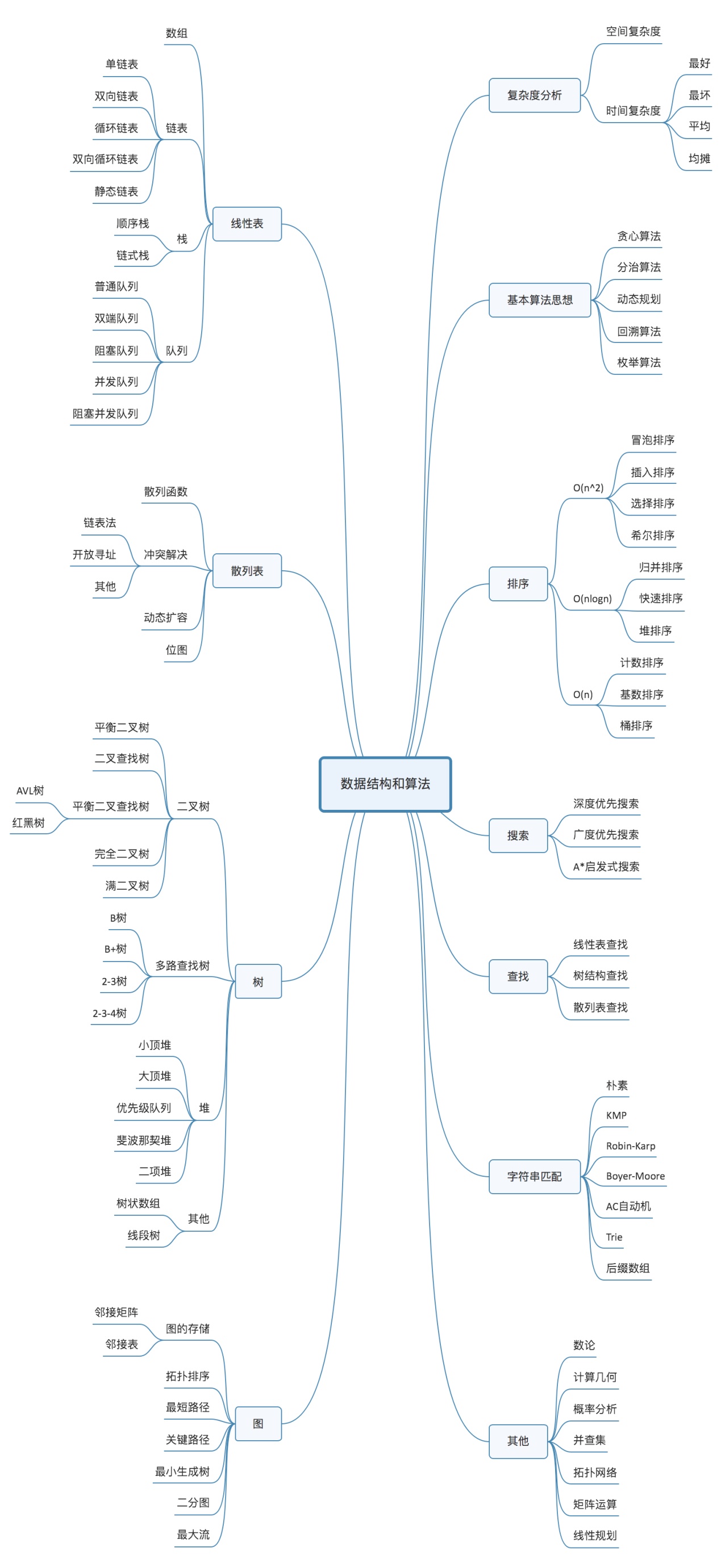
数据结构就是指一组数据的存储结构。算法就是操作数据的一组方法



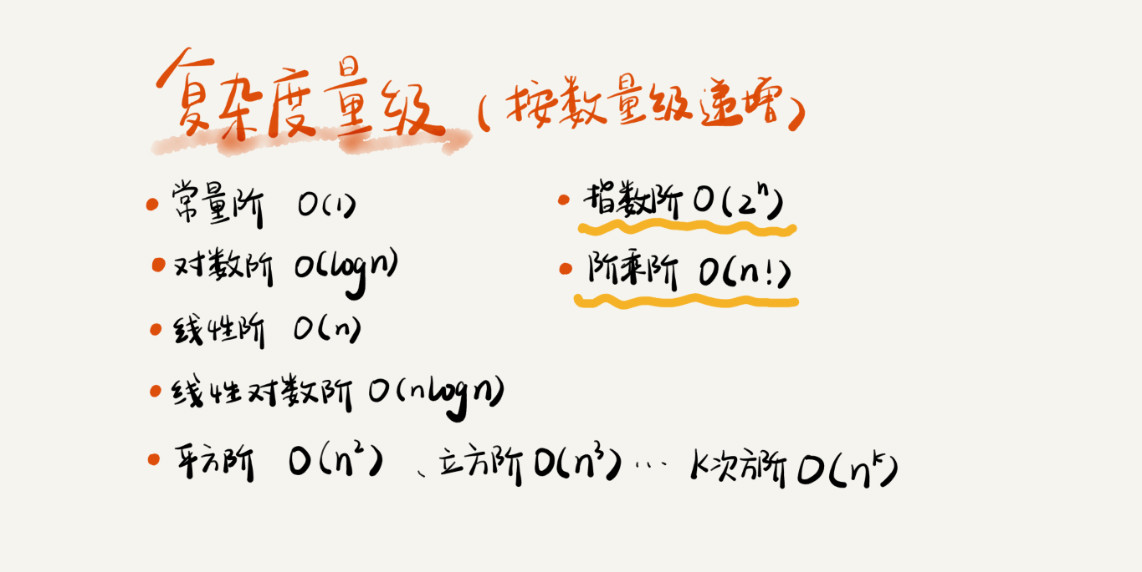
**03 | 复杂度分析（上）：如何分析、统计算法的执行效率和资源消耗？**

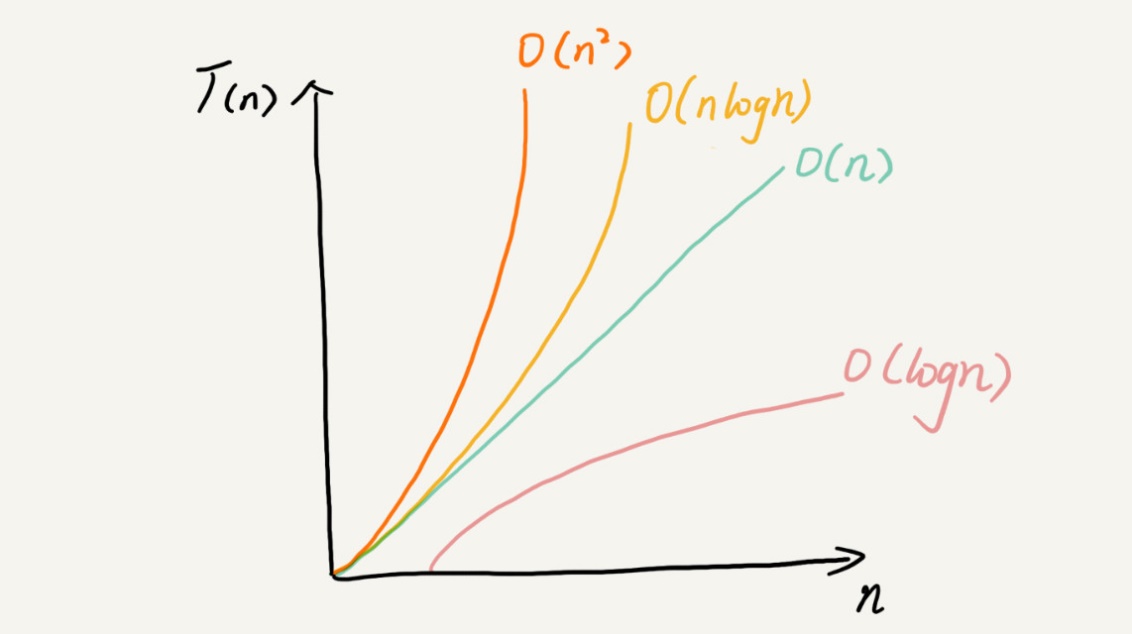
我们都知道，数据结构和算法本身解决的是“快”和“省”的问题，即如何让代码运行得更快，如何让代码更省存储空间。

## 大 O 时间复杂度表示法。

大 O 时间复杂度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。：总的时间复杂度就等于量级最大的那段代码的时间复杂度。

几种常见时间复杂度实例分析





对于刚罗列的复杂度量级，我们可以粗略地分为两类，多项式量级和非多项式量级。其中，非多项式量级只有两个：O(2n) 和 O(n!)。我们把时间复杂度为非多项式量级的算法问题叫作 NP（Non-Deterministic Polynomial，非确定多项式）问题。

## 空间复杂度

空间复杂度全称就是渐进空间复杂度（asymptotic space complexity），表示算法的存储空间与数据规模之间的增长关系。

我们常见的空间复杂度就是 O(1)、O(n)、O(n2 )，像 O(logn)、O(nlogn) 这样的对数阶复杂度平时都用不到

## 平均情况时间复杂度

## 均摊时间复杂度

# 05 | 数组：为什么很多编程语言中数组都从0开始编号？

在大部分编程语言中，数组都是从 0 开始编号的，但你是否下意识地想过，为什么数组要从 0 开始编号，而不是从 1 开始呢？ 从 1 开始不是更符合人类的思维习惯吗？

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）从 1 开始编号，每次随机访问数组元素都多了一次减法运算，对于 CPU 来说，就是多了一次减法指令。

常常会问数组和链表的区别，

“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)

int main(int argc, char\* argv[]){

int i = 0;

int arr[3] = {0};

for(; i<=3; i++){

arr[i] = 0;

printf("hello world\n");

}

return 0;

}

我们知道，在 C 语言中，只要不是访问受限的内存，所有的内存空间都是可以自由访问的。根据我们前面讲的数组寻址公式，a[3]也会被定位到某块不属于数组的内存地址上，而这个地址正好是存储变量 i 的内存地址，那么 a[3]=0 就相当于 i=0，所以就会导致代码无限循环。函数体内的局部变量存在栈上，且是连续压栈。在Linux进程的内存布局中，栈区在高地址空间，从高向低增长。变量i和arr在相邻地址，且i比arr的地址大，所以arr越界正好访问到i。当然，前提是i和arr元素同类型，否则那段代码仍是未决行为。

ArrayList 最大的优势就是可以将很多数组操作的细节封装起来。比如前面提到的数组插入、删除数据时需要搬移其他数据等。另外，它还有一个优势，就是支持动态扩容。

# 06 | 链表（上）：如何实现LRU缓存淘汰算法?

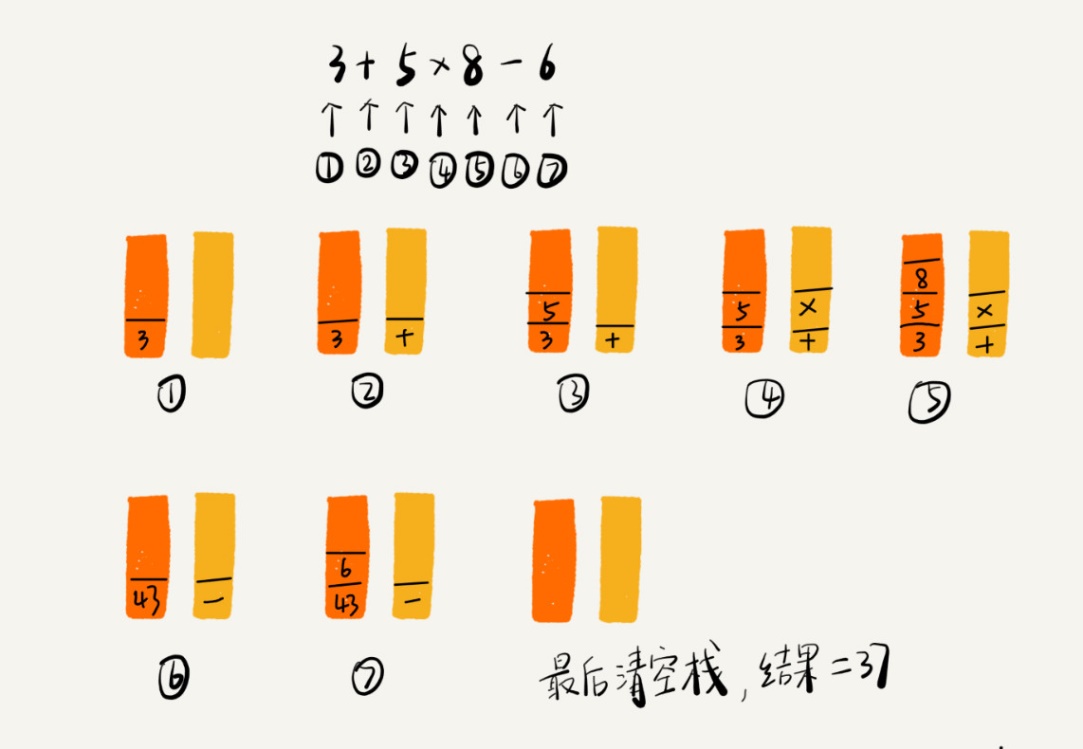
LRU： least recent used 最近最少使用 LFU: least frequent used 最少使用 FIFO: first in first out 先进先出 单链表： node1->node2->node3 循环链表: node1->node2->node3->node1 双链表: node1<->node2<->node3<->node4 链表优势： 插入、删除性能比数组强，内存容易申请，方便扩容 链表的劣势：随机访问、遍历性能比数组弱，可能造成内存碎片 数组优势：随机访问性能好，连续内存方便cpu预取 数组的劣势：大数组内存不好申请，不方便扩容 双链表表比单链表更耗费内存 双链表比单链表在删除性能上更强 LRU leas recent used： 1、单链表 2、对头放最新的，对尾放最旧的 3、若数据命中，则删除数据，插入头部 4、若数据不中，容量空闲，则插入对头 5、若数据不中，没容量，则删除队尾，将数据插入对头 优化：用散列保存每个元素位置，O(1)查找位置，直接做操作不用遍历了

# 07 | 链表（下）：如何轻松写出正确的链表代码？

所以，我精选了 5 个常见的链表操作。你只要把这几个操作都能写熟练，不熟就多写几遍，我保证你之后再也不会害怕写链表代码。单链表反转链表中环的检测两个有序的链表合并删除链表倒数第 n 个结点求链表的中间结点

# 08 | 栈：如何实现浏览器的前进和后退功能？

后进者先出，先进者后出，这就是典型的“栈”结构。

利用栈可已实现加减乘除我将 3+5\*8-6 这个表达式的计算过程画成了一张图，你可以结合图来理解我刚讲的计算过程。

# 09 | 队列：队列在线程池等有限资源池中的应用

先进者先出，这就是典型的“队列”。

如何实现循环队列

空和满的判断条件

(Tail+1)%capacity=head 队列满了

Tail==head 队列为空

# 10 | 递归：如何用三行代码找到“最终推荐人”？

有两个最难理解的知识点，一个是动态规划，另一个就是递归。

之后我们要讲的很多数据结构和算法的编码实现都要用到递归，比如 DFS 深度优先搜索、前中后序二叉树遍历等等

这就是一个非常标准的递归求解问题的分解过程，去的过程叫“递”，回来的过程叫“归”

## 递归需要满足的三个条件

递归需要满足的三个条件刚刚这个例子是非常典型的递归，那究竟什么样的问题可以用递归来解决呢？我总结了三个条件，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何为子问题？子问题就是数据规模更小的问题。比如，前面讲的电影院的例子，你要知道，“自己在哪一排”的问题，可以分解为“前一排的人在哪一排”这样一个子问题。2. 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样比如电影院那个例子，你求解“自己在哪一排”的思路，和前面一排人求解“自己在哪一排”的思路，是一模一样的。3. 存在递归终止条件把问题分解为子问题，把子问题再分解为子子问题，一层一层分解下去，不能存在无限循环，这就需要有终止条件。

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。

1. 递归代码要警惕堆栈溢出
   1. 。递归调用超过一定深度（比如 1000）之后，我们就不继续往下再递归了，直接返回报错
2. 递归代码要警惕重复计算
   1. 为了避免重复计算，我们可以通过一个数据结构（比如散列表）来保存已经求解过的 f(k)

怎么将递归代码改写为非递归代码？我们刚说了，递归有利有弊，利是递归代码的表达力很强，写起来非常简洁；而弊就是空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗时较多等问题。所以，在开发过程中，我们要根据实际情况来选择是否需要用递归的方式来实现。

## 那是不是所有的递归代码都可以改为这种迭代循环的非递归写法呢？

笼统地讲，是的。因为递归本身就是借助栈来实现的，只不过我们使用的栈是系统或者虚拟机本身提供的，我们没有感知罢了。如果我们自己在内存堆上实现栈，手动模拟入栈、出栈过程，这样任何递归代码都可以改写成看上去不是递归代码的样子。

但是这种思路实际上是将递归改为了“手动”递归，本质并没有变，而且也并没有解决前面讲到的某些问题，徒增了实现的复杂度。

# 11 | 排序（上）：为什么插入排序比冒泡排序更受欢迎？

如何分析一个“排序算法”？

排序算法的执行效率对于排序算法执行效率的分析，我们一般会从这几个方面来衡量：

1. 最好情况、最坏情况、平均情况时间复杂度
2. 时间复杂度的系数、常数 、低阶

3. 比较次数和交换（或移动）次数

排序算法的内存消耗

原地排序（Sorted in place）。原地排序算法，就是特指空间复杂度是 O(1) 的排序算法。

排序算法的稳定性

稳定性。这个概念是说，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。

第一，冒泡排序是原地排序算法吗？冒泡的过程只涉及相邻数据的交换操作，只需要常量级的临时空间，所以它的空间复杂度为 O(1)，是一个原地排序算法

第二，冒泡排序是稳定的排序算法吗？

在冒泡排序中，只有交换才可以改变两个元素的前后顺序。为了保证冒泡排序算法的稳定性，当有相邻的两个元素大小相等的时候，我们不做交换，相同大小的数据在排序前后不会改变顺序，所以冒泡排序是稳定的排序算法。第三，冒泡排序的时间复杂度是多少？最好情况下，要排序的数据已经是有序的了，我们只需要进行一次冒泡操作，就可以结束了，所以最好情况时间复杂度是 O(n)。而最坏的情况是，要排序的数据刚好是倒序排列的，我们需要进行 n 次冒泡操作，所以最坏情况时间复杂度为 O(n2)。

为什么插入排序的性能大于冒泡排序，

我们把执行一个赋值语句的时间粗略地计为单位时间（unit\_time），然后分别用冒泡排序和插入排序对同一个逆序度是 K 的数组进行排序。用冒泡排序，需要 K 次交换操作，每次需要 3 个赋值语句，所以交换操作总耗时就是 3\*K 单位时间。而插入排序中数据移动操作只需要 K 个单位时间。

# 12 | 排序（下）：如何用快排思想在O(n)内查找第K大元素？

## 归并排序的原理

我们先来看归并排序（Merge Sort）。

分治思想跟我们前面讲的递归思想很像。是的，分治算法一般都是用递归来实现的。分治是一种解决问题的处理思想，递归是一种编程技巧

递推公式：

merge\_sort(p…r) = merge(merge\_sort(p…q), merge\_sort(q+1…r))

终止条件：

p >= r 不用再继续分解

从刚刚的分析，我们可以得到一个重要的结论：不仅递归求解的问题可以写成递推公式，递归代码的时间复杂度也可以写成递推公式。

1. 归并排序是稳定的排序算法吗？是
2. 归并排序的时间复杂度是多少？O(nlogn)
3. 归并排序的空间复杂度是多少？

归并排序的时间复杂度任何情况下都是 O(nlogn)，看起来非常优秀

所以空间复杂度是 O(n)。

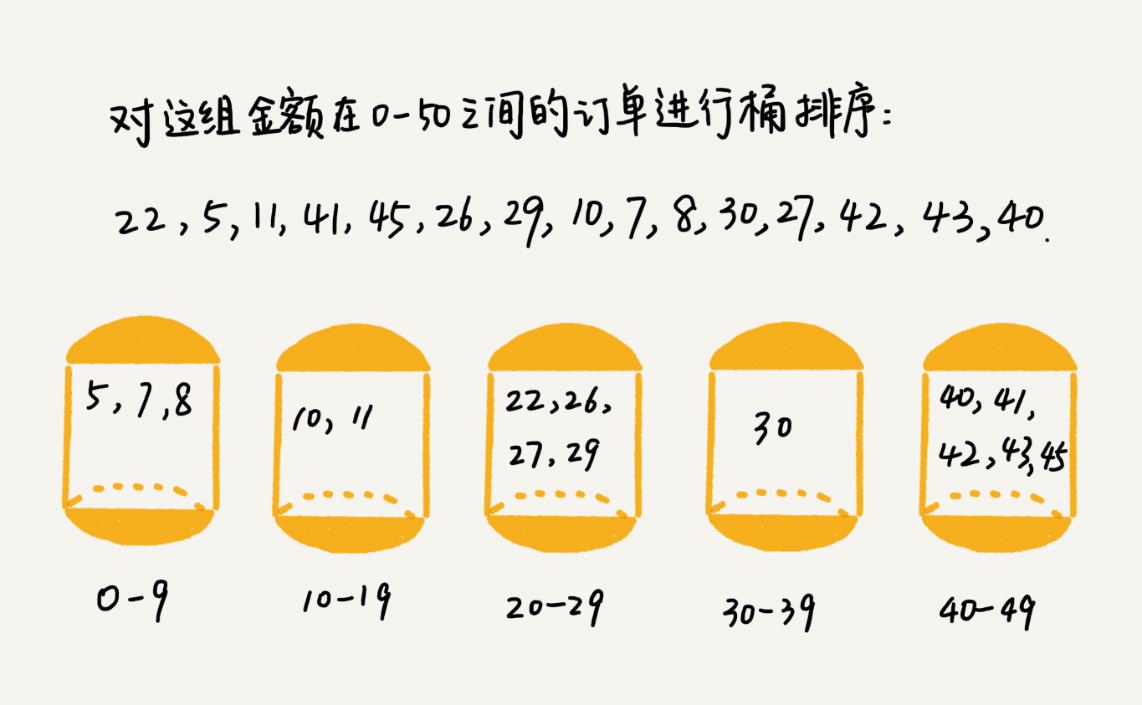
# 13 | 线性排序：如何根据年龄给100万用户数据排序？

我会讲三种时间复杂度是 O(n) 的排序算法：桶排序、计数排序、基数排序。

## 桶排序（Bucket sort）

订单数据排序

首先，我们来看桶排序。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。



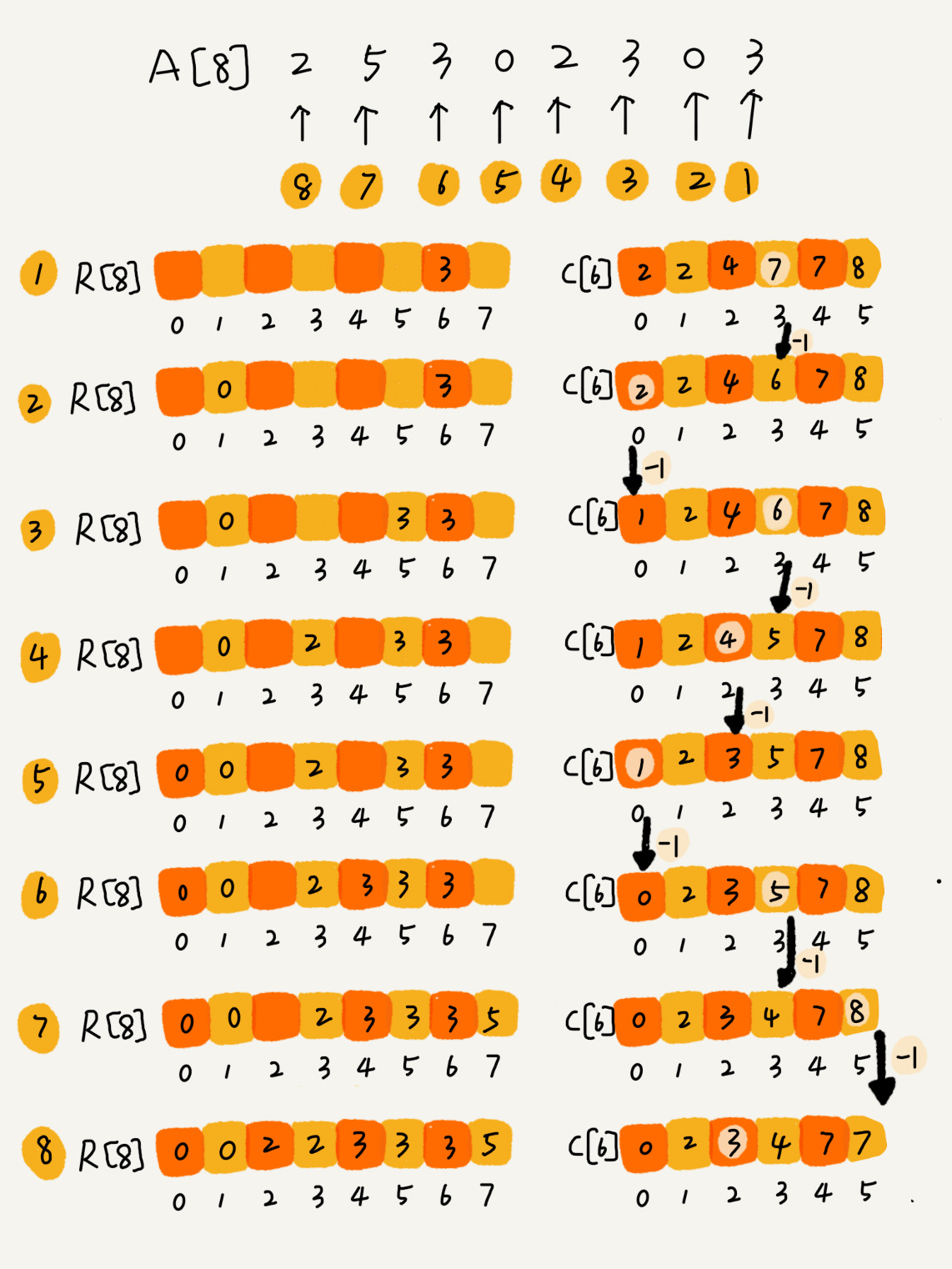
，要排序的数据需要很容易就能划分成 m 个桶，并且，桶与桶之间有着天然的大小顺序。这样每个桶内的数据都排序完之后，桶与桶之间的数据不需要再进行排序。

桶排序比较适合用在外部排序中。所谓的外部排序就是数据存储在外部磁盘中，数据量比较大，内存有限，无法将数据全部加载到内存中。

## 计数排序（Counting sort）

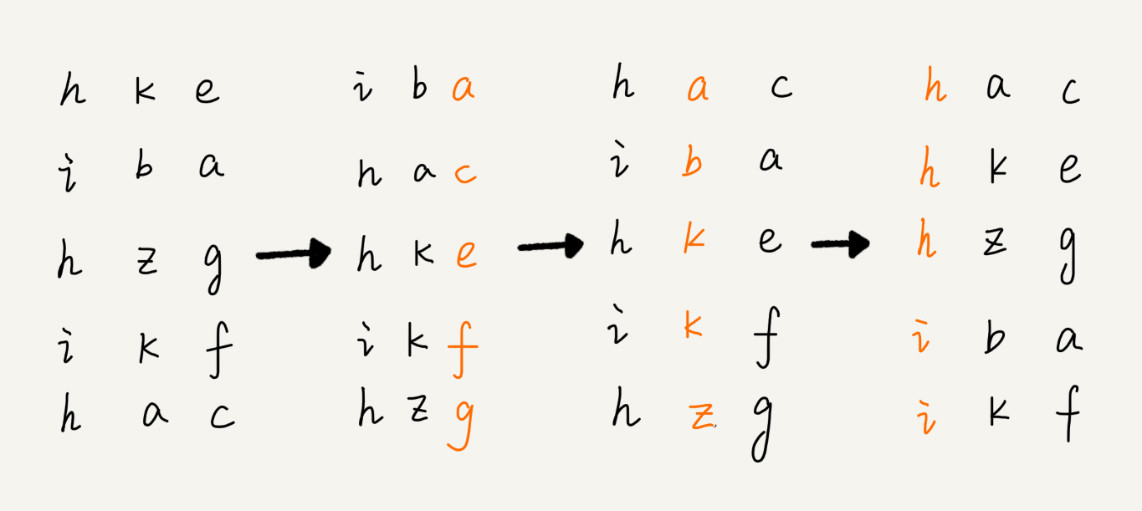
成绩统计

我个人觉得，计数排序其实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。



计数排序只能用在数据范围不大的场景中，如果数据范围 k 比要排序的数据 n 大很多，就不适合用计数排序了。而且，计数排序只能给非负整数排序，如果要排序的数据是其他类型的，要将其在不改变相对大小的情况下，转化为非负整数。

基数排序（Radix sort）



手机号码排序

注意，这里按照每位来排序的排序算法要是稳定的，否则这个实现思路就是不正确的。因为如果是非稳定排序算法，那最后一次排序只会考虑最高位的大小顺序，完全不管其他位的大小关系，那么低位的排序就完全没有意义了。

# 14 | 排序优化：如何实现一个通用的、高性能的排序函数？

## 如何选择合适的排序算法？



如果对小规模数据进行排序，可以选择时间复杂度是 O(n2) 的算法；如果对大规模数据进行排序，时间复杂度是 O(nlogn) 的算法更加高效。所以，为了兼顾任意规模数据的排序，一般都会首选时间复杂度是 O(nlogn) 的排序算法来实现排序函数。

 Java排序使用的是快排和优化后的归并排序

## 如何优化快速排序？

我们先来看下，为什么最坏情况下快速排序的时间复杂度是 O(n2) 呢？我们前面讲过，如果数据原来就是有序的或者接近有序的，每次分区点都选择最后一个数据，那快速排序算法就会变得非常糟糕，时间复杂度就会退化为 O(n2)。实际上，这种 O(n2) 时间复杂度出现的主要原因还是因为我们分区点选得不够合理。

1. 三数取中法我们从区间的首、尾、中间，分别取出一个数，然后对比大小，取这 3 个数的中间值作为分区点。这样每间隔某个固定的长度，取数据出来比较，将中间值作为分区点的分区算法，肯定要比单纯取某一个数据更好。但是，如果要排序的数组比较大，那“三数取中”可能就不够了，可能要“五数取中”或者“十数取中”
2. 随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。这种方法并不能保证每次分区点都选的比较好，但是从概率的角度来看，也不大可能会出现每次分区点都选得很差的情况，所以平均情况下，这样选的分区点是比较好的。时间复杂度退化为最糟糕的 O(n2) 的情况，出现的可能性不大

# 15 | 二分查找（上）：如何用最省内存的方式实现快速查找功能？

今天我们讲一种针对有序数据集合的查找算法：二分查找（Binary Search）算法，也叫折半查找算法。二分查找的思想非常简单，很多非计算机专业的同学很容易就能理解，但是看似越简单的东西往往越难掌握好，想要灵活应用就更加困难。

二分查找是我们目前为止遇到的第一个时间复杂度为 O(logn) 的算法。后面章节我们还会讲堆、二叉树的操作等等，它们的时间复杂度也是 O(logn)。我这里就再深入地讲讲 O(logn) 这种对数时间复杂度。这是一种极其高效的时间复杂度，有的时候甚至比时间复杂度是常量级 O(1) 的算法还要高效。为什么这么说呢？因为 logn 是一个非常“恐怖”的数量级，即便 n 非常非常大，对应的 logn 也很小。比如 n 等于 2 的 32 次方，这个数很大了吧？大约是 42 亿。也就是说，如果我们在 42 亿个数据中用二分查找一个数据，最多需要比较 32 次。我们前面讲过，用大 O 标记法表示时间复杂度的时候，会省略掉常数、系数和低阶。对于常量级时间复杂度的算法来说，O(1) 有可能表示的是一个非常大的常量值，比如 O(1000)、O(10000)。所以，常量级时间复杂度的算法有时候可能还没有 O(logn) 的算法执行效率高。