1. **实现和经验分析**

不论是去解决一个其他方法不能解决的大规模问题，还是为系统的关键部分提供一种高效的实现，为了高效地利用算法，都需要理解算法的**性能特征**。

理解算法的性能特征的第一步是进行**经验分析**：

经验分析面临的第一个挑战是开发一个正确、完整的算法实现(因此，有可能希望在投入更大精力实现该算法之前，对类似的程序进行数序分析或经验分析)；

第二个挑战是确定输入数据的特性，以及对进行的实验有直接影响到其他因素。典型地，我们有三个基本选择：利用**实际数据，随机数据或伪数据。**实际数据可以测试程序的真正开销；随机数据确保实验测试的是算法，而不是测试数据；伪数据确保程序可以处理可能的任何输入。

对于算法的性能特征，最常犯的错误是忽略了算法的性能特征(即常常接受一个慢速算法而不愿意去处理更复杂的算法)和过多地关注算法的性能特征(如果一个程序只运行几微妙，把这个程序运行时间改进10倍意义并不大)。

1. **算法分析(数学分析)**

我们对算法进行数学分析的目的是：

* 比较同一任务的不同算法；
* 预测算法在新环境下的能力；
* 设置算法中的参数值。

算法分析的基本步骤：

1. 明确算法基于的抽象操作，从而从实现中把分析分离出来；
2. 研究数据，为算法的输入建立模型；通常考虑以下方法：一是假设输入是随机的，然后研究程序在均匀分布下的性能；二是寻求伪输入，然后研究程序在最坏情况下的性能。
3. **函数的增长(常用的描述算法性能特征的数学函数)**

算法的运行时间一般会与以下某个函数成正比：

**1** 一个程序的所有指令只执行一次或者之多只执行几次，此时程序的运行时间为常量。

**logN** 当程序的运行时间为对数是，程序随N的增长而稍微增长。通常在求解一个大规模问题的程序中，它把问题变成一些小的子问题，每一步都把问题的规模缩小几分之几，就会出现这样的运行时间。

**N** 当程序的运行时间为线性时，通常对每个输入元素只做了少量的处理工作。这种情况对于一个必须处理N个输入(或产生N个输出)的算法是最优的。

**NlogN** 当把问题分解成小问题，且独立求解只问题，然后把这些子问题好的解组合成原有问题的解时，会出现NlogN的运行时间。

当算法的运行时间为平方时，算法只适用于规模相对较小的问题。 二次运行时间一般出现在需要处理所有数据项对(也许是双层嵌套循环)的算法中。

类似地，处理三个数据项的算法(或许是三层嵌套循环)的运行时间为立方，这样的算法只适用于小规模的问题。

一个指数运行时间的算法很难在实际中使用。

**其余常用函数和常数：**

|  |  |  |  |
| --- | --- | --- | --- |
| 函数 | 名称 | 特殊值 | 近似 |
|  | floor函数 |  | x |
|  | ceil函数 |  | x |
| lgN | 以2为底的对数 | lg1024=10 | 1.44lnN |
|  | 斐波那契数 |  |  |
|  | 调和数 |  |  |
| N! | 阶乘函数 | 10!=3628800 |  |
| lg(N!) |  | lg(100!)520 | NlgN-1.44N |
| e = 2.71828…  = 0.57721… (欧拉常数)  = (1+)/2 = 1.61803…  ln2 = 0.693147…  lge = 1/ln2 = 1.44269… | | | |

离散自然对数函数称为**调和数**，第N个调和数由以下方程定义：

自然对数lnN是曲线1/x在1和N之间与X轴所夹区域的面积；调和数是计算1/n在1和N之间与x轴所夹区域的面积。

**斐波那契数列**由以下公式定义：

两个相邻的斐波那契数列的比例接近黄金分割比率。

阶乘表示了N个对象的所有排列。

**二项分布及泊松分布：**

1. **大O分布**

定义： 如果存在常数和，对于所有N>,有g(N)<f(N),则称函数g(N)是O(f(N))的。

大O符号有三个作用：

* 限制忽略数学公式中的低阶项时产生的误差；
* 限制由于忽略对程序的总运行时间贡献较小的某些部分时产生的错误；
* 允许我们按照算法总运行时间的上界对算法进行分类。

大O符号允许我们在操作近似数学表达式时，只记录首项，而可以忽略其他低阶项，最终可使我们给出对于所分析的数学表达式的精确近似的简洁陈述。

如果函数f(N)渐近大于另一个函数g(N)(即当N→∞时，g(N)/f(N) →0),我们有时用**约等于**f(N)表示f(N)+O(g(N))。比如N(N-1)/2,我们可以近似表示为/2。

类似地，如果我们能够证明当g(N)渐近小于f(N)时，算法的运行时间cf(N)+g(N),则说算法的运行时间与f(N)**成正比**。

1. **基本递归方程**

**公式1：** 如果程序的循环通过输入每次减少一项，递归公式为：

其解为约等于/2。

**公式2：** 如果程序每次使输入减半，递归公式为：

其解为约等于。

**公式3：** 如果程序每次使输入减半，但需要检查输入的每一项，递归公式为：

其解为约等于。

**公式4：** 如果程序把输入分成两半，但在划分之前、划分之中以及划分之后需要线性遍历输入，递归公式为：

其解为约等于。(分治法)

**公式5：** 如果程序把输入分成两半，然后做一些需要常量时间的其他工作，递归公式为：

其解为约等于。

1. **保证、预测及局限性**

**保证：**算法的最坏情况下的性能。

困难：最坏情况下所需的时间可能更实际数据所需的时间存在很大的

差距；具有较好的最坏情况性能的算法可能比其他一些算法复杂得多

(设计这样的算法也要困难得多)；

**预测：**算法的平均情况下的性能可以是我们预测程序的运行时间。

困难：输入模型可能没有精确地表征实际中遇到的输入，也可能就没有

自然的输入模型；分析可能需要深奥的数学推理；有时还需要知道运行时间分布的标准方差或其他事实，这些可能更难推出。

**复杂度：** 确定给定问题的最佳算法在最坏的情况下的运行时间，误差不超过一个常数因子，这个函数称为问题的复杂度。

如果我们可以证明求解某个问题的算法的最坏情况下的运行时间是O(f(N)),则称f(N)是这个问题复杂度的一个**上界。**

当对算法的任何修改都不能在改进算法的运行时间时，此时算法的运行时间即为复杂度的**下界**。

当复杂度研究得出算法的上界与下界匹配时，我们可以确信试图设计比已知最好算法更快的算法是徒劳无益的，此时我们可以开始把注意力放在实现上。