第一章　算法的基本概念

1.1　引言

算法设计与分析在计算机科学与技术中的地位
算法（Algorithm）一词的由来。

1.1.1　算法的定义和特征

欧几里德算法：

算法 1.1　欧几里德算法
输入：正整数 m,n
输出：m,n 的最大公因子

```
1. int euclid(int m,int n)
2. {
3.     int    r;
4.     do {
5.         r = m % n;
6.         m = n;
7.         n = r;
8.     } while(r)
9.     return m;
10. }
```

一、算法的定义：

定义 1.1　算法是解某一特定问题的一组有穷规则的集合。

二、算法的特征：

1．有限性。算法在执行有限步之后必须终止。

2．确定性。算法的每一个步骤，都有精确的定义。要执行的每一个动作都是清晰的、无歧义的。

3．输入。一个算法有 0 个或多个输入，它是由外部提供的，作为算法开始执行前的初始值，或初始状态。算法的输入是从特定的对象集合中抽取的。

4．输出。一个算法有一个或多个输出，这些输出，和输入有特定的关系，实际上是输入的某种函数。不同取值的输入，产生不同结果的输出。

5．能行性。算法的能行性指的是算法中有待实现的运算，都是基本的运算。原则上可以由人们用纸和笔，在有限的时间里精确地完成。

1.1.2　算法设计的例子，穷举法

一、穷举法，是从有限集合中，逐一列举集合的所有元素，对每一个元素逐一判断和处理，从而找出问题的解。

二、例

例 1.1　百鸡问题。

“鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、母、雏各几何？”

a：公鸡只数，b：母鸡只数，c：小鸡只数。约束方程：

$$a+b+c=100 \qquad (1.1.1)$$

$$5a+3b+c/3=100 \qquad (1.1.2)$$

$$c\%3=0 \qquad (1.1.3)$$

1。第一种解法：

a、b、c的可能取值范围：0 ~ 100，对在此范围内的，a、b、c的所有组合进行测试，凡是满足上述三个约束方程的组合，都是问题的解。

把问题转化为用n元钱买n只鸡，n为任意正整数，则方程（1.1.1）、（1.1.2）变成：

$$a+b+c=n \qquad (1.1.4)$$

$$5a+3b+c/3=n \qquad (1.1.5)$$

算法 1.2　百鸡问题

输入：所购买的三种鸡的总数目 n

输出：满足问题的解的数目 k,公鸡,母鸡,小鸡的只数 g[],m[],s[]

```
 1. void chicken_question(int n,int &k,int g[],int m[],int s[])
 2. {
 3.    int  a,b,c;
 4.    k = 0;
 5.    for (a=0;a<=n;a++)
 6.       for (b=0;b<=n;b++)
 7.          for (c=0;c<=n;c++) {
 8.             if ((a+b+c==n)&&(5*a+3*b+c/3==n)&&(c%3==0)) {
 9.                g[k] = a;
10.                m[k] = b;
11.                s[k] = c;
12.                k++;
13.             }
```

```
14.            }
15.        }
16.    }
17. }
```

执行时间：外循环：$n+1$次，

中间循环：$(n+1)\times(n+1)$次，

内循环：$(n+1)\times(n+1)\times(n+1)$次。

当$n=100$时，内循环的循环体执行次数大于 100 万次。

2。第二种解法：

公鸡只数：$0\sim n/5$

母鸡只数：$0\sim n/3$

母鸡只数：$c=n-a-b$。

算法 1.3 改进的百鸡问题

输入：所购买的三种鸡的总数目 n

输出：满足问题的解的数目 k,公鸡,母鸡,小鸡的只数 g[],m[],s[]

```
 1. void chicken_problem(int n,int &k,int g[],int m[],int s[])
 2. {
 3.    int    i,j,a,b,c;
 4.    k = 0;
 5.    i = n / 5;
 6.    j = n / 3;
 7.    for (a=0;a<=i;a++)
 8.       for (b=0;b<=j;b++) {
 9.          c = n – a – b;
10.          if ((5*a+3*b+c/3==n)&&(c%3==0)) {
11.             g[k] = a;
12.             m[k] = b;
13.             s[k] = c;
14.             k++;
15.          }
16.       }
17.    }
18. }
```

执行时间：外循环：$n/5+1$

内循环：$(n/5+1)\times(n/3+1)$

当 $n=100$ 时，内循环的循环体的执行次数为 $21\times 34=714$ 次。

对某类特定问题，在规模较小的情况下，穷举法往往是一个简单有效的方法。

例 1.2 货郎担问题。

n 个城市，分别用 1 到 n 的数字编号，问题归结为在有向赋权图 $G=<V,E>$ 中，寻找一条路径最短的哈密尔顿回路。其中，$V=\{1,2,\cdots\cdots n\}$，表示城市顶点，边 $(i,j)\in E$ 表示城市 i 到城市 j 的距离，$i,j=1,2,\cdots\cdots n$。

图的邻接矩阵 C：表示各个城市之间的距离，称为费用矩阵。

数组 T：表示售货员的路线，依次存放旅行路线中的城市编号。

售货员的每一条路线，对应于城市编号 $1,2\cdots\cdots n$ 的一个排列。n 个城市共有 $n!$ 个排列，采用穷举法逐一计算每一条路线的费用，从中找出费用最小的路线，便可求出问题的解。

算法 1.4 穷举法版本的货郎担问题

输入：城市个数 n，费用矩阵 c[][]

输出：旅行路线 t，最小费用 min

```
1. void salesman_problem(int n,float &min,int t[],float c[][])
2. {
3.     int p[n],i = 1;
4.     float  cost;
5.     min = MAX_FLOAT_NUM;
6.     while (i <= n!) (
7.         产生 n 个城市的第 i 个排列于 p;
8.         cost =  路线 p 的费用;
9.         if (cost < min) {
10.            把数组 p 的内容拷贝到数组 t;
11.            min = cost;
12.        }
13.        i++;
14.    }
15. }
```

执行时间： while 循环执行 $n!$ 次。

表 1.1 算法 1.4 的执行时间随 n 的增长而增长的情况

n	$n!$	n	$n!$	n	$n!$	n	$n!$
5	120μs	9	362ms	13	1.72h	17	11.27year
6	720μs	10	3.62s	14	24h	18	203year

7	5.04ms	11	39.9s	15	15day	19	3857year
8	40.3ms	12	479.0s	16	242day	20	77146year

1.1.3 算法的复杂性分析

问题：效率和方法。

问题一：如何设计算法，算法的设计方法。

问题二：如何分析算法，算法的复杂性分析。

用算法的复杂性来衡量算法的效率。算法的时间复杂性和算法的空间复杂性。算法的时间复杂性越高，算法的执行时间越长；反之，执行时间越短。算法的空间复杂性越高，算法所需的存储空间越多；反之越少。

1.2 算法的时间复杂性

一、算法复杂性的度量？

二、如何分析和计算算法的复杂性？

1.2.1 算法的输入规模和运行时间的阶

一、算法的输入规模和运行时间

令百鸡问题的第一、二两个算法，其最内部的循环体每执行一次，需 1μs 时间。

	$n=100$ 的内循环次数	时间	$n=10000$ 的内循环次数	时间
第一个算法	100 万次	1s	10000^3	11 天零 13 小时
第二个算法	714 次	714μs	$(10000/5+1)\times(10000/3+1)$	6.7 秒

$n=2^{20}$　选择排序需 6.4 天　合并排序需 20 秒

算法的执行时间随问题规模的增大而增长的情况。

二、算法运行时间的评估

不能准确地计算算法的具体执行时间

不需对算法的执行时间作出准确地统计（除非在实时系统中）

1、计算模型：RAM 模型（随机存取机模型）、图灵机模型等

2、初等操作：所有操作数都具有相同的固定字长；所有操作的时间花费都是一个常数时间间隔。算术运算；比较和逻辑运算；赋值运算，等等；

例：输入规模为 n，百鸡问题的第一个算法的时间花费，可估计如下：

$$T_1(n)\le 1+2(n+1)+n+1+2(n+1)^2+(n+1)^2+16(n+1)^3+4(n+1)^3$$
$$=20n^3+63n^2+69n+27 \qquad (1.1.6)$$

可把 $T_1(n)$ 写成：

$$T_1^*(n)\approx c_1n^3 \qquad c_1>0 \qquad (1.1.7)$$

这时，称$T_1^*(n)$的阶是n^3。

百鸡问题的第一个算法的时间花费：

$$T_2(n)\le 1+2+2+1+2\times(n/5+1)+n/5+1+2\times(n/5+1)\times(n/3+1)+$$
$$(3+10+4)\times(n/5+1)\times(n/3+1)$$
$$=\frac{19}{15}n^2+\frac{161}{15}n+28 \qquad (1.1.8)$$

同样，随着n的增大，$T_2(n)$也可写成：

$$T_2^*(n)\approx c_2n^2 \qquad c_2>0 \qquad (1.1.9)$$

这时，称$T_2^*(n)$的阶是n^2。把$T_1^*(n)$和$T_2^*(n)$进行比较，有：

$$T_1^*(n)/T_2^*(n)=\frac{c_1}{c_2}n \qquad (1.1.10)$$

当n很大时，c_1/c_2的作用很小。

3、算法时间复杂性的定义：

定义 1.2 设算法的执行时间$T(n)$，如果存在$T^*(n)$，使得：

$$\lim_{n\to\infty}\frac{T(n)-T^*(n)}{T(n)}=0 \qquad (1.1.11)$$

就称$T^*(n)$为算法的渐近时间复杂性。

表 1.2 表示时间复杂性的阶为$\log n$, n, $n\log n$, n^2, n^3, 2^n，当$n=2^3, 2^4, \cdots\cdots, 2^{16}$ 时，算法的渐近运行时间。这里假定每一个操作是$1ns$。

表 1.3 表示对不同时间复杂性的算法，计算机速度提高后，可处理的规模n_2和n_1的关系。

表 1.2　不同时间复杂性下不同输入规模的运行时间

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
8	3 n	8 n	24 n	64 n	512 n	256 n
16	4 n	16 n	64 n	256 n	4.096 μ	65.536 μ
32	5 n	32 n	160 n	1.024 μ	32.768 μ	4294.967 ms
64	6 n	64 n	384 n	4.096 μ	262.144 μ	5.85 c
128	7 n	128 n	896 n	16.384 μ	1997.152 μ	10^{20} c
256	8 n	256 n	2.048 μ	65.536 μ	16.777 ms	10^{58} c
512	9 n	512 n	4.608 μ	262.144 μ	134.218 ms	10^{135} c
1024	10 n	1.024 μ	10.24 μ	1048.576 μ	1073.742 ms	10^{289} c
2048	11 n	2.048 μ	22.528 μ	4194.304 μ	8589.935 ms	10^{598} c
4096	12 n	4.096 μ	49.152 μ	16.777 ms	68.719 s	10^{1214} c
8192	13 n	8.196 μ	106.548 μ	67.174 ms	549.752 s	10^{2447} c

16384	14 n	16.384 μ	229.376 μ	268.435 ms	1.222 h	10^{4913} c
32768	15 n	32.768 μ	491.52 μ	1073.742 ms	9.773 h	10^{9845} c
65536	16 n	65.536 μ	1048.576 μ	4294.967 ms	78.187 h	10^{19709} c

n：纳秒　μ：微秒　ms：毫秒　s：秒　h：小时　d：天　y：年　c：世纪

表 1.3　计算机速度提高 10 倍后，不同算法复杂性求解规模的扩大情况

算法	A_1	A_2	A_3	A_4	A_5	A_6
时间复杂性	n	$n\log n$	n^2	n^3	2^n	$n!$
n_2 和 n_1 的关系	$10\,n_1$	$8.38\,n_1$	$3.16\,n_1$	$2.15\,n_1$	$n_1+3.3$	n_1

4、多项式时间算法和指数时间算法。

1.2.2　运行时间的上界，O 记号

一、O 记号的定义：

定义 1.3　令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N\to R_+$，函数 $g: N\to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n\ge n_0$，都有 $f(n)\le cg(n)$，就称函数 $f(n)$ 的阶至多是 $O(g(n))$。

因此，如果存在 $\lim\limits_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}\neq\infty$$

即意味着：

$$f(n)=O(g(n))$$

含义：$f(n)$ 的增长最多象 $g(n)$ 的增长那样快。称 $O(g(n))$ 是 $f(n)$ 的上界。

二、例：百鸡问题的第二个算法，由式（1.1.8）有：

$$T_2(n)\le\frac{19}{15}n^2+\frac{161}{15}n+28$$

取 n_0=28，对 $\forall n\ge n_0$，有：

$$\begin{aligned}T_2(n)&\le\frac{19}{15}n^2+\frac{161}{15}n+n\\&=\frac{19}{15}n^2+\frac{176}{15}n\\&\le\frac{19}{15}n^2+\frac{176}{15}n^2\\&=13n^2\end{aligned}$$

令 $c=13$，并令 $g(n)=n^2$，有：

$$T_2(n) \le cn^2 = cg(n)$$

所以，$T_2(n) = O(g(n)) = O(n^2)$。

这时，如果有一个新算法，其运行时间的上界低于以往解同一问题的所有其它算法的上界，就认为建立了一个解该问题所需时间的新上界。

1.2.3 运行时间的下界，Ω记号

一、Ω记号的定义：

定义 1.4 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N \to R_+$，函数 $g: N \to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n \ge n_0$，都有 $f(n) \ge cg(n)$， 就称函数 $f(n)$ 的阶至少是 $\Omega(g(n))$。

因此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \ne 0$$

即意味着：

$$f(n) = \Omega(g(n))$$

含义： $f(n)$ 的增长至少象 $g(n)$ 那样快。表示解一个特定问题的任何算法的时间下界。

二、例：百鸡问题的第二个算法第 11、12、13、14 行，仅在条件成立时才执行，其执行次数未知。假定条件都不成立，这些语句一次也没有执行，该算法的执行时间至少为：

$$\begin{aligned} T_2(n) &\ge 1+2+2+1+2\times(n/5+1)+n/5+1+2\times(n/5+1)\times(n/3+1)+ \\ &\quad (3+10)\times(n/5+1)\times(n/3+1) \\ &= n^2 + \frac{43}{5}n + 24 \\ &\ge n^2 \end{aligned}$$

当取 $n_0 = 1$ 时， $\forall n \ge n_0$，存在常数 $c = 1$， $g(n) = n^2$，使得：

$$T_2(n) \ge n^2 = cg(n)$$

三、结论 1.1 $f(n)$ 的阶是 $\Omega(g(n))$，当且仅当 $g(n)$ 的阶是 $O(f(n))$。

1.2.4 运行时间的准确界，Θ记号

百鸡问题的第二个算法，运行时间的上界是 $13n^2$，下界是 n^2，这表明不管输入规模如何变化，该算法的运行时间都界于 n^2 和 $13n^2$ 之间。这时，用记号 Θ 来表示这种情况，认为这个算法的运行时间是 $\Theta(n^2)$。Θ 记号表明算法的运行时间有一个较准确的界。

一、Θ记号的定义如下：

定义 1.5 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N \to R_+$，函数 $g: N \to R_+$，若存在自然数 n_0 和两个正常数 $0 \le c_1 \le c_2$，使得对所有的 $n \ge n_0$， 都有：

$$c_1 g(n) \le f(n) \le c_2 g(n)$$

就称函数 $f(n)$ 的阶是 $\Theta(g(n))$。

因此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c$$

即意味着：

$$f(n)=\Theta(g(n))$$

其中，c 是大于 0 的常数。

二、例：

例 1.3 常函数 $f(n)=4096$。

令 $n_0=0$，$c=4096$，使得对 $g(n)=1$，对所有的 n 有：

$f(n)\le 4096\times 1=cg(n)$ ∴ $f(n)=O(g(n))=O(1)$

$f(n)\ge 4096\times 1=cg(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(1)$。

$cg(n)\le f(n)\le cg(n)$ ∴ $f(n)=\Theta(1)$。

例 1.4 线性函数 $f(n)=5n+2$。

令 $n_0=0$，当 $n\ge n_0$ 时，有 $c_1=5$，$g(n)=n$，使得：

$f(n)\ge 5n=c_1g(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(n)$。

令 $n_0=2$，当 $n\ge n_0$ 时，有：$c_2=6$，$g(n)=n$

$$\begin{aligned} f(n) &\le 5n+n \\ &= 6n \\ &= c_2g(n) \end{aligned}$$ ∴ $f(n)=O(g(n))=O(n)$。

同时，有：

$c_1g(n)\le f(n)\le c_2g(n)$ ∴ $f(n)=\Theta(n)$。

例 1.5 平方函数 $f(n)=8n^2+3n+2$。

令 $n_0=0$，当 $n\ge n_0$ 时，有 $c_1=8$，$g(n)=n^2$，使得：

$f(n)\ge 8n^2=c_1g(n)$ ∴ $f(n)=\Omega(g(n))=\Omega(n^2)$。

令 $n_0=2$，当 $n\ge n_0$ 时，有：$c_2=12$，$g(n)=n^2$

$$\begin{aligned} f(n) &\le 8n^2+3n+n \\ &\le 12n^2 \\ &= c_2g(n) \end{aligned}$$ ∴ $f(n)=O(g(n))=O(n^2)$。

同时，有：

$c_1g(n)\le f(n)\le c_2g(n)$ ∴ $f(n)=\Theta(n^2)$。

结论 1.2 令：

$$f(n)=a_kn^k+a_{k-1}n^{k-1}+\cdots\cdots+a_1n+a_0 \qquad a_k>0$$

则有：$f(n)=O(n^k)$，且 $f(n)=\Omega(n^k)$，因此，有：$f(n)=\Theta(n^k)$。

例 1.6 指数函数 $f(n)=5\times 2^n+n^2$。

令 $n_0=0$，当 $n\geq n_0$ 时，有 $c_1=5$，$g(n)=2^n$，使

$$f(n)\geq 5\times 2^n=c_1 g(n) \qquad \therefore\ f(n)=\Omega(g(n))=\Omega(2^n)。$$

令 $n_0=4$，当 $n\geq n_0$ 时，有：$c_2=6$，$g(n)=2^n$

$$\begin{aligned} f(n) &\leq 5\times 2^n+2^n \\ &\leq 6\times 2^n \\ &= c_2 g(n) \end{aligned} \qquad \therefore\ f(n)=O(g(n))=O(2^n)。$$

同时，有：

$$c_1 g(n)\leq f(n)\leq c_2 g(n) \qquad \therefore\ f(n)=\Theta(2^n)。$$

例 1.7 对数函数 $f(n)=\log n^2$。

因为：$\log n^2=2\log n$

令 $n_0=1$，$c_1=1$，$c_2=3$，$g(n)=\log n$，有：

$$c_1 g(n)\leq 2\log n\leq c_2 g(n) \qquad \therefore\ \log n^2=\Theta(\log n)。$$

结论 1.2' 对任何正常数 k，都有：$\log n^k=\Theta(\log n)$

例 1.8 函数 $f(n)=\sum_{j=1}^{n}\log j$

$$\begin{aligned} \sum_{j=1}^{n}\log j &\leq \sum_{j=1}^{n}\log n \\ &= n\log n \end{aligned}$$

令 $n_0=1$，$c_1=1$，$g(n)=n\log n$，有：

$$\sum_{j=1}^{n}\log j\leq c_1 g(n) \qquad \therefore\ \sum_{j=1}^{n}\log j=O(g(n))=O(n\log n)$$

另一方面，假定 n 是偶数，

$$\begin{aligned} \sum_{j=1}^{n}\log j &\geq \sum_{j=1}^{n/2}\log\frac{n}{2} \\ &= \frac{n}{2}\log\frac{n}{2} \\ &= \frac{n}{2}(\log n-1) \\ &= \frac{n}{4}(\log n+\log n-2) \end{aligned}$$

因此，令 $n_0=4$， $c_2=1/4$， $g(n)=n\log n$，对所有的 $n\ge n_0$，都有：

$$\sum_{j=1}^{n}\log j \ge \frac{1}{4}n\log n$$
$$=c_2 g(n)$$
$$=\Omega(g(n))$$
$$=\Omega(n\log n)$$

因此，有：

$$c_2 g(n)\le \sum_{j=1}^{n}\log j \le c_1 g(n)$$

所以，

$$\sum_{j=1}^{n}\log j=\Theta(g(n))=\Theta(n\log n)$$

结论 1.2’’ $\log n!=\Theta(n\log n)$。

1.2.5 复杂性类型和 o 记号

一、复杂性的分类

定义 1.6 令 R 是函数集合 F 上的一个关系， $R\subseteq F\times F$，有

$$R=\{<f,g>\mid f\in F\wedge g\in F\wedge f(n)=\Theta(g(n))\}$$

则 R 是自反、对称、传递的等价关系，它诱导的等价类，称阶是 $g(n)$ 的复杂性类型的等价类。

所有常函数的复杂性类型都是 $\Theta(1)$；

所有线性函数的复杂性类型都是 $\Theta(n)$；

所有的 2 阶多项式函数的复杂性类型都是 $\Theta(n^2)$，如此等等。

例： $f(n)=4096$， $g(n)=3n+2$， $\exists\ n_0=1$， $c=1365$， $\forall\ n\ge n_0$，有 $f(n)\le cg(n)$。 $\therefore f(n)=O(g(n))=O(n)$。

又：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{4096}{3n+2}=0$$ $\therefore f(n)\ne\Omega(g(n))$，则： $f(n)\ne\Theta(g(n))$。

$\therefore$ $f(n)=4096$ 和 $g(n)=3n+2$ 是属于不同复杂性类型的函数。

例： $\log 2^n=n$ $\log n!=\Theta(n\log n)$ $\log n!\le cg(n\log n)$

由 $n<n\log n$ $\exists\ n_0\ge 0$， $c\ge 0$ ，$\forall\ n\ge n_0$

有 $2^n\le cn!$ $\therefore\ 2^n=O(n!)$

$$\lim_{n\to\infty}\frac{2^n}{n!}=\lim_{n\to\infty}\frac{2\cdot 2\cdots 2}{1\cdot 2\cdots n}=0 \qquad \therefore\ 2^n\neq\Omega(n!)$$

2^n 和 $n!$ 是属于不同复杂性类型的函数。

例： $\log n!=\Theta(n\log n)$ $\log 2^{n^2}=n^2>n\log n$

$n!=O(2^{n^2})$ $2^{n^2}\neq O(n!)$。

这两个函数也是属于不同复杂性类型的函数。

二、o 记号的定义

定义 1.7 令 N 为自然数集合，R_+ 为正实数集合。函数 $f: N\to R_+$，函数 $g: N\to R_+$，若存在自然数 n_0 和正常数 c，使得对所有的 $n\geq n_0$，都有

$$f(n)<cg(n)$$

就称函数 $f(n)$ 是 $o(g(n))$。

由此，如果存在 $\lim_{n\to\infty} f(n)/g(n)$，则

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0 \qquad \text{即意味着} \qquad f(n)=o(g(n))$$

结论 1.3 $f(n)=o(g(n))$ 当且仅当 $f(n)=O(g(n))$ 而 $g(n)\neq O(f(n))$。

复杂性类型体系：用偏序关系 $f(n)\prec g(n)$ 表示 $f(n)=o(g(n))$。

$$1\prec\log\log n\prec\log n\prec\sqrt{n}\prec n^{3/4}\prec n\prec n\log n\prec n^2\prec 2^n\prec n!\prec 2^{n^2}$$

1.3 算法的时间复杂性分析

1.3.1 循环次数的统计

一、循环次数表示乘以一个常数因子的运行时间

例 1.9 计算多项式：

$$P(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots\cdots+a_1x+a_0$$

Horner 法则改写： $P(x)=(\cdots(a_nx+a_{n-1})x+\cdots\cdots+a_1)x+a_0$

算法 1.5 计算多项式

输入：存放多项式系数的数组 A[],实数 x,多项式的阶 n

输出：多项式的值

```
1. float polynomial(float A[],float x,int n)
2. {
3.     int  i;
4.     float value;
5.     for (i=n;i>0;i--) {
```

```
6.        value = A[i] * x + A[i-1];
7.    return value;
8. }
```

c_1：循环控制变量 i 赋初值所花费的单位时间

c_2：变量 i 的测试及递减、以及值 *value* 的计算所花费的单位时间

算法的执行时间 $T(n)$ 为：　　$T(n)=c_1+c_2n$

$$=\Theta(n)$$

例 1.10　把数组中 n 个元素由小到大进行排序。

算法 1.6　冒泡算法

输入：数组 A[],元素个数 n

输出：按递增顺序排序的数组 A[]

```
1. template <class T>
2. void bubble(Type A[],int n)
3. {
4.    int i,k;
5.    for (k=n-1;k>0;k--) {
6.       for (i=0;i<k;i++) {
7.           if (A[i] > A[i+1]) {
8.               swap(A[i],A[i+1]);
9.           }
10.       }
11.   }
12. }
13. void swap(Type &x,Type &y)
14. {
15.   Type temp;
16.   temp = x;
17.   x = y;
18.   y = temp;
19. }
```

$\bar{c}$：辅助操作的执行时间

c：循环体的平均执行时间

算法总的执行时间 $T(n)$ 为：

$$T(n)=((n-1)+(n-2)+\cdots\cdots+1)\ c+\bar{c}$$

$$= \frac{c}{2} n(n-1) + \bar{c}$$

$$= \Theta(n^2)$$

例 1.11 选手的竞技淘汰比赛。

有 $n = 2^k$ 位选手进行竞技淘汰比赛，最后决出冠军的选手。假定用如下的函数：

```
BOOL comp(Type mem1,Type mem2)
```

模拟两位选手的比赛，若 *mem*1 胜则返回 *TRUE* ，否则返回 *FALSE* 。

并假定可以在常数时间 *c* 内完成函数 comp 的执行。

算法 1.7 竞技淘汰比赛

输入：选手成员 group[],选手个数 n

输出：冠军的选手

```
 1. Type game(Type group[],int n)
 2. {
 3.     int j,i = n;
 4.     while (i>1) {
 5.         i = i / 2;
 6.         for (j=0;j<i;j++)
 7.             if (comp(group[j+i],group[j]);
 8.                 group[j] = group[j+i];
 9.     }
10.     return group[0];
11. }
```

因为 $n = 2^k$ ，第 4 行的 while 循环的循环体共执行 *k* 次。

在每一次执行时，第 6 行的 for 循环的循环体，其执行次数分别为 $n/2, n/4, \cdots\cdots, 1$，

函数 comp 可以在常数时间内完成。

算法的执行时间 $T(n)$ 为：

$$T(n) = \frac{n}{2} + \frac{n}{4} + \cdots\cdots + \frac{n}{n}$$

$$= n\left(\frac{1}{2} + \frac{1}{4} + \cdots\cdots + \frac{1}{2^k}\right)$$

$$= n\left(1 - \frac{1}{2^k}\right)$$

$$= n - 1$$

$$= \Theta(n)$$

例 1.12 对 *n* 张牌进行 *n* 次洗牌，洗牌规则如下：在第 *k* 次洗牌时（ $k = 1 \cdots n$ ），对第 *i*

张牌（$i=1\cdots n/k$）随机地产生一个小于 n 的正整数 d，互换第 i 张牌和第 d 张牌的位置。

算法 1.8 洗牌

输入：牌 A[],牌的张数 n

输出：洗牌后的牌 A[]

```
 1. template <class Type>
 2. void shuffle(Type A[],int n)
 3. {
 4.    int  i,k,m,d;
 5.    random_seed(0);
 6.    for (k=1;k<=n;k++) {
 7.       m = n / k ;
 8.       for (i=1;i<=m;i++) {
 9.          d = random(1,n);
10.          swap(A[i],A[d]);
11.       }
12.    }
13. }
```

函数 random_seed：为随机数发生器产生随机数种子，需常数时间

函数 random：产生一个 1 到 n 之间的随机数，需常数时间

第 6 行开始的 for 循环的循环体共执行 n 次。

第 8 行开始的内部 for 循环的循环体，其执行次数依次为：

$$n,\lfloor n/2\rfloor,\lfloor n/3\rfloor,\cdots\cdots\lfloor n/n\rfloor$$

算法的执行时间 $T(n)$ 为内部 for 循环的循环体的执行次数乘以一个常数时间，因此，有：

$$T(n)=\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$$

因为：

$$\sum_{i=1}^{n}(\frac{n}{i}-1)\ \le\ \sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor\le\sum_{i=1}^{n}\frac{n}{i}$$

由调和级数的性质，有：

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1$$

因此：

$$\frac{\log(n+1)}{\log e} \le \sum_{i=1}^{n} \frac{1}{i} \le \frac{\log n}{\log e} + 1$$

所以：

$$\frac{1}{\log e} n\log(n+1) - n \le \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \le \frac{1}{\log e} n\log n + n$$

由此得出：

$$T(n) = \Theta(n\log n)$$

1.3.2 基本操作频率的统计

一、基本操作的定义

定义 1.8 算法中的某个初等操作，如果它的最高执行频率，和所有其它初等操作的最高执行频率，相差在一个常数因子之内，就说这个初等操作是一个基本操作。

初等操作的执行频率，可正比于任何其它操作的最高执行频率

基本操作的选择，必须反映出该操作随着输入规模的增加而变化的情况

二、用基本操作的执行频率估计算法的时间复杂性

例 1.13 合并两个有序的子数组

假定 A 是一个具有 m 个元素的整数数组，给定三个下标：p,q,r，$0 \le p \le q \le r < m$，使得 $A[p] \sim A[q]$，$A[q+1] \sim A[r]$ 分别是两个以递增顺序排序的子数组。把这两个子数组按递增顺序合并在 $A[p] \sim A[r]$ 中。

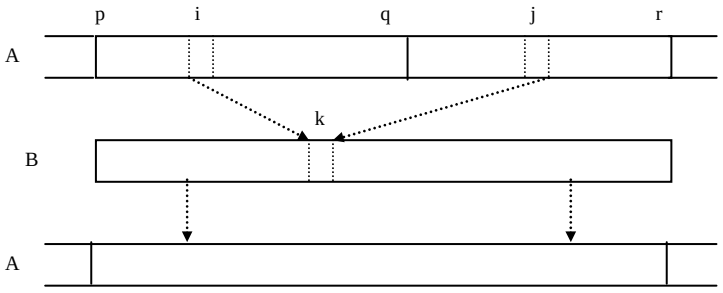
算法 1.9 合并两个有序的子数组

输入：整数数组 A[],下标 p,q,r,元素个数 m。A[p]~A[q]及 A[q+1]~A[r]已按递增顺序排序

输出：按递增顺序排序的子数组 A[p]~A[r]

```
 1. void merge(int A[],int p,int q,int r,int m)
 2. {
 3.     int  *bp = new int[m];          /* 分配缓冲区,存放被排序的元素 */
 4.     int  i,j,k;
 5.     i = p;   j = q + 1;   k = 0;
 6.     while (i<=q && j<=r) {          /* 逐一判断两子数组的元素 */
 7.         if (A[i]<=A[j])             /* 按两种情况,把小的元素拷贝到缓冲区*/
 8.             bp[k++] = A[i++];
 9.         else
10.             bp[k++] = A[j++];
11.     }
12.     if (i==q+1)                     /* 按两种情况,处理其余元素 */
```

```
13.        for (;j<=r;j++)
14.            bp[++] = A[j++];;       /* 把A[j]~A[r]拷贝到缓冲区 */
15.    else
16.        for (;i<=q;i++)
17.            bp[++] = A[i++];        /* 把A[i]~A[q]拷贝到缓冲区 */
18.    k = 0;
19.    for (i=p;i<=r;i++)    /* 最后,把数组bp的内容拷贝导A[p]~A[r] */
20.        A[i++] = bp[k++];
21.    delete bp;
22. }
```



while 循环的循环次数、for 循环的循环次数未知

基本操作的选择：

1、数组元素的赋值操作作为基本操作，操作频率：$2n$，

1）、随输入规模的增大而增加

2）、执行频率与其它操作的执行频率相差一个常数因子

算法的时间复杂性为$\Theta(n)$。

2、数组元素的比较操作作为基本操作

令两个子数组的大小分别为n_1和n_2，其中，$n_1+n_2=n$。

合并两个数组时，数组元素的比较次数，最少为n_1，最多为$n-1$次。

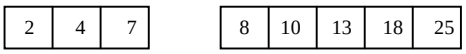


图1.1　合并两个有序数组时, 元素比较次数最少的情况

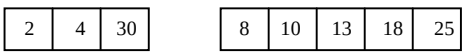


图1.2　合并两个有序数组时, 元素比较次数最多的情况

如果合并两个大小接近相同的有序数组，例如$n_1=\lfloor n/2 \rfloor$，$n_2=\lceil n/2 \rceil$，

数组元素的比较操作，操作频率：$n/2$

满足上述 1）2）。算法的时间复杂性仍然是$\Theta(n)$。

例 1.14　菜园四周种了n棵白菜，并按顺时针方向由 1 到n编号。收割时，从编号 1 开

始，按顺时针方向每隔两棵白菜收割一棵，直到全部收割完毕为止。按收割顺序列出白菜的编号。

数组 A：存放白菜的编号，初值为$1,\cdots,n$。当白菜被收割后，从数组中删去相应元素

数组 B：按收割顺序存放被收割白菜的编号

算法 1.10 收割白菜

输入：白菜棵数 n

输出：按收割顺序存放白菜编号的数组 B[]

```
 1. void reap(int B[],int n)
 2. {
 3.    int i,j,k,s,t;
 4.    int *A = new int[n];
 5.    j = 0;   k = 3;   s = n;
 6.    for (i=0;i<n;i++)
 7.       A[i] = i + 1;
 8.    while (j<n) {
 9.       t = s;   s = 0;
10.       for (i=0;i<t;i++) {
11.          if (--k!=0)
12.             A[s++] = A[i];              /* 未被收割的白菜 */
13.          else {
14.             B[j++] = A[i];   k = 3;     /* 被收割的白菜 */
15.          }
16.       }
17.    }
18.    delete A;
19. }
```

while 循环 for 循环的循环次数未知

基本操作的选择：14 行的赋值操作需要执行 n 次，12 行的赋值操作，需要执行 $2n$ 次

算法的运行时间为 $\Theta(n)$。

1.3.3 计算步的统计

一、计算步

定义 1.9 计算步是一个语法或语义意义上的程序段，该程序段的执行时间与输入实例无关。

例：

```
flag=(a+b+c==n)&&(5*a+3*b+c/3==n)&&(c%3==0);
a=b;
```

和输入规模无关。连续 200 个乘法操作可为一个计算步，*n* 次加法不能作为一个计算步计算步所表示的计算量，可能有很大的差别。

二、计算步的统计

把全局变量 *count* 嵌入实现算法的程序中，每执行一个计算步，*count* 就加 1。算法运行结束时，*count* 的值，就是算法所需执行的计算步数。

随着输入实例的不同，按这种方式统计出来的计算步数也不同。它有助于了解算法的执行时间随输入实例的变化而变化的情况。如果输入实例的规模增大 10 倍，所需执行的计算步数也增加 10 倍，就可以认为运行时间随着 *n* 的增大而线性增加。

1.3.4 最坏情况和平均情况

一、影响运行时间的因素

问题规模的大小、输入的具体数据（除算法的性能外）

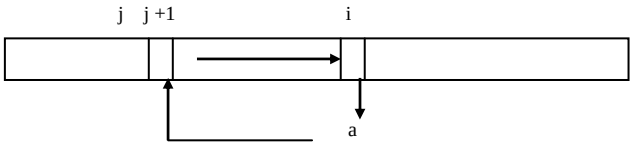
例 1.15 用插入法对 *n* 个元素的数组 A，按递增顺序进行排序。

算法 1.11 用插入法按递增顺序排序数组 A

输入：n 个元素的整数数组 A[],数组元素个数 n

输出：按递增顺序排序的数组 A[]

```
 1. void insert_sort(int A[],int n)
 2. {
 3.    int  a,i,j;
 4.    for (i=1;i<n;i++) {
 5.       a = A[i];
 6.       j = i – 1;
 7.       while (j>=0 && A[j]>a) {
 8.          A[j+1] = A[j];
 9.          j--;
10.       }
11.       A[j+1] = a;
12.    }
13. }
```



1. 初始数组已按递增顺序排列，执行时间既是$O(n)$的，也是$\Omega(n)$，所以，是$\Theta(n)$的。

2. 初始数组按递减顺序排列，每一个元素$A[i],1\le i\le n-1$，都和它前面的i个元素进行比较，则整个算法执行的元素比较次数为：

$$\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$$

在这种情况下，算法的执行时间是$O(n^2)$的，也是$\Omega(n^2)$的，所以是$\Theta(n^2)$的。

二、算法时间复杂性的三种分析

最坏情况的分析、平均情况的分析、和最好情况的分析。

1.3.5 最坏情况分析

下界和上界不一致的情况：

例 1.16 对已经排序过的、具有n个元素的数组A，检索是否存在元素x。当n是奇数时，用二叉检索算法检索；当n是偶数时，用线性检索算法检索。

算法 1.12 分别采用线性检索算法和二叉检索算法进行检索的算法

输入：给定 n 个已排序的元素的数组 A[],及元素 x

输出：若 x = A[j],1≤j≤n,输出 j,否则输出 0

```
1. int linear_search(int A[],int n,int x);
2. int binary_search(int A[],int n,int x);
3. int serach(int A[],int n,int x)
4. {
5.    if ((n%2)==0)
6.        return linear_search(A,n,x);
7.    else
8.        return binary_search(A,n,x);
10. }
```

这个算法在n是偶数时，调用线性检索算法进行检索；在n是奇数时，调用二叉检索算法进行检索。线性检索算法如下：

算法 1.13 线性检索算法

输入：给定 n 个已排序过的元素的数组 A[],及元素 x

输出：若 x = A[j],0≤j≤n-1,输出 j,否则输出-1

```
1.  int linear_search(int A[],int n,int x);
2.  {
3.     int j = 0;
4.     while (j<n && x!=A[j])
```

```
5.          j++;
6.      if ((j<n)&&(x==A[j]))
7.          return j;
8.      else
9.          return -1;
10.  }
```

二叉检索算法如下：

算法 1.14 二叉检索算法

输入：给定 n 个已排序过的元素的数组 A[],及元素 x

输出：若 x = A[j],0≤j≤n-1,输出 j,否则输出-1

```
1.  int binary_search(int A[],int n,int x)
2.  {
3.      int mid,low = 0,high = n – 1,j = -1;
4.      while (low<=high && j<0) {
5.          mid = (low + high) / 2;
6.          if (x==A[mid]) j = mid;
7.          else if (x<A[mid]) high = mid –1;
8.          else low = mid + 1;
9.      }
10.     return j;
11.  }
```

线性检索算法的最坏情况：数组中不存在元素 x，或元素 x 是数组的最后一个元素

时间复杂性：$O(n)$、$\Omega(n)$、$\Theta(n)$

二叉检索算法的最坏情况：数组中不存在元素 x，或元素 x 是数组的第一个元素、或最后一个元素

时间复杂性是 $O(\log n)$、$\Omega(\log n)$、$\Theta(\log n)$

search 在最坏情况下的时间复杂性。$O(n)$ 的，也是 $\Omega(\log n)$。

1.3.6 平均情况分析

在平均情况下，算法的运行时间取算法在所有可能输入的平均运行时间。

预先知道输入的出现概率，即所有输入的分布情况。

例 1.17 插入排序算法 insert_sort 的平均情况分析。

数组 A 中的元素为 $\{x_1, x_2, \cdots\cdots, x_n\}$，并且 $x_i \neq x_j, 1 \leq i, j \leq n, i \neq j$。

n 个元素共有 $n!$ 种排列，假定，每一种排列的概率相同。

前面$i-1$个元素已按递增顺序排序，把元素x_i插入到合适位置的i种可能：

$j=1$： x_i是序列中最小的，需执行$i-1$次比较；

$j=2$： x_i是这个序列中第二小的，仍需执行$i-1$次比较；

$j=3$： x_i是这个序列中第三小的，需执行$i-2$次比较；

……

$j=i$： x_i是这个序列中最大的，需执行 1 次比较。

当$2\le j\le i$时，算法需执行的比较次数为$i-j+1$。

这i种可能性的概率相同，都是$1/i$。元素x_i插入到合适的位置的平均比较次数T_i：

$$T_i=\frac{i-1}{i}+\sum_{j=2}^{i}\frac{i-j+1}{i}$$

$$=\frac{i-1}{i}+\sum_{j=1}^{i-1}\frac{j}{i}$$

$$=1-\frac{1}{i}+\frac{1}{2}(i-1)$$

$$=\frac{1}{2}+\frac{i}{2}-\frac{1}{i}$$

分别把$x_2, x_3, \cdots\cdots, x_n$插入到序列中的合适位置，所需的平均比较总次数$T$为：

$$T=\sum_{i=2}^{n}T_i=\sum_{i=2}^{n}(\frac{1}{2}+\frac{i}{2}-\frac{1}{i})$$

$$=\frac{1}{2}(n-1)+\frac{1}{2}\sum_{i=2}^{n}i-\sum_{i=1}^{n}\frac{1}{i}+1$$

$$=\frac{1}{2}(n-1)+\frac{1}{4}(n\ (n+1)-2)+1-\sum_{i=1}^{n}\frac{1}{i}$$

$$=\frac{1}{4}(n^2+3n)-\sum_{i=1}^{n}\frac{1}{i}$$

因为：

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1$$

所以：

$$T\approx\frac{1}{4}(n^2+3n)-\ln n$$

由此可得，插入排序算法 insert_sort 在平均情况下的时间复杂性是 $\Theta(n^2)$。

例 1.18　冒泡排序算法在平均情况下的下界分析。

算法 1.15　改进的冒泡算法

输入：被排序的数组 A[],数组的元素个数 n

输出：按递增顺序排序的数组 A[]

```
template <class Type>
void bubble_sort(Type A[],int n)
{
    int i,k,flag;
    k = n - 1,   flag = 1;
    while (flag) {
        k = k – 1,  flag = 0;
        for (i=0;i<=k;i++) {
            if (A[i] > A[i+1]) {
                swap(A[i],A[i+1]);
                flag = 1;
            }
        }
    }
}
```

1、最好的情况：所有初始数据都顺序排序。至少是 $\Omega(n)$。

2、最坏的情况：所有初始数据都逆序排序。执行次数为：

$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} i=\frac{1}{2}n(n-1)$$

最坏情况下的运行时间至多是 $O(n^2)$。

3、平均情况下运行时间下界：

定义 1.10　设 $a_1,a_2,\cdots,a_n$ 是集合 $\{1,2,\cdots,n\}$ 的一个排列，如果 $i<j$ 且 $a_i>a_j$，则对偶 (a_i,a_j) 称为该排列的一个逆序。

例：排列 3，4，1，5 有逆序 $(3,1)$ 及 $(4,1)$。若使元素按序排列，至少必须交换两次。

交换两个相邻元素，则逆序的总数将增 1 或减 1。

不断地交换两个相邻元素，使其逆序个数往减少的方向改变，当逆序个数减少为 0 时，就是一个有序的排列了。

排列中逆序的数目，是算法所执行的元素比较次数的下界。

n 个元素共有 $n!$ 种排列，所有排列的平均逆序的个数，也就是算法所执行的平均比较次数的下界。

例如，集合 $A=\{1,2,3\}$ 有如下 3!= 6 种排列：

排 列	逆序数目 k
1 2 3	0
1 3 2	1
2 1 3	1
2 3 1	2
3 1 2	2
3 2 1	3

令 $S(k)$ 是逆序个数为 k 时的排列数目，则有：

$$S(0)=1 \qquad S(1)=2 \qquad S(2)=2 \qquad S(3)=1$$

记 $mean(n)$ 为 n 个元素集合的所有排列的逆序的平均个数。则具有 3 个元素集合的逆序的平均个数为：

$$\begin{aligned} mean(3) &= \frac{1}{3!}(S(0)\cdot 0+S(1)\cdot 1+S(2)\cdot 2+S(3)\cdot 3) \\ &= \frac{1}{6}(1\cdot 0+2\cdot 1+2\cdot 2+1\cdot 3) \\ &= 1.5 \end{aligned}$$

n 个元素的集合的所有排列：

最好的情况下，所有的元素都已经是顺序排列的了，该排列的逆序个数为 0；

最坏的情况下，所有的元素都是逆序排列的，该排列的逆序个数为 $n(n-1)/2$。

逆序的平均个数为：

$$mean(n)=\frac{1}{n!}\sum_{k=0}^{n(n-1)/2} k\,S(k)$$

Donald E.Knuth 对逆序的分布规律进行了研究，他利用生成函数的性质进行了复杂的推导，得出了下面的公式：

$$\begin{aligned} mean(n) &= \sum_{k=1}^{n}\frac{k-1}{2} \\ &= \frac{1}{4}n(n-1) \end{aligned}$$

因此，冒泡排序在平均情况下的运行时间的下界是 $\Omega(n^2)$。Donald E.Knuth 对冒泡排序在平均情况下的运行时间也进行了研究，可参考文献[6]。

1.4　算法的空间复杂性

一、算法的空间复杂性，指的是为解一个问题实例而需要的存储空间。

二、两种处理方法

1、算法所需要存储空间，仅是算法所需要的工作空间。

例：线性检索算法 linear_search、叉检索算法 binary_search，空间复杂性是 $\Theta(1)$。

合并算法 merge，空间复杂性是 $\Theta(n)$。

令 $T(n)$ 和 $S(n)$ 分别表示算法的时间复杂性和空间复杂性，那么，一般情况下有 $S(n)=O(T(n))$。

2、算法所需要存储空间为算法在运行时所占用的内存空间的总和，包括存放输出入数据的变量单元、程序代码、工作变量、常数、以及运行时的引用型变量所占用的空间、及递归栈所占用的空间。

算法所需要的存储空间 S_A 可表示为：

$$S_A = c + S(n)$$

c：程序代码、常数等固定部分，

$S(n)$：是与输入规模有关的部分。输出入数据所占用的空间、工作空间、递归栈

在分析算法的空间复杂性时，主要考虑的是 $S(n)$。

在很多问题中，时间和空间是一个对立面。为算法分配更多的空间，可以使算法运行得更快。反之，当空间是一个重要因素时，有时，需要用算法的运行时间去换取空间。

1.5　最优算法

1、已知问题Π的任何算法的运行时间是 $\Omega(f(n))$，则对以时间 $O(f(n))$ 求解问题Π的任何算法，都认为是最优算法。

2、运行时间同阶的算法，常数因子小的算法，优于常数因子大的算法。

3、时间复杂性渐近阶的确定，与 n_0 及常数 c 的选取有关，当规模很小时，复杂性阶低的算法，不一定比复杂性阶高的算法更有效。