第一章　算法效率分析基础

整理者：Yu
资料整理自互联网，仅供学习用途

2017 年 3 月 6 日

通常，对于一个给定的算法，我们要做两项分析。第一是从数学上证明算法的正确性，这一步主要用到形式化证明的方法及相关推理模式，如循环不变式、数学归纳法等。而在证明算法是正确的基础上，第二部就是分析算法的时间复杂度。算法的时间复杂度反映了程序执行时间随输入规模增长而增长的量级，在很大程度上能很好反映出算法的优劣与否。因此，作为程序员，掌握基本的算法时间复杂度分析方法是很有必要的。算法执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。而度量一个程序的执行时间通常有两种方法。

事后统计的方法 这种方法可行，但不是一个好的方法。该方法有两个缺陷：一是要想对设计的算法的运行性能进行评测，必须先依据算法编制相应的程序并实际运行；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优势。

事前分析估算的方法 因事后统计方法更多的依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。因此人们常常采用事前分析估算的方法。在编写程序前，依据统计方法对算法进行估算。一个用高级语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

- 算法采用的策略、方法
- 编译产生的代码质量
- 问题的输入规模
- 机器执行指令的速度

一个算法是由控制结构（顺序、分支和循环3种）和原操作（指固有数据类型的操作）构成的，则算法时间取决于两者的综合效果。为了便于比较同一个问题的不同算法，通常的做法是，从算法中选取一种对于所研究的问题（或算法类型）来说是基本操作的原操作，以该基本操作的重复执行的次数作为算法的时间量度。

为什么要进行算法分析?

- 预测算法所需要的资源
  - 计算时间（CPU 消耗）
  - 内存空间（RAM 消耗）
  - 通信时间（带宽消耗）
- 预测算法的运行时间

  - 在给定输入规模时，所执行的基本操作数量。
  - 或者称为算法复杂度（Algorithm Complexity）

如何衡量算法复杂度?

- 内存（Memory）
- 时间（Time）
- 指令的数量（Number of Steps）
- 特定操作的数量
- 磁盘访问数量
- 网络包数量
- 渐进复杂度（Asymptotic Complexity）

算法的运行时间与什么相关?

- 取决于输入的数据的初始状态。（例如：如果数据已经是排好序的，时间消耗可能会减少。）
- 取决于输入数据的规模。（例如：10个数排序和10^8个数排序）
- 取决于运行时间的上限。（因为运行时间的上限是对使用者的承诺。）

算法分析的种类:

- 最坏情况（Worst Case）：任意输入规模的最大运行时间。（Usually）
- 平均情况（Average Case）：任意输入规模的期待运行时间。（Sometimes）
- 最佳情况（Best Case）：通常最佳情况不会出现。（Bogus）

例如，在一个长度为n 的列表中顺序搜索指定的值，则

(1) 最坏情况：n 次比较

(2) 平均情况：n/2 次比较

(3) 最佳情况：1 次比较

而实际中，我们一般仅考量算法在最坏情况下的运行情况，也就是对于规模为n 的任何输入，算法的最长运行时间。这样做的理由是:

(1) 一个算法的最坏情况运行时间是在任何输入下运行时间的一个上界（Upper Bound）。

(2) 对于某些算法，最坏情况出现的较为频繁。

(3) 大体上看，平均情况通常与最坏情况一样差。

算法分析要保持大局观（Big Idea），其基本思路:

(1) 忽略掉那些依赖于机器的常量。

(2) 关注运行时间的增长趋势。

比如：$T(n) = 73n^3 + 29n^2 + 8888$ 的趋势就相当于T(n) = $\Theta(n^3)$。
渐近记号（Asymptotic Notation）通常有O、Θ 和Ω 记号法。Θ 记号渐进地给出了一个函数的上界和下界，当只有渐近上界时使用O 记号，当只有渐近下界时使用Ω 记号。尽管技术上Θ 记号较为准确，但通常仍然使用O 记号表示。例如,线性复杂度$O(n)$ 表示每个元素都要被处理一次。平方复杂度$O(n^2)$ 表示每个元素都要被处理n 次。

表 1: 渐进记号表示法

Notation	Intuition	Informal Definition
$f(n) \in O(g(n))$	f is bounded above by g asymptotically	$\exists n > n_0, \lvert f(n)\rvert \leq g(n) \cdot k$
$f(n) \in \Omega(g(n))$	f is bounded below by g asymptotically	$\exists n > n_0, \lvert f(n)\rvert \geq g(n) \cdot k$
$f(n) \in \Theta(g(n)$	f is bounded above and below by g asymptotically	$\exists n > n_0, g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$

表 2: 基本的渐进效率类型

复杂度	标记符号
常数（Constant）	$O(1)$
对数（Logarithmic）	$O(\log_2 n)$
线性（Linear）	$O(n)$
平方（Quadratic）	$O(n^2)$
立方（Cubic）	$O(n^3)$
指数（Exponential）	$O(2^n), O(k^n), O(n!)$

求解算法的时间复杂度的具体步骤:

(1) 找出算法中的基本语句；算法中执行次数最多的那条语句就是基本语句，通常是最内层循环的循环体。

(2) 计算基本语句的执行次数的数量级；只需计算基本语句执行次数的数量级，这就意味着只要保证基本语句执行次数的函数中的最高次幂正确即可，可以忽略所有低次幂和最高次幂的系数。这样能够简化算法分析，并且使注意力集中在最重要的一点上：增长率。

(3) 用大O记号表示算法的时间性能。将基本语句执行次数的数量级放入大O记号中。如果算法中包含嵌套的循环，则基本语句通常是最内层的循环体，如果算法中包含并列的循环，则将并列循环的时间复杂度相加。例如:

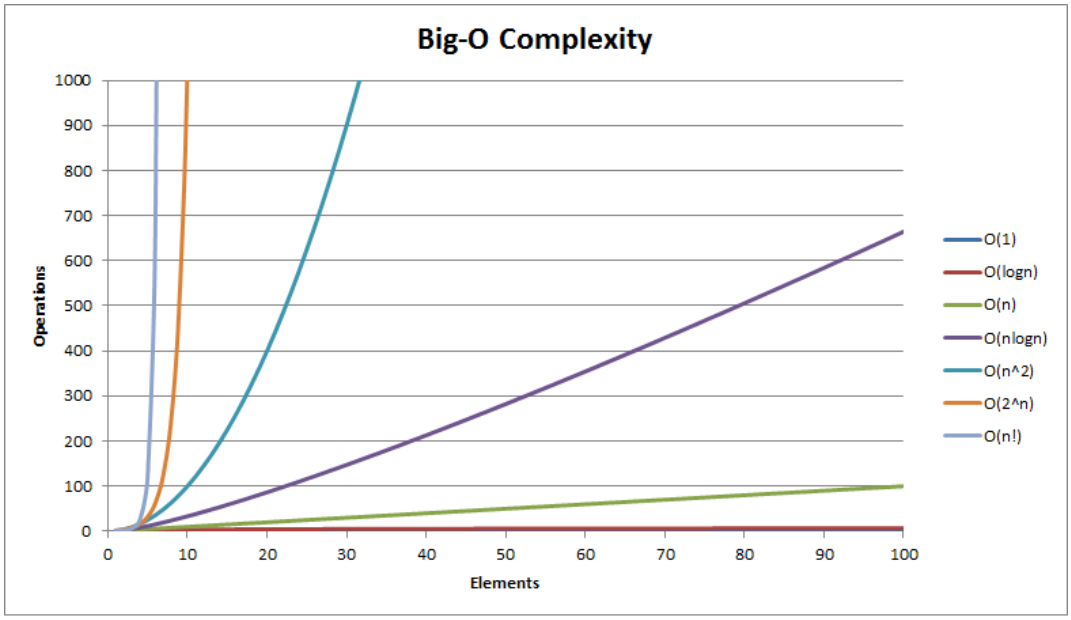

图 1: 基本的渐进效率图像

```
for (i=1; i<=n; i++)
    x++;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        x++;
```

第一个for循环的时间复杂度为$O(n)$，第二个for循环的时间复杂度为$O(n^2)$，则整个算法的时间复杂度为$O(n+n^2)=O(n^2)$。

$O(1)$表示基本语句的执行次数是一个常数，一般来说，只要算法中不存在循环语句，其时间复杂度就是$O(1)$。其中$O(log_2^n)$、$O(n)$、$O(nlog_2^n)$、$O(n^2)$和$O(n^3)$称为多项式时间，而$O(2^n)$和$O(n!)$称为指数时间。计算机科学家普遍认为前者（即多项式时间复杂度的算法）是有效算法，把这类问题称为P（Polynomial,多项式）类问题，而把后者（即指数时间复杂度的算法）称为NP（Non-Deterministic Polynomial, 非确定多项式）问题。

一般来说多项式级的复杂度是可以接受的，很多问题都有多项式级的解——也就是说，这样的问题，对于一个规模是n的输入，在n^k的时间内得到结果，称为P问题。有些问题要复杂些，没有多项式时间的解，但是可以在多项式时间里验证某个猜测是不是正确。比如问4294967297是不是质数？如果要直接入手的话，那么要把小于4294967297的平方根的所有素数都拿出来，看看能不能整除。还好欧拉告诉我们，这个数等于641和6700417的乘积，不是素数，很好验证的，顺便麻烦转告费马他的猜想不成立。大数分解、Hamilton回路之类的问题，都是可以多项式时间内验证一个“解”是否正确，这类问题叫做NP问题。

在计算算法时间复杂度时有以下几个简单的程序分析法则:

(1) 对于一些简单的输入输出语句或赋值语句,近似认为需要$O(1)$时间

(2) 对于顺序结构,需要依次执行一系列语句所用的时间可采用大O下”求和法则”。求和法则:是指若算法

的2个部分时间复杂度分别为$T_1(n)=O(f(n))$和$T_2(n)=O(g(n))$, 则$T_1(n)+T_2(n)=O(max(f(n),g(n)))$.

(3) 对于选择结构,如if语句,它的主要时间耗费是在执行then字句或else字句所用的时间,需注意的是检验条件也需要$O(1)$时间

(4) 对于循环结构,循环语句的运行时间主要体现在多次迭代中执行循环体以及检验循环条件的时间耗费,一般可用大O下"乘法法则"。乘法法则: 是指若算法的2个部分时间复杂度分别为$T_1(n)=O(f(n))$和$T_2(n)=O(g(n))$,则$T_1*T_2=O(f(n)*g(n))$

(5) 对于复杂的算法,可以将它分成几个容易估算的部分,然后利用求和法则和乘法法则技术整个算法的时间复杂度另外还有以下2个运算法则:(a) 若$g(n)=O(f(n))$,则$O(f(n))+O(g(n))=O(f(n))$; (b) $O(cf(n))=O(f(n))$,其中c是一个正常数

下面分别对几个常见的时间复杂度进行示例说明

(1)、$O(1)$

```
temp=i;
i=j;
j=temp;
```

以上三条单个语句的频度均为1，该程序段的执行时间是一个与问题规模n无关的常数。算法的时间复杂度为常数阶，记作$T(n)=O(1)$。注意：如果算法的执行时间不随着问题规模n的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是$O(1)$。

(2)、$O(n^2)$

```
sum=0;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      sum++;
```

因为$\Theta(2n^2+n+1)=n^2$（Θ即：去低阶项，去掉常数项，去掉高阶项的常参得到），所以$T(n)=O(n^2)$;

```
for (i=1;i<n;i++)
{
    y=y+1;              (1)
    for (j=0;j<=(2*n);j++)
        x++;            (2)
}
```

语句1的频度是$n-1$,语句2的频度是$(n-1)*(2n+1)=2n^2-n-1$, $f(n)=2n^2-n-1+(n-1)=2n^2-2$,又$\Theta(2n^2-2)=n^2$,该程序的时间复杂度$T(n)=O(n^2)$. 一般情况下，对步进循环语句只需考虑循环体中语句的执行次数，忽略该语句中步长加1、终值判别、控制转移等成分，当有若干个循环语句时，算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度$f(n)$决定的。

(3)、$O(n)$

```
int a=0,b=1;                (1)
for (i=1;i<=n;i++){         (2)
```

```
    s=a+b;          (3)
    b=a;            (4)
    a=s;            (5)
}
```

语句1的频度:2,语句2的频度:n, 语句3的频度:n-1, 语句4的频度:n-1, 语句5的频度:n-1, $T(n)=2+n+3(n-1)=4n-1=O(n)$.

(4)、$O(log_2n)$

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

语句1的频度是1,设语句2的频度是$f(n)$,则：$2*f(n)<=n;f(n)<=log_2n$,取最大值$f(n)=log_2n, T(n)=O(log_2n)$

算法时间复杂度分析是一个很重要的问题，任何一个程序员都应该熟练掌握其概念和基本方法，而且要善于从数学层面上探寻其本质，才能准确理解其内涵。