



上海师范大学  
Shanghai Normal University

# 《算法设计与分析》

## 1-算法分析基础 (Fundamentals)

杨启哲

上海师范大学信机学院计算机系

2025 年 9 月 18 日



- › 从 Fibonacci 数列开始
- › 算法分析基础



- 从 Fibonacci 数列开始
- 算法分析基础

12 世纪，意大利数学家斐波那契 (Leonardo Fibonacci) 如下描述了兔子生长的数目：

- 第一个月初有一对刚出生的兔子。
- 第二个月后（第三个月初）它们可以生育。
- 每月每对可生育的兔子会诞生下一对新兔子。
- 兔子永不死去。



易得每个月的兔子数量是如下的数列：1, 1, 2, 3, 5, 8, 13, ...

Leonardo Fibonacci(1170-1250)

## 斐波那契数列 (Fibonacci sequence)

斐波那契数列  $F_n$  的定义如下：

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

如何计算 Fibonacci 数列？

通项公式？

## Fibonacci 数列通项公式

$$F_n = \frac{\sqrt{5}}{5} \cdot \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right]$$

计算机如何计算一个无理数，甚至是无理数的幂？

我们可以让计算机取迭代从而避免无理数的计算。

## 第一个算法: $\text{Fib}_1(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

```
1: if  $n=0$  then  
2:   return 0  
3: else if  $n=1$  then  
4:   return 1  
5: end if  
6: return  $\text{Fib}_1(n-1) + \text{Fib}_1(n-2)$ 
```



## 三个问题



上海师范大学  
Shanghai Normal University

面对一个算法，我们需要考虑如下三个问题：

1. 这个算法是正确的么？
2. 这个算法需要耗费多少时间？
3. 有更快的算法么？



# 三个问题



上海师范大学  
Shanghai Normal University

面对一个算法，我们需要考虑如下三个问题：

1. 这个算法是正确的么？
2. 这个算法需要耗费多少时间？
3. 有更快的算法么？





面对一个算法，我们需要考虑如下三个问题：

1. 这个算法是正确的么？
2. 这个算法需要耗费多少时间？
3. 有更快的算法么？

令  $T(n)$  为运行  $\text{Fib}_1(n)$  所需要执行的基本操作次数。

- 当  $n < 2$  时，可以发现该算法执行的操作次数非常少，因此此时  $T(n) \leq 2$ 。
- 当  $n \geq 2$  时， $\text{Fib}_1(n)$  执行的基本操作次数为

$$T(n) = T(n-1) + T(n-2) + 3$$

### $T(n)$ 有多大？

很不幸， $T(n)$  比斐波那契数列的第  $n$  项还要大！（ $T(n) \geq F_n$ ，它是一个指数算法！）

如果用该算法计算  $F_{400}$ ，则需要执行  $T(400) \approx 2^{277}$  次基本操作！

目前最快的超级计算机 [Frontier](#) 每秒可以执行约  $10^{18}$  次基本操作，这意味着即使在这台机器上  $\text{Fib}_1(400)$  也要耗时  $2^{200}$  秒，而地球诞生至今也不过经过了  $2^{60}$  秒。



## 三个问题



上海师范大学  
Shanghai Normal University

面对一个算法，我们需要考虑如下三个问题：

1. 这个算法是正确的么？
2. 这个算法需要耗费多少时间？
3. 有更快的算法么？

### 第二个算法: $\text{Fib}_2(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

```
1: if  $n=0$  then
2:   return 0
3: end if
4: Define Array  $f[0, \dots, n]$ 
5:  $f[0] \leftarrow 0, f[1] \leftarrow 1$ 
6: for  $i \leftarrow 2$  to  $n$  do
7:    $f[i] \leftarrow f[i-1] + f[i-2]$ 
8: end for
9: return  $f[n]$ 
```

- 这个算法是正确的么？ 显然正确
- 这个算法需要耗费多少时间？  
由于存储下来了之前的结果，在  $\text{Fib}_2(n)$  中，循环仅执行了  $n - 1$  次。因此  $\text{Fib}_2(n)$  的基本操作次数关于  $n$  是线性的。  
 $\text{Fib}_2(n)$  是一个多项式时间的算法，我们可以很快的计算出  $F_{400}$  了！
- 有更快的算法么？

运用矩阵的一些运算，我们可以发现  $F_1, F_2, F_0$  满足下列等式

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

同样地，我们有：

$$\begin{bmatrix} F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

因此，我们可以求得相应的一般式：

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

令  $X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$  为相应的矩阵，如果我们求得  $X^n$ ，则可以很快的求出相应的  $F_n$ 。

如何求  $X^n$ ？

二分！通过不断的二分，我们可以只用  $O(\log n)$  次

矩阵乘法就可以求得  $X^n$ 。

$$\begin{array}{c} X^{23} \\ | \\ X \times (X^{11})^2 \\ | \\ X \times (X^5)^2 \\ | \\ X \times (X^2)^2 \\ | \\ (X)^2 \end{array}$$

二分计算  $X^{23}$  的流程

### 第三个算法: $\text{Fib}_3(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

- 1: **Define**  $X \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ ,  $Y \leftarrow \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$
- 2: **Calculate:**  $X \leftarrow X^n$
- 3:  $Y \leftarrow X$
- 4: **return**  $Y_{11}$



根据我们之前的讨论， $\text{Fib}_3(n)$  只需要进行  $O(\log n)$  次算术操作并可以获得  $F_n$ ，那么我们是否可以说它是一个更快的算法，并且相比于  $\text{Fib}_2(n)$  提升了指数级的效率？

并不可以！

## 不同的基本操作次数

尽管看上去  $\text{Fib}_3(n)$  只用了对数次算术操作，但是与  $\text{Fib}_2(n)$  相比：

- $\text{Fib}_2(n)$  的基本操作是加法。
- $\text{Fib}_3(n)$  的基本操作是乘法。

乘法操作和加法操作一样快么？

重新考虑两个数的加法，事实上如果两个长度为  $n$  的二进制数相加，我们需要进行  $O(n)$  次基本操作。

$$\begin{array}{r} 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1 \\ +\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1 \end{array}$$

图: 二进制数相加举例

因此对于 Fib<sub>2</sub>(n) 来说，其需要进行  $O(n^2)$  次基本操作。

而对于 Fib<sub>3</sub>(n) 来说, 其需要进行  $O(\log n)$  次乘法操作, 假设一次乘法操作需要  $M(n)$  次基本操作, 则 Fib<sub>3</sub>(n) 的基本操作次数为  $O(M(n)\log n)$ 。

所以是否存在:

$$M(n)\log n < n^2?$$

这取决于  $M(n)$  是否能比  $O(n^2)$  更快, 即我们能否以少于  $O(n^2)$  次基本操作的代价完成两个长度为  $n$  的二进制数的乘法。

我们将在后续的课程给出答案。

## 本节总结

- 了解清楚我们所面对的问题。
- 给出一个解决方案，也就是相应的算法。
- 对于给出的算法，我们需要考虑如下三个问题：
  1. 这个算法是正确的么？
  2. 这个算法需要耗费多少时间？
  3. 有更快的算法么？

## 运行时间

如何来衡量算法的运行时间？



- 从 Fibonacci 数列开始
- 算法分析基础

## ► 算法分析基础

### 算法时间估计

如果仅关注于一个算法对于某个输入运行了多少秒是没有意义的，因为即使考虑的是同一个问题，算法的运行时间会受到各个因素的影响，比如：

- 硬件上来说，CPU、内存、缓存等都会影响算法的运行时间。
- 软件上来说，使用的语言、编译器、操作系统等也都会影响算法的运行时间。
- 随着科技的发展，计算机的速度只会运行的越来越快。

## 独立性

因此在考察算法的运行时间时，我希望我们得到的结果是**独立的**，这是指：

- 独立于所使用的语言、编译器、操作系统等。
- 独立于科技的发展。

我们需要一些数学的方法。回想在计算 Fibonacci 数的例子里，我们看到，通过一些基本运算的次数来估计算法的运行时间是很有必要的。

## 算法的运行时间

一个算法的运行时间可以理解为：

$$\text{运行时间} = \sum_{\text{所有的操作}} \text{操作次数} \times \text{该操作所需的时间}$$

但我们有必要去考虑所有的操作么？



### 1-SUM

输入: 数组  $a[n]$

输出: 数组中元素为 0 的个数

```
1: count  $\leftarrow$  0
2: for i  $\leftarrow$  1 to n do
3:   if  $a[i] == 0$  then
4:     count  $\leftarrow$  count + 1
5:   end if
6: end for
7: return count
```

在这样一个算法中，有如下的操作：变量声明，变量赋值，小于判断，相等判断，加法

### 1-SUM 的运行次数

因此对于上述算法中的任一次运行，可能的操作次数为：

- 变量声明：2 次
- 变量赋值：2 次
- 小于判断： $n + 1$  次
- 相等判断： $n$  次
- 加法： $n \sim 2n$  次

但我们可以看到，整个算法进行了  $n$  次循环，任何一个操作执行的次数都是  $n$  的常数倍，因此我们只需要考虑循环的次数即可。

我们只需要估计其中一个基本运算甚至某些度量，保证其他运算至多是它的常数倍即可。

再次回顾计算 Fibonacci 数列的例子，即使是  $\text{Fib}_1(n)$ ，在计算很小的输入时我们也能很快的获得答案，效率甚至会比  $\text{Fib}_2(n)$ ,  $\text{Fib}_3(n)$  都要更快。

## 第一个算法: $\text{Fib}_1(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

```
1: if  $n=0$  then
2:   return 0
3: else if  $n=1$  then
4:   return 1
5: end if
6: return  $\text{Fib}_1(n-1) + \text{Fib}_1(n-2)$ 
```

## 第二个算法: $\text{Fib}_2(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

```
1: if  $n=0$  then
2:   return 0
3: end if
4: Define Array  $f[0, \dots, n]$ 
5:  $f[0] \leftarrow 0, f[1] \leftarrow 1$ 
6: for  $i \leftarrow 2$  to  $n$  do
7:    $f[i] \leftarrow f[i-1] + f[i-2]$ 
8: end for
9: return  $f[n]$ 
```

## 第三个算法: $\text{Fib}_3(n)$

输入: 正整数  $n$

输出: 第  $n$  个 Fibonacci 数  $F_n$

```
1: Define  $X \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ ,  $Y \leftarrow \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$ 
2: Calculate:  $X \leftarrow X^n$ 
3:  $Y \leftarrow X$ 
4: return  $Y_{11}$ 
```

因此，小规模输入的运行时间没有意义，我们要考虑的时大规模输入的情况下算法的运行时间。



## 估计算法运行时间的考虑因素

- 我们关注的衡量标准是独立的，与机器等无关。
- 我们需要关注的是相对的、近似的时间，而不是绝对时间。
- 我们需要关注的是大规模输入的情况，而不是小规模输入的情况。

比较下述两个算法:

## 算法 First

输入: 数组  $a[n]$ ,  $a[j] = j, 1 \leq j \leq n$

输出:  $\sum_{j=1}^n j$

- 1:  $\text{sum} \leftarrow 0$
- 2: **for**  $i \leftarrow 1$  to  $n$  **do**
- 3:      $\text{sum} \leftarrow \text{sum} + a[i]$
- 4: **end for**
- 5: **return**  $\text{sum}$

## 算法 Second

输入: 正整数  $n$

输出:  $\sum_{j=1}^n j$

- 1:  $\text{sum} \leftarrow 0$
- 2: **for**  $i \leftarrow 1$  to  $n$  **do**
- 3:      $\text{sum} \leftarrow \text{sum} + i$
- 4: **end for**
- 5: **return**  $\text{sum}$

	First	Second
输入规模	$n$	$\log n$
运行时间	$n$	$n$
相互关系	线性	指数

不同的输入规模对于算法的运行时间有着不同的影响!

## 一些常用的输入规模的测度

- 排序和搜索问题：数组或表中元素的个数。
- 图问题：图中顶点的个数和边的个数。
- 计算几何：点、边、线段或者多边形等的数目。
- 矩阵运算：输入矩阵的维数。
- 数论算法和密码学：用来表示输入数的位数 (一般为  $\log n$ )



## 算法分析基础

最好运行时间, 平均运行时间和最坏运行时间

### 在有序数组中搜索相应的元素

给定一个有序数组  $A[1, \dots, n]$  和一个元素  $x$ ，请问  $x$  是否在数组中？存在的话请返回相应的下标，否则请返回  $-1$ 。

我们下面提供两个不同的搜索算法，一个即从头开始搜索，我们称为线性搜索 (Linear Search)，另一个则是二分搜索 (Binary Search)。



## 线性搜索 LinearSearch

**输入:** 有序数组  $a[1, \dots, n]$  和元素  $x$

**输出:**  $x$  在数组中的下标, 不存在则返回  $-1$

```
1:  $j \leftarrow 1$ 
2: while  $j < n$  and  $x \neq a[j]$  do
3:    $j \leftarrow j + 1$ 
4: end while
5: if  $x = a[j]$  then
6:   return  $j$ 
7: else
8:   return  $-1$ 
9: end if
```

## 二分搜索 BinarySearch

**输入:** 有序数组  $a[1, \dots, n]$  和元素  $x$

**输出:**  $x$  在数组中的下标, 不存在则返回  $-1$

```
1:  $low \leftarrow 1$ ,  $high \leftarrow n$ ,  $j \leftarrow 0$ 
2: while  $low \leq high$  and  $j = 0$  do
3:    $mid \leftarrow \lfloor (low + high) / 2 \rfloor$ 
4:   if  $x = a[mid]$  then
5:      $j \leftarrow mid$ 
6:   else if  $x < a[mid]$  then
7:      $high \leftarrow mid - 1$ 
8:   else  $low \leftarrow mid + 1$ 
9:   end if
10: end while
11: return  $j$ 
```

考察如下的一个  $n$  元数组：

$$1, 2, 3, \dots, n-1, n$$

### 寻找不同的 $x$

- $x = 1$  时，LinearSearch 需要执行 1 次比较操作，BinarySearch 需要执行  $\log n$  次比较操作。
- $x = n$  时，LinearSearch 需要执行  $n$  次比较操作，BinarySearch 需要执行  $\log n$  次比较操作。
- $x = n/2$  时，LinearSearch 需要执行  $n/2$  次比较操作，BinarySearch 需要执行 1 次比较操作。

可以看到，在面对不同的  $x$  的时候，有的时候 LinearSearch 会更快些，有的时候 BinarySearch 会更快些。

## 定义 1

[最好运行时间].

算法的最好运行时间指的是在所有输入规模为  $n$  的输入中，时间最短的那个。

## 定义 2

[最坏运行时间].

算法的最小运行时间指的是在所有输入规模为  $n$  的输入中，时间最长的那个。

## 定义 3

[平均运行时间].

算法的平均运行时间指的是在所有输入规模为  $n$  的输入中，算法的平均运行时间。

在 LinearSearch 和 BinarySearch 中，算法运行的最好时间，最坏时间、平均时间都分别是什么？

	LinearSearch	BinarySearch
最好运行时间	1	1
最坏运行时间	$n$	$\log n$
平均运行时间	$O(n)$	$O(\log n)$

平均时间的分析需要一些概率论的知识，我们这里先不给出详细的证明。

## ► 算法分析基础

渐进符号

前面我们讨论到，其实我们只对算法在大规模的输入上的运行时间感兴趣。那么假设一个算法的运行时间  $T(n)$  满足：

$$T(n) = 2n^3 + 192832n^2 + 1223n + 322\log n + 293239$$

我们是否需要关注后面那些复杂的项数？

---

**理论上不需要！** 当  $n$  足够大的时候，后面都将比  $n^3$  小，因此我们会有  $T(n) < 3n^3$ 。

因此对应这样一个算法，它的运行时间主要是由  $n^3$  这一项决定的，甚至在很多时候我们可以忽略掉这一项上的系数。(尽管有的时候非常重要) 换句话说， $n^3$  是可以用来衡量该算法运行时间的一个指标，即某种渐近运行时间，我们也称之为它的**阶**。

## 定义 4

## [大 O 符号].

令  $f(n), g(n)$  是两个从自然数集到非负实数集的两个函数，如果存在一个自然数  $n_0$  和常数  $c > 0$ ，使得

$$\forall n \geq n_0, f(n) \leq cg(n)$$

则称  $f(n)$  是  $O(g(n))$  的。

因此如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在，那么：

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty \text{ 蕴含着 } f(n) = O(g(n))$$

## 补充说明

大 O 符号不严格的说，可以视为提供了某种上界，即  $f$  的大小不会比  $g$  的某个常数倍大。



### 例 5.

1.  $n^2 + 3n + 1 = O(n^2)$ .
2.  $\log n^2 = O(\log n)$ .
3.  $\log n! = O(n \log n)$ .
4.  $2n^{0.0001} + 3(\log n)^{100} = O(n^{0.0001})$ .
5.  $2^n + 100n^{100} = O(2^n)$ .
6.  $n^n + 2^n + 4n^5 = O(2^{n \log n})$ .
7.  $n^2 + 3n + 1 = O(n^3)$ .



## 定义 6

## [大 $\Omega$ 符号].

令  $f(n), g(n)$  是两个从自然数集到非负实数集的两个函数, 如果存在一个自然数  $n_0$  和常数  $c > 0$ , 使得

$$\forall n \geq n_0, f(n) \geq cg(n)$$

则称  $f(n)$  是  $\Omega(g(n))$  的。

因此如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在, 那么:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq 0 \text{ 蕴含着 } f(n) = \Omega(g(n))$$

## 补充说明

大  $\Omega$  符号不严格的说, 可以视为提供了某种下界, 即  $f$  的大小不会比  $g$  的某个常数倍小。

### 例 7.

1.  $n^2 + 3n + 1 = \Omega(n^2)$ .
2.  $\log n^k = \Omega(\log n)$ .
3.  $\log n! = \Omega(n \log n)$ .
4.  $n! = \Omega(2^n)$ .

---

由定义可知:  $f = O(g) \Leftrightarrow g = \Omega(f)$

是否存在  $f, g$ , 使得  $f = O(g)$  并且  $f = \Omega(g)$ ?

## 定义 8

[大  $\Theta$  符号].

令  $f(n), g(n)$  是两个从自然数集到非负实数集的两个函数, 如果存在一个自然数  $n_0$  和常数  $c_1, c_2 > 0$ , 使得

$$\forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

则称  $f(n)$  是  $\Theta(g(n))$  的。

因此如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在, 那么:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \text{ 蕴含着 } f(n) = \Theta(g(n))$$

其中  $c$  是一个大于 0 的常数。

显然,  $f(n) = \Theta(g(n))$  当且仅当  $f(n) = O(g(n)), f(n) = \Omega(g(n))$ 。



### 例 9.

1.  $n^2 + 3n + 1 = \Theta(n^2)$ .
2.  $\log n^2 = \Theta(\log n)$ .
3.  $\log n! = \Theta(n \log n)$ .
4.  $2n^{0.0001} + 3(\log n)^{100} = \Theta(n^{0.0001})$ .
5.  $2^n + 100n^{100} = \Theta(2^n)$ .
6.  $n^n + 2^n + 4n^5 = \Theta(2^{n \log n})$ .

## 定义 10

[小 o 符号].

令  $f(n), g(n)$  是两个从自然数集到非负实数集的两个函数，**如果对于任意的常数  $c > 0$  都存在自然数  $n_0$** ，使得

$$\forall n \geq n_0, f(n) < cg(n)$$

则称  $f(n)$  是  $o(g(n))$  的。

因此如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在，那么：

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \text{ 蕴含着 } f(n) = o(g(n))$$

## 补充说明

小 o 符号不严格的说，可以视为提供了某种更大的关系，即相比于  $g$  在  $n$  足够大时可以忽略掉  $f$  的大小。

小 O 符号可以更清楚的表示上界的关系。比如在之前的例子中，我们有：

- $n^2 + 3n + 1 = O(n^2)$ .
- $n^2 + 3n + 1 = O(n^3)$ .

但事实上，我们可以更清楚的表示这种关系，即：

$$n^2 + 3n + 1 = o(n^3) \text{ 但是 } n^2 + 3n + 1 \neq o(n^2)$$

### 例 11.

1.  $\log n! = o(n^2)$ .
2.  $n = o(n \log n)$ .

同样的，我们可以更精确的来描述一些下界的关系。

## 定义 12

[小  $\omega$  符号].

令  $f(n), g(n)$  是两个从自然数集到非负实数集的两个函数，**如果对于任意的常数  $c > 0$  都存在自然数  $n_0$ ，使得**

$$\forall n \geq n_0, f(n) > c g(n)$$

则称  $f(n)$  是  $\omega(g(n))$  的。

因此如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在，那么：

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \text{ 蕴含着 } f(n) = \omega(g(n))$$

## 运算技巧

1.  $\sum_{j=1}^n j = \frac{n(n+1)}{2} = \Theta(n^2)$ 。
2.  $\sum_{j=0}^n c^j = \frac{c^{n+1}-1}{c-1} = \Theta(c^n)$ 。
3.  $\sum_{j=1}^n j^k = \Theta(n^{k+1})$ 。
4.  $H_n = \sum_{j=1}^n \frac{1}{j} = \Theta(\ln n) = \Theta(\log n)$ 。
5.  $\sum_{j=1}^n \log j = \Theta(n \log n)$
6.  $f(x)$  递增时  $\int_{m-1}^n f(x) dx \leq \sum_{j=m}^n f(j) \leq \int_m^{n+1} f(x) dx$ 。
7.  $f(x)$  递减时  $\int_m^{n+1} f(x) dx \leq \sum_{j=m}^n f(j) \leq \int_{m-1}^n f(x) dx$ 。



## 定义 13

[复杂性类].

令  $R$  是复杂性函数集合上定义的一个等价关系:

$$fRg \text{ 当且仅当 } f(n) = \Theta(g(n))$$

由该等价关系导出的等价性类被称为复杂性类。

我们也用  $f \prec g$  表示  $f(n) = o(g(n))$ , 则有:

$$1 \prec \log \log n \prec \log n \prec n^{0.75} \prec n \prec n \log n \prec n^{1.5} \prec 2^n \prec n! \prec 2^{2^n} \dots$$

假设一台电脑每秒可以执行  $10^6$  次基本操作，那么我们可以估计出不同阶下的运行速度：

阶	名称	算法实例	$n = 1000$	$n = 2000$
1	常数	返回数组某个位置的元素	立即	立即
$\log n$	对数	二分搜索	立即	立即
$n$	线性	线性搜索	立即	立即
$n \log n$	线性对数 (linearithmic)	归并排序	立即	立即
$n^2$	平方 (quadratic)	选择排序	~ 1 秒	~ 2 秒
$2^n$	指数 (exponential)	汉诺威塔 (Hanoi)	几乎永久	几乎永久

我们前面关注的都是算法的时间复杂性：

- 对于运行时间来说，越快越好。

同样，算法也有空间复杂性，对其消耗的空间进行分析：

- 对于运行空间来说，越少越好。

---

## 空间复杂性与时间复杂性的关系

空间复杂性  $\leq$  时间复杂性

$f$  和  $g$  满足什么关系,  $f = O(g)$ ,  $f = \Omega(g)$ ,  $f = \Theta(g)$ ,  $f = o(g)$ ,  $f = \omega(g)$ ?

1.  $f(n) = n - 100$ ,  $g(n) = n - 200$   $f = \Theta(g)$
2.  $f(n) = n^{1/2}$ ,  $g(n) = n^{2/3}$   $f = o(g)$
3.  $f(n) = 100n + \log n$ ,  $g(n) = n + (\log n)^2$   $f = \Theta(g)$
4.  $f(n) = \log 2n$ ,  $g(n) = \log 3n$   $f = \Theta(g)$
5.  $f(n) = 10 \log n$ ,  $g(n) = \log n^2$   $f = \Theta(g)$
6.  $f(n) = \sqrt{n}$ ,  $g(n) = (\log n)^{10}$   $f = \omega(g)$
7.  $f(n) = (\log n)^{\log n}$ ,  $g(n) = \frac{n}{\log n}$   $f = \omega(g)$
8.  $f(n) = n^{1/2}$ ,  $g(n) = 5^{\log_2 n}$   $f = o(g)$
9.  $f(n) = \sum_{i=1}^n i^k$ ,  $g(n) = n^{k+1}$   $f = \Theta(g)$

请分析下列算法的时间复杂性:

### 算法 1.9: Count2

输入: 正整数  $n$

输出: 第 5 步的执行次数  $\text{count}$

```

1:  $\text{count} \leftarrow 0$ 
2: for  $i \leftarrow 1$  to  $n$  do
3:    $m \leftarrow \lfloor \frac{n}{i} \rfloor$ 
4:   for  $j \leftarrow 1$  to  $m$  do
5:      $\text{count} \leftarrow \text{count} + 1$ 
6:   end for
7: end for
8: return  $\text{count}$ 
    
```

$\Theta(n \log n)!$

请分析下列算法的时间复杂性:

## 算法 1.10: Count3

输入:  $n = 2^k$ ,  $k$  为正整数

输出: 第 5 步的执行次数 count

```

1: count  $\leftarrow$  0
2: i  $\leftarrow$  1
3: while i  $\leq$  n do
4:   for j  $\leftarrow$  1 to i do
5:     count  $\leftarrow$  count + 1
6:   end for
7:   i  $\leftarrow$  2i
8: end while
9: return count
    
```

$\Theta(n)!$

请分析下列算法的时间复杂性:

### 算法 1.11: Count4

输入:  $n = 2^k$ ,  $k$  为正整数

输出: 第 4 步的执行次数 count

```
1: count  $\leftarrow$  0
2: while  $n \geq 1$  do
3:   for  $j \leftarrow 1$  to  $n$  do
4:     count  $\leftarrow$  count + 1
5:   end for
6:    $n \leftarrow \frac{n}{2}$ 
7: end while
8: return count
```

$\Theta(n)!$

请分析下列算法的时间复杂性:

### 算法 1.11: Count5

输入:  $n = 2^{2^k}$ ,  $k$  为正整数

输出: 第 6 步的执行次数 count

```
1: count  $\leftarrow$  0
2: for  $i \leftarrow 1$  to  $n$  do
3:    $j \leftarrow 2$ 
4:   while  $j \leq n$  do
5:      $j \leftarrow j^2$ 
6:     count  $\leftarrow$  count + 1
7:   end while
8: end for
9: return count
```

$\Theta(n \log \log n)!$



让我们在考察一下渐近符号，我们说  $O(n)$  不关注系数，那么系数真的不重要么？

---

- 事实上，在实际运行中，系数也是十分重要的，毕竟从  $2n$  到  $n$ ，那也是翻了一倍的提升。为此，有很多 fine-grained complexity 的研究。
- 另一方面，存在一些系数超大的算法，在实际中并没有很好的体现，比如用来解线性规划的椭球法，其理论复杂度优于单纯形法，但在实际中并没有很好的应用。
- 但是，对复杂性的探究有助于我们认知到问题的困难性，因此复杂性结论依旧有着不可撼动的重要意义。



## 本节内容

- 算法的时间估计, 输入规模
- 最好运行时间, 平均运行时间和最坏运行时间
- 渐进符号