算法分析

维基百科，自由的百科全书

在计算机科学中，**算法分析**（英语：Analysis of algorithm）是分析执行一个给定算法需要消耗的计算资源数量（例如计算时间，存储器使用等）的过程。算法的效率或复杂度在理论上表示为一个函数。其定义域是输入数据的长度（通常考虑任意大的输入，没有上界），值域通常是执行步骤数量（时间复杂度）或者存储器位置数量（空间复杂度）。算法分析是计算复杂度理论的重要组成部分。

理论分析常常利用渐近分析估计一个算法的复杂度，并使用大O符号、大Ω符号和大Θ符号作为标记。举例，二分查找所需的执行步骤数量与查找列表的长度之对数成正比，记为 $O(\log n)$，简称为“对数时间”。通常使用渐近分析的原因是，同一算法的不同具体实现的效率可能有差别。但是，对于任何给定的算法，所有符合其设计者意图的实现，它们之间的性能差异应当仅仅是一个系数。

精确分析算法的效率有时也是可行的，但这样的分析通常需要一些与具体实现相关的假设，称为计算模型。计算模型可以用抽象机器来定义，比如图灵机。或者可以假设某些基本操作在单位时间内可完成。

假设二分查找的目标列表总共有 n 个元素。如果我们假设单次查找可以在一个时间单位内完成，那么至多只需要 $\log n+1$ 单位的时间就可以得到结果。这样的分析在有些场合非常重要。

算法分析在实际工作中是非常重要的，因为使用低效率的算法会显著降低系统性能。在对运行时间要求极高的场合，耗时太长的算法得到的结果可能是过期或者无用的。低效率算法也会大量消耗计算资源。

目录

时间资源消耗

时间复杂度分析和如何定义“一步操作”有紧密联系。作为算法分析成立的一项基本要求，单步操作必须能够在确定的常量时间内完成。

实际情况很复杂。举例，有些分析方法假定两个数相加是单个步骤，但这假定可能不成立。若被分析的算法可以接受任意大的数，则无法保证相加操作能够在确定的时间内完成。

通常有两种定义消耗的方法：[1] [2] [3] [4] [5]

- 单一消耗：每一步操作的消耗定义为一个常量，与参与运算的数据的大小无关。
- 对数消耗：每一步操作的消耗，均与参与运算的数据的长度（位数）成正比。

后者更难以应用，所以只在必要时使用。一个例子是对接受任意精度数据的算法（比如密码学中用到的一些算法）的分析。

人们常常忽略一点：算法的效率的理论界限，通常建立在比实际情况更加严格的假定之上。因此在实际中，算法效率是有可能突破理论的界限的。[6]

经验分析的缺陷

算法是平台无关的，也即一个算法可以在任意计算机、任意操作系统上、用任意编程语言实现。因此，算法性能的相对好坏，不能仅仅通过基于运行记录的经验来判断。

举例：一个程序在大小为 n 的有序数组中搜索元素。假设该程序在一台先进的电脑 A 上用线性搜索实现，在一台老旧的电脑 B 上用二分搜索实现。性能测试的结果可能会如下：

数组长度 n	计算机 A 的运行时间（以纳秒计）	计算机 B 的运行时间（以纳秒计）
15	7	100,000
65	32	150,000
250	125	200,000
1,000	500	250,000

通过这些数据，很容易得出结论说计算机 A 运行的算法比计算机 B 的算法要高效得多。但假如输入的数组长度显著增加的话，很容易发现这个结论的错误。 以下是另一组数据：

数组长度 n	计算机 A 的运行时间（以纳秒计）	计算机 B 的运行时间（以纳秒计）
15	7	100,000
65	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
63,072 × 10^{12}	31,536 × 10^{12}纳秒，约等于 1 年	1,375,000 纳秒，或 1.375 毫秒

计算机 A 运行的线性搜索算法具有线性时间。它的运行时间直接与输入规模成正比。输入大小若加倍，运行时间同样加倍。而计算机 B 运行的二分搜索算法具有对数时间。输入大小若加倍，运行时间仅仅增加一个常量，在此例中是 25,000 纳秒。即使计算机 A 明显性能更强，在输入不断增加的情况下，计算机 B 的运行时间终究也会比计算机 A 更短，因为它运行的算法的增长率小得多。

增长的阶

非正式地，如果一个关于 n 的函数 $f(n)$，乘以一个系数以后，能够为某个算法在输入数据大小 n 足够大的情况下的运行时间提供一个上界，那么称此算法按该函数的阶增长。一个等价的描述是，当输入大小 n 大于某个 n_0 时，存在某个常数 c，使得算法的运行时间总小于 $c \times f(n)$。常用大O符号对此进行描述。比如，插入排序的运行时间随数据大小二次增长，那么插入排序具有 $O(n^2)$ 的时间复杂度。大O符号通常用于表示某个算法在最差情况下的运行时间，但也可以用来表述平均情况的运行时间。比如，快速排序的最坏运行时间是 $O(n^2)$，但是平均运行时间则是 $O(n \log n)$。[7]

运行时间复杂度的分析

分析一个算法的最坏运行时间复杂度时，人们常常作出一些简化问题的假设，并分析该算法的结构。以下是一个例子：

```
1    从输入值中获取一个正数
2    if n > 10
3        print "耗时可能较长，请稍候……"
4    for i = 1 to n
5        for j = 1 to i
6            print i * j
7    print "完成！"
```

一台给定的电脑执行每一条指令的时间是确定[8]的，并可以用 DTIME 描述。 假设第 1 步操作需时 T_1，第 2 步操作需时 T_2，如此类推。

步骤 1、2、7 只会运行一次。应当假设在最坏情况下，步骤 3 也会运行。步骤 1 至 3 和步骤 7 的总运行时间是：

$$T_1 + T_2 + T_3 + T_7$$

步骤 4、5、6 中的循环更为复杂。步骤 4 中的最外层循环会执行 (n + 1) 次（需要一次执行来结束 for 循环，因此是 (n + 1) 次而非 n 次），因此会消耗 $T_4 \times (n + 1)$ 单位时间。内层循环则由 i 的值控制，它会从 1 迭代到 n。 第一次执行外层循环时，j 从 1 迭代到 1，因此内层循环也执行一次，总共耗时 T_6 时间。以及内层循环的判断语句消耗 $3T_5$ 时间。

所以，内层循环的总共耗时可以用一个等差级数表示：

$$T_6 + 2T_6 + 3T_6 + \cdots + (n-1)T_6 + nT_6$$

上式可被因式分解[9]为：

$$T_6\left[1 + 2 + 3 + \cdots + (n-1) + n\right] = T_6\left[\frac{1}{2}(n^2 + n)\right]$$

类似地，可以分析内层循环的判断语句：

$$2T_5 + 3T_5 + 4T_5 + \cdots + (n-1)T_5 + nT_5 + (n+1)T_5$$

$$= T_5 + 2T_5 + 3T_5 + 4T_5 + \cdots + (n-1)T_5 + nT_5 + (n+1)T_5 - T_5$$

上式可被分解为：

$$T_5\left[1+2+3+\cdots+(n-1)+n+(n+1)\right]-T_5$$

$$=\left[\frac{1}{2}(n^2+n)\right]T_5+(n+1)T_5-T_5$$

$$=T_5\left[\frac{1}{2}(n^2+n)\right]+nT_5$$

$$=\left[\frac{1}{2}(n^2+3n)\right]T_5$$

因此该算法的总运行时间为：

$$f(n)=T_1+T_2+T_3+T_7+(n+1)T_4+\left[\frac{1}{2}(n^2+n)\right]T_6+\left[\frac{1}{2}(n^2+3n)\right]T_5$$

改写一下：

$$f(n)=\left[\frac{1}{2}(n^2+n)\right]T_6+\left[\frac{1}{2}(n^2+3n)\right]T_5+(n+1)T_4+T_1+T_2+T_3+T_7$$

通常情况下，一个函数的最高次项对它的增长率起到主导作用。在此例里，n^2 是最高次项，所以有结论 $f(n)=O(n^2)$。

严格证明如下：证明 $\left[\frac{1}{2}(n^2+n)\right]T_6+\left[\frac{1}{2}(n^2+3n)\right]T_5+(n+1)T_4+T_1+T_2+T_3+T_7\leq cn^2,\ n\geq n_0$

$$\left[\frac{1}{2}(n^2+n)\right]T_6+\left[\frac{1}{2}(n^2+3n)\right]T_5+(n+1)T_4+T_1+T_2+T_3+T_7$$

$$\leq(n^2+n)T_6+(n^2+3n)T_5+(n+1)T_4+T_1+T_2+T_3+T_7\ (n\geq 0)$$

令 k 为一个常数，其大于从 T_1 到 T_7 所有的数。

$$T_6(n^2+n)+T_5(n^2+3n)+(n+1)T_4+T_1+T_2+T_3+T_7\leq k(n^2+n)+k(n^2+3n)+kn+5k$$

$$=2kn^2+5kn+5k\leq 2kn^2+5kn^2+5kn^2\ (n\geq 1)=12kn^2$$

因此有

$\left[\frac{1}{2}(n^2+n)\right]T_6+\left[\frac{1}{2}(n^2+3n)\right]T_5+(n+1)T_4+T_1+T_2+T_3+T_7\leq cn^2,n\geq n_0$，其中 $c=12k,n_0=1$

还可以假定所有步骤全部消耗相同的时间，它的值比 T_1 到 T_7 中任意一个都大。这样的话，这个算法的运行时间就可以这样来分析：[10]

$$4+\sum_{i=1}^{n}i\leq 4+\sum_{i=1}^{n}n=4+n^2\leq 5n^2\ (n\geq 1)=O(n^2).$$

其他运算资源的增长率分析

运用与分析时间相同的方法可以分析其他运算资源的消耗情况，比如存储器空间的消耗。例如，考虑以下一段管理一个文件的内存使用的伪代码：

```
while (文件打开)
    令 n = 文件大小
    for n 每增长 100kb
        为该文件分配多一倍的内存空间
```

在这个例子里，当文件大小 n 增长的时候，内存消耗会以指数增长，或 $O(2^n)$。这个速度非常快，很容易使得资源消耗失去控制。

参见

- 平摊分析
- 渐近分析
- 渐近时间复杂度
- 计算时间

- 大O符号
- 计算复杂性理论
- 主定理
- NP-完全
- 数值分析
- 多项式时间
- 程序优化
- 性能分析
- 可扩放性
- 平滑分析
- 时间复杂度，包括常见算法的增长率列表

注释

1. Alfred V. Aho; John E. Hopcroft; Jeffrey D. Ullman. The design and analysis of computer algorithms. Addison-Wesley Pub. Co. 1974., section 1.3
2. Juraj Hromkovič. Theoretical computer science: introduction to Automata, computability, complexity, algorithmics, randomization, communication, and cryptography. Springer. 2004: 177–178. ISBN 978-3-540-14015-3.
3. Giorgio Ausiello. Complexity and approximation: combinatorial optimization problems and their approximability properties. Springer. 1999: 3–8. ISBN 978-3-540-65431-5.
4. Wegener, Ingo, Complexity theory: exploring the limits of efficient algorithms, Berlin, New York: Springer-Verlag: 20, 2005, ISBN 978-3-540-21045-0
5. Robert Endre Tarjan. Data structures and network algorithms. SIAM. 1983: 3–7. ISBN 978-0-89871-187-5.
6. Examples of the price of abstraction? (http://cstheory.stackexchange.com/questions/608/examples-of-the-price-of-abstraction), cstheory.stackexchange.com
7. 在算法分析的场合里，常用 $\log$ 或 $\lg$ 作为 $\log_2$ 的简称。
8. 但这对量子计算机不成立。
9. 可用数学归纳法证明 $1+2+3+\cdots+(n-1)+n=\frac{n(n+1)}{2}$
10. 比起上面的方法，这个方法忽略了结束循环的判断语句所消耗的时间，但很明显可以证明这种忽略不影响最后结果。

参考文献

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford. Introduction to Algorithms. Chapter 1: Foundations Second. Cambridge, MA: MIT Press and McGraw-Hill. 2001: 3–122. ISBN 0-262-03293-7.
- Sedgewick, Robert. Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching 3rd. Reading, MA: Addison-Wesley Professional. 1998. ISBN 978-0-201-31452-6.
- Knuth, Donald. The Art of Computer Programming. Addison-Wesley.
- Greene, Daniel A.; Knuth, Donald E. Mathematics for the Analysis of Algorithms Second. Birkhäuser. 1982. ISBN 3-7643-3102-X.
- Goldreich, Oded. Computational Complexity: A Conceptual Perspective. Cambridge University Press. 2010. ISBN 978-0-521-88473-0.