑色节点隔开的； 4. 每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点；

**9）堆**

  堆（Heap）是一种特殊的树。 1. 堆是一个完全二叉树； 2. 堆中每一个节点的值都必须大于等于或小于等于其子树中每个节点的值，前者叫大顶堆，后者叫小顶堆。

  完全二叉树比较适合用数组来存储。数组中下标为 i 的节点的左子节点，就是下标为 i*2 的节点，右子节点就是下标为 i*2+1 的节点，父节点就是下标为 2/i​ 的节点。

  将堆进行调整，让其重新满足堆的特性，这个过程叫做堆化（heapify）。堆化非常简单，就是顺着节点所在的路径，向上或者向下，对比，然后交换。

  堆这种数据结构几个非常重要的应用：优先级队列、求 Top K 和求中位数。

**10）图**

  图（Graph）和树比起来，这是一种更加复杂的非线性表结构。

  树中的元素称为节点，图中的元素就叫做顶点（vertex）。图中的一个顶点可以与任意其他顶点建立连接关系。把这种建立的关系叫做边（edge）。度（degree）就是跟顶点相连接的边的条数。

  把这种边有方向的图叫做“有向图”。以此类推，把边没有方向的图就叫做“无向图”。在有向图中，把度分为入度（In-degree）和出度（Out-degree）。

**11）Trie树**

  Trie 树，也叫“字典树”。它是一个树形结构，一种专门处理字符串匹配的数据结构，可解决在一组字符串集合中快速查找某个字符串的问题。

  Trie 树的本质，就是利用字符串之间的公共前缀，将重复的前缀合并在一起。

  每次查询时，如果要查询的字符串长度是 k，那只需要比对大约 k 个节点，就能完成查询操作。

  跟原本那组字符串的长度和个数没有任何关系。所以说，构建好 Trie 树后，在其中查找字符串的时间复杂度是 O(k)，k 表示要查找的字符串的长度。

## 三、算法

**1）递归**

  只要同时满足以下三个条件，就可以用递归来解决。 1. 一个问题的解可以分解为几个子问题的解。 2. 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样。 3. 存在递归终止条件。

  写递归代码的关键就是找到如何将大问题分解为小问题的规律，并且基于此写出递推公式，然后再推敲终止条件，最后将递推公式和终止条件翻译成代码。

  对于递归代码，这种试图想清楚整个递和归过程的做法，实际上是进入了一个思维误区。很多时候，我们理解起来比较吃力，主要原因就是自己给自己制造了这种理解障碍。

  因此，编写递归代码的关键是，只要遇到递归，我们就把它抽象成一个递推公式，不用想一层层的调用关系，不要试图用人脑去分解递归的每个步骤。

**2）排序**

  分析一个排序算法，要从几个方面入手： 1. 排序算法的执行效率，衡量方面： \* 最好情况、最坏情况、平均情况时间复杂度。 \* 时间复杂度的系数、常数 、低阶。 \* 比较次数和交换（或移动）次数。 2. 排序算法的内存消耗，针对排序算法的空间复杂度，我们还引入了一个新的概念，原地排序（Sorted in place），特指空间复杂度是 O(1) 的排序算法。 3. 排序算法的稳定性，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

  常见的排序算法有： 1. 冒泡排序（Bubble Sort），只会操作相邻的两个数据。每次冒泡操作都会对相邻的两个元素进行比较，看是否满足大小关系要求。如果不满足就让它俩互换。 2. 插入排序（Insertion Sort），取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。 3. 选择排序（Selection Sort），思路有点类似插入排序，也分已排序区间和未排序区间，但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾。 4. 归并排序（Merge Sort），如果要排序一个数组，先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。 5. 快速排序（Quick Sort），利用的也是分治思想，如果要排序数组中下标从 p 到 r 之间的一组数据，选择 p 到 r 之间的任意一个数据作为 pivot（分区点）。遍历 p 到 r 之间的数据，将小于 pivot 的放到左边，将大于 pivot 的放到右边，将 pivot 放到中间。归并排序和快速排序的区别： \* 归并排序的处理过程是由下到上的，先处理子问题，然后再合并。归并排序虽然是稳定的、时间复杂度为 O(nlogn) 的排序算法，但是它是非原地排序算法。 \* 快排正好相反，它的处理过程是由上到下的，先分区，然后再处理子问题。快速排序通过设计巧妙的原地分区函数，可以实现原地排序。 6. 桶排序（Bucket Sort），将要排序的数据分到几个有序的桶里，每个桶里的数据再单独进行排序。 7. 计数排序（Counting Sort），桶排序的一种特殊情况。当要排序的 n 个数据，所处的范围并不大的时候，比如最大值是 k，就可以把数据划分成 k 个桶。每个桶内的数据值都是相同的，省掉了桶内排序的时间。 8. 堆排序，堆排序的过程大致分解成两个大的步骤，首先将数组原地建成一个堆，建堆结束之后，数组中的数据已经是按照大顶堆的特性来组织的。数组中的第一个元素就是堆顶，也就是最大的元素。把它跟最后一个元素交换，那最大元素就放到了下标为 n 的位置。

**3）二分查找**

  二分查找（Binary Search）针对的是一个有序的数据集合，查找思想有点类似分治思想。每次都通过跟区间的中间元素对比，将待查找的区间缩小为之前的一半，直到找到要查找的元素，或者区间被缩小为 0。

  容易出错的 3 个地方 1. 循环退出条件，注意是 low<=high，而不是 low。 2. mid 的取值，如果 low 和 high 比较大的话，两者之和就有可能会溢出。改成 low + ((high - low) >> 1) 即可。 3. low 和 high 的更新，low=mid+1，high=mid-1。注意这里的 +1 和 -1。

  二分查找的时间复杂度是 O(logn)，查找数据的效率非常高。不过，并不是什么情况下都可以用二分查找，它的应用场景是有很大局限性的。 1. 二分查找依赖的是顺序表结构，简单点说就是数组。 2. 二分查找针对的是有序数据。 3. 数据量太小不适合二分查找。 4. 数据量太大也不适合二分查找。

  变形问题： 1. 查找第一个值等于给定值的元素。 2. 查找最后一个值等于给定值的元素。 3. 查找第一个大于等于给定值的元素。 4. 查找最后一个小于等于给定值的元素。

**4）哈希算法**

  哈希算法历史悠久，业界著名的哈希算法也有很多，比如 MD5、SHA 等。

  将任意长度的二进制值串映射为固定长度的二进制值串，这个映射的规则就是哈希算法，而通过原始数据映射之后得到的二进制值串就是哈希值。

  设计一个优秀的哈希算法需要满足的几点要求： 1. 从哈希值不能反向推导出原始数据（所以哈希算法也叫单向哈希算法）； 2. 对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同； 3. 散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小； 4. 哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值。

  哈希算法的应用选了最常见的七个，分别是安全加密、唯一标识、数据校验、散列函数、负载均衡、数据分片、分布式存储。

**5）字符串匹配**

1. BF 算法是Brute Force的简称，中文叫作暴力匹配算法，也叫朴素匹配算法。在字符串 A 中查找字符串 B，那字符串 A 就是主串，字符串 B 就是模式串。把主串的长度记作 n，模式串的长度记作 m。
2. RK 算法全称叫 Rabin-Karp 算法，每次检查主串与子串是否匹配，需要依次比对每个字符。通过哈希算法对主串中的 n-m+1 个子串分别求哈希值，然后逐个与模式串的哈希值比较大小。
3. BM 算法全称叫 Boyer-Moore 算法， 其性能是著名的KMP 算法的 3 到 4 倍。在模式串与主串匹配的过程中，当模式串和主串某个字符不匹配的时候，能够跳过一些肯定不会匹配的情况，将模式串往后多滑动几位。
4. KMP 算法的核心思想和 BM 算法非常相近。假设主串是 a，模式串是 b。在模式串与主串匹配的过程中，当遇到不可匹配的字符的时候，希望找到一些规律，可以将模式串往后多滑动几位，跳过那些肯定不会匹配的情况。
5. 多模式串匹配算法只需要扫描一遍主串，就能在主串中一次性查找多个模式串是否存在，从而大大提高匹配效率。对敏感词字典进行预处理，构建成 Trie 树结构。经典的多模式串匹配算法：AC 自动机。

**6）贪心算法**

  贪心算法（greedy algorithm）有很多经典的应用，比如霍夫曼编码（Huffman Coding）、Prim 和 Kruskal 最小生成树算法、还有 Dijkstra 单源最短路径算法。

  贪心算法解决问题的步骤： 1. 第一步，当看到这类问题的时候，首先要联想到贪心算法：针对一组数据，定义了限制值和期望值，希望从中选出几个数据，在满足限制值的情况下，期望值最大。 2. 第二步，尝试看下这个问题是否可以用贪心算法解决：每次选择当前情况下，在对限制值同等贡献量的情况下，对期望值贡献最大的数据。 3. 第三步，举几个例子看下贪心算法产生的结果是否是最优的。

  实际上，用贪心算法解决问题的思路，并不总能给出最优解。贪心算法的题目包括分糖果、钱币找零、区间覆盖等。

**7）分治算法**

  分治算法（divide and conquer）的核心思想其实就是四个字，分而治之 ，也就是将原问题划分成 n 个规模较小，并且结构与原问题相似的子问题，递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

  分治算法的递归实现中，每一层递归都会涉及这样三个操作： 1. 分解：将原问题分解成一系列子问题； 2. 解决：递归地求解各个子问题，若子问题足够小，则直接求解； 3. 合并：将子问题的结果合并成原问题。

  分治算法能解决的问题，一般需要满足下面这几个条件： 1. 原问题与分解成的小问题具有相同的模式； 2. 原问题分解成的子问题可以独立求解，子问题之间没有相关性，这一点是分治算法跟动态规划的明显区别； 3. 具有分解终止条件，也就是说，当问题足够小时，可以直接求解； 4. 可以将子问题合并成原问题，而这个合并操作的复杂度不能太高，否则就起不到减小算法总体复杂度的效果了。

**8）回溯算法**

  回溯的处理思想，有点类似枚举搜索。枚举所有的解，找到满足期望的解。为了有规律地枚举所有可能的解，避免遗漏和重复，把问题求解的过程分为多个阶段。

  每个阶段，都会面对一个岔路口，先随意选一条路走，当发现这条路走不通的时候（不符合期望的解），就回退到上一个岔路口，另选一种走法继续走。

  回溯算法的应用包括深度优先搜索、八皇后、0-1 背包问题、图的着色、旅行商问题、数独、全排列、正则表达式匹配等。

**9）动态规划**

  动态规划（Dynamic Programming）比较适合用来求解最优问题，比如求最大值、最小值等等。

  它的主要学习难点跟递归类似，那就是，求解问题的过程不太符合人类常规的思维方式。

  把问题分解为多个阶段，每个阶段对应一个决策。记录每一个阶段可达的状态集合（去掉重复的），然后通过当前阶段的状态集合，来推导下一个阶段的状态集合，动态地往前推进。

  一个模型”指的是动态规划适合解决的问题的模型。这个模型定义为“多阶段决策最优解模型”。“三个特征”分别是最优子结构、无后效性和重复子问题。 1. 最优子结构指的是，问题的最优解包含子问题的最优解。反过来说就是，可以通过子问题的最优解，推导出问题的最优解。 2. 无后效性有两层含义，第一层含义是，在推导后面阶段的状态的时候，只关心前面阶段的状态值，不关心这个状态是怎么一步一步推导出来的。第二层含义是，某阶段状态一旦确定，就不受之后阶段的决策影响。 3. 重复子问题。如果用一句话概括一下，那就是，不同的决策序列，到达某个相同的阶段时，可能会产生重复的状态。

解决动态规划问题，一般有两种思路。 1. 状态转移表法。状态表一般都是二维的，所以你可以把它想象成二维数组。其中，每个状态包含三个变量，行、列、数组值。根据决策的先后过程，从前往后递推关系，分阶段填充状态表中的每个状态。 2. 状态转移方程法。根据最优子结构，写出递归公式，也就是所谓的状态转移方程。