数据结构和算法速查速记手册

有很多读者反馈，学了数据结构和算法之后，不用过不几天就忘光了。当然，我之前在公众号中也反复强调过，对于有些复杂数据结构和算法，原理和实现容易忘记是很正常的，而且，忘记也并不代表白学。

不过，对于一些基础的数据结构和算法，你应该牢记它们的原理和实现，因为它们既是学习更加高级的数据结构和算法的基础，也是LeetCode刷题的基础。关于哪些是基础的数据结构和算法，哪些是高级的数据结构和算法，我之前在公众号中已经讲过，这里就不再重复罗列了。

为了方便你复习，我整理了这份数据结构和算法速查速记手册，罗列每种数据结构和算法的核心知识点，在你复习的时候，只需要看这份手册即可，不用抱着一个大部头从头读到尾，大大节省你的时间。而且，它也可以作为一份速查手册，当你刷题时，对某个数据结构的某个知识点有所遗忘，可以通过这个手册迅速查找回忆。除此之外，在手册中，每一个数据结构和算法都明确指明了掌握程度，比如熟练编写某某的代码实现等，让你知道怎么才算真正掌握。

01

—

数组、链表、栈、队列

01. 数组

概念与特性	1. 数组是线性表，用一组连续的内存空间存储一组具有相同类型的数据； 2. 最大的特性是支持按照下标O(1)时间复杂度内快速访问数组元素； 3. 一维数组寻址公式：a[i]_addr = base_addr + i * data_type_size；
操作与复杂度	1. 随机访问时间复杂度是O(1)； 2. 在数组中间任意位置插入数据的时间复杂度是O(n)； 3. 删除数组中任意位置数据的时间复杂度是O(n)；
应用场景	数组是其他数据结构和算法的实现基础，比如栈、队列、堆、二分查找等
其他知识点	1. 数组需要连续的内存空间，对内存的要求较高； 2. 数组中的数据连续存储，对CPU缓存友好； 3. 大部分编程语言中，数组下标都是从0开始编号； 4. 大部分编程语言中，都提供了容器类型以支持动态数组（动态扩容）； 5. 编程语言中的数组类型并不等同于数据结构中讲的数组；
掌握程度	能够自己动手实现一个动态数组类。

02. 链表

概念与特性	1. 链表是线性表，不需要连续的内存空间来存储元素，通过指针将串联每个链表中的结点； 2. 常用的链表结构有：单链表、双向链表、循环链表，其中双向链表因为支持在O(1)时间复杂度内找到前驱结点，在实际开发中最常用；
操作与复杂度	1. 跟数组对比，查找第i个元素的时间复杂度是O(n)； 2. 在已知前驱结点的情况下，单链表中插入数据的时间复杂度是O(1)； 3. 在已知前驱结点的情况下，单链表中删除数据的时间复杂度是O(1)； 注意：上面的插入、删除操作，都是针对已知前驱结点的情况，如果未知前驱结点，在单链表中插入、删除数据时间复杂度是O(n)，而在双向链表中插入、删除数据的时间复杂度仍然是O(1)。这也是双向链表比单链表更常用的主要原因。
应用场景	链表是其他数据结构和算法的实现基础，比如跳表、散列表等；
其他知识点	1. 链表中的数据不连续存储，对CPU缓存不友好； 2. 在实际的编程中，可定义有头链表，也可以定义无头链表；有头链表指的是链表中的头结点不存储数据；
掌握程度	1. 熟练实现单链表、双向链表、循环链表的定义和操作；

	2. 熟练实现经典的链表题目，比如反转链表、链表求中间结点、合并有序链表、删除链表倒数第K个结点等；

03. 栈

概念与特性	1. 栈是一种操作受限的线性表，只能在一端插入删除数据； 2. 栈的最大特性是先进后出；
操作与复杂度	1. 入栈操作，在栈顶放入数据，时间复杂度是O(1)； 2. 出栈操作，从栈顶取出数据，时间复杂度是O(1)；
应用场景	1. 函数调用栈； 2. 编译器利用栈来实现表达式求值； 3. 浏览器中的前进后退功能的实现也会用到栈；
其他知识点	1. 栈既可以用数组来实现，也可以用链表来实现； 2. 基于数组实现的支持动态扩容的栈的插入操作的均摊时间复杂度是O(1)；
掌握程度	1. 熟练利用数组实现一个栈； 2. 熟练利用链表实现一个栈； 3. 掌握基于数组实现的支持动态扩容的栈的插入操作的时间复杂度分析； 4. 用栈检查括号是否匹配，比如：{[()](){[]}或[{()}([])]等都为合法格式，而{[}()]或[({)]为不合法的格式；

04. 队列

概念与特性	1. 队列是一种操作受限的线性表，只能在两端插入、删除数据； 2. 队列的最大特性是先进先出；
操作与复杂度	1. 入队操作，在队尾插入数据，时间复杂度是O(1)； 2. 出队操作，从队头取出数据，时间复杂度是O(1)；
应用场景	队列常用在有限资源池中，用于排队请求，比如数据库连接池等；
其他知识点	1. 队列既可以用数组来实现，也可以用链表来实现； 2. 最长使用的队列是基于数组实现的循环队列；
掌握程度	熟练实现一个循环队列，重点是掌握队列的判空和判满条件；

02

递归、排序、二分查找

05. 递归

概念与特性	函数调用函数自身的编程方式叫做递归，调用为”递“，返回为”归“。
三个条件	1. 一个问题的解可以分解为多个子问题的解； 2. 分解之后的子森缇，除了数据规模不同，求解思路跟原问题相同； 3. 存在递归终止条件；
编程技巧	1. 寻找将大问题分解为小问题求解的规律； 2. 找出递推公式和终止条件，将其直接翻译成代码； 3. 切记不要人肉一层一层的递归； 换句话说，也就是：如果一个问题A可以分解为若干子问题B、C、D，我们可以假设子问题B、C、D已经解决，在此基础上思考如何解决问题A。我们只需要思考问题A与子问题B、C、D两层之间的关系即可，不需要一层一层往下思考子问题与子子问题，子子问题与子子子问题之间的关系。
应用场景	递归是一种应用非常广泛编程技巧，很多数据结构和算法的编码实现都要用到递归，比如快排、归并排序、DFS、二叉树遍历、回溯等；
其他知识点	1. 避免堆栈溢出（限制调用层次；递归改为迭代；尾递归优化）； 2. 避免重复计算（利用备忘录）；
掌握程度	1. 熟练编写斐波那契数列、全排列、八皇后、快速排序；归并排序、DFS、二叉树遍历、链表反转递归实现等； 2. 掌握递归算法的时间、空间复杂度分析；其中时间复杂度通过递推公式或者递归树来分析；空间复杂度跟递归函数调用栈深度成正比；

06. 排序

概念与特性	1. 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； 2. 原地：不额外申请非常量级的空间来临时存储排序数据；原地排序算法并不一定空间复杂度是O(1)，空间复杂度是O(1)的排序算法一定是原地排序算法，比如快速排序是原地排序算法，但因为用到递归，函数调用栈会消耗非常量级的空间，所以，空间复杂度并非O(1)，是O(logn)。	
O(n^2)	冒泡排序	冒泡排序是稳定原地排序算法。 整个冒泡排序过程包含多遍冒泡操作。每次冒泡操作都会遍历整个数组，依次对相邻的元素进行比较，看是否满足大小关系要求，如果不满足，就将它们互换位置。一次冒泡操作会让至少一个元素移动到它应该在的位置，重复n次，就完成了n个数据的排序工作。
	插入排序	插入排序是稳定原地排序算法。 首先，我们将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有一个元素，就是数组中的第一个元素。插入算法的核心思想是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。重复这个过程，直到未排序区间中元素为空，算法结束。
	选择排序	选择排序算法是非稳定原地排序算法。

		其实现思路有点类似插入排序，也分已排序区间和未排序区间。但不同点在于，选择排序算法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。
O(nlogn)	快速排序	快速排序是非稳定原地排序算法。空间复杂度是O(logn)。 如果要排序数组中下标从p到r之间的一组数据，我们选择p到r之间的任意一个数据作为pivot（分区点），然后，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。经过这一步骤之后，p到r之间的数据就被分成了三个部分。假设pivot现在所在位置的下标是q，那p到q−1之间数据都小于pivot，中间是pivot，q+1到r之间的数据都大于pivot。根据分治、递归的处理思想，我们递归排序下标从p到q−1之间的数据和下标从q+1到r之间的数据，直到区间缩小为1，就说明所有的数据都有序了。 递推公式： quickSort(p…r)=quickSort(p…q−1) & quickSort(q+1…r)
	归并排序	归并排序是稳定非原地排序算法。空间复杂度是O(n)。 如果要排序一个数组，我们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。 递推公式： mergeSort(p…r)=merge(mergeSort(p…q), mergeSort(q+1…r))
O(n)	桶排序	桶排序，顾名思义，会用到“桶”，核心思想是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。桶内排完序之后，再把每个桶里的数据按照顺序依次取出，组成的序列就是有序的了。 要排序的数据需要很容易就能划分成m个桶，并且，桶与桶之间有着天然的大小顺序。这样每个桶内的数据都排完序之后，桶与桶之间的数据不需要再进行排序。
	计数排序	实际上，计数排序是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，我们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。
	基数排序	基数排序对要排序的数据也是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系：如果a数据的高位比b数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，可以使用其他线性排序算法来排序，否则，基数排序的时间复杂度就无法做到O(n)了。
应用场景	工程中的排序函数一般使用O(nlogn)的快排、归并或者堆排序作为主排序算法，当数据规模较小时，转而选择使用更加简单的插入排序。	
其他知识点	为了避免快速排序时间复杂度退化为极端情况O(n^2)，我们使用更加高级的分区点选择方式，比如三数取中法、随机法等。	

掌握程度	1. 熟练掌握冒泡、插入、选择、快速、归并排序的原理、代码实现； 2. 熟练掌握快速、归并排序的时间和空间复杂度分析； 3. 掌握桶排序、计数排序、基数排序的原理；

07. 二分查找

概念与特性	二分查找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。
操作与复杂度	二分查找的时间复杂度是O(logn)。
二分查找变体	• 变体一：查找第一个值等于给定值的元素 • 变体二：查找最后一个值等于给定值的元素 • 变体三：查找第一个大于等于给定值的元素 • 变体四：查找最后一个小于等于给定值的元素
掌握程度	熟练掌握二分查找、二分查找变体的代码实现。

03

散列表、位图、哈希算法

08. 散列表

概念与特性	散列表的英文翻译是“Hash Table”，所以，我们平时也叫它“哈希表”或者“Hash表”。实际上，散列表是数组的一种扩展，由数组演化而来，底层依赖数组支持按下标随机访问的特性，所以，可以说，如果没有数组，就没有散列表。
操作与复杂度	理论上讲，散列表插入、删除、查找数据的时间复杂度是O(1)。更加准确一点，性能跟装载因子成相关性。
散列冲突解决	1. 开放寻址法，如果一旦出现散列冲突，就通过重新探测新位置的方法来解决冲突。 2. 链表法，它是一种更加常用的解决散列冲突办法，相比开放寻址法，它要简单得多。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。
应用场景	1. 支持快速插入、删除、查找数据的动态集合； 2. 编程语言中常用的Map结构，比如Java中的HashMap； 3. 跟有序链表组合使用，实现LRU缓存淘汰算法；
其他知识点	1. 支持动态扩容的散列表，为了避免一次性扩容导致的耗时过多，我们可以将扩容操作穿插在插入操作的过程中，分批完成。

	2. 为了提高散列表的性能，对基于链表法解决散列冲突的链表，当链表长度大于等于8时，将链表转化成红黑树。当红黑树中节点个数小于等于6时，又会将红黑树转化成链表。
掌握程度	1. 掌握散列表的原理，冲突解决方案； 2. 掌握LRU缓存淘汰算法的实现原理；

09. 位图

概念与特性	1. 位图是一种特殊的散列表，集合中的元素跟位图中的某一位一一对应。 2. 布隆过滤器是对位图的优化，为得是节省内存空间，比起位图来说，它使用多个散列函数对应多个位来确定一个元素。
操作与复杂度	1. 位图支持插入、删除、查找操作，时间复杂度是O(1)，空间复杂度跟数据范围大小有关。 2. 布隆过滤器支持插入、查找，一般不支持删除，时间复杂度是O(1)，空间消耗较位图要小。
应用场景	1. 搜索引擎爬虫URL去重； 2. 垃圾邮件判别；
其他知识点	布隆过滤器存在误判的情况，不过，只有在判断存在的情况下，才有可能发生误判，也就是说，判定存在有可能并不存在。如果某个数据经过布隆过滤器判断不存在，那说明这个数字真的不存在，这种情况是不会有误判的。
掌握程度	1. 掌握位图的代码实现； 2. 掌握布隆过滤器的原理和应用；

10. 哈希算法

概念与特性	哈希算法的定义和原理非常简单，一句话就能概括：将任意长度的二进制值串映射为固定长度的二进制值串，这个映射的规则就叫“哈希算法”。原始数据映射之后得到的二进制值串就叫“哈希值”。一般来讲，哈希算法还要满足下面几点： • 从哈希值不能反向推导出原始数据（因此哈希算法也叫单向哈希算法）； • 对输入数据非常敏感，哪怕原始数据只修改了一个二进制位，最后得到的哈希值也大不相同； • 散列冲突的概率要很小，不同原始数据对应相同哈希值的概率非常小； • 哈希算法的执行效率要高，对较长的文本，也能快速计算出哈希值。
应用场景	• 应用一：安全加密：任何哈希算法都会存在哈希冲突，但冲突概率一般会很小。越复杂的哈希算法越难破解，但计算耗时也就越长。所以，选择哈希算法要权衡安全性和计算耗时来决定。 • 应用二：唯一标识：哈希算法可以对大数据做信息摘要，通过一个较短的二进制编码来表示很大的数据。 • 应用三：数据校验：校验数据的完整性和正确性。 • 应用四：散列函数：它对哈希算法的要求比较特别，更加看重平均性和执行效率。 • 应用五：负载均衡：利用哈希算法替代映射表，可以实现一个会话粘滞的负载均衡策略。

	• 应用六：数据分片：通过哈希算法对处理的海量数据进行分片，多机分布式处理，可以突破单机资源的限制。 • 应用七：分布式存储：利用一致性哈希算法，可以解决缓存等分布式系统的扩容、缩容导致数据大量搬移的难题。
掌握程度	了解哈希算法的七个应用即可。

04

—

二叉树、BST、BBST、递归树、B+树

11. 二叉树

概念与特性	二叉树，顾名思义，每个节点最多有两个“叉”，也就是两个子节点，分别是左子节点和右子节点。不过，二叉树并不要求每个节点都有两个子节点，有的节点可能只有左子节点，有的节点可能只有右子节点。

二叉树的存储	存储一棵二叉树一般有两种方法：一种是基于指针或者引用的二叉链式存储法，一种是基于数组的顺序存储法。
	
	我们先来看比较简单、直观的链式存储法。从图中你应该可以很清楚地看到，每个节点有三个字段，其中一个存储数据，另外两个是指向左右子节点的指针。我们只要拎住根节点，就可以通过左右子节点的指针，把整棵树都串起来。这种存储方式比较常用。大部分二叉树代码都是通过这种结构来实现的。
	
	链式存储法用代码实现出来，就是下面这个样子。
	
	public class Node { // 节点的定义
	  public int data;
	  public Node left;
	  public Node right;
	}
	
	我们再来看，基于数组的顺序存储法。
	
	我们用数组来存储所有的节点。如果节点X存储在数组中下标为i的位置，下标为i*2的位置存储的就是左子节点，下标为2*i+1的位置存储的就是右子节点。反过来，下标为i/2的位置存储就是它的父节点。
	
	通过这种方式，我们只要知道根节点存储的位置（一般情况下，为了方便计算父子节点下标，根节点会存储在下标为1而不是0的位置），这样就可以通过下标计算，把整棵树都串起来。

二叉树遍历	前序遍历的递推公式： preOrder(r) = print r->preOrder(r->left)->preOrder(r->right) 中序遍历的递推公式： inOrder(r) = inOrder(r->left)->print r->inOrder(r->right) 后序遍历的递推公式：

	postOrder(r) = postOrder(r->left)->postOrder(r->right)->print r
掌握程度	熟练掌握二叉树的存储和前中后序遍历的代码实现。

12. 二叉查找树

概念与特性	二叉查找树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。它不仅仅支持快速查找数据，还支持快速插入、删除数据。
查找操作	先取根节点，如果要查找的数据等于根节点的值，那根节点就是要找的节点，直接返回。如果要查找的数据比根节点的值小，按照二叉查找树的定义，要查找的数据只有可能出现在左子树中，那就在左子树中递归查找；同理，如果要查找的数据比根节点的值大，那就在右子树中递归查找。
插入操作	为了简化插入，新插入的数据一般都放在叶子节点上。我们从根节点开始，依次比较要插入的数据和节点的大小关系，来寻找合适的插入位置。 如果要插入的数据比节点数据值大，并且节点的右子树为空，就将新数据直接插到右子节点的位置；如果不为空，就再递归遍历右子树，查找插入位置。同理，如果要插入的数据比节点数据值小，并且节点的左子树为空，就将新数据插入到左子节点的位置；如果不为空，就再递归遍历左子树，查找插入位置。
删除操作	二叉查找树的查找、插入操作都比较简单，但是，它的删除操作就比较复杂了 。针对待删除节点的子节点个数的不同，我们需要分三种情况来处理。 • 第一种情况是，如果要删除的节点没有子节点，我们只需要直接将父节点中，指向要删除节点的指针置为null即可。比如删除下图中的节点55。 • 第二种情况是，如果要删除的节点只有一个子节点（只有左子节点或者右子节点），我们只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。 • 第三种情况是，如果要删除的节点有两个子节点，这就比较复杂了。我们需要找到这个节点的右子树中的“最小节点”，把它替换到要删除的节点上。然后再删除掉这个“最小节点”，因为“最小节点”肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。
时间复杂度	大部分情况下，二叉查找树上操作的时间复杂度跟树的高度成正比，对于比较平衡的二叉查找树，插入、删除、查找的时间复杂度是O(logn)。极端情况下，如果二叉查找树退化为链表，那插入、删除、查找的时间复杂度退化为O(n)。
掌握程度	1. 熟练掌握二叉查找树的查找、插入、删除操作的代码实现。 2. 熟练掌握二叉查找树上操作的时间复杂度分析；

13. 平衡二叉查找树

概念与特性	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。从这个定义来看，完全二叉树、满二叉树其实都是平衡二叉树。除此之外，AVL也是严格的平衡二叉树。 很多平衡二叉树其实并没有严格符合上面的定义（树中任意一个节点的左右子树的高度相差不能大于1），比如更常用的红黑树，从根节点到各个叶子节点的最长路径，有可能会比最短路径大一倍。
存在的意义	二叉查找树在频繁的动态更新过程中，可能会出现树的高度远大于的情况，从而导致各个操作的效率下降。极端情况下，二叉树会退化为链表，相应地，时间复杂度会退化为O(n)。平衡二叉查找树是为了解决这个复杂度退化的问题而产生的。
红黑树	顾名思义，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，红黑树还需要满足以下几个要求： • 根节点是黑色的； • 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； • 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； • 对于每个节点，从该节点到其叶子节点的所有路径，都包含相同数目的黑色节点；
掌握程度	1. 了解平衡二叉查找树的定义、存在的意义； 2. 了解红黑树的定义；

14. 递归树

概念与特性	递归的思想就是将大问题分解为小问题来求解，然后再将小问题分解为小小问题。这样一层一层地分解，直到问题的数据规模被分解得足够小，不用再继续递归分解为止。如果我们把这个一层一层的分解过程画成图，它其实就是一棵树。我们给这棵树起一个名字，叫作递归树。
应用场景	递归树用来分析那些通过递推公式比较难分析的递归代码的时间复杂度。
掌握程度	熟练掌握快排、斐波那契数列、全排列生成等递归代码的时间复杂度分析。

15. B+树

概念与特性	B+树它通过将多叉树索引存储在磁盘中，做到了时间、空间的平衡，既保证了执行效率，又节省了内存 B+树的特点： 每个节点中子节点的个数不能超过m，也不能小于m/2； 根节点的子节点个数可以不超过m/2，这是一个例外； m叉树只存储索引，并不真正存储数据； 通过链表将叶子节点串联在一起，这样可以方便按区间查找； 一般情况，根节点会被存储在内存中，其他节点存储在磁盘中。
应用场景	数据库索引。

掌握程度	了解B+树的定义、操作。

05

—

堆、堆排序

16. 堆

概念与特性	只要满足下面两点的二叉树就是堆（大顶堆）： 堆是一个完全二叉树； 堆中每一个节点的值都必须大于等）其子树中每个节点的值； 如果将定义中第二条里的”大于等于“换为”小于等于“，那对应的就是小顶堆的定义。
堆的存储	完全二叉树比较适合用数组来存储。用数组来存储完全二叉树是非常节省存储空间的。因为我们不需要存储左右子节点的指针，单纯地通过对数组下标的运算，就可以找到一个节点的左右子节点和父节点。 对于数组中下标为i的节点，其左子节点是下标为i*2的节点，右子节点是下标为i*2+1的节点，父节点是下标为i/2的节点。
插入数据	将数据放入数组的末尾，并进行从下往上的堆化操作。堆化的过程非常简单，就是顺着节点所在的路径，向上对比，如果上下两个节点不符合堆中定义的大小关系的话，就将其交换。
查询最大值	直接去堆顶元素，也就是下标为1的数组元素。
删除最大值	把最后一个节点放到堆顶，然后利用从上往下的堆化方法，对父子节点进行对比，对于不满足父子节点大小关系的，互换两个节点，并且重复进行这个过程，直到父子节点之间满足大小关系为止。
应用场景	维护动态集合最值、堆排序、优先级队列、求TOP K、求中位数等。
掌握程度	1. 熟练掌握堆的存储和三个核心操作的代码实现。 2. 熟练掌握对的经典应用。

17. 堆排序

概念与特性	堆排序是一种非稳定、原地、时间复杂度为O(nlogn)的排序算法。包含建堆和排序两个过程。
建堆	下标从n/2+1到n的节点都是叶子节点。我们对下标从n/2到1的元素进行从上往下的堆化，下标是n/2+1到n的叶子节点，不需要堆化。 建堆过程的时间复杂度是O(n)，这个分析稍微有点复杂，需要掌握。

排序	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了。数组中的第一个元素就是堆顶，也就是最大元素。我们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。交换之后的堆顶元素，需要通过自上而下堆化的方法，将其移动到合适的位置。堆化完成之后，堆中只剩下n−1个元素，我们再取堆顶元素，跟下标为n−1的元素交换位置，然后再堆化交换之后的堆顶元素，一直重复这个过程，直到最后堆中只剩下一个元素，排序工作就完成了。 排序过程的时间复杂度是O(nlogn)。
掌握程度	1. 熟练掌握堆排序的代码实现； 2. 熟练掌握堆排序的复杂度分析；

06

—

跳表、并查集、线段树、树状数组

18. 跳表

概念与特性	跳表使用空间换时间的设计思路，通过构建多级索引来提高查询的效率，实现了基于链表的“二分查找”。
操作与复杂度	跳表是一种动态数据结构，支持快速的插入、删除、查找操作，时间复杂度都是O(logn)。 跳表的空间复杂度是O(n)。不过，跳表的实现非常灵活，可以通过改变索引构建策略，有效平衡执行效率和内存消耗。
应用场景	跳表支持区间查询，这是红黑树实现不了的。
掌握程度	掌握跳表的实现原理和复杂度分析。

19. 并查集

概念与特性	并查集就是用来根据对象两两之间的直接关系来快速查询任意两个对象之间是否存在关系（直接或间接的）。 实际上，并查集从本质上来讲是一组集合，存在（直接或间接）关系的对象组成一个集合，不存在任何关系的对象被分割到不同的集合。在这组集合上有两个主要的操作，一个是“并”（union），一个是“查”（find）。
基于树的实现	基于树的并查集实现思路，使用树来表示集合，并且使用树的根节点作为集合“代表”来标识一个集合。在基于树的并查集实现思路中，每个节点并不没有直接存储集合的代表，而是利用父节点指针，沿着此节点到根节点的路径往上追溯来寻找根节点。

	• 按秩合并：当合并两棵树的时候，为了让树尽可能的”矮”，我们把高度小的树拼接到高度大的树上。所以，我们需要记录树的高度，也就是树的“代表”（根节点）的高度，我们给它起了一个新的名字，叫做秩（Rank）。按秩合并的意思就是按秩的大小来合并。如果两棵树的秩不同，合并后新树的秩不变。如果两棵树的秩相同，合并后新树的秩加一。 • 路径压缩：查找某个对象所在集合的“代表”，也就是查找某个结点的根节点。在通过父指针从这个结点往上追溯的过程中，我们可以将路径上的所有节点的父指针都更新为指向根节点。这样，路径上的所有节点就都可以快速地寻找到根节点了。
复杂度分析	实际上，union()操作的时间复杂度很好分析，是O(1)。find()操作的时间复杂度是比对数级还小、接近线性的一个量级。严密的分析和证明非常繁琐，我们不展开讲解，但我们可以换一种不严谨的方式去理解，find()操作的过程会触发路径压缩，多次find()操作之后，所有的节点的父节点都直接指向根节点，所以，后续find()操作就变成了O(1)时间复杂度的了。
掌握程度	1. 了解并查集的原理和代码实现。 2. 掌握并查集的应用。

20. 线段树

概念与特性	假设要处理的数据的最大值是m，并且都为正整数。我们构建一棵特殊的二叉树。每个节点代表一个区间，包含三个基本数据：区间起始点、结束点、统计值（比如数据个数）。根节点表示最大的区间[1, m]，根节点的左右子节点分别代表[1, m]区间的一半，左子节点代表前半部分[1, m/2]，右子节点代表后半部分[m/2+1, m]。以此类推，逐级拆分，直到节点只包含一个数据为止。
线段树的存储	线段树使用数组来存储。我们知道，堆是完全二叉树，所以，使用数组的存储方式，数组中间没有空洞，不会浪费存储空间，但是，线段树并不是完全二叉树，所以，会浪费一定的存储空间，不过，线段树的叶子节点主要集中在最后两层，整体接近完全二叉树，尽管有空洞，但不会很多，可以接受。
操作与复杂度	线段树能实现在动态数据集合之上高效地插入、删除、区间统计，其时间复杂度跟线段树的高度有关，是O(logm)。需要注意的是，此处的m是数据集合的最大值，而非数据个数。尽管线段树的时间复杂度比较低，但空间复杂度稍高，需要较多的额外的存储空间。如果数据集合的最大值是m，那需要4*m大小的数组才能存的下所有的区间节点。
应用场景	主要用来区间统计，比如统计落在某个区间的数据个数，统计某个区间的数据之和、最大值、最小值，甚至可以统计某个区间的第K大元素。
掌握程度	1. 了解线段树的原理和代码实现、复杂度分析。 2. 掌握线段树的应用；

21. 树状数组

概念与特性	在一个包含n个数据的数组a中，数组中的数据有可能会更新，为了实现快速的更新元素和求前缀和操作，我们在数组a之上，构建树状数组c。

	用c[i]表示以a[i]为结尾的数据片段的和。c[i]中包含元素的个数就等于i的最后一个为1的二进制位的值，比如c[6]，i=6的最后一个为1的二进制位值是2^1，所以，c[6]包含两个元素的和，又以a[6]结尾，所以，c[6]=a[5]+a[6]。
操作与复杂度	利用树状数组，更新元素和求前缀和操作的时间复杂度可以做到O(logn)。
应用场景	树状数组应用范围比较小，经典应用是求前缀和，当然，也可以求区间和。所有可以用树状数组来解决的问题，都可以用线段树来解决。反过来则不成立，并不是所有线段树能解决的问题，都可以用树状数组来解决。尽管树状数组应用范围比线段树要局限，但在它能够解决的问题上，比起线段树，它的代码实现更加简单，空间消耗更少。
掌握程度	1. 了解树状数组的原理和代码实现。 2. 掌握树状数组的应用。

07

—

字符串匹配算法

22. 单模式字符串匹配算法

BF算法	BF算法是最简单、粗暴的字符串匹配算法，它的实现思路是，拿模式串跟主串中所有长度为m的子串匹配，看是否有能匹配的子串。所以，时间复杂度也比较高，是O(n*m)，n、m表示主串和模式串的长度。
RK算法	RK算法是借助哈希算法对BF算法进行改造，即对每个子串分别求哈希值，然后拿子串的哈希值与模式串的哈希值比较，减少了子串与模式串比较的时间。RK算法的时间复杂度是O(n)，跟BF算法相比，效率提高了很多。不过这样的效率取决于哈希算法的设计方法，如果存在冲突的情况下，时间复杂度可能会退化。极端情况下，哈希算法大量冲突，时间复杂度就退化为O(n*m)。
BM算法	BM算法核心思想是，利用模式串本身的特点，在模式串中某个字符与主串不能匹配的时候，将模式串往后多滑动几位，以此来减少不必要的字符比较，提高匹配的效率。BM算法构建的规则有两类：坏字符规则和好后缀规则。好后缀规则可以独立于坏字符规则使用。因为坏字符规则的实现比较耗内存，为了节省内存，我们可以只用好后缀规则来实现BM算法。 为了提高算法的执行效率，对于坏字符规则，我们通过散列表，来实现快速查找坏字符在模式串中出现的位置。对于好后缀规则，我们事先计算suffix数组和prefix数组，来实现快速地在模式串中查找好后缀可以匹配的子串，以及可匹配模式串前缀子串的最长后缀子串。
KMP算法	KMP算法和BM算法的本质非常类似，都是根据规律，在遇到坏字符的时候，把模式串往后多滑动几位。

	BM算法有两个规则，坏字符和好后缀。KMP算法借鉴BM算法的思想，可以总结成好前缀规则。这里面最难懂的就是next数组的计算。如果用最笨的方法来计算，确实不难，但是效率会比较低。所以，KMP算法使用了一种类似动态规划的方法，按照下标i从小到大，依次计算next[i]，并且next[i]的计算通过前面已经计算出来的next[0]，next[1]，……，next[i-1]来推导。
应用场景	因为BF算法逻辑简单、代码实现简单，对于处理小规模的字符串匹配，其性能又跟更加高效的KMP、BM算法相差无几，甚至因为逻辑简单而更加高效，所以，在实际的软件开发中，非常常用，并且大部分编程语言，都提供了相应的函数实现。 在实际的软件开发中，如果字符串匹配是核心、高频、耗时多的操作，比如，像网络安全入侵检测这样的应用场景，字符串匹配是其中最核心的操作，尽管每个模式串可能都不长，单一聚焦在一次字符串匹配上，可能用什么算法都相差无几，但微小性能差距累加起来，整个应用的运行效率就相差很多。这种情况下，更加高效的BM、KMP算法就排上用场了。
掌握程度	1. 熟练掌握BF算法的原理和代码实现。 2. 了解RK、BM、KMP算法的原理和代码实现。

23. 多模式字符串匹配算法

Trie树	Trie树，也叫“字典树”。顾名思义，它是一个树形结构，用来专门处理字符串匹配，解决在一组字符串集合中快速查找某个字符串的问题。 Trie树的本质是利用字符串之间的公共前缀，将重复的前缀合并在一起。其中，根节点不包含任何信息。每个节点表示字符串中的一个字符，从根节点到红色节点的一条路径表示字符串集合中的一个字符串。
AC自动机	AC自动机算法，全称是Aho-Corasick算法。其实，多模式串匹配中Trie树跟AC自动机之间的关系，就等同于单模式串匹配中BF算法跟KMP算法之间的关系。所以，AC自动机实际上就是在Trie树之上，加了类似KMP算法的next数组，只不过此处的next数组是构建在树上，叫做失败指针。
应用场景	多模式串匹配更常用AC自动机，比如敏感词过滤。精确匹配查找问题更适合用散列表或者红黑树来解决。对于模糊查找，比如查找前缀匹配的字符串，更加适合用Trie树，比如搜索关键词提示功能。
掌握程度	1. 熟练掌握Trie树的原理和代码实现、应用。 2. 了解AC自动机的原理和代码实现。

08

图和图算法

24. 图的表示

概念	图（Graph）是一种非线性表结构，除此之外，前面讲到的树也是非线性表结构。不过，跟树比起来，图更加复杂。树中的元素叫作节点，对应地，图中的元素叫作顶点（vertex）。如下图所示，图中的一个顶点可以与任意其他顶点建立连接关系，我们把这种连接关系叫作边（edge）。
其他重要概念	无向图、有向图、加权图、入度、出度
图的存储	邻接矩阵、邻接表
掌握程度	熟练掌握图的两种存储方法及代码实现。

25. 图的算法

搜索算法	BFS&DFS	广度优先搜索，通俗的理解就是，地毯式层层推进，从起始顶点开始，依次往外遍历。广度优先搜索逐层遍历，需要借助队列来实现。深度优先搜索用的是回溯思想，非常适合用递归实现。换种说法，深度优先搜索是借助栈来实现的。
	单源最短路径	单源最短路径算法不仅能求得单个源点到单个终点的最短路径，它还能一次性求得单个源点到多个终点（图中所有其他顶点）的最短路径。求解单源点到单终点的最短路径的需求更普遍。 Dijkstra算法是构建在有向有权图之上的，并且它要求不能存在负权边，对于存在负权边的有权图，对应的最短路径算法是Bellman-Ford算法。 Dijkstra算法借助优先级队列，每次从队列中，取出与起始顶点最近的顶点，然后更新其他相邻顶点的dist值，然后将新扩展出来的顶点（没有加入过优先级队列的）加入到优先级队列中。重复上面的过程，直到目标顶点出队列。
	多源最短路径	Floyd-Warshall算法简称为Floyd算法。它像Dijkstra算法一样，既可以处理有向图，也可以处理无向图。除此之外，它允许图中存在负权边，但不允许存在负权环（也就是图中存在包含负权边的环）。 Floyd算法的核心代码只有5行，三层for循环，但要理解这5行代码并不简单。它利用的是动态规划的处理思想，符合动态规划典型的多阶段决策模型，每个阶段都引入一个中转顶点，看通过中转顶点是否能缩小顶点之间的距离。下一阶段基于上一阶段的更新结果，引入新的中转顶点，直到所有的中转顶点都引入之后，dist数组中存储的就是顶点之间的最短距离。 多源最短路径问题也可以通过运行V次单源最短路径来解决，相对于这种方法，针对稠密图，Floyd算法在执行效率上更有优势。除此之外，在空间复杂度、编码复杂度上，Floyd算法也具有绝对优势。
	A*算法	A*算法属于一种启发式搜索算法（Heuristically Search Algorithm）。实际上，启发式搜索算法并不仅仅只有A*算法，还有很多其他算法，比如IDA*算法、蚁群算法、遗传算法、模拟退火算法等。如果感兴趣，你可以自行研究下。

		启发式搜索算法利用估价函数，避免“跑偏”，贪心地朝着最有可能到达终点的方向前进。这种算法找出的路线并不是最短路线。但是，实际的软件开发中的路线规划问题，我们往往并不需要非得找最短路线。所以，鉴于启发式搜索算法能很好地平衡路线质量和执行效率，它在实际的软件开发中的应用更加广泛。
拓扑排序	拓扑排序是指通过局部的依赖关系、先后顺序，推导满足所有局部依赖关系的执行序列。解决拓扑排序问题的经典算法有两种：Kahn算法和DFS算法。 Kahn算法利用的是贪心算法思想。核心思想是：每个顶点的入度表示这个顶点依赖多少个其他顶点。如果某个顶点的入度变成了0，就表示这个顶点没有依赖的顶点了，或者说这个顶点依赖的顶点都已经执行了。 DFS算法需要将邻接表转化成逆邻接表来处理。核心思想是：通过深度优先递归遍历，优先输出某个节点可达的所有节点，然后再输出自己。	
最小生成树	最小生成树针对的是无向有权连通图。假设图有V个顶点和E条边，那连通所有的顶点只需要V-1条边，多了就会存在环。这V-1条边和V个顶点构成一棵树，叫做生成树。一个图的生成树有很多个，其中边的权重之和最小的那个生成树，被叫做最小生成树。 最小生成树的两种经典算法都利用了贪心算法思想。其中，Kruskal算法借助于并查集，起初将每个顶点看做一个集合，然后，按照权重从小到大考察每条边。如果边对应的两个顶点在不同的集合中，则将这条边放入最小生成树，并将集合合并，否则，丢弃这条边。Prim算法的处理思路跟Dijkstra算法非常类似。它先初始化一棵空的最小生成树，然后不停地往外扩展寻找跟这棵树直接相连的边中最小的边，如果这条边加入到最小生成树中不会形成环，那就将其加入，否则，继续考察其他边。	
最大流	最大流算法有很多，大体上可以分为两类：基于增广路径（Augmenting Path）的算法和基于推送-重贴标签（Push Relabel）的算法。本节重点介绍了基于增广路径的Ford-Fulkerson方法，以及它的一个具体实现Edmonds-Karp算法。除此之外，我们还介绍最大流的一个经典应用：最大二分匹配。 Ford-Fulkerson方法的基本原理非常简单，通过不停地在残余网络中寻找增广路径来求解最大流。不过，我们还需要对基本原理做修正，通过增加反向边来实现对之前的增广路径重新“改道”。不过，Ford-Fulkerson方法并没有给出寻找增广路径的具体实现，所以，它只能称为方法而不是算法。其中，通过广度优先搜索来寻找增广路径的算法被叫做Edmonds-Karp算法。	
掌握程度	1. 熟练掌握BFS、DFS的原理和代码实现； 2. 掌握Dijkstra算法的原理和代码实现； 3. 了解Floyd算法的原理和代码实现； 4. 了解A*算法的原理和代码实现； 5. 掌握拓扑排序算法的原理和代码实现； 6. 掌握最小生成树算法的原理和代码实现； 7. 了解最大流算法原理和代码实现；	

09

—

贪心、分治、回溯、动态规划

24. 算法思想

贪心	实际上，贪心算法适用的场景比较有限。这种算法思想更多的是指导设计基础算法。比如最小生成树算法、单源最短路径算法，这些算法都用到了贪心算法。从我个人的学习经验来讲，不要刻意去记忆贪心算法的原理，多练习才是最有效的学习方法。 对于贪心算法，最难的一块是如何将要解决的问题抽象成贪心算法模型，只要这一步搞定之后，贪心算法的编码一般都很简单。虽然很多时候贪心算法解决问题的正确性看起来显而易见，但要严谨地证明算法能够得到最优解，并不是件容易的事。所以，很多时候，我们只需要多举几个例子，验证一下贪心算法的解决方案能得到最优解就可以了。
分治	分治算法（divide and conquer）的核心思想其实就是四个字：分而治之，详细点讲，就是将原问题划分成n个规模更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。 分治看起来有点像递归。分治算法是一种处理问题的思想，递归是一种编程技巧。实际上，分治算法一般都比较适合用递归来实现。分治算法的递归实现中，每一层递归都会涉及下面这样三个操作： 分解：将原问题分解成一系列子问题； 解决：递归地求解各个子问题，若子问题足够小，则直接求解； 合并：将子问题的结果合并成原问题。
回溯	回溯算法思想非常简单，大部分情况下，都是用来解决广义的搜索问题，也就是，从一组可能的解中，选出一个满足要求的解。回溯算法非常适合用递归来实现，在实现的过程中，剪枝操作是提高回溯效率的一种技巧。利用剪枝，可以提前终止不能满足要求的情况，从而提高搜索效率。
动态规划	什么样的问题适合用动态规划解决？这些问题可以总结概括为“一个模型三个特征”。其中，“一个模型”指的是问题可以抽象成分阶段决策最优解模型。“三个特征”指的是最优子结构、无后效性和重复子问题。 动态规划有两种解题思路，分别是状态转移表法和状态转移方程法。其中，状态转移表法解题思路大致可以概括为：回溯算法实现–定义状态–画递归树–找重复子问题–画状态转移表–根据递推关系填表–将填表过程翻译成代码。状态转移方程法的大致思路可以概括为：找最优子结构–写状态转移方程–将状态转移方程翻译成代码。
掌握程度	1. 熟练掌握贪心算法经典问题：分糖果、最短服务时间、区间覆盖； 2. 熟练掌握分治算法经典问题：快排、归并排序； 3. 熟练掌握回溯算法经典问题：八皇后、全排列、0–1背包、正则表达式； 4. 熟练掌握动态规划经典问题：背包问题、杨辉三角、走格子、硬币找零、莱文斯坦编辑距离、最长公共子串、最长递增子序列