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？

2018-09-26 王争

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？
朗读人：修阳 19′42′′ | 9.04M

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。所以，执行效率是算法一个非常重要的考量指标。那如何来衡量你编写的算法代码的执行效率呢？这里就要用到我们今天要讲的内容：时间、空间复杂度分析。

其实，只要讲到数据结构与算法，就一定离不开时间、空间复杂度分析。而且，我个人认为，**复杂度分析是整个算法学习的精髓，只要掌握了它，数据结构和算法的内容基本上就掌握了一半。**

复杂度分析实在太重要了，因此我准备用两节内容来讲。希望你学完这个内容之后，无论在任何场景下，面对任何代码的复杂度分析，你都能做到“庖丁解牛”般游刃有余。

为什么需要复杂度分析？

你可能会有些疑惑，我把代码跑一遍，通过统计、监控，就能得到算法执行的时间和占用的内存大小。为什么还要做时间、空间复杂度分析呢？这种分析方法能比我实实在在跑一遍得到的数据更准确吗？

首先，我可以肯定地说，你这种评估算法执行效率的方法是正确的。很多数据结构和算法书籍还给这种方法起了一个名字，叫**事后统计法**。但是，这种统计方法有非常大的局限性。

1. 测试结果非常依赖测试环境

测试环境中硬件的不同会对测试结果有很大的影响。比如，我们拿同样一段代码，分别用 Intel Core i9 处理器和 Intel Core i3 处理器来运行，不用说，i9 处理器要比 i3 处理器执行的速度快很多。

还有，比如原本在这台机器上 a 代码执行的速度比 b 代码要快，等我们换到另一台机器上时，可能会有截然相反的结果。

2. 测试结果受数据规模的影响很大

后面我们会讲排序算法，我们先拿它举个例子。对同一个排序算法，待排序数据的有序度不一样，排序的执行时间就会有很大的差别。极端情况下，如果数据已经是有序的，那排序算法不需要做任何操作，执行时间就会非常短。除此之外，如果测试数据规模太小，测试结果可能无法真实地反应算法的性能。比如，对于小规模的数据排序，插入排序可能反倒会比快速排序要快！

所以，**我们需要一个不用具体的测试数据来测试，就可以粗略地估计算法的执行效率的方法**。这就是我们今天要讲的时间、空间复杂度分析方法。

大 O 复杂度表示法

算法的执行效率，粗略地讲，就是算法代码执行的时间。但是，如何在不运行代码的情况下，用“肉眼”得到一段代码的执行时间呢？

这里有段非常简单的代码，求 1,2,3..n 的累加和。现在，我就带你一块来估算一下这段代码的执行时间。

复制代码

```
1 int cal(int n) {
2   int sum = 0;
3   int i = 1;
4   for (; i <= n; ++i) {
5     sum = sum + i;
6   }
7   return sum;
8 }
```

从 CPU 的角度来看，这段代码的每一行都执行着类似的操作：**读数据-运算-写数据**。尽管每行代码对应的 CPU 执行的个数、执行的时间都不一样，但是，我们这里只是粗略估计，所以可以假设每行代码执行的时间都一样，为 unit_time。在这个假设的基础之上，这段代码的总执行时间是多少呢？

第 2、3 行代码分别需要 1 个 unit_time 的执行时间，第 4、5 行都运行了 n 遍，所以需要 2n*unit_time 的执行时间，所以这段代码总的执行时间就是 (2n+2)*unit_time。可以看出来，所有代码的执行时间 T(n) 与每行代码的执行次数成正比。

按照这个分析思路，我们再来看这段代码。

复制代码

```
1 int cal(int n) {
2   int sum = 0;
3   int i = 1;
4   int j = 1;
5   for (; i <= n; ++i) {
6     j = 1;
7     for (; j <= n; ++j) {
8       sum = sum +  i * j;
9     }
10   }
11 }
```

我们依旧假设每个语句的执行时间是 unit_time。那这段代码的总执行时间 T(n) 是多少呢？

第 2、3、4 行代码，每行都需要 1 个 unit_time 的执行时间，第 5、6 行代码循环执行了 n 遍，需要 2n * unit_time 的执行时间，第 7、8 行代码循环执行了 n^2遍，所以需要 $2n^2$ * unit_time 的执行时间。所以，整段代码总的执行时间 T(n) = $(2n^2+2n+3)$*unit_time。

尽管我们不知道 unit_time 的具体值，但是通过这两段代码执行时间的推导过程，我们可以得到一个非常重要的规律，那就是，所有代码的执行时间 T(n) 与每行代码的执行次数 n 成正比。

我们可以把这个规律总结成一个公式。注意，大 O 就要登场了！

$$T(n) = O(f(n))$$

我来具体解释一下这个公式。其中，T(n) 我们已经讲过了，它表示代码执行的时间；n 表示数据规模的大小；f(n) 表示每行代码执行的次数总和。因为这是一个公式，所以用 f(n) 来表示。公式中的 O，表示代码的执行时间 T(n) 与 f(n) 表达式成正比。

所以，第一个例子中的 T(n) = O(2n+2)，第二个例子中的 T(n) = $O(2n^2+2n+3)$。这就是大 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。

当 n 很大时，你可以把它想象成 10000、100000。而公式中的低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略。我们只需要记录一个最大量级就可以了，如果用大 O 表示法表示刚讲的那两段代码的时间复杂度，就可以记为：T(n) = O(n)； T(n) = $O(n^2)$。

时间复杂度分析

前面介绍了大 O 时间复杂度的由来和表示方法。现在我们来看下，如何分析一段代码的时间复杂度？我这儿有三个比较实用的方法可以分享给你。

1. 只关注循环执行次数最多的一段代码

我刚才说了，大 O 这种复杂度表示方法只是表示一种变化趋势。我们通常会忽略掉公式中的常量、

低阶、系数，只需要记录一个最大阶的量级就可以了。所以，我们在分析一个算法、一段代码的时间复杂度的时候，也只关注循环执行次数最多的那一段代码就可以了。这段核心代码执行次数的 n 的量级，就是整段要分析代码的时间复杂度。

为了便于你理解，我还拿前面的例子来说明。

复制代码

```
1 int cal(int n) {
2   int sum = 0;
3   int i = 1;
4   for (; i <= n; ++i) {
5     sum = sum + i;
6   }
7   return sum;
8 }
```

其中第 2、3 行代码都是常量级的执行时间，与 n 的大小无关，所以对于复杂度并没有影响。循环执行次数最多的是第 4、5 行代码，所以这块代码要重点分析。前面我们也讲过，这两行代码被执行了 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

我这里还有一段代码。你可以先试着分析一下，然后再往下看跟我的分析思路是否一样。

复制代码

```
int cal(int n) {
   int sum_1 = 0;
   int p = 1;
   for (; p < 100; ++p) {
     sum_1 = sum_1 + p;
   }

   int sum_2 = 0;
   int q = 1;
   for (; q < n; ++q) {
     sum_2 = sum_2 + q;
   }

   int sum_3 = 0;
   int i = 1;
   int j = 1;
   for (; i <= n; ++i) {
     j = 1;
     for (; j <= n; ++j) {
       sum_3 = sum_3 +  i * j;
     }
   }

   return sum_1 + sum_2 + sum_3;
 }
```

这个代码分为三部分，分别是求 sum_1、sum_2、sum_3。我们可以分别分析每一部分的时间复杂度，然后把它们放到一块儿，再取一个量级最大的作为整段代码的复杂度。

第一段的时间复杂度是多少呢？这段代码循环执行了 100 次，所以是一个常量的执行时间，跟 n 的规模无关。

这里我要再强调一下，即便这段代码循环 10000 次、100000 次，只要是一个已知的数，跟 n 无关，照样也是常量级的执行时间。当 n 无限大的时候，就可以忽略。尽管对代码的执行时间会有很大影响，但是回到时间复杂度的概念来说，它表示的是一个算法执行效率与数据规模增长的变化趋势，所以不管常量的执行时间多大，我们都可以忽略掉。因为它本身对增长趋势并没有影响。

那第二段代码和第三段代码的时间复杂度是多少呢？答案是 $O(n)$ 和 $O(n^2)$，你应该能容易就分析出来，我就不啰嗦了。

综合这三段代码的时间复杂度，我们取其中最大的量级。所以，整段代码的时间复杂度就为 $O(n^2)$。也就是说：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。那我们将这个规律抽象成公式就是：

如果 $T1(n)=O(f(n))$，$T2(n)=O(g(n))$；那么 $T(n)=T1(n)+T2(n)=max(O(f(n)), O(g(n))) =O(max(f(n),$

g(n))).

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

我刚讲了一个复杂度分析中的加法法则，这儿还有一个乘法法则。类比一下，你应该能“猜到”公式是什么样子的吧？

如果 T1(n)=O(f(n))，T2(n)=O(g(n))；那么 T(n)=T1(n)*T2(n)=O(f(n))*O(g(n))=O(f(n)*g(n)).

也就是说，假设 T1(n) = O(n)，T2(n) = O(n^2)，则 T1(n) * T2(n) = O(n^3)。落实到具体的代码上，我们可以把乘法法则看成是嵌套循环，我举个例子给你解释一下。

复制代码

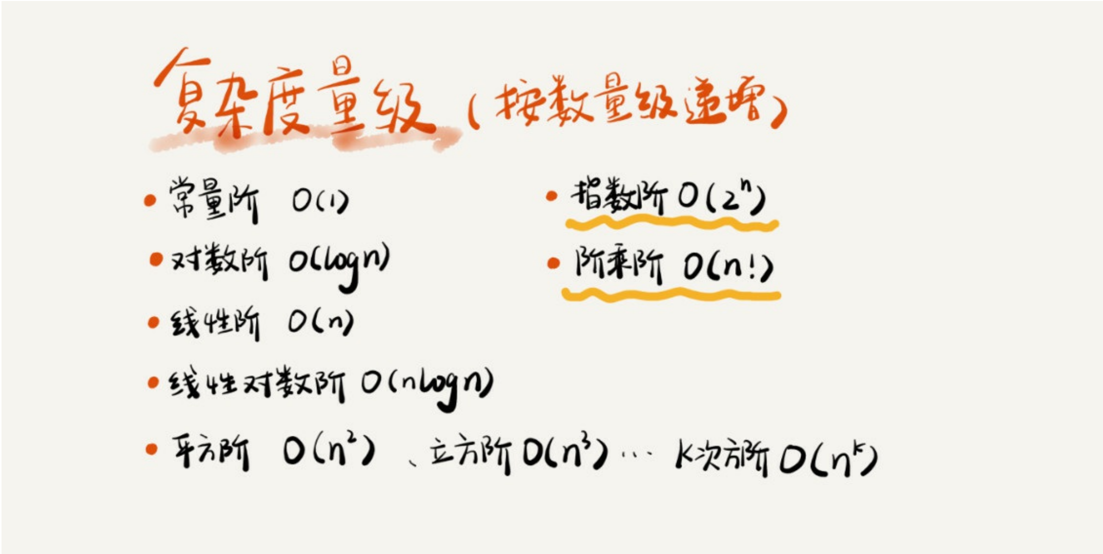
```
1int cal(int n) {
2   int ret = 0;
3   int i = 1;
4   for (; i < n; ++i) {
5     ret = ret + f(i);
6   }
7 }
8
9 int f(int n) {
10  int sum = 0;
11  int i = 1;
12  for (; i < n; ++i) {
13    sum = sum + i;
14  }
15  return sum;
16 }
```

我们单独看 cal() 函数。假设 f() 只是一个普通的操作，那第 4～6 行的时间复杂度就是，T1(n) = O(n)。但 f() 函数本身不是一个简单的操作，它的时间复杂度是 T2(n) = O(n)，所以，整个 cal() 函数的时间复杂度就是，T(n) = T1(n) * T2(n) = O(n*n) = O(n^2)。

我刚刚讲了三种复杂度的分析技巧。不过，你并不用刻意去记忆。实际上，复杂度分析这个东西关键在于“熟练”。你只要多看案例，多分析，就能做到“无招胜有招”。

几种常见时间复杂度实例分析

虽然代码千差万别，但是常见的复杂度量级并不多。我稍微总结了一下，这些复杂度量级几乎涵盖了你今后可以接触的所有代码的复杂度量级。



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：$O(2^n)$ 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。

当数据规模 n 越来越大时，非多项式量级算法的执行时间会急剧增加，求解问题的执行时间会无限增长。所以，非多项式时间复杂度的算法其实是非常低效的算法。因此，关于 NP 时间复杂度我就不展开讲了。我们主要来看几种常见的多项式时间复杂度。

1. O(1)

首先你必须明确一个概念，O(1) 只是常量级时间复杂度的一种表示方法，并不是指只执行了一行代码。比如这段代码，即便有 3 行，它的时间复杂度也是 O(1），而不是 O(3)。

复制代码

```
int i = 8;
int j = 6;
int sum = i + j;
```

我稍微总结一下，只要代码的执行时间不随 n 的增大而增长，这样代码的时间复杂度我们都记作 O(1)。或者说，一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是O(1)。

2. O(logn)、O(nlogn)

对数阶时间复杂度非常常见，同时也是最难分析的一种时间复杂度。我通过一个例子来说明一下。

复制代码

```
i=1;
while (i <= n)  {
  i = i * 2;
}
```

根据我们前面讲的复杂度分析方法，第三行代码是循环执行次数最多的。所以，我们只要能计算出这行代码被执行了多少次，就能知道整段代码的时间复杂度。

从代码中可以看出，变量 i 的值从 1 开始取，每循环一次就乘以 2。当大于 n 时，循环结束。还记得我们高中学过的等比数列吗？实际上，变量 i 的取值就是一个等比数列。如果我把它一个一个列出来，就应该是这个样子的：

$$2^0 \quad 2^1 \quad 2^2 \cdots 2^k \cdots 2^x = n$$

所以，我们只要知道 x 值是多少，就知道这行代码执行的次数了。通过 $2^x=n$ 求解 x 这个问题我们想高中应该就学过了，我就不多说了。$x=\log_2 n$，所以，这段代码的时间复杂度就是 $O(\log_2 n)$。

现在，我把代码稍微改下，你再看看，这段代码的时间复杂度是多少？

复制代码

```
i=1;
while (i <= n)  {
  i = i * 3;
}
```

根据我刚刚讲的思路，很简单就能看出来，这段代码的时间复杂度为 $O(\log_3 n)$。

实际上，不管是以 2 为底、以 3 为底，还是以 10 为底，我们可以把所有对数阶的时间复杂度都记为 O(logn)。为什么呢？

我们知道，对数之间是可以互相转换的，$\log_3 n$ 就等于 $\log_3 2 * \log_2 n$，所以 $O(\log_3 n) = O(C * \log_2 n)$，其中 $C=\log_3 2$ 是一个常量。基于我们前面的一个理论：在采用大 O 标记复杂度的时候，可以忽略系数，即 O(Cf(n)) = O(f(n))。所以，$O(\log_2 n)$ 就等于 $O(\log_3 n)$。因此，在对数阶时间复杂度的表示方法里，我们忽略对数的“底”，统一表示为 O(logn)。

如果你理解了我前面讲的 O(logn)，那 O(nlogn) 就很容易理解了。还记得我们刚讲的乘法法则吗？如果一段代码的时间复杂度是 O(logn)，我们循环执行 n 遍，时间复杂度就是 O(nlogn) 了。而且，O(nlogn) 也是一种非常常见的算法时间复杂度。比如，归并排序、快速排序的时间复杂度都是 O(nlogn)。

3. O(m+n)、O(m*n)

我们再来讲一种跟前面都不一样的时间复杂度，代码的复杂度由两个数据的规模来决定。老规矩，先看代码！

复制代码

```
int cal(int m, int n) {
  int sum_1 = 0;
  int i = 1;
  for (; i < m; ++i) {
    sum_1 = sum_1 + i;
  }

  int sum_2 = 0;
  int j = 1;
  for (; j < n; ++j) {
    sum_2 = sum_2 + j;
  }

  return sum_1 + sum_2;
}
```

从代码中可以看出，m 和 n 是表示两个数据规模。我们无法事先评估 m 和 n 谁的量级大，所以我们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。所以，上面代码的时间复杂度就是 O(m+n)。

针对这种情况，原来的加法法则就不正确了，我们需要将加法规则改为：T1(m) + T2(n) = O(f(m) + g(n))。但是乘法法则继续有效：T1(m)*T2(n) = O(f(m) * f(n))。

空间复杂度分析

前面，咱们花了很长时间讲大 O 表示法和时间复杂度分析，理解了前面讲的内容，空间复杂度分析方法学起来就非常简单了。

前面我讲过，时间复杂度的全称是**渐进时间复杂度**，**表示算法的执行时间与数据规模之间的增长关系**。类比一下，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**。

我还是拿具体的例子来给你说明。（这段代码有点“傻”，一般没人会这么写，我这么写只是为了方便给你解释。）

```
1void print(int n) {
2  int i = 0;
3  int[] a = new int[n];
4  for (i; i <n; ++i) {
5    a[i] = i * i;
6  }

7  for (i = n-1; i >= 0; --i) {
8    print out a[i]
9  }
10}
```

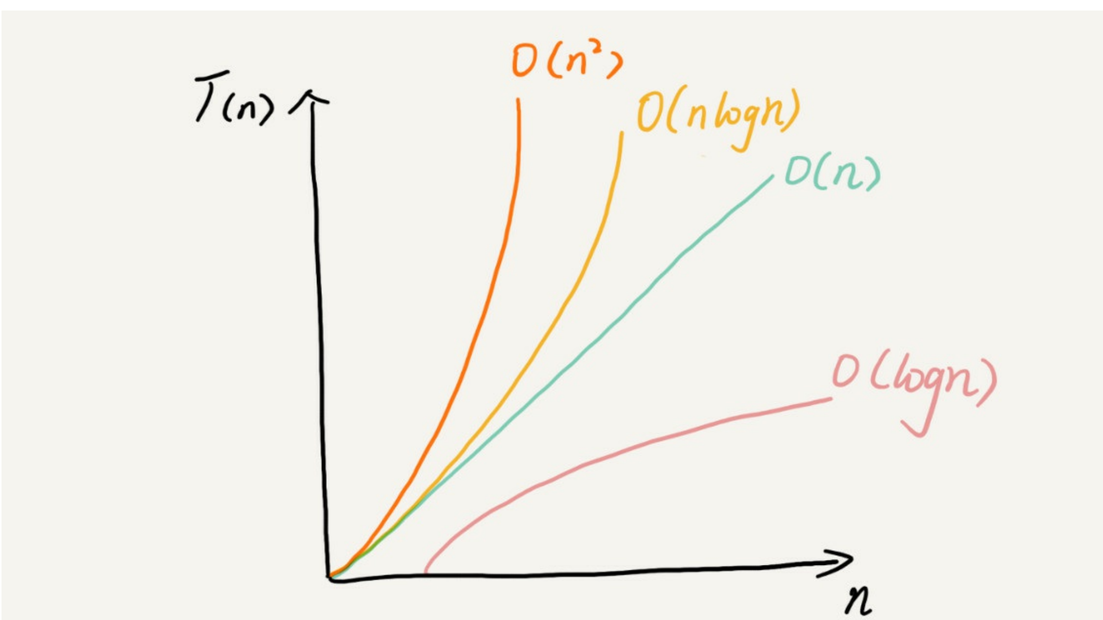
跟时间复杂度分析一样，我们可以看到，第 2 行代码中，我们申请了一个空间存储变量 i，但是它是常量阶的，跟数据规模 n 没有关系，所以我们可以忽略。第 3 行申请了一个大小为 n 的 int 类型数组，除此之外，剩下的代码都没有占用更多的空间，所以整段代码的空间复杂度就是 O(n)。

我们常见的空间复杂度就是 O(1)、O(n)、O(n^2)，像 O(logn)、O(nlogn) 这样的对数阶复杂度平时都用不到。而且，空间复杂度分析比时间复杂度分析要简单很多。所以，对于空间复杂度，掌握刚我说的这些内容已经足够了。

内容小结

基础复杂度分析的知识到此就讲完了，我们来总结一下。

复杂度也叫渐进复杂度，包括时间复杂度和空间复杂度，用来分析算法执行效率与数据规模之间的增长关系，可以粗略地表示，越高阶复杂度的算法，执行效率越低。常见的复杂度并不多，从低阶到高阶有：O(1)、O(logn)、O(n)、O(nlogn)、O(n^2)。等你学完整个专栏之后，你就会发现几乎所有的数据结构和算法的复杂度都跑不出这几个。



复杂度分析并不难，关键在于多练。 之后讲后面的内容时，我还会带你详细地分析每一种数据结构和算法的时间、空间复杂度。只要跟着我的思路学习、练习，你很快就能和我一样，每次看到代码

的时候，简单的一眼就能看出其复杂度，难的稍微分析一下就能得出答案。

课后思考

有人说，我们项目之前都会进行性能测试，再做代码的时间复杂度、空间复杂度分析，是不是多此一举呢？而且，每段代码都分析一下时间复杂度、空间复杂度，是不是很浪费时间呢？你怎么看待这个问题呢？

欢迎留言和我分享，我会第一时间给你反馈。

写留言

精选留言

xr

256

我不认为是多此一举，渐进时间，空间复杂度分析为我们提供了一个很好的理论分析的方向，并且它是宿主平台无关的，能够让我们对我们的程序或算法有一个大致的认识，让我们知道，比如在最坏的情况下程序的执行效率如何，同时也为我们交流提供了一个不错的桥梁，我们可以说，算法1的时间复杂度是O(n)，算法2的时间复杂度是O(logN)，这样我们立刻就对不同的算法有了一个“效率”上的感性认识。

当然，渐进式时间，空间复杂度分析只是一个理论模型，只能提供给粗略的估计分析，我们不能直接断定就觉得O(logN)的算法一定优于O(n), 针对不同的宿主环境，不同的数据集，不同的数据量的大小，在实际应用上面可能真正的性能会不同，个人觉得，针对不同的实际情况，进而进行一定的性能基准测试是很有必要的，比如在统一一批手机上(同样的硬件，系统等等)进行横向基准测试，进而选择适合特定应用场景下的最有算法。

综上所述，渐进式时间，空间复杂度分析与性能基准测试并不冲突，而是相辅相成的，但是一个低阶的时间复杂度程序有极大的可能性会优于一个高阶的时间复杂度程序，所以在实际编程中，时刻关心理论时间，空间度模型是有助于产出效率高的程序的，同时，因为渐进式时间，空间复杂度分析只是提供一个粗略的分析模型，因此也不会浪费太多时间，重点在于在编程时，要具有这种复杂度分析的思维。

2018-09-26

作者回复

写得很好。理解的到位

2018-09-26

姜威 169

总结

一、什么是复杂度分析？

1.数据结构和算法解决是“如何让计算机更快时间、更省空间的解决问题”。

2.因此需从执行时间和占用空间两个维度来评估数据结构和算法的性能。

3.分别用时间复杂度和空间复杂度两个概念来描述性能问题，二者统称为复杂度。

4.复杂度描述的是算法执行时间（或占用空间）与数据规模的增长关系。

二、为什么要进行复杂度分析？

1.和性能测试相比，复杂度分析有不依赖执行环境、成本低、效率高、易操作、指导性强的特点。

2.掌握复杂度分析，将能编写出性能更优的代码，有利于降低系统开发和维护成本。

三、如何进行复杂度分析？

1.大O表示法

1）来源

算法的执行时间与每行代码的执行次数成正比，用T(n) = O(f(n))表示，其中T(n)表示算法执行总时间，f(n)表示每行代码执行总次数，而n往往表示数据的规模。

2）特点

以时间复杂度为例，由于时间复杂度描述的是算法执行时间与数据规模的增长变化趋势，所以常量阶、低阶以及系数实际上对这种增长趋势不产决定性影响，所以在做时间复杂度分析时忽略这些项。

2.复杂度分析法则

1）单段代码看高频：比如循环。

2）多段代码取最大：比如一段代码中有单循环和多重循环，那么取多重循环的复杂度。

3）嵌套代码求乘积：比如递归、多重循环等

4）多个规模求加法：比如方法有两个参数控制两个循环的次数，那么这时就取二者复杂度相加。

四、常用的复杂度级别？

多项式阶：随着数据规模的增长，算法的执行时间和空间占用，按照多项式的比例增长。包括，O(1)（常数阶）、O(logn)（对数阶）、O(n)（线性阶）、O(nlogn)（线性对数阶）、O(n^2)（平方阶）、O(n^3)（立方阶）

非多项式阶：随着数据规模的增长，算法的执行时间和空间占用暴增，这类算法性能极差。包括，O(2^n)（指数阶）、O(n!)（阶乘阶）

五、如何掌握好复杂度分析方法？

复杂度分析关键在于多练，所谓熟能生巧。

2018-09-26

作者回复

总结的很棒

2018-09-26

芳芳 53

糟糕，是看不懂的感觉

2018-09-26

Orcsir 43

老师，代码片段把行号也写上吧。

2018-09-26

作者回复

嗯嗯 我联系运营加上

2018-09-26

w 24

第二个例子中，第6.7行为什么是2n平方遍而不是n平方遍呢？

2018-09-26

作者回复

因为两层循环 一层是n 两层是n*n。不信你自己令n=5 自己算算

2018-09-26

有一天 14

一直有一个很纠结的问题，烦请解答一下：O具体是哪一个英文字母的缩写？

2018-09-26

作者回复

不是英文缩写 就是一个数学符号而已

2018-09-26

吕宁 14

老师好，我们上算法课，老师讲到存储一个二进制数，输入规模（空间复杂度）是O(logn) bit。请问如何理解？

2018-09-26

作者回复

比如8用二进制表示就是3个bit。16用二进制表示就是4个bit。以此类推 n用二进制表示就是logn个bit

2018-09-26

起名好难 13

文章里也说了，性能测试这种是受环境所影响的。作为程序员，我们能做的就是尽可能的降低复杂度，才能让代码在不同的环境下以最快的效率执行。至于是不是浪费时间，我觉得其实是个伪命题。首先按刚刚分析过程来看，通过熟悉练习，简单的代码是可以直接看出来复杂度的也就是不费时间；而比较复杂的代码就容易"一不小心"太"复杂"了，这个时候，为了代码质量考虑分析复杂度的时间也并不浪费。再有甚者，我们学习这个分析法，我觉得更多的是要明白这个理念，在写代码的时候就能关注一下这方面的问题，毕竟复杂的代码在写的过程往往是先分析整体逻辑结构的，并且写的过程也需要不断思考，了解这个理念后才能在写的过程中也思考关注这个点。不然，复杂的一段代码一旦写成，日后因为性能问题重构，更费时间。

以上是对课后题的思考，欢迎批评指正。

另: 感觉加法法则那个图，maxf(n)+g(n) 换成max(f(n)+g(n))会不会更好些？

2018-09-26

作者回复

理解的非常透彻 非常有逻辑性 很赞。ps 图画错了 我联系运营改下

2018-09-26

dickwxyz@126.com 11

没有看懂，所以，我们只要知道 x 值是多少，就知道这行代码执行的次数了。通过 2x=n 求解 x 这个问题我们想高中应该就学过了，我就不多说了。 这里的x不就是代码里的n吗，时间复杂度不是O(n)吗？

2018-09-26

作者回复

i 第一次等于1

第2次等于2

第3次等于2*2

第4次等于2*2*2

.....

第x次等于2的x-1次方

......

那第几次之后等于n呢

也就是2的x-1次方等于n求解x

x粗略点讲就近似于logn

也就是代码执行了logn遍就退出循环了

所以根据大o标记法 为logn

2018-09-26

最爱小黑黑

11

睡前刷一遍 明早起来再细看一遍 加油各位！

2018-09-26

陈华应

7

有必要，基准测试是事后，也是理论验证，有时候O(n)未必一定比O(1)效率低。
复杂度分析是理论，整体趋势上反应了一个算法的时间或者空间利用率与数据规模的渐进关系，并且像程序员之间使用设计模式来讨论代码设计一样，说出名字就大致知道代码是如何组织的，大O也是一样。
随着自己使用大O分析代码复杂度的熟练程度增加，判断一段代码的复杂度可能分分钟的事情，甚至更快。

2018-09-26

作者回复

理解的很透彻！

2018-09-26

wistbean

6

--------总结一下--------

事后统计的局限性：

1.不同的环境测试结果差异大
2.数据本身规模会影响结果（如数据规模小，结果无法真实反应出算法的性能问题）

所以：需要复杂度分析

时间复杂度：

表示法：T(n) = O(n)； T(n) = O(n2)。
T(n) 代表代码执行时间
(n)代表每行代码的执行次数总和

也就是每行代码的执行次数总和越大，那么代码执行就需要更多的时间。

时间复杂度分析方法：
1.只关注执行次数最多的那段代码。
2.加法法则：总复杂度取量级最大的那段代码。
3.乘法法则：代码之间复杂度有嵌套情况，将各个复杂度相乘得到总复杂度。

复杂度常见案例：

非多项式量级：
O（2^n）
O(n!)
n越大，算法执行时间急剧增加，相对低效

多项式量级：
O(1) : 不存在n的影响因素
O（logn）、O（nlogn）：对阶时间复杂度，最难分析之一。
O（m+n）、O（m*n）：当数据规模无法评估哪个较大时，加法法则失效，可以使用 O（m+n）的方式。

空间复杂度分析
类比于时间复杂度就是：存储空间与数据规模的增长关系。

-----------课后思考--------------

不认为是浪费时间，性能的基准测试可能会受到环境，数据规模本身的影响，对时间复杂度、空间复杂度进行分析至少对一些复杂度能够做出判断，写出相对效率高的代码，此外，还能提升自己分析复杂度的思维和效率的认知。

2018-09-26

作者回复

写得很好

2018-09-26

Dwyane

6

03
大家好，这是我的总结：

公式中的低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略
1. 只关注循环执行次数最多的一段代码
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的乘积

只要代码的执行时间不随 n 的增大而增长，这样代码的时间复杂度我们集作O(1)
不同数据规模，无法评估 m 和 n 的量级大，所以不能利用加法法则，去掉某一个，而是 O(m+n)

空间复杂度：表示算法的存储空间与数据规模之间的增加关系

额外说一下：log3n 就等于 log32 * log2n 其实是利用换底公示推导，有疑问的搜一下。

2018-09-26

作者回复

是换底公式

2018-09-26

短迪大魔王 5

很有必要，现在是大数据时代，如果是矩阵计算，那就是on，如果是传统双for遍历那就是on²，做lr不依托矩阵都要天荒地老，那神经网络尤其是rnn就不用做了，即使是84万文本数据，长度为20个词，用单机gpu加速要跑七天。双for是几天那？经典例子是马踏棋盘，没优化代码跑几天，优化了又看不懂，问问老师如何对代码做优化，因为优化了就读不懂了有没有？
空间复杂度也有必要，还是nlp的例子，如果是embending的话，内存开销和磁盘开销都小的多，虽然现在分布式允许无限大，但是生产环境要把数据传到hdfs，再传到训练集群上，这都有网络传输开销啊，其二是可能没有这个权限，不安全。其三，生成npy文件不能shuffle，很不便利，也不允许分割，所以事先要想好空间要怎么来。当然时间更重要，敏捷迭代。

2018-09-26

ChaoYrAx 5

老师 空间复杂度 和时间复杂度的 具体区别是什么，我怎么看上去像一样的

2018-09-26

作者回复

一个表示内存的消耗 一个表示执行的快慢

2018-09-26

realEago 5

看不懂别慌，也别忙着总结，先读五遍文章先，无他，唯手熟尔~

2018-09-26

作者回复

说的太好了 我这里也没葵花宝典 学还是得靠自己

2018-09-26

冯剑 5

在分析多项式复杂度的时候，有根据输入规模确定复杂度O(m+n),我的理解是 假设n是相对比较大的值，那么这个复杂度O(m+n)<=O(2n)，2是常量，这样的话复杂度不就是O(n),请问下，O(n)和O(m+n)的区别在什么地方？有什么应用场景能体现出二者不同

2018-09-26

五岳寻仙 5

老师讲得太好了！记得上学时，刚接触大O，直接懵，并且丝毫没意识到复杂度分析的重要性，还觉得为什么不直接讲干货呢。现在想想真愚蠢！
今天课程的感悟：
第一，复杂度分析的重要性。我觉得再怎么强调其重要性也不为过，不具备这种意识，就很难摆脱底层码农的处境。
第二，理解“大”和“小”。科普书《从1到无穷大》中举到一个例子：比较自然数和奇数的个数，答案是一样多。当数据规模大到一定程度时，靠感觉是不够的，必须借助理论工具，大O表示法就是一个强有力的工具。
第三，抓主要矛盾。刚开始分析算法复杂度的时候，系数常数都考虑得很细致，这只是方便理解。真正在实际运用中，系数和常数都可以忽略，我们真正关心的是 算法是常数复杂度，对数复杂度还是线性复杂度等，它们之间有天壤之别，决定了计算量是几亿次还是几万亿次。

课后思考我觉得 xr 同学回答得非常好！我自己从中学到了很多。我举一个自己的例子：
我曾经需要对3G条数据排序，使用了内省排序，时间复杂度是nlogn，花了半个小时左右。假如我

用选择排序，时间复杂度是n方，计算量是前者的几千万倍，在我有生之年也等不到排序结束。

2018-09-26

□

5

考研数据结构会考到□

2018-09-26

秋凯

4

看了下上面的总结，我想补充一下

学习复杂度的主要目的就一个：方便后续定性比较算法性能，比如归并是nlogn，插入是n^2，所以归并快，而且快不少。

下一期是不是讲最坏情况和最好情况？

2018-09-26

作者回复

是的。还有平均 均摊

2018-09-26