数据结构与算法之美——概要内容记录

入门篇：

**01 为什么要学习数据结构和算法？**

1.直接好处是能够有写出性能更优的代码。

2.算法，是一种解决问题的思路和方法，有机会应用到生活和事业的其他方面。

3.长期来看，大脑思考能力是个人最重要的核心竞争力，而算法是为数不多的能够有效训练大脑思考能力的途径之一。

（One day， you can write your own params）

**02 抓住重点，系统高效地学习数据结构与算法**

首先，算法和数据结构是为了解决实际问题的，解决的过程越简洁，使用的空间和时间越少，是可以认为这种方法月优秀的，所以入门的第一步，必须学会**复杂度分析**。

接下来是数据结构和算法的大致知识图谱(./algorithm-map)。

课程内容包括10个数据结构+10个算法：

10数据结构：数组、链表、栈、队列、散列表、二叉树、堆、跳表、图、Trie树

10算法：递归，排序，二分查找，搜索，哈希，贪心算法，分治，回溯，动态规划，字符串匹配。

学习的结构：来源，自身特点，适用问题类型，实际场景应用。

**03 复杂复分析(上)：如何分析、统计算法的执行效率和资源消耗**

1 为什么需要进行复杂度分析？

正常思路下，事后统计法（代码跑完，统计、监控时间和占用空间即可），但是这种方法局限性太大：

i 测试结果对于测试环境的极度依赖

ii 测试结果受数据规模影响很大

复杂度分析旨在**用粗略估计的方式来估算算法的执行效率**。

分为：时间复杂度T(n)&空间复杂度

2 复杂度表示——O

设每一行代码的运行时间为固定的unit\_time，那么对于整个代码块而言总的运行时间就有T(n) = *f*(n)\*unit\_time, 即T(n)和*f*(n)成正比，于是我们将整个结果进行统(zhu)一(jue)表(deng)示(chang)：T(n) = O( *f* (n) )。

可以从描述中得到大O时间复杂度表示的并不是代码真正的执行时间，二十表示代码执行时间随苏剧规模增长的变化趋势，因此叫做**渐进时间复杂度**，简称**时间复杂度**。

当n很大时，公式中低阶、常量以及高阶前的系数对于增长趋势的影响几乎是可以忽略的，故只需记录最大量级，即有了T(n2)这样的表达。

3 时间复杂度分析方法

1 分析一个算法、一段代码的时间复杂度的时候，只关注循环执行次数最多的那段代码就可以了。

2 加法法则：总复杂度等于量级最大的那段代码的复杂度。

3 乘法法则：嵌套代码的复杂度等于嵌套内外代码复杂度乘积。

几种常见时间复杂度实例分析

常量阶 O(1)

对数阶 O(logn)

线性阶 O(n) 多项式量级

线性对数阶 O(nlogn)

K次方阶 O(nk) (k≥2)

-------------------------------

指数阶 O(2n) 非多项式量级

阶乘阶 O(n!)

1. O(1):

常数级时间复杂度的一种表示方法，而不是实际只执行一行代码。一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行代码，时间复杂度也是O(1)。

2 O(logn)、O(nlogn):

示例代码如下：

i=1;

while (i <= n) {

**i = i \* 2;**

}

带有log的时间复杂度有明显的特点就是其本身在循环的时候存在一个不断加速的过程（上述代码粗体），因此代码的实际执行次数变成了求2*x*= *n*的过程，得出的结果即执行次数*x*。

而由于系数可忽略，故O(logn)中不指定底数。

3 O(m+n)、O(m\*n)

这是一类有两个数据的规模决定代码的时间复杂度

def cal(m, n):

sum\_1, sum\_2 = 0, 0

for i in range(m):

sum\_1 += i

for j in range(n):

sum\_2 += j

return sum\_1 + sum\_2

由于m, n规模未知（且互不相等），因此不能单纯的直接采用加法法则而是O(*f*(m)+*g*(n))

但是乘法规则继续有效。

4 空间复杂度分析

同时间复杂度一样，空间复杂度全称**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**。

一般的空间复杂度在初始化定义部分就能直接找到（数组列表之类，但是对于链表这一类需要另需注意），此后的代码部分只负责搬运数据进行计算。~~和申请空间毫无关系，当然这些操作仅对C系语言和java而言，对于python为首的胶水语言，这类方法就不好使了，python之中大部分代码间有很大的粘连性，因此单从代码的初始化~~

**04 复杂复分析(下)：浅析最好、最坏、平均、均摊时间复杂度**

1最好、最坏情况时间复杂度

引入的原因：当出现查找一类的逻辑时，代码的找寻效率出现波动。

这个时候需要引入最好情况的时间复杂度、最坏情况时间复杂度、平均情况时间复杂度。

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。

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。

而平均情况时间复杂度的计算需要考虑所有情况的概率，所以也叫加权平均时间复杂度

**很多时候，我们使用一个复杂度就可以满足需求了。只有同一块代码在不同的情况下，时间复杂度有量级的差距，我们才会使用这三种复杂度表示法来区分**。

2 均摊时间复杂度

两个条件满足时使用：1）代码在绝大多数情况下是低级别复杂度，只有极少数情况是高级别复杂度；2）低级别和高级别复杂度出现具有时序规律。均摊结果一般都等于低级别复杂度。

基础篇：

**05 数组：为什么很多编程语言中数组都从0开始编号？**

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。

☞部分关键词：

i 线性表：数据排成一条线的结构，每个线性表上的数据最多只有两个方向

代表结构：数组、链表、队列、栈等。

非线性表：数据之间的关系不是单纯的前后关系。

代表结构：二叉树、图、堆等。

ii 连续的内存空间和相同的数据类型

有了这两个条件，才能实现对内部数据的随机访问，同时也导致了数组的很多操作非常低效，如增删。

一般的数组寻址都是通过基址（定义时系统赋值）+偏值（就是数组的下标\*每个元素的大小（X-bit））的寻址方式：

a[i]\_address = base\_address + i \* data\_type\_size

这就是为什么看到的大部分数组，计数时是从0而不是1开始。

数组支持随机访问，根据下标随机访问的而时间复杂度为O(1)。

有效的“增”“删”：

增：

如果按照传统意义，当向数组中某位置插入一个新的元素，这个时候后面的每一个个体都需要后移1位。而简单的办法则是在没有要求有序的情况下，可以允许插入位置的数据直接移动到末位添加，而原位置则占用其原本的位置。

删：

同样的处于连续性的考量，会需要搬运数据，而无序时，直接通过增加记录的方式表示该位置的数据已经被删除，即无法索引，而当没有更多存储空间时，则触发真正的删除操作，这样就大大减少了删除操作导致的数据搬移。

数组越界问题：

1 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。

2 java中的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ArrayList，我们就完全不需要关心底层的扩容逻辑，ArrayList 已经帮我们实现好了。每次存储空间不够的时候，它都会将空间自动扩容为 1.5 倍大小。

数组的用武之地：

1.Java ArrayList 无法存储基本类型，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。

2. 如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。

3. 还有一个是我个人的喜好，当要表示多维数组时，用数组往往会更加直观。比如 Object[][] array；而用容器的话则需要这样定义：ArrayList > array。

06 链表(上)： 如何实现LRU缓存淘汰算法

概念先导：

1 缓存：为提高数据读取性能的计数，在硬件设计、如那件开发中都有着非常广泛的应用，常见CPU缓存，数据库缓存、浏览器缓存。

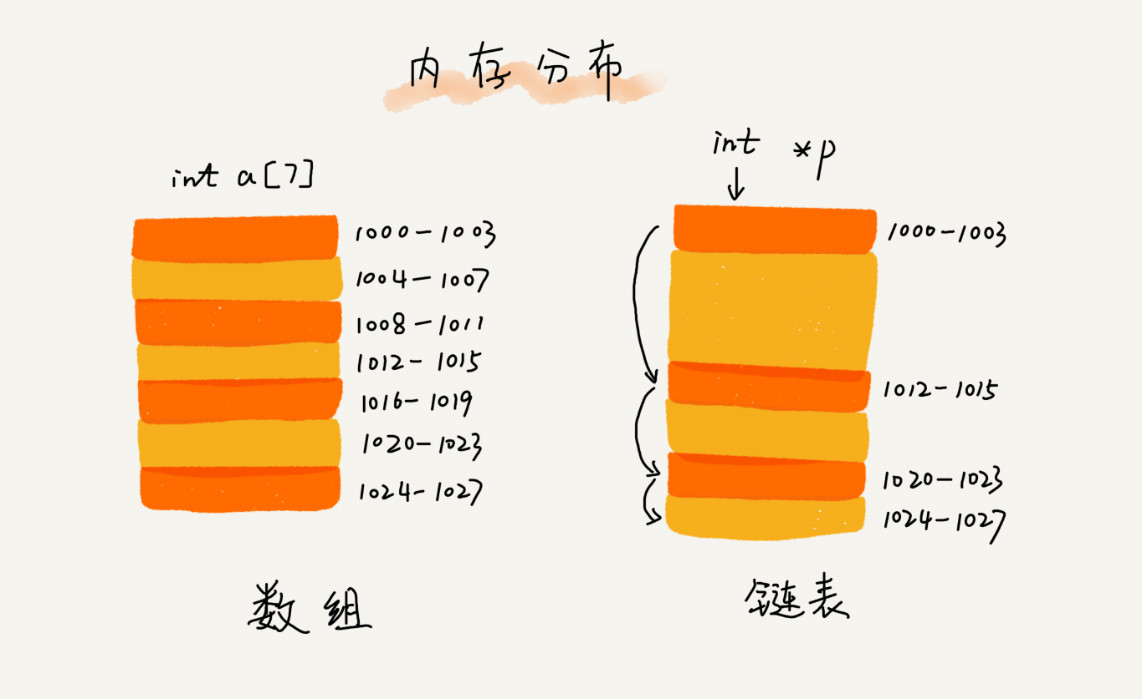
2 缓存淘汰策略：当缓存被用满时，如何判断数据的去留就由设定好的淘汰策略来实现。常见的淘汰策略：FIFO(先进先出)，LFU(最少使用)，LRU(最近最少使用)

五花八门的链表结构

相比数组，链表较为复杂（指针连接结构）

与数组对比：

i底层存储结构上：



数组向内存申请的是一块连续的存储空间，如果（内存）没有（足够大小的连续存储空间）直接就申请失败了。

链表则不同，通过指针将一组零散的内存块串联起来使用，一般不存在申请失败的情况。

常见的3种链表结构：单~，双向~，循环~。

单（向）链表：（结点：data + 后继指针next）

头节点（基址）与尾节点（空址NULL）：

插入删除O(1)——改变指针即可。

查找O(n)——从头部一个一个查找。

环形链表：尾节点指向头结点数据基址。

双向链表（结点：前驱指针prev + data + 后继指针next）

占用更多空间，某些情况下的插入、删除操作比单链表还要简单高效。

LRU问题：

维护一个单向有序链表即可，越接近尾部，表示访问时间越早，当一个新数据被访问时：

2 数据不在缓存链表中：

链表已满，删尾增头

链表未满，直接插入头结点

1 数据遍历，发现有该数据，删除+插头

07 链表(下)：手写链表代码的正确姿势

1 指针的含义和各语言中的定义方式

将某个变量赋值给指针，实际上就是将这个变量的地址赋值给指针，或者反过来说，指针中存储了这个变量的内存地址，指向了这个变量，通过指针就能找到这个变量。

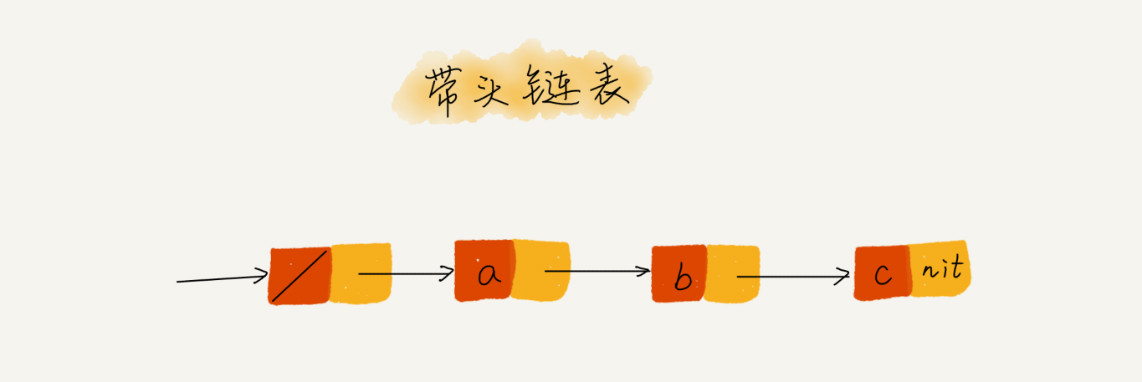
2 警惕指针丢失和内存泄漏

插入结点时，一定要注意操作的顺序，要先将结点 x 的 next 指针指向结点 b，再把结点 a 的 next 指针指向结点 x

删除链表结点时，也一定要记得手动释放内存空间，否则，也会出现内存泄漏的问题。当然，对于像 Java 这种虚拟机自动管理内存的编程语言来说，就不需要考虑这么多了。

3 利用哨兵简化实现难度

针对链表的插入、删除操作，需要对插入第一个结点和删除最后一个结点的情况进行特殊处理。如果我们引入哨兵结点，在任何时候，不管链表是不是空，head 指针都会一直指向这个哨兵结点。



4 留意边界处理条件

我经常用来检查链表代码是否正确的边界条件有这样几个：

如果链表为空时，代码是否能正常工作？

如果链表只包含一个结点时，代码是否能正常工作？

如果链表只包含两个结点时，代码是否能正常工作？

代码逻辑在处理头结点和尾结点的时候，是否能正常工作？

05 举例画图、辅助思考

**06 多写多练，没有捷径**

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