时间复杂度

在计算机科学中，算法的**时间复杂度**是一个函数，它定量描述了该算法的运行时间。这是一个代表算法输入值的字符串的长度的函数。时间复杂度常用大O符号表述，不包括这个函数的低阶项和首项系数。使用这种方式时，时间复杂度可被称为是渐近的，亦即考察输入值大小趋近无穷时的情况。例如，如果一个算法对于任何大小为n（必须比n_0大）的输入，它至多需要$5n^3 + 3n$的时间运行完毕，那么它的渐近时间复杂度是$O(n^3)$。

为了计算时间复杂度，我们通常会估计算法的操作单元数量，每个单元执行的时间都是相同的。因此，总运行时间和算法的操作单元数量最多相差一个常量系数。

相同大小的不同输入值仍可能造成算法的执行时间不同，因此我们通常使用算法的最坏情况复杂度，记为$T(n)$，定义为任何大小的输入n所需的最大执行时间。另一种较少使用的方法是平均情况复杂度，通常有特别指定才会使用。时间复杂度可以用函数$T(n)$的自然特性加以分类，举例来说，有著$T(n) = O(n)$的算法被称作“线性时间算法”；而$T(n) = O(M^n)$和$M^n = O(T(n))$，其中$M \geq n > 1$的算法被称作“指数时间算法”。

目录

常见时间复杂度列表

以下表格统整了一些常用的时间复杂度类别。表中，$\text{poly}(x) = x^{O(1)}$，也就是$x$的多项式。

名称	复杂度类	运行时间（$T(n)$）	运行时间举例	算法举例
常数时间		$O(1)$	10	判断一个二进制数的奇偶
反阿克曼时间		$O(\alpha(n))$		并查集的单个操作的平摊时间
迭代对数时间		$O(\log^* n)$		分散式圆环著色问题
对数对数时间		$O(\log\log n)$		有界优先队列的单个操作[1]
对数时间	DLOGTIME	$O(\log n)$	$\log n$，$\log n^2$	二分搜索
幂对数时间		$(\log n)^{O(1)}$	$(\log n)^2$	
（小于1次）幂时间		$O(n^c)$，其中 $0 < c < 1$	$n^{\frac{1}{2}}$，$n^{\frac{2}{3}}$	K-d树的搜索操作
线性时间		$O(n)$	n	无序数组的搜索
线性迭代对数时间		$O(n\log^* n)$		莱姆德·赛德尔的三角分割多边形算法
线性对数时间		$O(n\log n)$	$n\log n$，$\log n!$	最快的比较排序
二次时间		$O(n^2)$	n^2	冒泡排序、插入排序
三次时间		$O(n^3)$	n^3	矩阵乘法的基本实现，计算部分相关性
多项式时间	P	$2^{O(\log n)} = n^{O(1)}$	n，$n\log n$，n^{10}	线性规划中的卡马卡演算法，AKS质数测试
准多项式时间	QP	$2^{(\log n)^{O(1)}}$		关于有向斯坦纳树问题最著名的$O(\log^2 n)$近似算法
次指数时间（第一定义）	SUBEXP	$O(2^{n^\epsilon})$，对任意的 $\varepsilon > 0$	$O(2^{(\log n)^{\log\log n}})$	假设复杂性理论推测，BPP包含在SUBEXP中。[2]
次指数时间（第二定义）		$2^{o(n)}$	$2^{n^{1/3}}$	用于整数分解与图形同构问题的著名演算法
指数时间	E	$2^{O(n)}$	$1.1^n, 10^n$	使用动态规划解决旅行推销员问题
阶乘时间		$O(n!)$	$n!$	通过暴力搜索解决旅行推销员问题
指数时间	EXPTIME	$2^{\mathrm{poly}(n)}$	$2^n, 2^{n^2}$	
双重指数时间	2-EXPTIME	$2^{2^{\mathrm{poly}(n)}}$	2^{2^n}	在预膨胀算术中决定一个给定描述的真实性

常数时间

若对于一个算法，$T(n)$的上界与输入大小无关，则称其具有**常数时间**，记作$O(1)$时间。一个例子是访问数组中的单个元素，因为访问它只需要一条指令。但是，找到无序数组中的最小元素则不是，因为这需要遍历所有元素来找出最小值。这是一项线性时间的操作，或称$O(n)$时间。但如果预先知道元素的数量并假设数量保持不变，则该操作也可被称为具有常数时间。

虽然被称为“常数时间”，运行时间本身并不必须与问题规模无关，但它的**上界**必须是与问题规模无关的确定值。举例，“如果a > b则交换a、b的值”这项操作，尽管具体时间会取决于条件“a > b”是否满足，但它依然是常数时间，因为存在一个常量t使得所需时间总**不超过**t。

以下是一个常数时间的代码片段：

```
int index = 5;
int item = list[index];
if (condition true) then
   perform some operation that runs in constant time
else
   perform some other operation that runs in constant time
for i = 1 to 100
   for j = 1 to 200
      perform some operation that runs in constant time
```

如果$T(n) = O(c)$，其中c是一个常数，这记法等价于标准记法$T(n) = O(1)$。

对数时间

若算法的$T(n) = \mathbf{O(\log n)}$，则称其具有**对数时间**。由于计算机使用二进制的记数系统，对数常常以2为底（即$\log_2 n$，有时写作$\lg n$）。然而，由对数的换底公式，$\log_a n$和$\log_b n$只有一个常数因子不同，这个因子在大O记法中被丢弃。因此记作O（$\log n$），而不论对数的底是多少，是对数时间算法的标准记法。

常见的具有对数时间的算法有二叉树的相关操作和二分搜索。

对数时间的算法是非常有效的，因为每增加一个输入，其所需要的额外计算时间会变小。

递归地将字符串砍半并且输出是这个类别函数的一个简单例子。它需要O（log n）的时间因为每次输出之前我们都将字符串砍半。这意味着，如果我们想增加输出的次数，我们需要将字符串长度加倍。

```
// 递归输出一个字符串的右半部分
var right = function(str)
{
    var length = str.length;

    // 辅助函数
    var help = function(index)
    {

        // 递归情况：输出右半部分
        if(index < length){

            // 输出从index到数组末尾的部分
            console.log(str.substring(index, length));

            // 递归调用：调用辅助函数，将右半部分作为参数传入
            help(Math.ceil((length + index)/2));
        }

        // 基本情况：什么也不做
    }
    help(0);
}
```

幂对数时间

对于某个常数k，若算法的$T(n) = \mathrm{O}((\log n)^k)$，则称其具有**幂对数时间**。例如，矩阵链排序可以通过

一个PRAM模型.[3]被在幂对数时间内解决。

次线性时间

对于一个演算法，若其符合$T(n) = o(n)$，则其时间复杂度为**次线性时间**（**sub-linear time**或**sublinear time**）。实际上除了符合以上定义的演算法，其他一些演算法也拥有次线性时间的时间复杂度。例如有$O(n^{1/2})$ 葛罗佛搜寻演算法。

常见的非合次线性时间演算法都采用了诸如平行处理（就像NC_1 matrix行列式计算那样）、非古典处理（如同葛罗佛搜寻那样），又或者选择性地对有保证的输入结构作出假设（如幂对数时间的二分搜寻）。不过，一些情况，例如在头log(n)位元中每个字串有一个位元作为索引的字串组就可能依赖于输入的每个位元，但又符合次线性时间的条件。

“次线性时间演算法”通常指那些不符合前一段的描述的演算法。它们通常运行于传统电脑架构系列并且不容许任何对输入的事先假设。[4]但是它们可以是随机化算法，而且必须是真随机算法除了特殊情况。

线性时间

如果一个算法的时间复杂度为$O(n)$，则称这个算法具有线性时间，或**$O(n)$**时间。非正式地说，这意味着对于足够大的输入，运行时间增加的大小与输入成线性关系。例如，一个计算列表所有元素的和的程序，需要的时间与列表的长度成正比。这个描述是稍微不准确的，因为运行时间可能显著偏离一个精确的比例，尤其是对于较小的n。

线性对数（准线性）时间

若一个算法时间复杂度T(n) = O(nlog n)，则称这个算法具有线性对数时间。因此，从其表达式我们也可以看到，线性对数时间增长得比线性时间要快，但是对于任何含有n，且n的幂指数大于1的多项式时间来说，线性对数时间却增长得慢。

多项式时间

强多项式时间与弱多项式时间

复杂度类

从多项式时间的概念出发，在计算复杂度理论中可以得到一些复杂度类。以下是一些重要的例子。

- **P**：包含可以使用确定型图灵机在多项式时间内解决的决定性问题。
- **NP**：包含可以使用非确定型图灵机在多项式时间内解决的决定性问题。
- **ZPP**：包含可以使用概率图灵机在多项式时间内零错误解决的决定性问题。
- **RP**：包含可以使用概率图灵机在多项式时间内解决的决定性问题，但它给出的两种答案中(是或否）只有一种答案是一定正确的，另一种则有几率不正确。
- **BPP**：包含可以使用概率图灵机在多项式时间内解决的决定性问题，它给出的答案有错误的概率在某个小于0.5的常数之内。
- **BQP**：包含可以使用量子图灵机在多项式时间内解决的决定性问题，它给出的答案有错误的概率在某个小于0.5的常数之内。

在机器模型可变的情况下，P在确定性机器上是最小的时间复杂度类。例如，将单带图灵机换成多带图灵机可以使算法运行速度以二次阶提升，但所有具有多项式时间的算法依然会以多项式时间运行。一种特定的抽象机器会有自己特定的复杂度类分类。

超越多项式（superpolynomial）时间

如果一个算法的时间$T(n)$没有任何多项式上界，则称这个算法具有**超越多项式时间**。在这种情况下，对于所有常数c我们都有$T(n) = \omega(n^c)$，其中n是输入参数，通常是输入的数据量（比特数）。指数时间显然属于超越多项式时间，但是有些算法仅仅是很弱的超越多项式算法。例如，Adleman-Pomerance-Rumely质数测试对于n比特的输入需要运行$n^{O(\log \log n)}$时间；对于足够大的n，这时间比任何多项式都快；但是输入要大得不切实际，时间才能真正超过低阶的多项式。

准多项式时间

准多项式时间演算法是运算慢于多项式时间的演算法，但不会像指数时间那么慢。对一些固定的$c > 0$，准多项式时间演算法的最坏情况运行时间是$2^{O((\log n)^c)}$。如果准多项式时间演算法定义中的常数“c”等于1，则得到多项式时间演算法；如果小于1，则得到一个次线性时间算法。

次指数时间

术语次指数时间用于表示某些演算法的运算时间可能比任何多项式增长得快，但仍明显小于指数。在这种状况下，具有次指数时间演算法的问题比那些仅具有指数演算法的问题更容易处理。“次指数”的确切定义并没有得到普遍的认同，[5]我们列出了以下两个最广泛使用的。

第一定义

如果一个问题解决的运算时间的对数值比任何多项式增长得慢，则可以称其为次指数时间。更准确地说，如果对于每个ε> 0，存在一个能于时间$O(2^{n^\varepsilon})$内解决问题的演算法，则该问题为次指数时间。所有这些问题的集合是复杂性SUBEXP，可以按照DTIME的方式定义如下。[2][6][7][8]

$$\mathrm{SUBEXP} = \bigcap_{\varepsilon>0} \mathrm{DTIME}\left(2^{n^\varepsilon}\right)$$

第二定义

一些作者将次指数时间定义为$2^{o(n)}$的运算时间。[9][10][11]该定义允许比次指数时间的第一个定义更多的运算时间。这种次指数时间演算法的一个例子，是用于整数因式分解的最著名古典演算法——普通数域筛选法，其运算时间约为$2^{\tilde{O}(n^{1/3})}$，其中输入的长度为n。另一个例子是图形同构问题的最著名演算法，其运算时间为$2^{O(\sqrt{n \log n})}$。

指数时间

若$T(n)$是以$2^{\mathrm{poly}(n)}$为上界，其中$\mathrm{poly}(n)$是n的多项式，则演算法被称为**指数时间**。更正规的讲法是：若$T(n)$对某些常数k是由$O(2^{n^k})$所界定，则演算法被称为**指数时间**。在确定性图灵机上认定为指数时间演算法的问题，形成称为**EXP**的复杂性级别。

$$\mathrm{EXP} = \bigcup_{c\in\mathbb{N}} \mathrm{DTIME}\left(2^{n^c}\right)$$

有时侯，指数时间用来指称具有$T(n) = 2^{O(n)}$的演算法，其中指数最多为n的线性函数。这引起复杂性等级**E**。

$$\mathrm{E} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}\left(2^{cn}\right)$$

双重指数时间

若$T(n)$是以$2^{2^{\mathrm{poly}(n)}}$为上界，其中$\mathrm{poly}(n)$是n的多项式，则演算法被称为双重指数时间。这种演算法属于复杂性等级2-EXPTIME。

$$\text{2-EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}\left(2^{2^{n^c}}\right)$$

众所周知的双重指数时间演算法包括：

- 预膨胀算术的决策程序
- 计算葛洛拿基底（在最差状况[12]）
- 实封闭体的量词消去至少耗费双重指数时间，[13]而且可以在这样的时间内完成。[14]

参见

- L-notation

参考资料

1. Mehlhorn, Kurt; Naher, Stefan. Bounded ordered dictionaries in O(log log N) time and O(n) space. Information Processing Letters. 1990, **35** (4): 183. doi:10.1016/0020-0190(90)90022-P.
2. Babai, László; Fortnow, Lance; Nisan, N.; Wigderson, Avi. BPP has subexponential time simulations unless EXPTIME has publishable proofs. Computational Complexity (Berlin, New York: Springer-Verlag). 1993, **3** (4): 307–318. doi:10.1007/BF01275486.
3. Bradford, Phillip G.; Rawlins, Gregory J. E.; Shannon, Gregory E. Efficient Matrix Chain Ordering in Polylog Time. SIAM Journal on Computing (Philadelphia: Society for Industrial and Applied Mathematics). 1998, **27** (2): 466–490. ISSN 1095-7111. doi:10.1137/S0097539794270698.
4. Kumar, Ravi; Rubinfeld, Ronitt. Sublinear time algorithms (PDF). SIGACT News. 2003, **34** (4): 57–67.
5. Aaronson, Scott. A not-quite-exponential dilemma. Shtetl-Optimized. 5 April 2009 [2 December 2009].
6. *Complexity Zoo*: Class SUBEXP: Deterministic Subexponential-Time (http://qwiki.stanford.edu/index.php/Complexity_Zoo:S#subexp)
7. Moser, P. Baire's Categories on Small Complexity Classes. Lecture Notes in Computer Science (Berlin, New York: Springer-Verlag). 2003: 333–342. ISSN 0302-9743.
8. Miltersen, P.B. DERANDOMIZING COMPLEXITY CLASSES. Handbook of Randomized Computing (Kluwer Academic Pub). 2001: 843.
9. **引用错误：没有为名为ETH的参考文献提供内容**
10. Kuperberg, Greg. A Subexponential-Time Quantum Algorithm for the Dihedral Hidden Subgroup Problem. SIAM Journal on Computing (Philadelphia: Society for Industrial and Applied Mathematics). 2005, **35** (1): 188. ISSN 1095-7111. doi:10.1137/s0097539703436345.
11. Oded Regev. A Subexponential Time Algorithm for the Dihedral Hidden Subgroup Problem with Polynomial Space. *arXiv:quant-ph/0406151v1 (http://arxiv.org/abs/quant-ph/0406151v1) [quant-ph (http://arxiv.org/archive/quant-ph)]*. 2004.
12. Mayr,E. & Mayer,A.: The Complexity of the Word Problem for Commutative Semi-groups and Polynomial Ideals. *Adv. in Math.* 46(1982) pp. 305-329
13. J.H. Davenport & J. Heintz: Real Quantifier Elimination is Doubly Exponential. *J. Symbolic Comp.* 5 (1988) pp. 29-35.

14. G.E. Collins: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. Proc. 2nd. GI Conference Automata Theory & Formal Languages (Springer Lecture Notes in Computer Science 33) pp. 134-183

- 本页面最后修订于2017年7月22日 (星期六) 00:44。