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

同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。算法分析的目的在于选择合适算法和改进算法。一个算法的评价主要从时间复杂度和空间复杂度来考虑。随着硬件技术的不断发展和价格走低，空间复杂度在大部分情况下已经不是需要重点关注的问题了，然而时间效率的重要性并没有减弱到这种程度。本章将主要介绍时间复杂度。

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。

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。

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
常量（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）	$O(2^n), O(k^n), O(n!)$

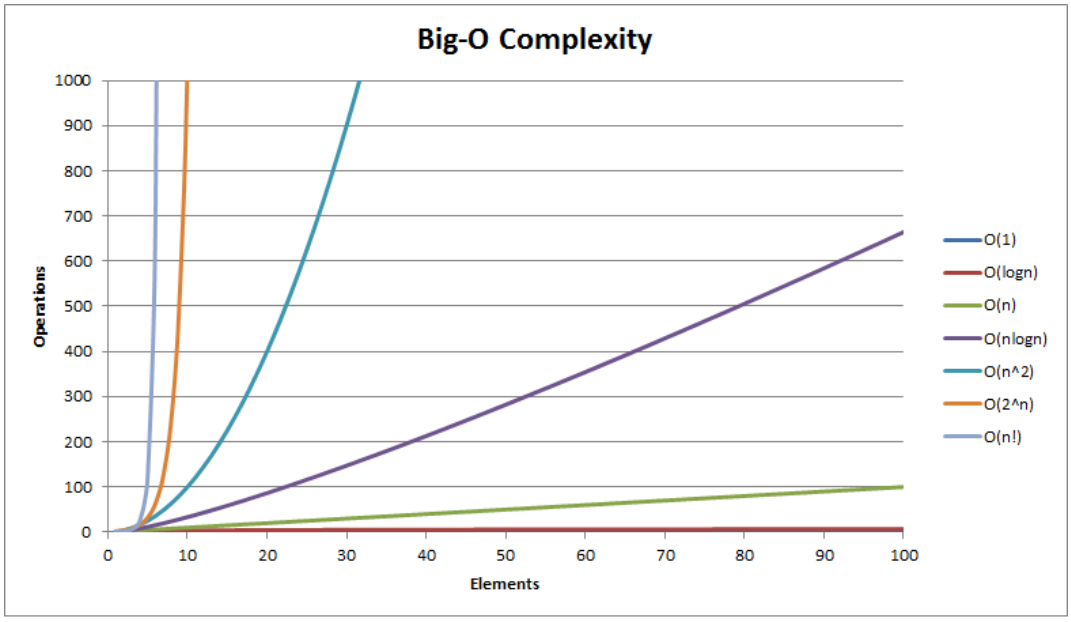

图 1: 基本的渐进效率图像

(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:

```
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(n2)，则整个算法的时间复杂度为O(n+n2)=O(n2)。

```
def FindMaxElement(array):
    max = array[0]
    for i in range(len(array):
        if array[i] > max:
          max = array[i]
```

又如以上代码，其中最基本的执行语句为比较array与max的大小，其执行数量约为$n*(n-1)/2$，所以算法复杂度为$O(n^2)$。