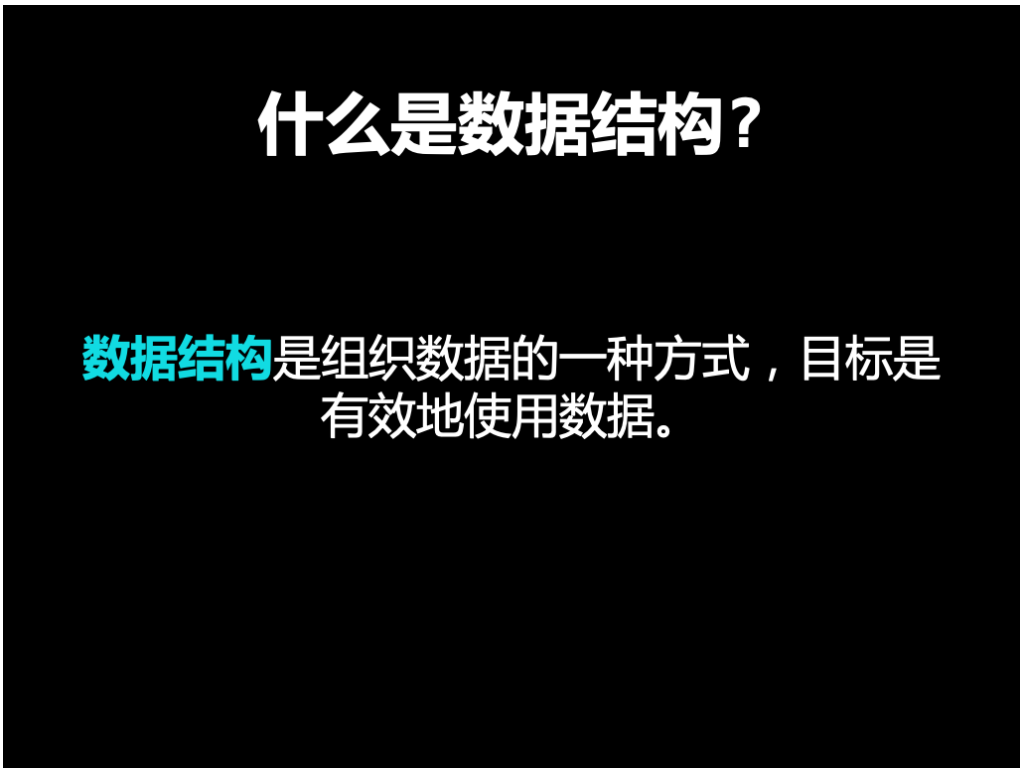
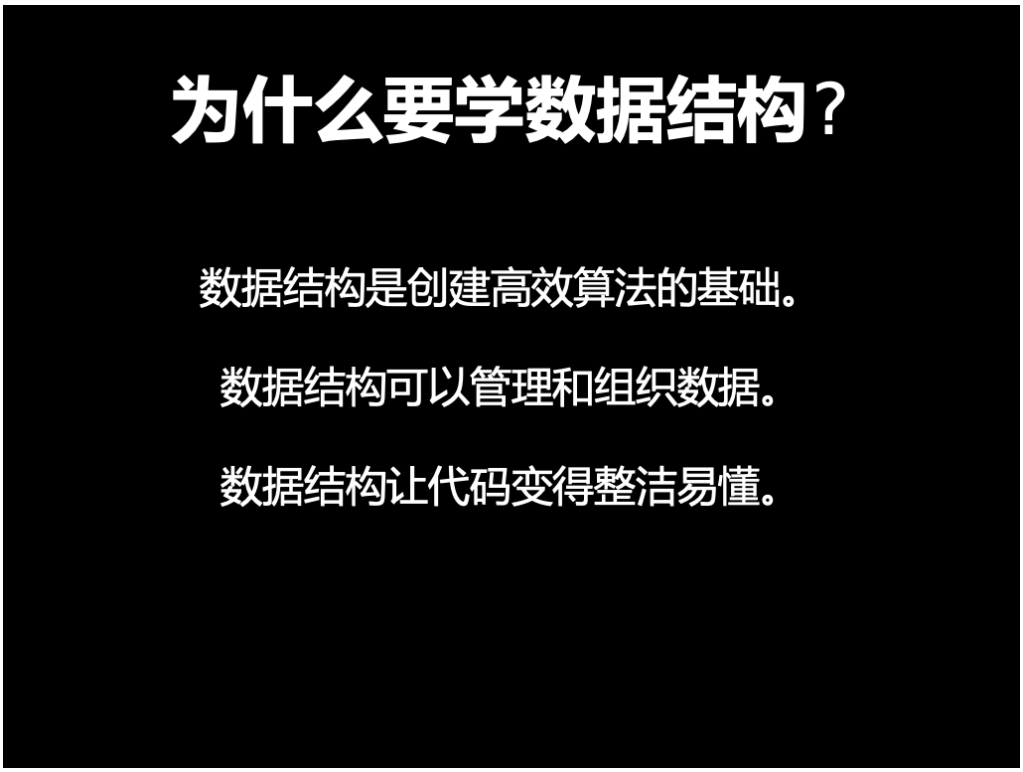
你好，欢迎来到波波新推出的数据结构课程。在前面的几个视频中，我会解释一些核心概念，理解这些概念，可以为你学习后续的课程打下一个良好的基础，因为在我们后续的课程中，我们会不断地应用到这些概念。好，让我们从基础开始。



那么什么是数据结构呢？我比较喜欢的一个定义是这样的：

数据结构是组织数据的一种方式，它的目标是有效的使用数据。

这就是数据结构的定义，它是一种组织数据的方式，方便我们后续有效地去访问、查询或者更新数据。



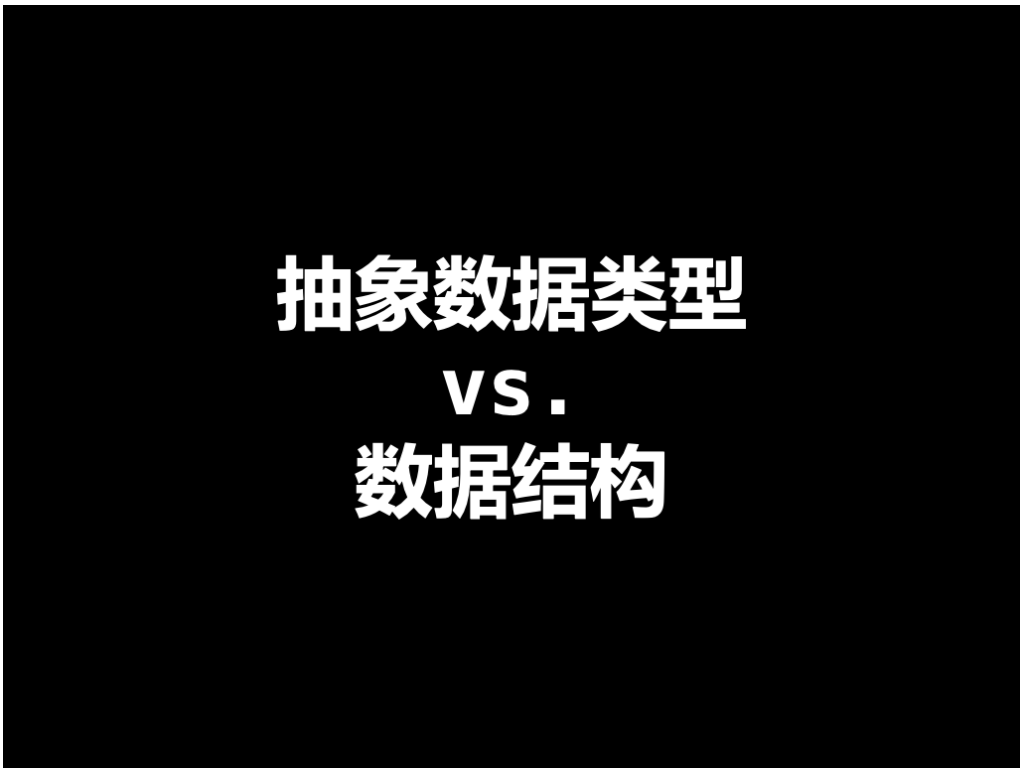
知道了什么是数据结构，那么为什么数据结构如此重要？我们为什么要学数据结构呢？

我这边总结了三点：

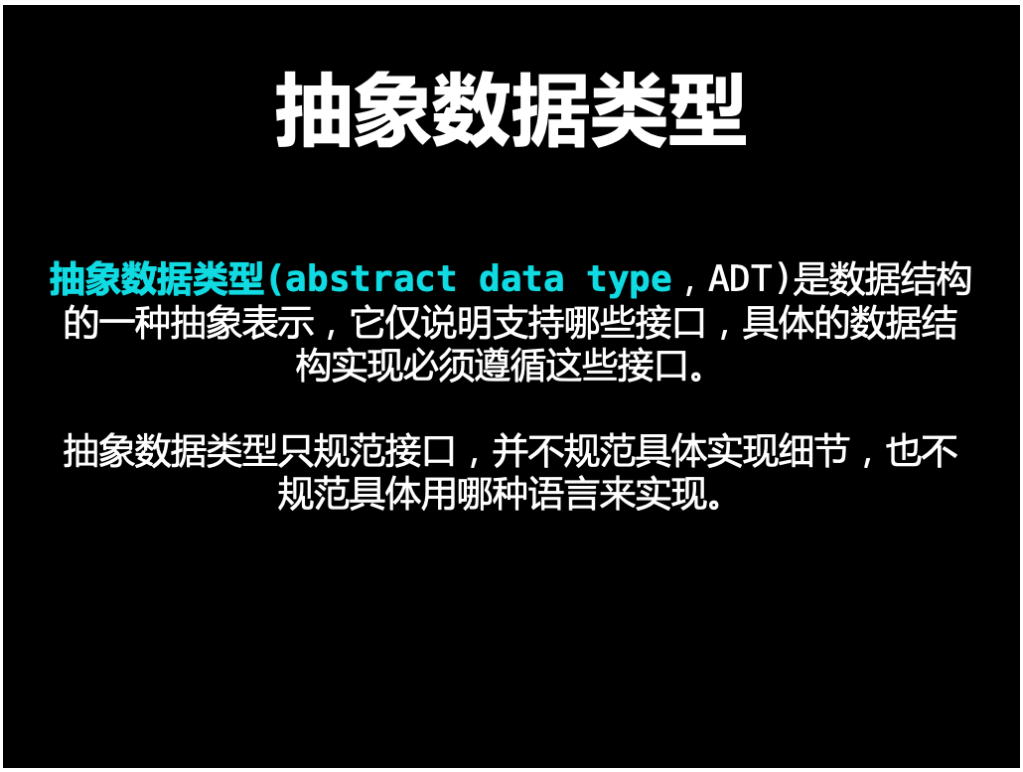
第一点是，数据结构是创建高效算法的基础。
第二点是，数据结构可以帮助我们管理和组织数据。

第三点是，数据结构让代码变得整洁清晰，并且易于理解。

顺便提一下，我发现优秀程序员和不合格的，或者一般程序员之间的主要差异在于，优秀程序员能够合理的选择和应用数据结构，来解决他们手头的编程任务。可以说，数据结构就是普通程序员和优先程序员之间的差异所在。这也说明了为什么每一个计算机专业的本科生都需要学习数据结构这门课程。



在正式讲解数据结构之前，我们先要来讲一下数据结构的抽象。这个说法本身有点抽象，其实我要讲的是抽象数据类型这个概念。



什么是抽象数据类型？它和数据结构有什么区别？好的，让我来回答这些问题：

抽象数据类型(英文称为abstract data type，简称ADT)，它是数据结构的一种抽象表示，它仅仅说明这个类型支持哪些接口，具体的数据结构实现必须遵循这些接口。

抽象数据类型只规范接口，并不规范具

体的实现细节，也不规范具体采用哪种语言来实现。

举一个例子，说到这个ADT，我经常将它比喻为从地点A到地点B的运输方式。我们都知道，从地点A到地点B的运输方式有很多。一些具体的运输方式包括像走路，开车，或者通过火车等等。这些具体的运输方式就好比是数据结构。

下面我们来看一些例子。

PPT的左边有一些抽象数据类型的例子，右边是具体的底层实现。比方说，列表List可以有两种实现方式，分别采用动态数组实现，或者采用链表来实现。它们都支持在列表中添加，移除和索引定位元素。

再比如队列Queue和字典Map抽象数据类型，它们都可以有多种实现方式。注意，在队列Queue的实现方式中，我添

加了基于栈的队列实现，实际上，我们确实可以用栈Stack来实现队列，虽然这种实现方式效率很差。

最后，我还在表中加了交通工具Vehicle，这个只是为了形象说明，如果交通工具是一种抽象，那么它可以有多种具体的实现，包括高尔夫球车，自行车，还有智能汽车。这些具体的交通工具都可以移动，转弯，或者停车，等等。

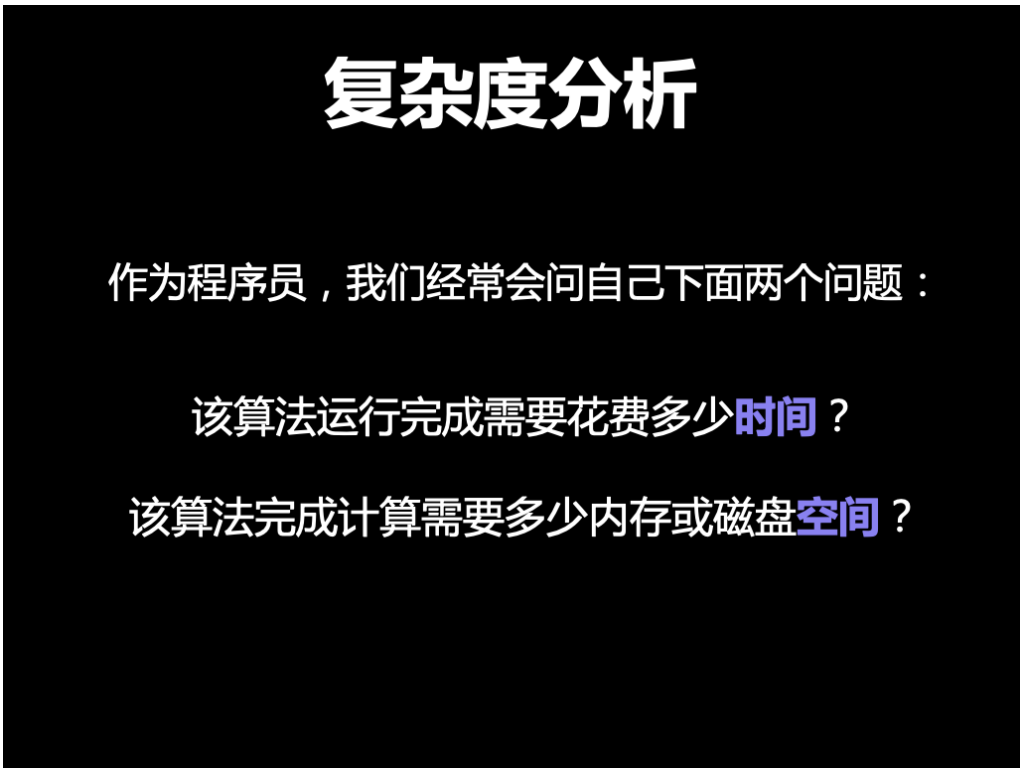
在实际应用中，人们可能会经常会混用数据结构和抽象数据类型这两个概念，比方说，某个人说Map，另外一个人说hash map，他们可能说得是同一个概念，但是经过我的解释，你现在应该可以理解两者的细微差异。

这里再强调一下，抽象数据类型仅定义它

支持哪些行为，或者说方法，但是它并不定义这些方法具体是如何实现的。

好的，第一节课就讲这些内容，后续还有更多内容等着你学习。下节课我们会讲计算复杂度和Big O标记，我们下节课再见。

好的，既然我们已经学过抽象数据类型，那么下面我们来进一步学习一下计算复杂度这个概念。理解计算复杂度，可以帮助我们分析算法的执行性能。

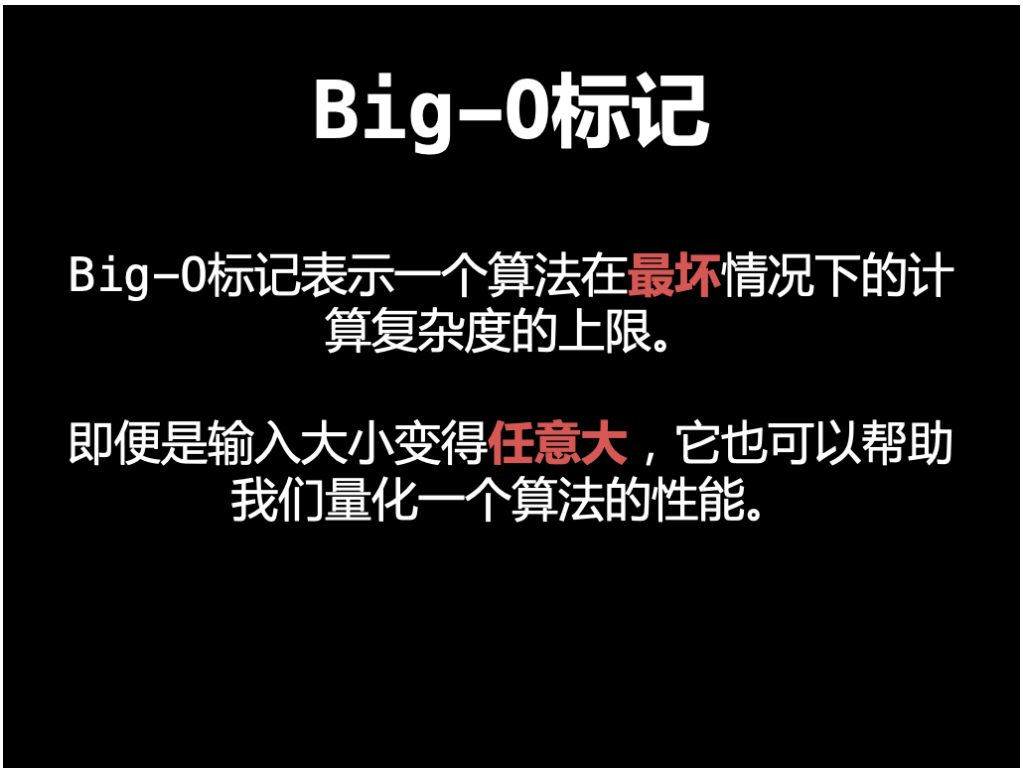


作为程序员，我们经常会问自己两个问题：

第一个问题是：这个算法运行完成需要花费多少时间？
第二个问题是：这个算法完成计算的话，需要多少内存或者磁盘空间？

如果你的程序在宇宙运行终结之前也运行不完，那它就基本没有什么用。或者，

虽然你的程序可以在常量时间内运行完成，但是它要占用巨大的空间，相当于互联网上所有文件字节数的总和，那么你的程序也没有什么用处。



为了能够量化说明算法运行的时间和空间需求，理论计算机科学家发明了Big-O标记(也称为大O标记)。当然还有big theta和big omega标记。但是我们这门课只关注big O，因为它表达的是最坏情况。

Big O标记只关注最坏情况，比方说，假定你有一个排序算法，如果我们用Big O分析它的复杂度，那么Big O关注

的是输入顺序最极端的一种情况。再举个例子，假如现在有一个数字列表，列表中每一个数字都是唯一的，并且这个列表是乱序的，然后你从列表的第一个元素开始，以顺序查找的方式查找数字7的索引位置，那么最坏情况的输入，7既不是出现在头部，也不是出现在中间，而是出现在最后一个位置。顺序查找方式的复杂度是线性的，它随着数组大小的增长而增长，因为你需要顺序查找数组中的每一个元素，直到找到7为止。同样的概念也适用于空间，你需要考虑到，对于任意的输入，我的算法所需要的内存或者磁盘空间，在最坏的情况下是多少。

另外，Big O真正关注的是当输入变得非常庞大的时候，你的算法的表现，对于小规模的输入，它并不关心。因此，对于在Big O后面增加的常量，或者在Big O前面添加

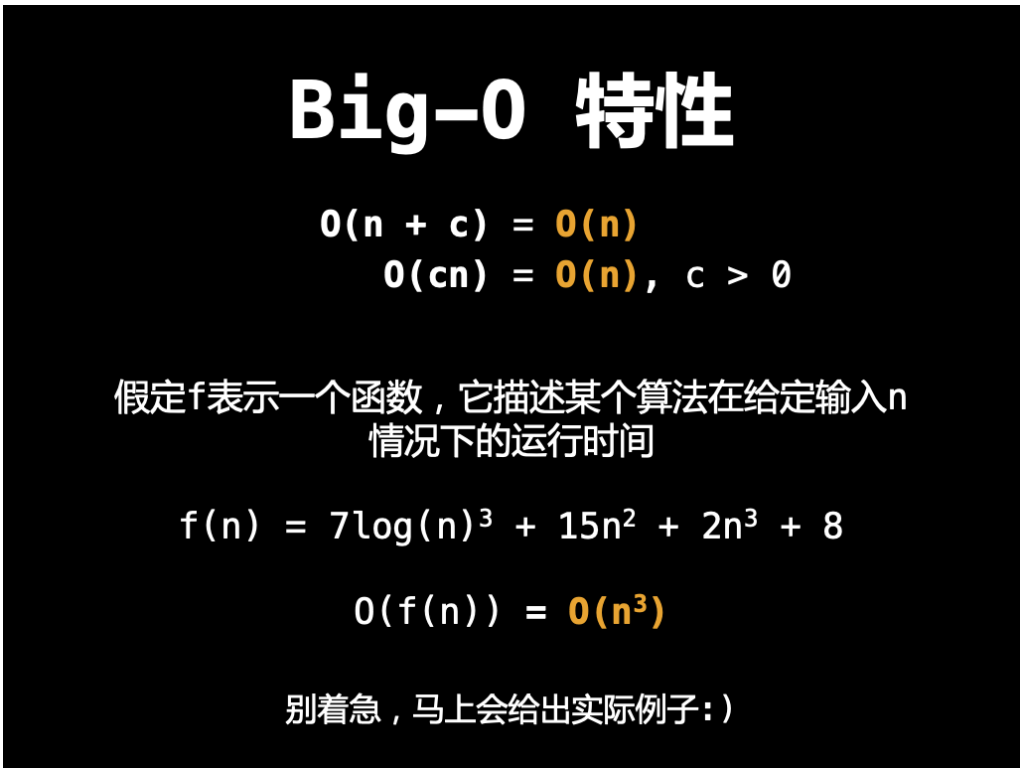
的乘数常量，这些常量都可以被忽略，

PPT上给出了一些常见的算法复杂度的Big O表示，其中n表示算法输入的大小。未来，在你的整个编程职业生涯中，你将会经常见到这些Big O表示。

1. 如果你的程序需要常量时间运行完成，那么它的复杂度是O(1)。
2. 如果你的算法需要对数级时间运行完成，那么的的复杂度是O(log(n))。

3. 同样的，线性级时间，对应O(n)。
4. 线性对数级(英文是Linearithmic)，对应O(nlog(n))。
5. 平方级，对应$O(n^2)$。
6. 立方级，对应$O(n^3)$。
7. 指数级，对应$O(b^n)$，其中b大于1。
8. 阶乘级，对应O(n!)。

需要提一下，除了上面这些常见的复杂度，中间还有一些复杂度，比方说n开二次方根，loglogn，还有n的5次方等等。实际上，大部分包含n的数学表达式，都可以用Big O标记来表达，并且是Big O合法的。

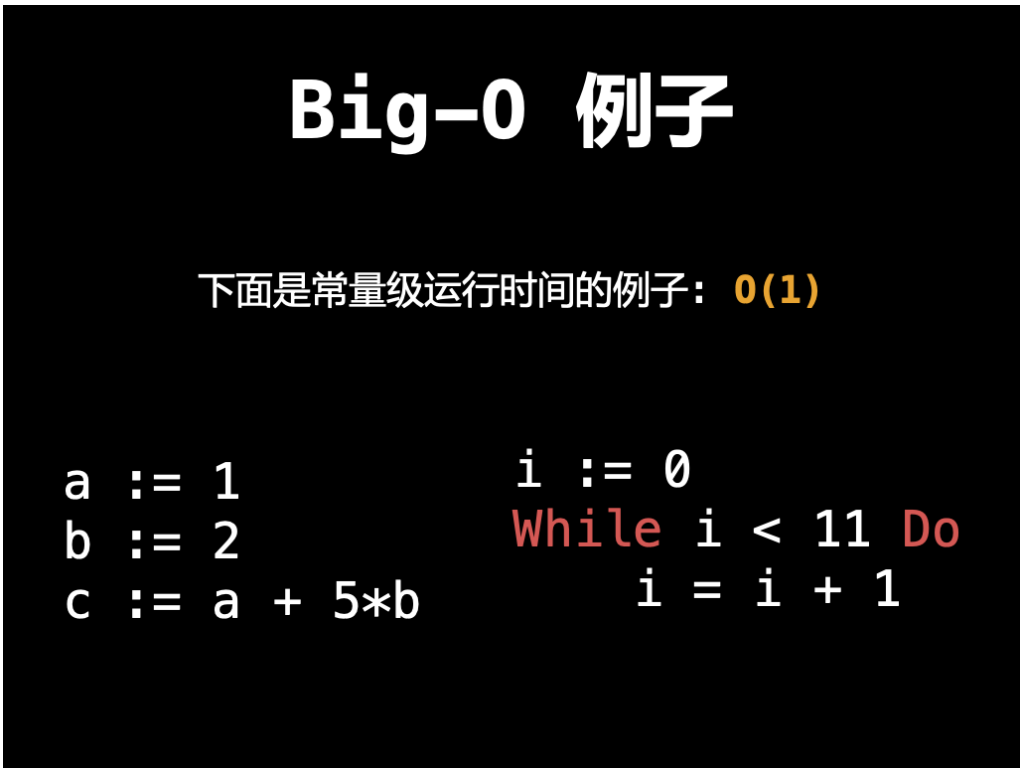


下面我们来学习Big-O的一些特性。先重申一下前面两个slide中我们所讲的内容，Big O仅仅关注输入变得非常巨大的情况，换句话说，它仅关注当输入值n变成无限大的时候，算法的复杂度状况。于是，我们可以得出Big O的前面两个特性。第一个特性是说，我们可以忽略在Big O标记中添加的常量，比方说PPT上的O(n + c)中的c。因为这里的n才是我们的输入大小，并且它是会变

的，所以当n变得非常大的时候，c的值一直是固定的，当n趋向无限大，c将最终消失，所以可以被忽略。其次，对于n前面的乘数常量也是一样的，即便c非常大也一样，当n的值趋向于无穷大的时候，c的值就会变得无足轻重。当然这些都只是理论上的，在实际场景中，当c=200亿的时候，它对你的算法的影响可能也是巨大的。

下面我们来看一个函数的例子，$f(n) = 7\log(n)^3 + 15n^2 + 2n^3 + 8$，在这个函数表达式中，当输入n变得非常大的时候，对f(n)的结果起决定性作用的项是$2n^3$，因为它是最大的项，同时我们知道前面的乘数2是可以忽略的，因此这个函数的最终复杂度是$O(n^3)$，也就是立方级复杂度。

那么，这个还不是实际的例子，别着急，我们马上会给出实际例子:)



下面我们来看一些Big-O应用的具体例子。

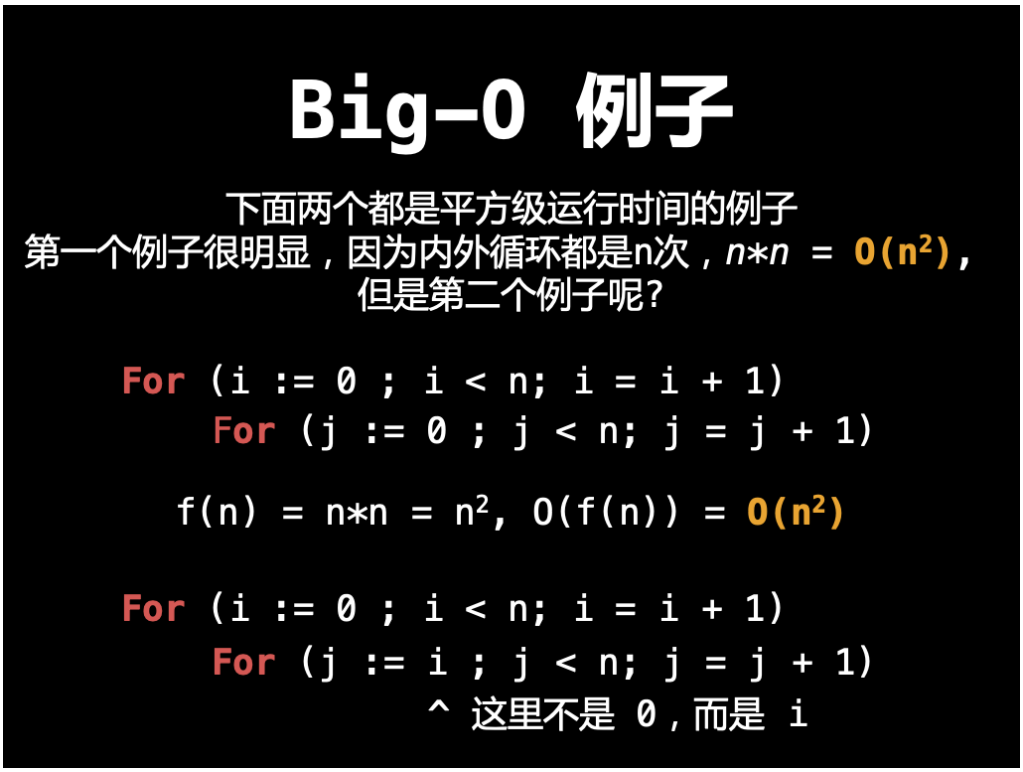
PPT上的两段代码都是常量级运行时间的例子，因为它们根本就不依赖于输入n。左边的例子是简单的算术运算，不依赖于n，所以肯定是常量运行时间。右边的例子虽然是一个循环，但是循环大小是固定的，所以也是常量运行时间。

下面来看两个线性级运行时间的例子。

左边是一个循环，i从0到n，每次将i的值加一，所以总共循环次数是n次，也就是说f(n) = n，它的运行复杂度是O(n)。

右边也是一个循环，i也是从0到n，但是每次i的值加3，所以总计运行的次数是n/3，也就是说f(n) = n/3，但是乘数1/3是可以忽略的常量，所以它的运算复杂

度是O(n)。

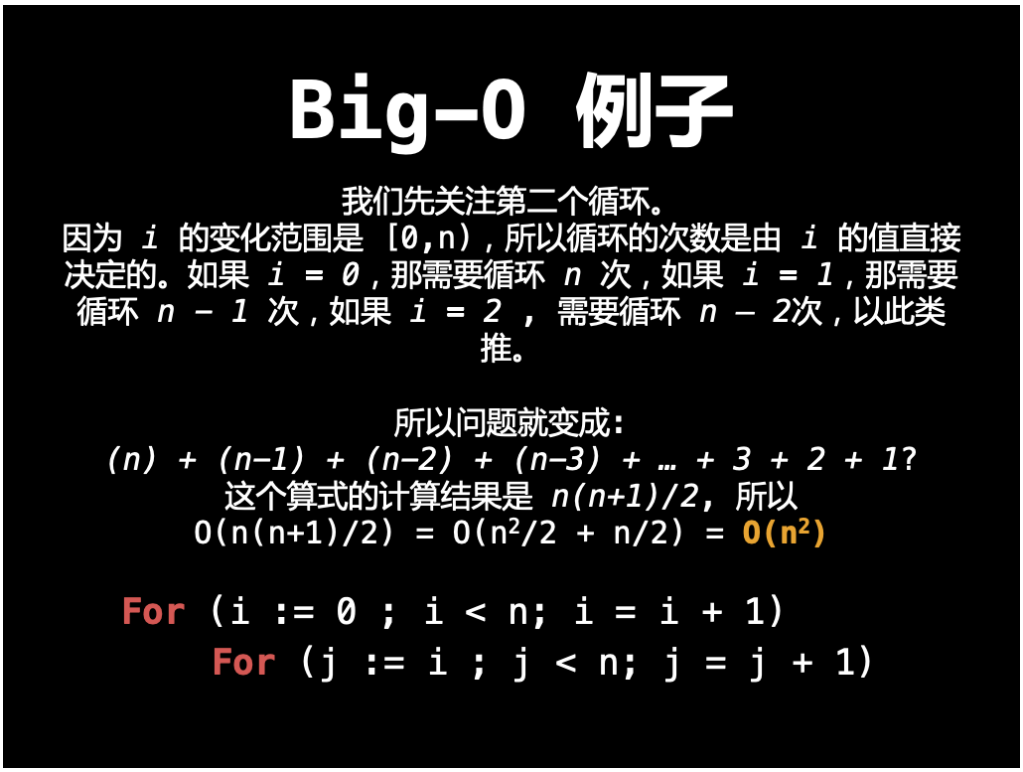


下面我们来看两个平方级运行时间的例子。

第一个例子很明显，因为内外循环都是n次，n * n 的复杂度就是 $O(n^2)$。

第二个例子稍微复杂一点，注意第二个循环，它的开始循环变量j的值不是0，而是i。波波建议你不妨暂停视频，先自己思考一下，这个算法的运行时间复杂

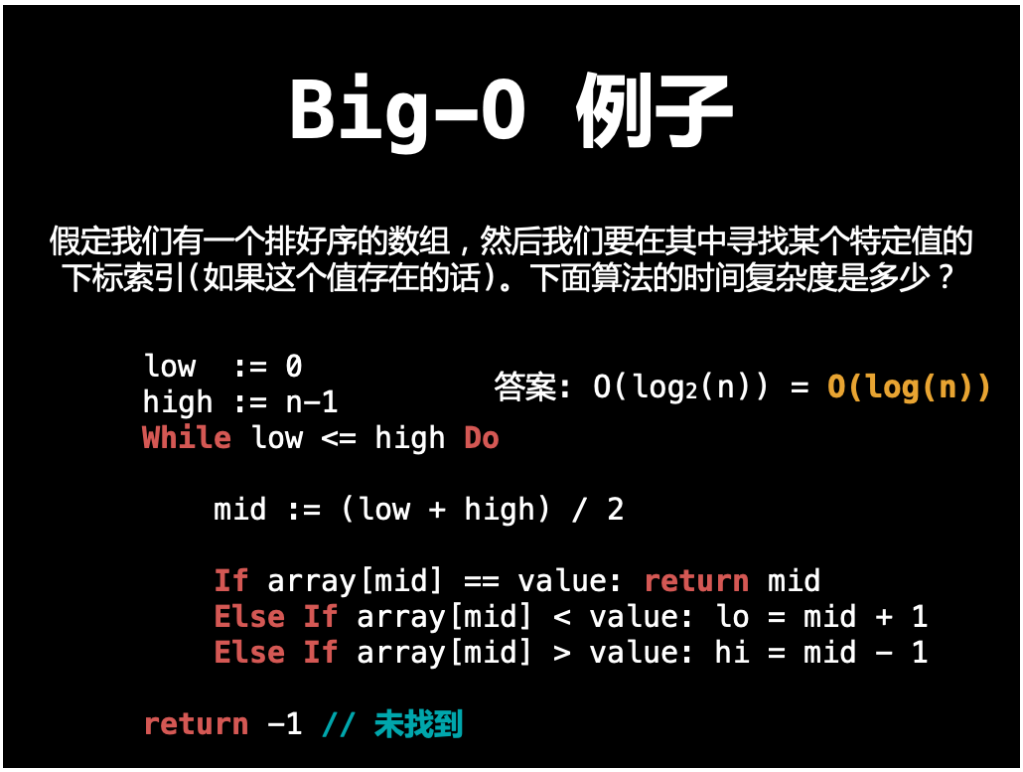
度是多少？



好，我们来解答这个问题。

我们主要关注第二个循环，因为第一个循环是很明显的。在第二个循环中，i的变化范围是从0到n，包括0，但是不包括n。所以循环的次数是由i的值直接决定的，如果i的值为0，那么就需要循环n次，如果i的值为1，那么就需要循环n-1次，如果i=2，那么就需要循环n-2次，以此类推。

所以这个问题就变成求：(n) + (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1，如果大家上过高中，学过数列运算，那么应该可以知道这个算式的计算结果是n(n+1)/2，也就是$n^2/2$ + n/2，忽略乘数常量1/2和后面的n/2，它的计算复杂度是$O(n^2)$。



下面是一个更复杂一点的例子。前面我们介绍过一些常见的算法复杂度，其中有对数(logarithmic)和线性对数(linearithmic)时间复杂度，你可能会问这两种复杂度有哪些具体例子。这里我就给出一个二分搜索算法的例子，这是一个非常经典的算法，时间复杂度是对数级的。

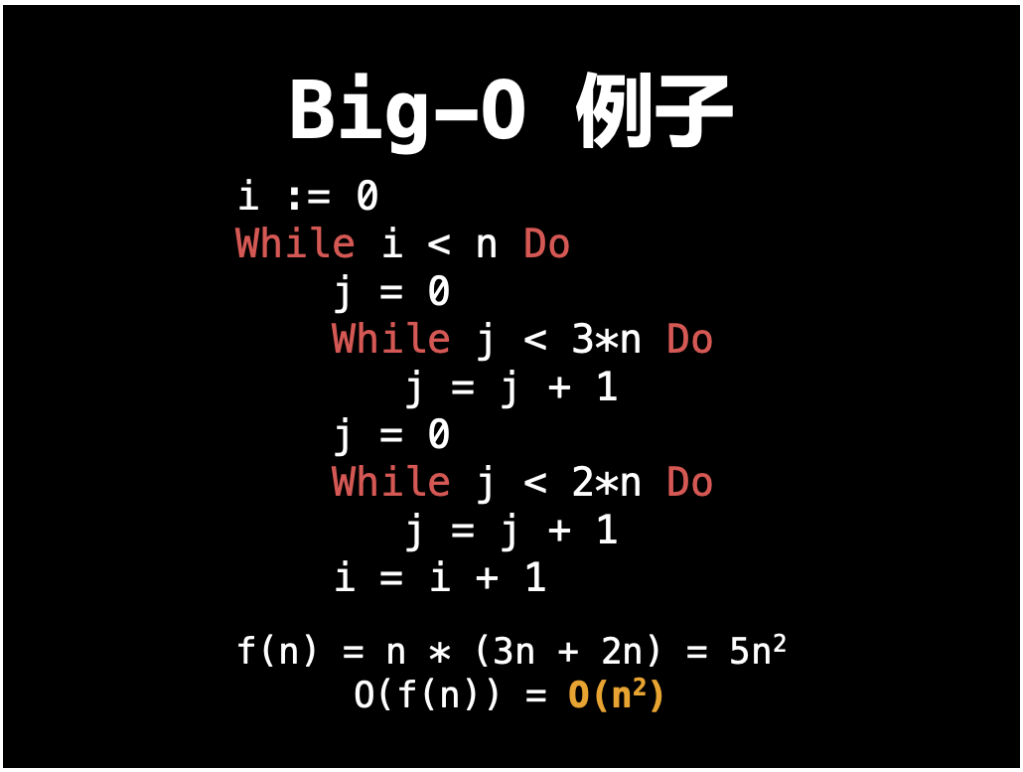
假定我们有一个已经排好序的数组，然

后我们要在其中寻找某个特定值的下标索引(如果这个值存在的话)。看下面这个算法，思考一下，它的算法复杂度是多少？

这个算法是这样工作的，我们先用两个指针变量low和high分别指向数组的两端，刚开始low = 0, high = n - 1。然后在两者之间找到一个中点mid，看这个中点的值是否是我们要找的值，如果找到就返回这个mid，如果没有找到，那么根据中点的值和要找的值的大小，我们可以判断要找的值是在左边还是右边的子数组当中，因为整个数组是排序的。然后我们可以忽略肯定不存在我们要找的值的那个子数组，然后相应地调整low和high指针变量的值，去另外一半子数组继续查找。利用这个算法思路一直查找下去，直到找到，或者找不到为止。当下标指针low>high的时候，就说明找不到。利用数学我们可以证明，对于任意输

入长度为n的数组，要查找的次数的最坏情况是$\log_2(n)$次，也就是说二分搜索算法的时间复杂度是O(log(n))。

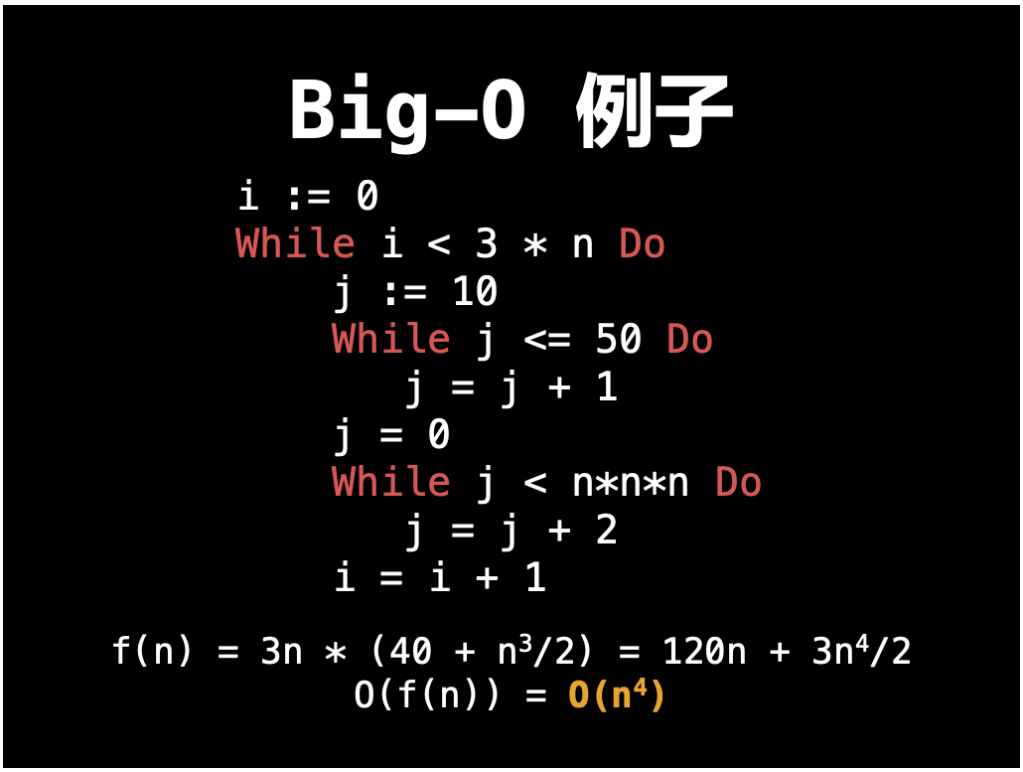
两分搜索算法是一个经典和优雅的算法，在实践中也经常会用到，希望大家记住它的算法和运算复杂度。



这里有一个看似有点复杂的例子，值得我们研究一下。首先，这个算法的外层循环的执行次数是n次。其次，这个算法的内层循环有两个，循环次数分别是3n和2n。要计算这个算法的时间复杂度，一般的规则是：

1. 不同层级的循环是相乘的关系；
2. 同一层级的循环是相加的关系；

根据这两条规则，我们可以得出算法的总的运算次数$f(n) = n * (3n + 2n) = 5n^2$，忽略乘数常量5，最后它的时间复杂度是$O(n^2)$。



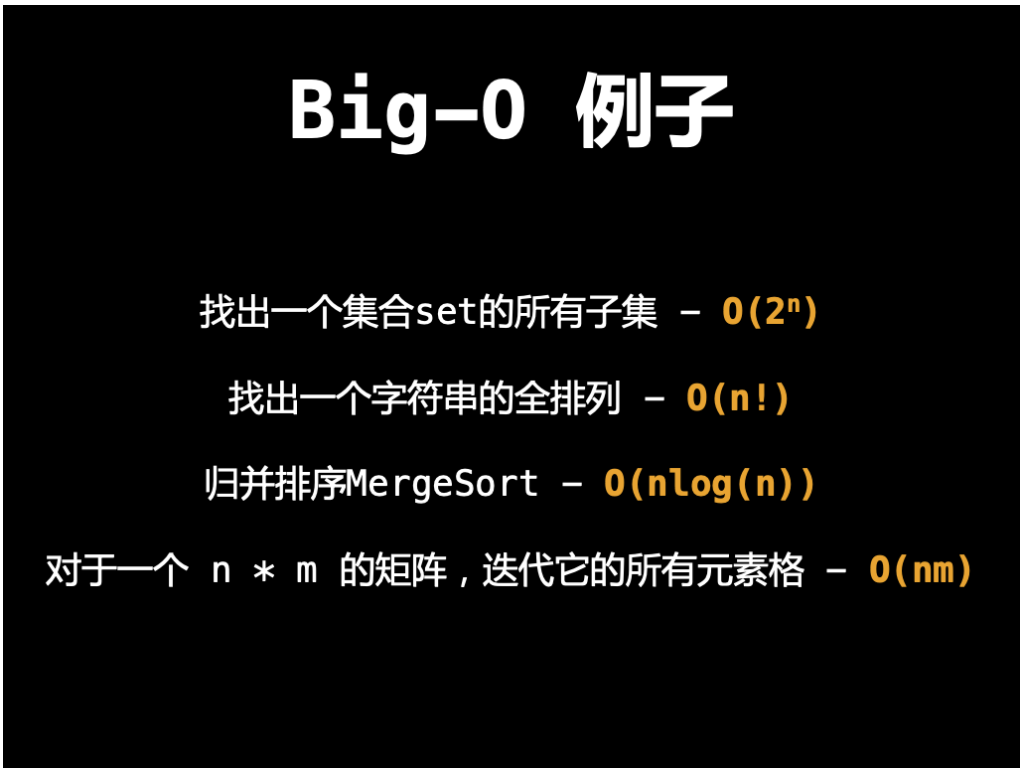
下面这个例子和前面的例子看上去有点像，但是其实是不一样的。

它的外层循环总共运行3 * n次，这个很明显。

它的内层循环有两个，上面一个，仔细看一下的话，运行次数是40次。下面一个稍微复杂一点，循环变量j界限是n * n * n，也就是n^3，但是j每次是增加2的，

所以总计循环次数是 $n^3/2$ 次。

同样，根据前面的提到的两条规则，我们可以得出$f(n) = 3n * (40 + n^3/2) = 3n/40 + 3n^4/2$，忽略小头120n，也忽略乘数常量3/2，最后得到该算法的复杂度是$O(n^4)$。



最后还有一些Big-O的例子，我这里简单说明一下：

1. 找出一个集合set的所有子集，复杂度是O(2ⁿ)。
2. 找出一个字符串的全排列，复杂度是O(n!)。
3. 归并排序MergeSort，复杂度是O(nlog(n))。
4. 最后，对于一个n * m的矩阵，如果

要迭代它的所有元素格，那么时间复杂度是O(nm)。

好，本节课算法复杂度和Big-O标记，我们就讲到这里，下节课我们来讲静态和动态数组，我们下节课再见！