第1章

绪论

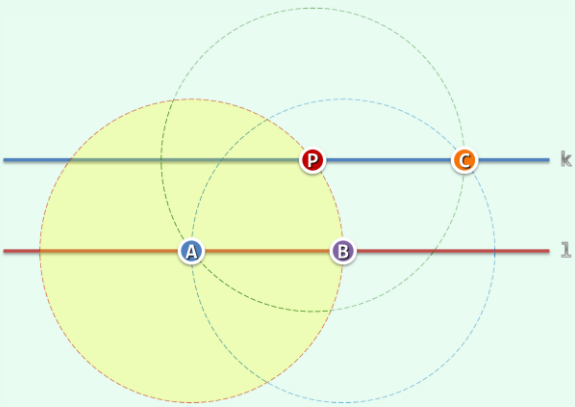
[1-1] 试借助基本的几何作图操作描述一个算法过程，实现“过直线外一点作其平行线”的功能。

【解答】

算法x1.1[①]给出了一个可行的算法，其原理及操作过程如图x1.1所示。

```
parallel(l, P)
输入：直线l及其外一点P
输出：经过P且平行于l的直线
1．以l上任意一点A为中心、以|AP|为半径作圆
2．在该圆与l的两个交点中，任取其一记作B
3．分别以B和P为中心、以|AP|为半径各作一圆
4．两圆相交于A及另一点，将后者记作C
5．过P和C绘制一条直线k
```

算法x1.1 过直线外一点作其平行线



图x1.1 过直线外一点作其平行线

[1-2] 《海岛算经》讨论了如下遥测海岛高度的问题：

今有望海岛，立两表，齐高三丈，前后相去千步，令后表与前表参相直。从前表却行一百二十三步，人目著地取望岛峰，与表末参合。从后表却行一百二十七步，人目著地取望岛峰，亦与表末参合。问岛高及去表各几何？

刘徽所给出的解法是：

以表高乘表间为实；相多为法，除之。所得加表高，即得岛高。
求前表去岛远近者，以前表却行乘表间为实。相多为法。除之，得岛去表数。

a） 该算法的原理是什么？

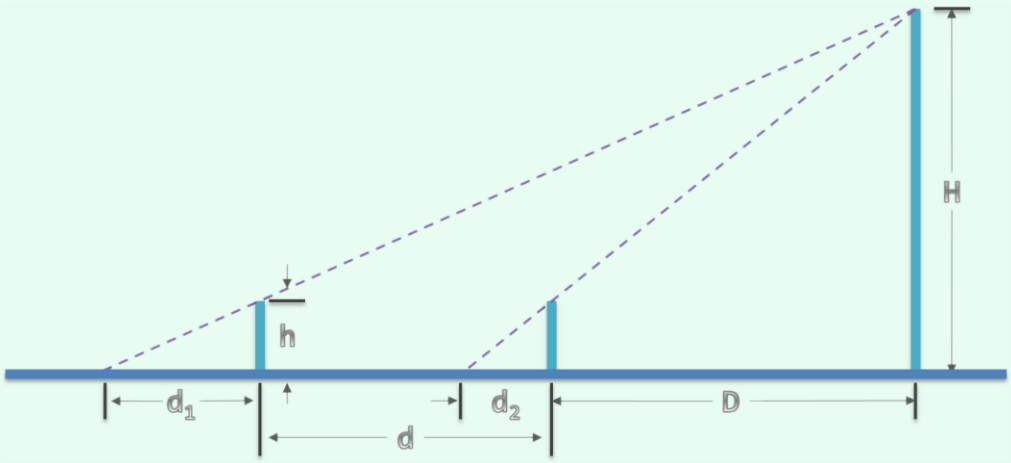
【解答】

刘徽通过设立两根立柱（表），利用光线直线传播的性质，根据海岛峰顶经过立柱顶端后到地面的投影位置，巧妙地计算出远方海岛的高度。

具体地如图x1.2所示，若将立柱高度（表高）记作h，后立柱、前立柱间距（表间）记作d，对应的投影距离分别记作d_1和d_2，则不难导出：

$$\text{海岛高度}H = d \times h / (d_1 - d_2) + h$$
$$= 1000 \times 3 / (127 - 123) + 3 = 753 \text{（丈）}$$
$$\text{海岛距离}D = d \times d_2 / (d_1 - d_2)$$
$$= 1000 \times 123 / (127 - 123) = 30750 \text{（步）}$$

[①] 为与主教材相互区别，本习题解析中所有插图、表格、算法、代码等编号前，均增加“x”标识



图x1.2 《海岛算经》算法原理

按古制，1丈合10尺，1步合6尺，3丈合5步，1里合300步，故前者为1255步（4里又55步），后者为102里又150步。

b） 试以伪代码形式描述该算法的过程。

【解答】

按照以上原理，计算海岛高度H的算法可形式地描述如代码x1.1所示：

```
1 float islandHeight(float d1, float d2, float d, float h) { //后表却步、前表却步、表间、表高
2    float pha = d1 - d2; //二去表相减为相多，以为法
3    float shi = d * h; //前后表相去为表间，以表高乘之为实
4    return shi / pha + h; //以法除之，加表高，即是岛高积步
5 }
```

代码x1.1 《海岛算经》中计算海岛高度的算法

而计算前立柱至海岛距离D的算法，可形式地描述如代码x1.2所示：

```
1 float islandDistance(float d1, float d2, float d) { //后表却步、前表却步、表间
2    float shi = d2 * d; //前去表乘表间（得一十二万三千步）
3    float pha = d1 - d2; //以相多（四步）为法
4    return shi / pha; //除之（得三万七百五十步；又以里法三百步除之，得一百二里一百五十步）
5 }
```

代码x1.2 《海岛算经》中计算海岛距离的算法

以上算法中的注释，引自（唐）李淳风等对原书的注解。

c） 该算法借助了哪些计算工具？

【解答】

可见，以上算法利用了（垂直地面、等长的）两根立柱、（沿直线传播的）光线，以及度量立柱高度、立柱间距及其投影距离的直尺。当然，人眼在此也不可或缺，否则难以确定投影位置。

[1-3] 试分别举出实例说明，在对包含 n 个元素的序列做起泡排序的过程中，可能发生以下情况：

a） 任何元素都无需移动（从而内循环仅执行一轮即可终止算法）；

【解答】

比如，所有元素已经按序排列。

b） 某元素会一度（朝着远离其最终位置的方向）逆向移动；

【解答】

比如序列：

{ n, n - 1, 1, 2, ..., n - 2 }

经首轮扫描交换后为

{ n - 1, 1, 2, ..., n - 2, n }

其中的次大元素n - 1，最终位置在初始位置的右侧，在上述过程中却向左侧移动一个单元。

c） 某元素的初始位置与其最终位置相邻，甚至已经处于最终位置，却需要参与 n - 1 次交换；

【解答】

当n为偶数时，考查序列：

{ n/2 + 2, n/2 + 3, ..., n, n/2 + 1, 1, 2, ..., n/2 }

其中元素n/2 + 1只需右移一个单元，即是最终位置。但按照起泡排序算法，在前n/2 - 1轮扫描交换中，它都左移一个单元。在此后的第n/2轮扫描交换中，它连续地右移n/2个单元，方抵达最终位置。整个过程累计参与n - 1次交换。

当n为奇数时，考查序列：

{ (n + 1)/2 + 1, (n + 1)/2 + 2, ..., n, (n + 1)/2, 1, 2, ..., (n - 1)/2 }

其中元素(n + 1)/2已经处于最终位置。但按照起泡排序算法，在前(n - 1)/2轮扫描交换中，它都左移一个单元。在此后的第n/2轮扫描交换中，它连续地右移(n - 1)/2个单元，方抵达最终位置。整个过程累计参与n - 1次交换。

d） 所有元素都需要参与 n - 1 次交换。

【解答】

仔细观察以上实例不难看出：对于序列中的任何一个元素，只要更小（大）的元素均处于其右（左）侧，则该元素或早或晚必然与其余元素各交换一次，累计n - 1次。也就是说，这类元素与其余的每个元素均构成一个逆序对（inversion）——参见习题[3-11]。

实际上，若所有元素均具有这种特性，则必为完全逆序的序列，比如：

{ n, n - 1, ..., 2, 1 }

其中共包含n(n - 1)/2个逆序对。

[1-4] 对 n 个整数的排序，能否保证在最坏情况下仍可在少于 $o(n)$ 的时间内完成？为什么？

【解答】

不能。这一结论，可以从几个方面来理解。

首先，为确定n个整数的排列次序，至少需要对每个元素访问一次。否则，即便其余n - 1个整数业已排序，在未读取出该整数的准确数值之前，仍无法确定整体的排列次序。

其次，同一组整数的输入次序可能不同，其中必有一种是完全错位的，即没有任何一个元素是就位的。此种情况下，每个整数都至少需要参与一次比较或者移动操作。

最后，即便排序结果已知，在输出的过程中每个整数也必须花费常数的时间。

在后面的第2.7节，将就此问题给出并证明一个更强的结论。

[1-5] 随着问题输入规模的不断扩大，同一算法所需的计算时间通常都呈单调递增趋势，但情况亦并非总是如此。试举实例说明，随着输入规模的扩大，同一算法所需的计算时间可能上下波动。

【解答】

例如，对任意整数n ≥ 2做素因子分解的一种蛮力算法是，反复地从2到n递增地逐一尝试。采用这一算法，对于不同的输入n，所需的时间T(n)如下表所示。

n	2	3	4	5	6	7	8	...
T(n)	1	2	2	4	3	6	3	...

n	47	48	49	50	51	52	53	...
T(n)	46	6	12	9	18	14	52	...

n	62	63	64	65	66	67	68	...
T(n)	31	10	6	16	13	66	18	...

实际上，任意素数n都对应于该算法的最坏情况，即：

$$T(n) = n - 1 = \mathcal{O}(n)$$

而任意形如n = 2^k的整数都对应于最好情况，即：

$$T(n) = k = \mathcal{O}(\log n)$$

因此，该算法的运行时间将在这两种极端情况之间，呈波动形式上下起伏。

[1-6] 在一台速度为 1G flops 的电脑上使用教材中代码 1.1 中的 bubblesort1A()算法，大致需要多长时间才能完成对全国人口记录的排序？

【解答】

输入规模按n = 10^9（十亿人口）计，计算量为n^2 = 10^18。

该电脑的计算能力按10^9计，则大致需要10^(18 - 9) = 10^9秒 = 30年。

根据数据结构和算法的渐进复杂度，凭借在实际计算环境中积累的经验，针对计算过程主要部分进行的此类粗略估算，也称作封底估算（back-of-the-envelope calculation），意指只需在废信封背面寥寥数行即可完成的估算。

好的封底估算不仅简便易行，而且可以大体上对数据结构和算法的实际性能做出较为准确的比较和判断。这种对总体计算效率的把握能力，也是优秀程序员必备的一项基本素质。

[1-7] 试用C++语言描述一个包含循环、分支、子函数调用，甚至递归结构的算法，要求具有常数的总体时间复杂度。

【解答】

一个综合多种情况的实例，如代码x1.3所示。

这个名为O1()的函数虽然设有一个循环，但无论其输入参数n有多大，循环控制变量i总是以1 + n/2013为步长，从0逐次递增至n。因此迭代步数大致为2013 = $O(1)$。

该函数还设有一个转向标志UNREACHABLE，但转向条件“1 + 1 != 2”永远无法满足，故这个转向分支实质上形同虚设。

以下，是经过条件判断之后对某个子函数doSomething()的“调用”。然而，这里的转向条件“n * n < 0”依然属于逻辑上的永非式，因此相应的调用绝不可能发生。

```
1  void O1( unsigned int n ) {
2     for ( unsigned int i = 0; i < n; i += 1 + n/2013 ) { //循环：但迭代至多2013次，与n无关
3  UNREACHABLE: //无法抵达的转向标志
4        if ( 1 + 1 != 2 ) goto UNREACHABLE; //分支：条件永非，转向无效
5        if ( n * n < 0 ) doSomething(n); //分支：条件永非，调用无效
6        if ( (n + i) * (n + i) < 4 * n * i ) doSomething( n ); //分支：条件永非，调用无效
7        if ( 2 == (n * n) % 5 ) O1( n + 1 ); //分支：条件永非，递归无效
8        int f = fib(n); if ( 12 < n && (sqrt(f) * sqrt(f) == f) ) O1( n - 1 ); //分支：条件永非
9     }
10 }
```

代码x1.3 包含循环、分支、子函数调用甚至递归结构，但具有常数时间复杂度的算法

目前某些高级的编译器，已经能够识别前一类完全由常数定义的永非式，并在编译过程中作相应的自动优化。然而不幸的是，对于由变量参与定义的这种（以及更为复杂的）逻辑条件，编译器尚不能有效地判别和优化。

例如，接下来经过条件判断“(n + i) * (n + i) < 4 * n * i”之后对子函数doSomething()的“调用”，也绝对不可能被执行。实际上，作为关于不等式的基本常识我们知道，非负整数的算术平均不可能小于其几何平均。

又如，再接下来经过条件判断“2 == (n * n) % 5”之后的“递归调用”，也绝对不可能被执行。实际上，由基本的数论知识不难验证，任意整数的平方关于5整除之后的余数，断乎不可能是2（或3）。

最后，经过条件判断“(12 < n) && (sqrt(f) * sqrt(f) == f)”之后的“递归调用”，依然绝对不可能被执行——实际上在Fibonacci数中，只有fib(0)、fib(1)、fib(2)、fib(12)是平方数，fib(n > 12)必然都不是。

不难理解，相对于前两种情况，后三种无效的分支语句几乎无法有效地辨别。由以上可见，对于程序时间复杂度的估算，不能完全停留和依赖于其外在的流程结构；更为准确而精细的分析，必然需要以对其内在功能语义的充分理解为基础。

[1-8] 试证明，在用对数函数界定渐进复杂度时，常底数的具体取值无所谓。

【解答】

设某函数的上界可表示为$f(n) = \mathcal{O}(\log_a n)$，其中$a > 1$为常数。

则对任一常数$b > 1$，因$\ln b / \ln a$为常数，故根据大$\mathcal{O}$记号的性质有：

$$f(n) = \mathcal{O}(\log_a n) = \mathcal{O}((\ln b / \ln a)\log_b n) = \mathcal{O}(\log_b n)$$

由此可见，无论更换为任一常数底，只会影响到常系数。

[1-9] 试证明，对于任何$\varepsilon > 0$，都有$\log n = \mathcal{O}(n^{\varepsilon})$。

【解答】

我们知道，函数$\ln n$增长得极慢，故总存在$M > 0$，使得$n > M$之后总有$\ln n < \varepsilon n$。

令$N = e^M$，则当$n > N$（即$\ln n > M$）之后，总有：$\ln(\ln n) < \varepsilon \ln n$，亦即：$\ln n < n^{\varepsilon}$。

实际上从另一角度来看，既然$n^{1-\varepsilon/2} \cdot \ln n$是凹函数（concave function，导数递减），$n^{1+\varepsilon/2}$是凸函数（convex function，导数递增），则也不难得出以上结论。

[1-10] 试证明，在大$\mathcal{O}$记号的意义下

a） 等差级数之和与其中最大一项的平方同阶；

【解答】

考查首项为常数x、公差为常数$d > 0$、长度为n的等差级数：

$$\{ x, x + d, x + 2d, \ldots, x + (n - 1)d \}$$

其中末项$(n - 1)d = \Theta(n)$，各项总和为：

$$(d/2)n^2 + (x - d/2)n = \Theta(n^2)。$$

b） 等比级数之和与其中最大一项同阶。

【解答】

考查首项为常数x、公比为常数$d > 1$、长度为n的等比级数：

$$\{ x, xd, xd^2, \ldots, xd^{n-1} \}$$

其中末项$xd^{n-1} = \Theta(d^n)$，各项总和为：

$$x(d^n - 1)/(d - 1) = \Theta(d^n)$$

[1-11] 若$f(n) = \mathcal{O}(n^2)$且$g(n) = \mathcal{O}(n)$，则以下结论是否正确：

a） $f(n) + g(n) = \mathcal{O}(n^2)$；

【解答】

正确。根据大$\mathcal{O}$记号的性质，多项式中的低次项可以忽略。

b） $f(n) / g(n) = \mathcal{O}(n)$；

【解答】

错误。

比如，对于$f(n) = n\log n = \mathcal{O}(n^2)$和$g(n) = 1 = \mathcal{O}(n)$，有$f(n)/g(n) = n\log n \neq \mathcal{O}(n)$。

c） g(n) = *O*(f(n))；

【解答】

错误。

比如，对于$f(n) = \log n = O(n^2)$和$g(n) = n = O(n)$，有$g(n) \neq O(f(n))$。

d） f(n) * g(n) = *O*(n^3)

【解答】

正确。

由大*O*记号定义，存在常数$c_1 > 0$和$N_1 > 0$，使得当$n > N_1$后总有$f(n) < c_1 n^2$。

同理，存在常数$c_2 > 0$和$N_2 > 0$，使得当$n > N_2$后总有$g(n) < c_2 n$。

于是，若令$c = c_1 c_2$，$N = \max(N_1, N_2)$，则当$n > N$后，总有：

$$f(n)*g(n) < cn^3 = O(n^3)$$

[1-12] 改进教材 13 页代码 1.2 中 countOnes()算法，使得时间复杂度降至

a） *O*(countOnes(n))，线性正比于数位 1 的实际数目；

【解答】

如代码x1.4所示，这里通过位运算的技巧，自低（右）向高（左）逐个地将数位1转置为0。

```
1 int countOnes1 ( unsigned int n ) { //统计整数二进制展开中数位1的总数：O(ones)正比于数位1的总数
2    int ones = 0; //计数器复位
3    while ( 0 < n ) { //在n缩减至0之前，反复地
4       ones++; //计数（至少有一位为1）
5       n &= n - 1; //清除当前最靠右的1
6    }
7    return ones; //返回计数
8 } //等效于glibc的内置函数int __builtin_popcount (unsigned int n)
```

代码x1.4 countOnes()算法的改进版

对于任意整数n，不妨设其最低（右）的数位1对应于2^k，于是n的二进制展开应该如下：

x x ... x 1 0 0 ... 0

其中数位x可能是0或1，而最低的k + 1位必然是"1 0 0 ... 0"，即数位1之后是k个0。

于是相应地，n - 1的二进制展开应该如下：

x x ... x 0 1 1 ... 1

也就是说，其最低的k + 1位与n恰好相反，其余的（更高）各位相同。

因此，二者做位与运算（n & (n - 1)）的结果应为：

x x ... x 0 0 0 ... 0

等效于将原n二进制展开中的最低位1转置为0。

以上计算过程，仅涉及整数的减法和位与运算各一次。若不考虑机器的位长限制，两种运算均可视作基本运算，各自只需*O*(1)时间。因此，新算法的运行时间，应线性正比于n的二进制展开中数位1的实际数目。

b） $O(\log_2 W)$，$W = O(\log_2 n)$为整数的位宽。

【解答】

一种可行的实现方式，如代码x1.5所示。

```
#define POW(c) (1 << (c)) //2^c
#define MASK(c) (((unsigned long) -1) / (POW(POW(c)) + 1)) //以2^c位为单位分组，相间地全0和全1
// MASK(0) = 55555555(h) = 01010101010101010101010101010101(b)
// MASK(1) = 33333333(h) = 00110011001100110011001100110011(b)
// MASK(2) = 0f0f0f0f(h) = 00001111000011110000111100001111(b)
// MASK(3) = 00ff00ff(h) = 00000000111111110000000011111111(b)
// MASK(4) = 0000ffff(h) = 00000000000000001111111111111111(b)

//输入：n的二进制展开中，以2^c位为单位分组，各组数值已经分别等于原先这2^c位中1的数目
#define ROUND(n, c) (((n) & MASK(c)) + ((n) >> POW(c) & MASK(c))) //运算优先级：先右移，再位与
//过程：以2^c位为单位分组，相邻的组两两捉对累加，累加值用原2^(c + 1)位就地记录
//输出：n的二进制展开中，以2^(c + 1)位为单位分组，各组数值已经分别等于原先这2^(c + 1)位中1的数目

int countOnes2 ( unsigned int n ) { //统计整数n的二进制展开中数位1的总数
   n = ROUND ( n, 0 ); //以02位为单位分组，各组内前01位与后01位累加，得到原先这02位中1的数目
   n = ROUND ( n, 1 ); //以04位为单位分组，各组内前02位与后02位累加，得到原先这04位中1的数目
   n = ROUND ( n, 2 ); //以08位为单位分组，各组内前04位与后04位累加，得到原先这08位中1的数目
   n = ROUND ( n, 3 ); //以16位为单位分组，各组内前08位与后08位累加，得到原先这16位中1的数目
   n = ROUND ( n, 4 ); //以32位为单位分组，各组内前16位与后16位累加，得到原先这32位中1的数目
   return n; //返回统计结果
} //32位字长时，O(log_2(32)) = O(5) = O(1)
```

代码x1.5 countOnes()算法的再改进版

这里运用了多种二进制位运算的技巧，其数学原理及计算过程，请读者参照所附的注解，细心理解、体会和记忆。

可见，若计算模型支持的整数字长为W，则对于任意整数$n \in [0, 2^W)$，都可在：

$$T(n) = O(\log_2 W) = O(\log W) = O(\log\log n)$$

时间内统计出n所含比特1的总数。

通常，$O(\log\log n)$可以视作常数。比如，就人类目前所能感知的整个宇宙范围而言，所有基本粒子的总数约为：

$$N = 10^{81} = 2^{270}$$

即便如此之大的N，也不过：

$$\log\log N = \log 270 < 9$$

而在目前主流计算环境中，unsigned int类型的位宽多为W = 32，有：

$$\log\log N = \log W = 5$$

[1-13] 实现教材 14 页代码 1.4 中 power2BF_I()算法的递归版，要求时间复杂度保持为 $O(n) = O(2^r)$。

【解答】

一种可行的实现方式，如代码x1.6所示。

```
1 __int64 power2BF ( int n ) { //幂函数2^n算法（蛮力递归版），n >= 0
2    return ( 1 > n ) ? 1 : power2BF ( n - 1 ) << 1; //递归
3 } //O(n) = O(2^r)，r为输入指数n的比特位数
```

代码x1.6 power2BF_I()算法的递归版

与原先的迭代版相比，该版本的原理完全一致，只不过计算方向恰好颠倒过来：

- 为计算出2^n，首先通过递归计算出2^(n - 1)，然后通过左移返回值加倍
- 经向下递归深入n层并抵达递归基power2BF(0)之后，再逆向地逐层返回
- 每返回一层，都执行一次加倍

就空间复杂度而言，该版本无形中提高至$O(n)$。相对于原就地的迭代版，反而有所倒退。

[1-14] 实现教材 21 页代码 1.8 中 power2()算法的迭代版，要求时间复杂度保持为 $O(\log n) = O(r)$。

【解答】

一种可行的实现方式，如代码x1.7所示。

```
1 __int64 power2_I ( int n ) { //幂函数2^n算法（优化迭代版），n >= 0
2    __int64 pow = 1; //O(1)：累积器初始化为2^0
3    __int64 p = 2; //O(1)：累乘项初始化为2
4    while ( 0 < n ) { //O(logn)：迭代log(n)轮，每轮都
5       if ( n & 1 ) //O(1)：根据当前比特位是否为1，决定是否
6          pow *= p; //O(1)：将当前累乘项计入累积器
7       n >>= 1; //O(1)：指数减半
8       p *= p; //O(1)：累乘项自乘
9    }
10   return pow; //O(1)：返回累积器
11 } //O(logn) = O(r)，r为输入指数n的比特位数
```

代码x1.7 power2()算法的迭代版

与原先的递归版相比，该版本的原理完全一致，只不过计算方向却恰好颠倒过来：由低到高，依次检查n二进制展开中的各比特，在该比特为1时累乘以累乘项p。

这里的辅助变量p，应始终等于各比特所对应的指数权重，亦即：

2^1, 2^2, 2^4, 2^8, 2^16, ...

因此，其初始值应置为：

2^1 = 2

而此后每经过一步迭代（并进而转向更高一位），p都会通过自平方完成更新。

不难看出，这个版本仅需$O(1)$的辅助空间，故就空间复杂度而言，较之原递归的版本有了很大改进。

以上算法不难推广至一般的情况。比如，对于任意的整数a和n，计算a^n的一个通用算法，可实现如代码x1.8所示。

```
1  __int64 power ( __int64 a, int n ) { //a^n算法：n >= 0
2     __int64 pow = 1; //O(1)
3     __int64 p = a; //O(1)
4     while ( 0 < n ) { //O(logn)
5        if ( n & 1 ) //O(1)
6           pow *= p; //O(1)
7        n >>= 1; //O(1)
8        p *= p; //O(1)
9     }
10    return pow; //O(1)
11 } //power()
```

代码x1.8 通用的迭代版幂函数算法

请读者参照注释，对该算法的时间复杂度做一分析。

[1-15] 考查最大元素问题：从 n 个整数中找出最大者。

a） 试分别采用迭代和递归两种模式设计算法，在线性时间内解决该问题；

b） 用 C++语言实现你的算法，并分析它们的复杂度。

【解答】

该问题迭代版算法一种可行的实现方式，如代码x1.9所示。

```
1 int maxI ( int A[], int n ) { //求数组最大值算法（迭代版）
2    int m = INT_MIN; //初始化最大值纪录，O(1)
3    for ( int i = 0; i < n; i++ ) //对全部共O(n)个元素，逐一
4       m = max ( m, A[i] ); //比较并更新，O(1)
5    return m; //返回最大值，O(1)
6 } //O(1) + O(n) * O(1) + O(1) = O(n + 2) = O(n)
```

代码x1.9 数组最大值算法（迭代版）

该问题线性递归版算法一种可行的实现方式，如代码x1.10所示。

```
1 int maxR ( int A[], int n ) { //数组求最大值算法（线性递归版）
2    if ( 2 > n ) //平凡情况，递归基
3       return A[n - 1]; //直接（非递归式）计算
4    else //一般情况，递归：在前n - 1项中的最大值与第n - 1项之间，取大者
5       return max ( maxR ( A, n - 1 ), A[n - 1] );
6 } //O(1) * 递归深度 = O(1) * (n + 1) = O(n)
```

代码x1.10 数组最大值算法（线性递归版）

该问题二分递归版算法一种可行的实现方式，如代码x1.11所示。

```
int maxR ( int A[], int lo, int hi ) { //计算数组区间A[lo, hi)的最大值（二分递归）
   if ( lo + 1 == hi ) //如遇递归基（区间长度已降至1），则
      return A[lo]; //直接返回该元素
   else { //否则（一般情况下lo + 1 < hi），则递归地
      int mi = ( lo + hi ) >> 1; //以中位单元为界，将原区间一分为二：A[lo, mi)和A[mi, hi)
      return max ( maxR ( A, lo, mi ), maxR ( A, mi, hi ) ); //计算子区间的最大值，再从中取大者
   }
} //O(hi - lo)，线性正比于区间的长度
```

代码x1.11 数组最大值算法（二分递归版）

以上诸版本时间复杂度的分析结论，请参见所附注释。

就空间复杂度而言，迭代版为$O(1)$，已属于就地算法。递归版的所需的空间量均取决于最大的递归深度，对二分递归而言为$O(\log n)$，对线性递归而言为$O(n)$。

[1-16] 考查如下问题：设S为一组共n个正整数，其总和为2m，判断是否可将S划分为两个不相交的子集，且各自总和均为m？美国总统选举即是该问题的一个具体实例：

若有两位候选人参选，并争夺n = 51个选举人团（50个州和1个特区）的共计2m = 538张选举人票，是否可能因两人恰好各得m = 269张，而不得不重新选举？

a） 试设计并实现一个对应的算法，并分析其时间复杂度；

【解答】

采用蛮力策略，逐一枚举S的每一子集，并统计其中元素的总和。一旦发现某个子集的元素总和恰为m，即成功返回。若直至枚举完毕均未发现此类子集，即失败返回。

b） 若没有其它（诸如限定整数取值范围等）附加条件，该问题可否在多项式时间内求解？

【解答】

此问题已被证明是NP完全的（NP-complete）。这意味着，就目前的计算模型而言，不存在可在多项式时间内回答此问题的算法。反过来，上述基于直觉的蛮力算法已属最优。

[1-17] 试证明，若每个递归实例仅需使用常数规模的空间，则递归算法所需的空间总量将线性正比于最大的递归深度。

【解答】

根据递归跟踪分析法，在递归程序的执行过程中，系统必须动态地记录所有活跃的递归实例。在任何时刻，这些活跃的递归实例都可按照调用关系，构成一个调用链，该程序执行期间所需的空间，主要用于维护上述调用链。不难看出，按照题目所给的条件，这部分空间量应线性正比于调用链的最大长度，亦即最大的递归深度。

在教材的4.2.1节，还将针对递归实现机制——函数调用栈——做详细的介绍。届时，我们将了解上述过程更多的具体细节。简而言之，以上所定义的调用链，实际上就对应于该栈中的所

有帧。在任何时刻，其中每一对相邻的帧，都对应于存在“调用与被调用”关系的一对递归实例。若各递归实例所需空间均为常数量，则空间占用量与栈内所含帧数成正比，并在递归达到最深层时达到最大。

总而言之由上可见，递归算法所需的空间总量，并不直接取决于计算过程中出现过的递归实例总数，与总体消耗的计算时间也没有必然的关系。

[1-18] 试采用递推方程法，分析教材 17 页代码 1.5 中线性递归版 sum()算法的空间复杂度。

【解答】

设采用该算法对长度为n的数组统计总和，所需空间量为S(n)，于是可得递推方程如下：

$$S(1) = \mathcal{O}(1)$$

$$S(n) = S(n - 1) + \mathcal{O}(1)$$

两式联合求解即得：

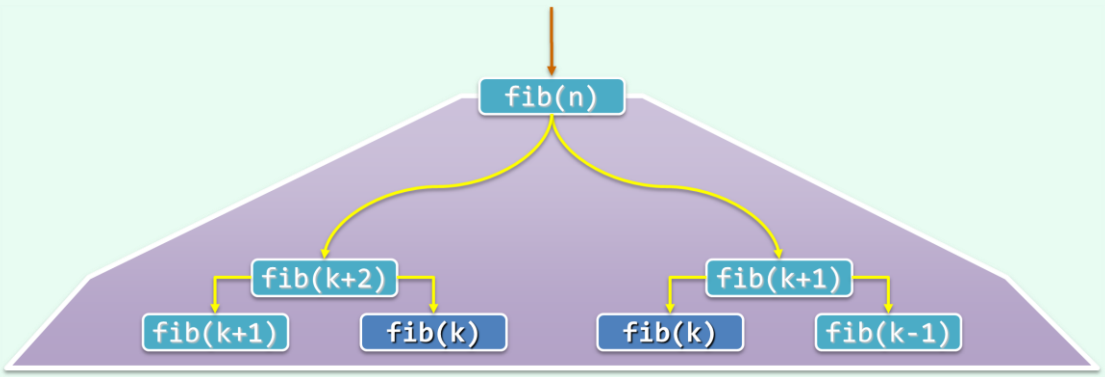
$$S(n) = \mathcal{O}(n)$$

[1-19] 考查如教材 24 页代码 1.12 所示的二分递归版 fib(n)算法，试证明：

a) 对任一整数 1 ≤ k ≤ n，形如 fib(k)的递归实例，在算法执行过程中都会先后重复出现 fib(n - k + 1)次；

【解答】

在该算法的递归跟踪图中，每向下递归深入一层，入口参数就减一（向左）或减二（向右）。在从入口fib(n)通往每一fib(k)递归实例的沿途，各递归实例的入口参数只能依次减一或减二。因此，fib(k)出现的次数，应该等于从n开始，经每次减一或减二，最终减至k的路径总数。



图x1.3 fib()算法中递归实例fib(k)的两种出现可能

考查这些路径的最后一步，如图x1.3所示无非两种可能：或由fib(k + 1)向左抵达fib(k)，或由fib(k + 2)向右抵达fib(k)。故若将fib(k)出现的次数记作F(k)，则可得递推式如下：

$$F(n) = 1$$

$$F(k) = F(k + 1) + F(k + 2)$$

两式联合求解，即得：

$$F(k) = fib(n - k + 1)$$

b） 该算法的时间复杂度为指数量级；

【解答】

该算法每个递归实例自身仅消耗常数时间，故总体运行时间应线性正比于递归实例的总数：

$$\sum_{k=0}^{n} F(k) = \sum_{k=0}^{n} fib(n - k + 1)$$

$$= \sum_{k=1}^{n+1} fib(k) = fib(n + 3) - 1 = \mathcal{O}(\Phi^n)$$

其中，$\Phi = (1 + \sqrt{5})/2 = 1.618$。

为便捷而准确地估计此类算法的时间和空间复杂度，我们不妨记住以下关于Φ的近似估计：

Φ^5 ≈ 10 , Φ^67 ≈ 10^14

Φ^3 ≈ 2^2 , Φ^36 ≈ 2^25

就以这里的二分递归版fib(n)算法为例。如读者编译并运行该算法，就会发现在其计算至第45项之后，即可感觉到明显的（秒量级以上的）延迟。

其实，只要注意到目前常见的CPU主频大致在1G Hz = 10^9 Hz量级，而且由以上近似经验值可估算出Φ^45 ≈ 10^9，即不难解释和理解上述现象。

c） 该算法的最大递归深度为 $\mathcal{O}(n)$；

【解答】

在该算法运行过程中的任一时刻，（在函数调用栈内）活跃的递归实例所对应的入口参数，必然从n（栈底）到k ≥ 0（栈顶）严格单调地递减。因此，活跃实例的总数不可能多于n个。

d） 该算法具有线性的空间复杂度。

【解答】

既然最大的递归深度不超过n，故由习题[1-18]的结论，该算法所需的空间亦不超过$\mathcal{O}(n)$。

[1-20] 考查 Fibonacci 数的计算。

a） 试证明，任意算法哪怕只是直接打印输出 fib(n)，也至少需要Ω(n)的时间；

（提示：无论以任何常数为进制，fib(n)均由Θ(n)个数位组成）

【解答】

从渐进角度看，fib(n) = $\Theta(\Phi^n)$。因此采用任何常数进制展开，fib(n)均由Θ(n)个数位组成。这就意味着，即便已知fib(n)的数值大小，将该数值逐位打印出来也至少需要Θ(n)时间。

b） 试参考教材 21 页代码 1.8 中 power2()算法设计一个算法，在 $\mathcal{O}(\log n)$时间内计算出 fib(n)；

【解答】

根据Fibonacci数的定义，可得如下矩阵形式的递推关系：

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} fib(k-1) \\ fib(k) \end{pmatrix} = \begin{pmatrix} fib(k) \\ fib(k+1) \end{pmatrix}$$

由此，可进一步得到如下通项公式：

$$\begin{pmatrix} fib(n) \\ fib(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} fib(0) \\ fib(1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

于是，若套用power2()算法的流程，只要将其中的整数平方运算sqr()换成矩阵的平方运算，即可实现fib(n)的计算。更重要的是，这里仅涉及2×2矩阵的计算，每次同样只需常数时间，故整体的运行时间也是$\mathcal{O}(\log n)$。

c） 以上结论是否矛盾？为什么？

【解答】

以上结论在表面上的确构成悖论。究其根源在于，以上对power2()与fib()等算法的时间复杂度分析都假定，整数的乘法、位移和打印等基本操作各自只需$\mathcal{O}(1)$时间——即采用所谓的常数代价准则（uniform cost criterion）——而这只是在一定程度上的近似。

设参与运算的整数（的数值）为k。不难看出，上述基本操作都需要逐个地读取k的二进制展开的每一有效比特位，故更为精确地，这些操作的时间成本应该线性正比于k的有效位的总数$\mathcal{O}(\log k)$——即采用所谓的对数代价准则（logarithmic cost criterion）。

当k不是很大时，两种准则之间的差异并不是不大；而当k很大甚至远远超出机器字长之后，二者之间的差异将不容忽略。仍以Fibonacci数为例。因$fib(n) = \Theta(\Phi^n)$，故该数列应以Φ为比率呈指数递增，各项的二进制展开长度$\log_2(\Phi^n)$则以匀速呈线性递增。根据习题[1-19]所给的估算经验，相邻项约相差$\log_2\Phi = 0.694$个比特，大致每隔36项相差25个比特。也就是说，自fib(48)后便会导致32位无符号整数的溢出，自fib(94)后便会导致64位无符号整数的溢出。

[1-21] 考查fib()算法的二分递归版、线性递归版和迭代版。

a） 分别编译这些算法，针对n = 64实际运行并测试对比；

b） 三者的运行速度有何差别？为什么？

【解答】

请读者通过实际动手，独立完成实测和对比任务，并根据测试结果给出结论。

需要特别留意的是，根据上述分析，中途很快即会发生字长溢出的现象。另外，若采用时间复杂度为$\mathcal{O}(\Phi^n)$的二分递归版，在计算出fib(64)之前我们大致需要等待：

$$\Phi^{64} / 10^9 = \Phi^{67-3} / 10^9 = (10^{14}/4) / 10^9$$
$$= 10^5 / 4 \text{ 秒} = 1/4 \text{ 天} = 6 \text{ 小时}$$

而多少令我们惊讶的是，若计算的目标是fib(92)，则需要等待

$$\Phi^{92} / 10^9 = \Phi^{67+25} / 10^9 = 10^{14+5} / 10^9$$
$$= 10^{10} \text{ 秒} = 3 \text{ 世纪}$$

但愿我们都能如此长寿！不过，即便不考虑电脑老化或故障等因素，在经过一个世纪之后，或许你倒是可以甚至应该更换一台更快速的新型电脑——当然，那台“电脑”未必仍然需要电能驱动，届时我们也将不再会因计算的中途停电而懊恼沮丧了。

[1-22] 参照教材 26 页代码 1.14 中迭代版 fibI()算法，实现支持如下接口的 Fib 类。

```
class Fib { //Fibonacci数列类
public:
   Fib(int n); //初始化为不小于n的最小Fibonacci项（如，Fib(6) = 8），O(logΦ(n))时间
   int get(); //获取当前Fibonacci项（如，若当前为8，则返回8），O(1)时间
   int next(); //转至下一Fibonacci项（如，若当前为8，则转至13），O(1)时间
   int prev(); //转至上一Fibonacci项（如，若当前为8，则转至5），O(1)时间
};
```

【解答】

Fib类一种可行的实现方式，如代码x1.12所示。

```
class Fib { //Fibonacci数列类
private:
   int f, g; //f = fib(k - 1), g = fib(k)。均为int型，很快就会数值溢出
public:
   Fib ( int n ) //初始化为不小于n的最小Fibonacci项
   { f = 1; g = 0; while ( g < n ) next(); } //fib(-1), fib(0)，O(log_phi(n))时间
   int get()  { return g; } //获取当前Fibonacci项，O(1)时间
   int next() { g += f; f = g - f; return g; } //转至下一Fibonacci项，O(1)时间
   int prev() { f = g - f; g -= f; return g; } //转至上一Fibonacci项，O(1)时间
};
```

代码x1.12 Fib类的实现

套用教材1.4.5节介绍的动态规划（dynamic programming）策略，这里也设置了两个内部的私有成员变量f和g，始终分别记录当前的一对相邻Fibonacci数，在构造函数中，它们分别被初始化为：

$$f = fib(-1) = 1$$

$$g = fib(0) = 0$$

为定位至不小于n的最小项，以下反复调用next()接口依次递增地遍历，直至首次达到或者超过n。因n与Fibonacci数的第$\log_\Phi(n)$项渐进地同阶，故整个遍历过程不超过$\mathcal{O}(\log_\Phi(n))$步。

next()接口对f和g的更新方式，与教材26页代码1.14完全一致：经过滚动式的叠加，使之继续指向下一对相邻的Fibonacci项。

取前一项的方法亦与此相似，只不过方向颠倒而已。

[1-23] 法国数学家 Edouard Lucas 于 1883 提出的 Hanoi 塔问题，可形象地描述如下：

有 n 个中心带孔的圆盘贯穿在直立于地面的一根柱子上，各圆盘的半径自底而上不断缩小；需要利用另一根柱子将它们转运至第三根柱子，但在整个转运的过程中，游离于这些柱子之外的圆盘不得超过一个，且每根柱子上的圆盘半径都须保持上小下大。

试将上述转运过程描述为递归形式，并进而实现一个递归算法。

【解答】

将三根柱子分别记作X、Y和Z，则整个转运过程可递归描述为：

```
为将X上的n只盘子借助Y转运至Z，只需（递归地）
    将X上的n - 1只盘子借助Z转运至Y
    再将X上最后一只盘子直接转移到Z
    最后再将Y上的n - 1只盘子借助X转运至Z
```

按照这一理解，即可如代码x1.13所示实现对应的递归算法。

```
1 // 按照Hanoi规则，将柱子Sx上的n只盘子，借助柱子Sy中转，移到柱子Sz上
2 void hanoi ( int n, Stack<Disk>& Sx, Stack<Disk>& Sy, Stack<Disk>& Sz ) {
3    if ( n > 0 )   { //没有盘子剩余时，不再递归
4       hanoi ( n - 1, Sx, Sz, Sy ); //递归：将Sx上的n - 1只盘子，借助Sz中转，移到Sy上
5       move ( Sx, Sz ); //直接：将Sx上最后一只盘子，移到Sz上
6       hanoi ( n - 1, Sy, Sx, Sz ); //递归：将Sy上的n - 1只盘子，借助Sx中转，移到Sz上
7    }
8 }
```

代码x1.13 Hanoi塔算法

关于时间复杂度，该算法对应的边界条件和递推式为：

$$T(1) = \mathcal{O}(1)$$

$$T(n) = 2*T(n - 1) + \mathcal{O}(1)$$

若令：

$$S(n) = T(n) + \mathcal{O}(1)$$

则有：

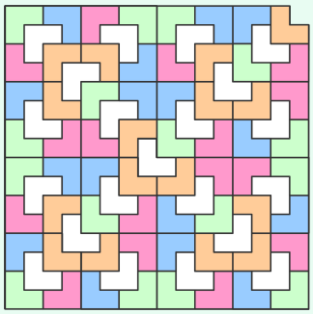
$$S(1) = \mathcal{O}(2)$$

$$\begin{aligned} S(n) &= 2 \cdot S(n - 1) \\ &= 2^2 \cdot S(n - 2) \\ &= 2^3 \cdot S(n - 3) \\ &= \dots \\ &= 2^{n-1} \cdot S(1) \\ &= 2^n \end{aligned}$$

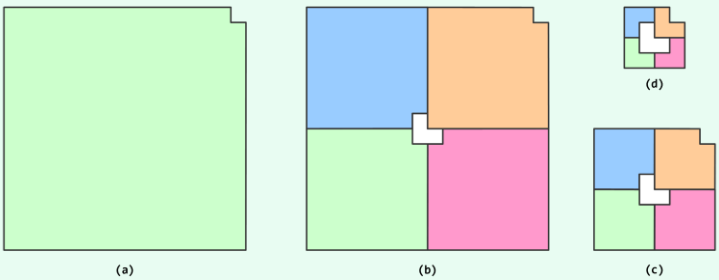
故有：

$$T(n) = \mathcal{O}(2^n)$$

[1-24] 如图 x1.4 所示，考查缺失右上角（面积为 $4^n - 1$）的 $2^n \times 2^n$ 棋盘，$n \geq 1$。



图x1.4 使用85块L形积木，可以恰好覆盖缺失一角的16×16棋盘



图x1.5 采用分治策略，将大棋盘的覆盖问题转化为四个小棋盘的覆盖问题

a） 试证明，使用由三个 1×1 正方形构成、面积为 3 的 L 形积木，可以恰好覆盖此类棋盘；

【解答】

将n称作此类棋盘的阶次，并对n做数学归纳。作为归纳基，n = 1时显然。

故假设阶次低于n的此类棋盘都可被L形积木覆盖，考查如图x1.5(a)所示的n阶棋盘。先将一块L形积木摆放至棋盘中心处，缺口方向与棋盘缺口一致。

于是如图(b)所示，棋盘的剩余部分可以划分为四个n - 1阶的棋盘。由归纳假设，它们均可被L形积木覆盖，故原n阶棋盘亦是如此。

b） 试给出一个算法，对于任意 n ≥ 1，给出覆盖方案；

【解答】

参照以上归纳证明的思路，即可得出构造覆盖方案的如下递归算法。

```
1  // 覆盖基准点在(x, y)的n ≥ 1阶棋盘
2  // 四种缺口方向，由(dx, dy)指定：(+1, +1)东北、(+1, -1)东南、(-1, +1)西北、(-1, -1)西南
3  // 算法的起始调用入口为cover(n, 0, 0, 1, 1)：基准点在(0, 0)、缺口朝向东北的n阶棋盘
4  void cover(int n, int x, int y, int dx, int dy) {
5     int s = 1 << (n-1); //子棋盘的边长：2^(n-1)
6     place(x + dx * (s - 1), y + dy * (s - 1), dx, dy); //首先用一块L形积木覆盖中心
7     if (1 < n) { //只要棋盘仍未完全覆盖，则继续递归地覆盖四个子棋盘
8        cover(n - 1, x,                 y,                 dx,  dy); //递归：覆盖西南方子棋盘
9        cover(n - 1, x + dx * s,        y + dy * s,        dx,  dy); //递归：覆盖东北方子棋盘
10       cover(n - 1, x + dx * (2*s - 1), y,               -dx,  dy); //递归：覆盖东南方子棋盘
11       cover(n - 1, x,                 y + dy * (2*s - 1), dx, -dy); //递归：覆盖西北方子棋盘
12    }
13 }
```

算法x1.2 缺角棋盘的覆盖算法

有兴趣的读者，不妨尝试实现与该递归算法对应的迭代版本。

实际上，该问题还可进一步推广：即便缺失的正方形不是位于角部，亦存在覆盖的方案。有兴趣的读者可自行给出证明，并参照以上算法设计出相应的算法。

c） 该算法的时间复杂度是多少？

【解答】

以上算法的每一递归实例本身只需常数时间，根据算法流程可得如下递推式：

$T(1) = \mathcal{O}(1)$

$T(n) = 4*T(n - 1) + \mathcal{O}(1)$

两式联合求解，即得：

$T(n) = \mathcal{O}(4^n)$

若以棋盘的边长N = 2^n（而非棋盘的阶次n）为依据，则有

$T(N) = \mathcal{O}(N^2)$

若以棋盘的面积M = 4^n - 1为依据，则有

$T(M) = \mathcal{O}(M)$

也可从另一角度简便而精准地估计出本算法的时间复杂度。为此，需要注意到以下事实：

每个递归实例，都对应于覆盖方案中的某一块L形积木，反之亦然

这就意味着，递归实例的总数恰好等于所用L形积木的总数，后者恰为棋盘面积的三分之一，渐进地即是$\mathcal{O}(M)$。

[1-25]《九章算术》记载的“中华更相减损术”可快速地计算正整数 a 和 b 的最大公约数，其过程如下：

```
1  令p = 1
2  若a和b不都是偶数，则转5)
3  令p = p×2，a = a/2，b = b/2
4  转2)
5  令t = |a - b|
6  若t = 0，则返回并输出a×p
7  若t为奇数，则转10)
8  令t = t/2
9  转7)
10 若a ≥ b，则令a = t；否则，令b = t
11 转5)
```

a） 按照上述流程，编写一个算法 int gcd(int a, int b)，计算 a 和 b 的最大公约数；

【解答】

运用“中华更相减损术”的最大公约数算法，可实现如代码 x1.14 所示。

```
1 __int64 gcdCN ( __int64 a, __int64 b ) { //assert: 0 < min(a, b)
2    int r = 0; //a和b的2^r形式的公因子
3    while ( ! ( ( a & 1 ) || ( b & 1 ) ) ) { //若a和b都是偶数
4       a >>= 1; b >>= 1; r ++; //则同时除2（右移），并累加至r
5    } //以下，a和b至多其一为偶
```


```
6    while ( 1 ) {
7       while ( ! ( a & 1 ) ) a >>= 1; //若a偶（b奇），则剔除a的所有因子2
8       while ( ! ( b & 1 ) ) b >>= 1; //若b偶（a奇），则剔除b的所有因子2
9       ( a > b ) ? a = a - b : b = b - a; //简化为：gcd(max(a, b) - min(a, b), min(a, b))
10      if ( 0 == a ) return b << r; //简化至平凡情况：gcd(0, b) = b
11      if ( 0 == b ) return a << r; //简化至平凡情况：gcd(a, 0) = a
12   }
13 }
```

代码x1.14 运用“中华更相减损术”的最大公约数算法

b） 与功能相同的欧几里得算法相比，这一算法有何优势？

【解答】

不妨将两个算法分别简称作“中”和“欧”。

首先可以证明，算法“中”的渐进时间复杂度依然是$O(\log(a + b))$。

考查该算法的每一步迭代，紧接于两个内部while循环之后设置一个断点，观察此时的a和b。实际上，在a和b各自剔除了所有因子2之后，此时它们都将是奇数。接下来，无论二者大小如何，再经一次互减运算，它们必然将成为一奇一偶。比如，不失一般性地设a > b，则得到：

a - b （偶）

b （奇）

再经一步迭代并重新回到断点时，前者至多是：

(a - b)/2

两个变量之和至多是：

$(a - b)/2 + b \le (a + b)/2$

可见，每经过一步迭代，a + b至少减少一半，故总体迭代步数不超过：

$\log_2(a + b)$

另外，尽管从计算流程来看，算法“中”的步骤似乎比算法“欧”更多，但前者仅涉及加减、位测试和移位（除2）运算，而不必做更复杂的乘除运算。因此，前者更适于在现代计算机上编程实现，而且实际的计算效率更高。

反之，无论是图灵机模型还是RAM模型[②]，除法运算在底层都是通过减法实现的。因此，对于算法“欧”所谓的“除法加速”效果，不可过于乐观——而在输入整数大小悬殊时，尤其如此。

最后，较之算法“欧”，算法“中”更易于推广至多个整数的情况。

② 关于这两种典型的计算模型的定义、性质及其关系，建议读者阅读文献[4]的第一章。

[1-26] 试设计并实现一个就地的算法 shift(int A[], int n, int k)，在 O(n)时间内将数组 A[0, n)中的元素整体循环左移 k 位。例如，数组 A[] = { 1, 2, 3, 4, 5, 6 }经 shift(A, 6, 2)之后，有 A[] = { 3, 4, 5, 6, 1, 2 }。（提示：利用教材 20 页代码 1.7 中 reverse()算法）

【解答】

一个可行的算法，如代码x1.15所示。

```
1 int shift2 ( int* A, int n, int k ) { //借助倒置算法，将数组循环左移k位，O(3n)
2    k %= n; //确保k <= n
3    reverse ( A, k ); //将区间A[0, k)倒置：O(3k/2)次操作
4    reverse ( A + k, n - k ); //将区间A[k, n)倒置：O(3(n - k)/2)次操作
5    reverse ( A, n ); //倒置整个数组A[0, n)：O(3n/2)次操作
6    return 3 * n; //返回累计操作次数，以便与其它算法比较：3/2 * (k + (n - k) + n) = 3n
7 }
```

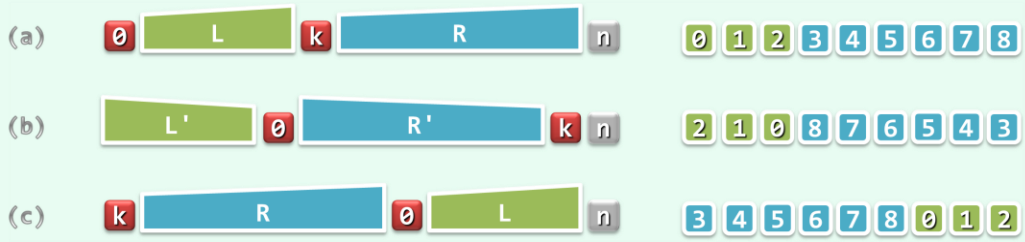
代码x1.15 借助reverse()算法在O(n)时间内就地移位

若在原向量V中前k个元素组成的前缀为L，剩余的（后缀）部分为R，则如图x1.6所示，经整体左移之后的向量应为：

$$R + L$$

这里约定，任意向量V整体倒置后的结果记作V'。于是该算法的原理来自如下恒等式：

$$R + L = (L' + R')'$$



图x1.6 借助reverse()算法在O(n)时间内就地移位的过程及原理

该算法的运行时间主要消耗于元素的互换操作。在整个三轮兑换中，每个元素至多参与两次互换操作。通常，每一对元素的互换需要3次移动操作，因此移动操作的累计次数不超过：

$$n \times 2 \times 3 / 2 = 3n$$

该算法的其它版本有可能只需更少的交换操作，故单就此指标而言，似乎更加“优于”以上版本。然而就实际的计算效率而言，以上版本却要远远优于其它版本。

究其原因在于，reverse()之类的操作所涉及的数据元素，在物理上是连续分布的，因此操作系统的缓存机制可以轻易地被激活，并充分发挥作用；其它版本的交换操作尽管可能更少，但数据元素在空间往往相距很远，甚至随机分布，缓存机制将几乎甚至完全失效。

在实际的算法设计与编程中，这些方面也是首先必须考虑的因素；在当下，面对规模日益膨胀的大数据，这方面的技巧对算法的实际性能更是举足轻重。在教材的8.2等节，我们还将结合B-树等数据结构，就此深入讨论。

[1-27] 试实现一个递归算法，对任意非负整数 m 和 n，计算以下 Ackermann 函数值：

$$\text{Ackermann}(m, n) = \begin{cases} n + 1 & (\text{若} m = 0) \\ \text{Ackermann}(m - 1, 1) & (\text{若} m > 0 \text{且} n = 0) \\ \text{Ackermann}(m - 1, \text{Ackermann}(m, n - 1)) & (\text{若} m > 0 \text{且} n > 0) \end{cases}$$

对于每一(m, n)组合，这个算法是否必然终止？

【解答】

以上定义本身就是递归式的，故不难将其转换为一个递归算法。请读者独立完成这一任务。

在可计算性理论中，Ackermann函数是典型的非原始递归的递归函数。尽管其定义和计算过程较为复杂，依然可以证明其计算过程必然终止，故对任何(m, n)参数组合均有明确的定义。以下，可以采用超限数学归纳法（transfinite induction）来证明上述论断。

为此，我们首先需要在所有非负整数的组合(m, n)之间，定义如下次序：

> 对于任何(m_1, n_1)与(m_2, n_2)，若$m_1 < m_2$，或者$m_1 = m_2$且$n_1 < n_2$，则称前者小于后者，记作$(m_1, n_1) < (m_2, n_2)$

实际上，所有的(m, n)组合与平面上第一象限内的整点一一对应。不难看出，任何两个整点都可按照这一定义比较大小，故这是一个全序。更重要地，该整点集的任何一个子集，都有最小元素——即该子集中的最左最低点（leftmost-then-lowest point）。其中特别地，全集的最小元素即为坐标原点(0, 0)。因此，如上定义的次序“<”，的确是一个良序（well order）。

由定义，任意形如(0, n)的输入都会立即终止——这可作为归纳基础。

作为归纳假设，不妨假定：对于任意小于(m, n)的输入，Ackermann函数均能终止。现考查输入参数为(m, n)时，该函数的可终止性。

依然由定义可见，此时可能引发的递归实例无非三类：

```
Ackermann(m - 1, 1)
Ackermann(m - 1, *)
Ackermann(m, n - 1)
```

可见，根据如上约定的次序，其对应的参数组合均小于(m, n)。故由归纳假设，以此参数组合对该函数的调用，亦必然会终止。

[1-28] 考查所谓咖啡罐游戏（Coffee Can Game）：在咖啡罐中放有 n 颗黑豆与 m 颗白豆，每次取出两颗：若同色，则扔掉它们，然后放入一颗黑豆；若异色，则扔掉黑豆，放回白豆。

a） 试证明，该游戏必然终止（当罐中仅剩一颗豆子时）；

【解答】

尽管游戏的每一步都有（同色或异色）两个分支，但不难验证：无论如何，每经过一次迭代，罐中豆子的总数（n + m）必然减一。因此就总体而言，罐中豆子的数目必然不断地单调递减，直至最终因不足两颗而终止。

b） 对于哪些(n，m)的组合，最后剩下的必是白豆？

【解答】

类似地，尽管这里有两个分支，但无论如何迭代，罐中白色豆子总数（m）的奇偶性始终保持不变。因此若最终仅剩一颗白豆，则意味着白色豆子始终都是奇数颗。反之，只要初始时白豆共计奇数颗，则最终剩余的也必然是一颗白豆。

[1-29] 序列Hailstone(n)是从n开始，按照以下规则依次生成的一组自然数：

$$\text{Hailstone}(n) = \begin{cases} \{1\} & (\text{若}n = 1) \\ \{n\} \cup \text{Hailstone}(n/2) & (\text{若}n\text{为偶数}) \\ \{n\} \cup \text{Hailstone}(3n+1) & (\text{若}n\text{为奇数}) \end{cases}$$

比如：

Hailstone(7) = { 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 }

试编写一个非递归程序[③]，计算Hailstone(n)的长度hailstone(n)。

【解答】

由题中所给的定义，可以直接导出该算法的递归实现。

该算法的一种可行的非递归实现方式，则如代码x1.16所示。

```
1  template <typename T> struct Hailstone { //函数对象：按照Hailstone规则转化一个T类对象
2     virtual void operator() ( T& e ) { //假设T可直接做算术运算
3        int step = 0; //转换所需步数
4        while ( 1 != e ) { //按奇、偶逐步转换，直至为1
5           ( e % 2 ) ? e = 3 * e + 1 : e /= 2;
6           step++;
7        }
8        e = step; //返回转换所经步数
9     }
10 };
```

代码x1.16 计算Hailstone(n)序列长度的“算法”

正如教材中已经指出的，“序列Hailstone(n)长度必然有限”的结论至今尚未得到证明，故以上程序未必总能终止，因而仍不能称作是一个真正的算法。

[1-30] 在分析并界定其渐进复杂度时，迭代式算法往往体现为级数求和的形式，递归式算法则更多地体现为递推方程的形式。针对这两类主要的分析技巧，参考文献[7]做了精辟的讲解和归纳。试研读其中的相关章节。

【解答】

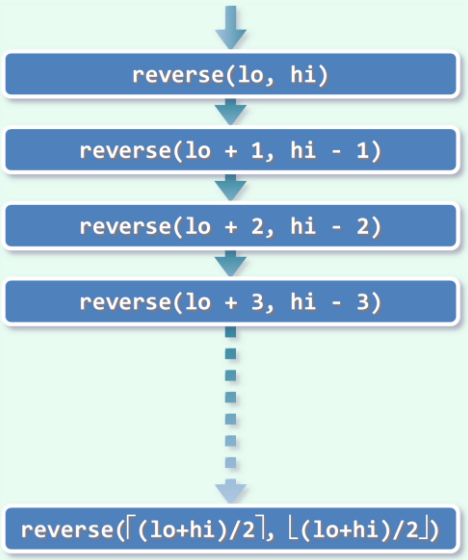
请读者独立完成研读任务，并根据自己的理解进行归纳总结。

[③] 据作者所知，“序列Hailstone(n)长度必然有限”的结论尚未得到证明，故你编写的程序可能并非一个真正的算法

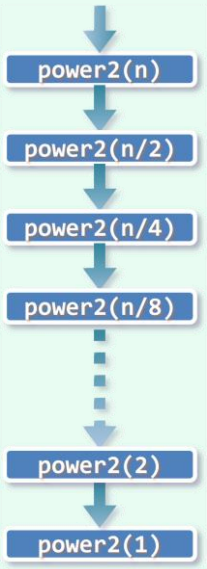
[1-31] 试针对教材 20 页代码 1.7 中的 reverse()算法和 21 页代码 1.8 中的 power2()算法，运用递归跟踪法分析其时间复杂度。

【解答】

reverse()算法的递归跟踪过程，如图x1.7所示。



图x1.7 reverse()算法的递归跟踪



图x1.8 power2()算法的递归跟踪

从中可以清楚地看出，每递归深入一层，入口参数lo和hi之间的差距必然缩小2，因此递归深度（亦即时间复杂度）为：

$$(hi - lo) / 2 = n/2 = O(n)$$

power2()算法的递归跟踪过程，如图x1.8所示。

类似地也可看出，每递归深入一层，入口参数n即缩小一半，因此递归深度（亦即时间复杂度）应为：

$$\log_2 n = O(\log n)$$

若按教材中的定义，将参数n所对应二进制展开的宽度记作r = logn，则由图x1.8看出，每递归深入一层，r都会减一，因此递归深度（亦即时间复杂度）应为：

$$O(r) = O(\log n)$$

这与上述分析殊途同归。

[1-32] 若假定机器字长无限，移位操作只需单位时间，递归不会溢出，且 rand()为理想的随机数发生器。试分析以下函数 F(n)，并以大 O 记号的形式确定其渐进复杂度的紧上界。

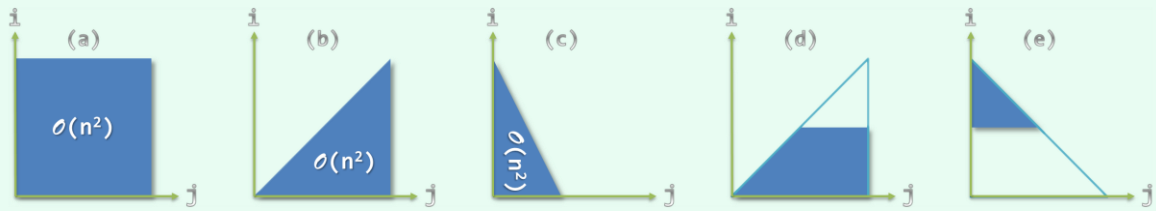
```
01) void F(int n) {
       for (int i = 0; i < n; i ++)
       for (int j = 0; j < n; j ++);
    }
```

【解答】

这是最基本的二重循环模式，其特点是循环控制变量均按算术级数变化。内循环的每次迭代只需$O(1)$时间，故执行时间取决于总体的迭代次数。因外、内循环的范围分别是$i \in [0, n)$和$j \in [0, n)$，故累计迭代次数为：

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}1=\sum_{i=0}^{n-1}n=n\times n=O(n^2)$$

实际上在如图x1.9(a)所示的(i, j)坐标平面上，此二重循环的执行过程对应于匀速地自下而上、由左到右逐行扫过矩形$[0, n) \times [0, n)$。因此，这一过程所需的时间也就对应于该矩形的面积，就渐进意义而言即是$O(n^2)$。



图x1.9 二重循环执行时间的对应图形

相对于以上严格的计算，基于这一图形式理解的估算不仅更加简捷，而且可以更好地体现通过大O记号刻画渐进复杂度总体趋势的要义。而对于接下来更为复杂的情况，这种方法的上述优势则尤为突出。

```
02) void F(int n) {
       for (int i = 0; i < n; i ++)
       for (int j = i; j < n; j ++);
    }
```

【解答】

这也是一种典型的二重循环模式，循环控制变量仍然按算术级数变化。与上例相比，外循环的范围依然是$i \in [0, n)$，但内循环的范围则变成$j \in [i, n)$，故累计迭代次数为：

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}1=\sum_{i=0}^{n-1}(n-i)=\sum_{(k=n-i)=1}^{n}k=n(n+1)/2=O(n^2)$$

也可套用以上基于图形的估算方法，在如图x1.9(b)所示的(i, j)坐标平面上，此二重循环的执行过程对应于匀速地自下而上、由左到右逐行扫过三角形$[0, n) \times [i, n)$。该三角形的面积大致为原矩形的一半——就渐进意义而言，时间复杂度依然是$O(n^2)$。

```
03) void F(int n) {
       for (int i = 0; i < n; i ++)
       for (int j = 0; j < i; j += 2013);
    }
```

【解答】

这里，外循环的范围依然是i ∈ [0, n)，但内循环控制变量j递增的步长改为2013，对应的范围变成j*2013 ∈ [0, i)。

套用以上基于图形的估算方法，在如图x1.9(c)所示的(i, j)坐标平面上，此二重循环的执行过程对应于匀速地自下而上、由左到右逐行扫过一个三角形。该三角形的高仍保持为n，底边压缩了2013倍，故其面积大致为原矩形的1/2/2013 = 1/4026倍——就渐进意义而言，时间复杂度依然是$O(n^2)$。

```
04) void F(int n) {
       for (int i = 0; i < n/2; i ++)
       for (int j = i; j < n; j ++);
    }
```

【解答】

这里，外循环和内循环的范围分别改为i ∈ [0, n/2)和j ∈ [i, n)。

依然沿用以上基于图形的估算方法，在如图x1.9(d)所示的(i, j)坐标平面上，此二重循环的执行过程对应于匀速地自下而上、由左到右逐行扫过一个梯形。该梯形的上底和下底分别长为n/2和n，高度为n/2，故其面积大致为原矩形的3/8倍——就渐进意义而言，其 时间复杂度依然是$O(n^2)$。

```
05) void F(int n) {
       for (int i = n/2; i < n; i ++)
       for (int j = 0; j < n - i; j ++);
    }
```

【解答】

这里，外循环的范围改为i ∈ [n/2, n)，内循环的范围是j ∈ [0, i)。

依然沿用以上基于图形的估算方法，在如图x1.9(e)所示的(i, j)坐标平面上，此二重循环的执行过程对应于匀速地自下而上、由左到右逐行扫过一个小三角形。该三角形的底和高均为n/2，故其面积大致为原矩形的1/8倍——就渐进意义而言，其时间复杂度依然是$O(n^2)$。


```
06) void F(int n) {
       for (int i = 0; i < n; i ++)
       for (int j = 1; j < n; j <<= 1);
    }
```

【解答】

请注意，在此二重循环模式中，尽管外循环的控制变量i仍在[0, n)内按算术级数变化，但内循环的控制变量j在[1, n)内却是按（以2为倍数的）几何级数变化，故累计迭代次数为：

$$\sum_{i=0}^{n-1}\sum_{(k=\log j)=0}^{\log n-1}1=\sum_{i=0}^{n-1}(\log n-1)=n\log n-n=O(n\log n)$$

这里所采用的左移操作，不仅因为可以便捷地实现控制变量j的倍增，同时更为重要的是，这也为复杂度的快速估算提供了线索和依据。我们知道，任意正整数的二进制展开的宽度，与其数值呈对数关系。具体地，数值的加倍对应于其展开宽度加一，反之亦然。

从这一角度考查此处的内循环可见，随着j以2为倍数不断递增，j的二进制展开宽度将以1为步长不断递增。每一轮内循环的迭代次数既然等于j从1至n的倍增次数，也应该就是n的二进制展开宽度，即log n。因此，n轮内循环共计耗时$O(n\log n)$。

```
07) void F(int n) {
       for (int i = 0; i < n; i ++)
       for (int j = 1; j < 2013; j <<= 1);
    }
```

【解答】

此处内循环的控制变量j尽管也是按几何级数递增，但其变化范围固定在[1, 2013)内。因此每一轮内循环只做常数（log 2013）次的迭代。故总体时间复杂度仅取决于外循环，为$O(n)$。

```
08) void F(int n) {
       for (int i = 1; i < n; i ++)
       for (int j = 0; j < n; j += i);
    }
```

【解答】

此处内循环的控制变量j尽管是在[0, n)内按算术级数递增，但步长并不固定。具体地，第i轮内循环采用的步长即为i，故需做n/i次迭代。于是，所有循环的累计迭代次数为：

$$\sum_{i=1}^{n-1}\sum_{(k=\frac{j}{i})=0}^{\frac{n}{i}-1}1=\sum_{i=1}^{n-1}\frac{n}{i}=n\sum_{i=1}^{n-1}\frac{1}{i}=O(n\log n)$$

这里需要借助关于调和级数的以下性质：

$$\sum_{i=1}^{n}\frac{1}{i}=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}=\ln n+\gamma+\Theta(\frac{1}{2n})$$

其中，$\gamma\approx 0.577216$为欧拉常数。

```
09) void F(int n) { for (int i = 0, j = 0; i < n; i += j, j ++); }
```

【解答】

这里的变量i和j均从0开始不断递增，每经过一步迭代，i递增j，j递增1。

表x1.1 函数F(n)中变量i和j随迭代不断递增的过程

迭代次序t	0	1	2	3	4	...
变量i	0	0 + 0	0 + 0 + 1	0 + 0 + 1 + 2	0 + 0 + 1 + 2 + 3	...
变量j	0	1	2	3	4	...

具体地，这一过程可以归纳如表x1.1所示。故经过k次迭代后，必有：

$$i = \sum_{t=0}^{k-1} t = k(k-1)/2$$

在循环退出之前，必有：

$$i = k(k-1)/2 < n$$，或等价地，$$k < \frac{1+\sqrt{1+8n}}{2}$$

故该函数的时间复杂度为$O(\sqrt{n})$。

```
10) void F(int n) { for (int i = 1, r = 1; i < n; i <<= r, r <<= 1); }
```

【解答】

这里的变量i和r均从1开始不断递增，每经过一步迭代，i递增为i·2^r，r递增为2·r。

表x1.2 函数F(n)中变量i和r随迭代不断递增的过程

迭代次序t	0	1	2	3	4	...
变量i	2^0	2^(0+1)	2^(0+1+2)	2^(0+1+2+4)	2^(0+1+2+4+8)	...
变量r	2^0 = 1	2^1 = 2	2^2 = 4	2^3 = 8	2^4 = 16	...

具体地，这一过程可以归纳如表x1.2所示。故经过k次迭代后，必有

$$i = \prod_{t=0}^{k-1} 2^{(2^t)} = 2^{\sum_{t=0}^{k-1} 2^t} = 2^{(2^k-1)}$$

在循环退出之前，必有

$$i = 2^{(2^k-1)} < n$$

亦即

$$2^k - 1 < \log n$$

$$2^k \le \log n$$

$$k \le \log\log n$$

故该函数的时间复杂度为$\mathcal{O}(\log\log n)$。

同样地，这里通过左移操作实现变量递增的方式，也为我们快捷地估算时间复杂度提供了新的视角和线索。从二进制展开的角度来看，变量r的展开宽度每次增加一位，而变量i则每次增加r位。也就是说，变量i的宽度将以（大致）加倍的指数速度膨胀，直至刚好超过logn。因此，总体的迭代次数应不超过logn的对数，亦即$\mathcal{O}(\log\log n)$。

```
11) void F(int n) { for (int i = 1; i < n; i = 1 << i); }
```

【解答】

每经一次迭代，i即增长至2^i。设经过k次迭代之后，因i ≥ n而退出迭代。

现颠倒原迭代的方向，其过程应等效于反复令n = $\log_2 n$，并经k次迭代之后有n ≤ 1。由此可知，若对n反复取对数直至其不大于1，则k等于其间所做对数运算的次数，记作k = $\log^* n$，读作“log-星-n”。

我们知道，指数函数增长的速度本来就很快，而按照i = 2^i规律增长的速度更是极其地快。因此不难理解，作为反函数的T(n) = $\mathcal{O}(\log^* n)$尽管依然是递增的，但增长的速度应极其地慢。

另一方面，既然此前习题[1-12]介绍的$\mathcal{O}(\log\log n)$通常可以视作常数，则$\mathcal{O}(\log^* n)$更应该可以。

不妨仍以人类目前所能感知的宇宙范围内，所有基本粒子的总数N = 10^81 = 2^270为例，不难验证有：

$$\log^* N < 5$$

```
12) int F(int n) { return (n > 0) ? G(G(n - 1)) : 0; }
    int G(int n) { return (n > 0) ? G(n - 1) + 2*n - 1 : 0; }
```

【解答】

首先，需要分析这两个函数的功能语义。不难验证，G(n) = n^2实现了整数的平方运算功能；相应地，F(n) = $((n - 1)^2)^2 = (n - 1)^4$。

接下来为分析时间复杂度，这里及以下将F(n)和G(n)的时间复杂度分别记作f(n)和g(n)。

G(n)属于线性递归（linear recursion），其原理及计算过程实质上可以表示为：

$$n^2 = (2n - 1) + (2n - 3) + \ldots + 5 + 3 + 1$$

也就是说，经递归n层计算前n个奇数的总和。因此，其运行时间为g(n) = $\mathcal{O}(n)$。

请注意，这里的F()并非递归函数，其本身只消耗$\mathcal{O}(1)$时间。不过，F(n)会启动G()的两次递归，入口参数分别为n - 1和$(n - 1)^2$。故综合而言，总体运行时间应为：

$$\begin{aligned} f(n) &= \mathcal{O}(1) + g(n - 1) + g((n - 1)^2) \\ &= \mathcal{O}(1) + \mathcal{O}(n - 1) + \mathcal{O}((n - 1)^2) \\ &= \mathcal{O}(n^2) \end{aligned}$$

需要强调的是，既然G($(n - 1)^2$)的递归深度为$(n - 1)^2$，故在实际运行时此类代码比较容易因递归过深而导致存储空间的溢出。

```
13) void F(int n) { for (int i = 1; i < n/G(i, 0); i ++); }
    int G(int n, int k) { return (n < 1) ? k : G(n - 2*k - 1, k + 1); }
```

【解答】

同样地，首先需要分析这两个函数的功能语义。不难验证，G(n, 0) = $\lceil\sqrt{n}\rceil$实现了整数的开方运算功能；相应地，F(n)只不过是以1为步长，令变量i从1递增到n/$\lceil\sqrt{n}\rceil$。

G(n)属于线性递归（linear recursion），其原理实质上与前一题相同，只不过计算过程相反——从1开始，依次从n中扣除各个奇数，直至n不再是正数。因此与前一题同理，共需递归$\lceil\sqrt{n}\rceil$层，其运行时间亦为：

$$g(n) = \mathcal{O}(\lceil\sqrt{n}\rceil)$$

这里的F()本身只是一个基本的迭代，递增的控制变量i初始值为1。在迭代终止时，应有：

$$i \geq n/\lceil\sqrt{i}\rceil$$

亦即：

$$i = \Theta(n^{2/3})$$

需要特别留意的是，函数F()中的循环每做一步迭代，都需要调用一次G(i, 0)以核对终止条件。故综合而言，这部分时间累计应为：

$$f(n) = \mathcal{O}(\sqrt{1}) + \mathcal{O}(\sqrt{2}) + \mathcal{O}(\sqrt{3}) + \ldots + \mathcal{O}(\sqrt{n^{2/3}})$$

$$= \mathcal{O}(\int_0^{n^{2/3}} \sqrt{x})$$

$$= \mathcal{O}(n)$$

即便计入F()自身所需的$\mathcal{O}(n^{2/3})$时间，已不足以影响这一结论。

在以上分析的基础上稍加体会即不难理解，对于函数F()而言，循环的终止条件实际上完全取决于输入参数n——迭代过程等效于变量i从1逐步递增至$n^{2/3}$。故就此问题而言，为提高算法的整体效率，应该首先直接估算出$n^{2/3}$，然后将其作为越界点。比如，可以改写函数G(n)并使之返回G(n) = $n^{1/3}$，进而得到n/G(n) = $n^{2/3}$，从而使得F()仅需调用一次G()。

实际上只要实现得法，具有以上新的操作语义的函数G(n)本身的耗时仅为$\mathcal{O}(n^{1/3})$。请读者根据以上提示，独立完成此项任务。

```
14) int F(int n) { return (n > 0) ? G(2, F(n - 1)) : 1; }
    int G(int n, int m) { return (m > 0) ? n + G(n, m - 1) : 0; }
```

【解答】

同样地，首先需要分析这两个函数的功能语义。不难验证有：

G(n, m) = n * m，实现了整数的乘法运算功能

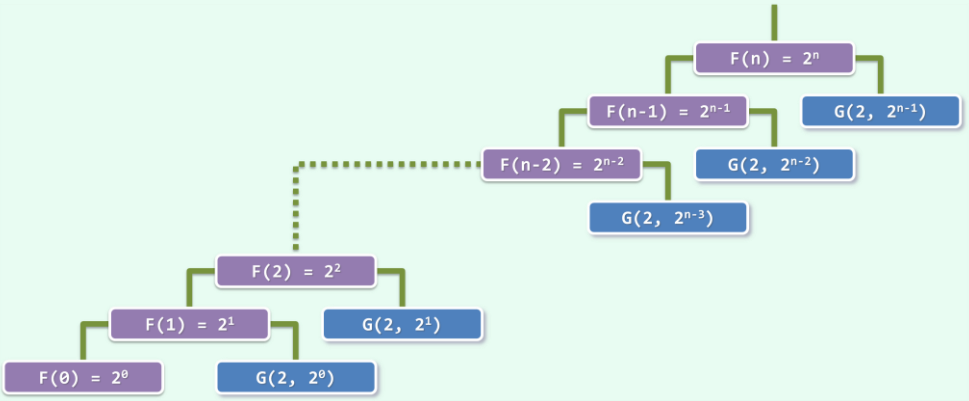
F(n) = 2^n，实现了2的整数次幂运算功能

接下来，分析这两个函数的时间复杂度。

G(n, m)的计算过程，实质上就是将n累加m次，故其运行时间为：

$$g(n, m) = \mathcal{O}(m)$$

从F(n)入口的递归跟踪过程，如图x1.10所示。



图x1.10 联合递归函数F(n)和G(n)的递归跟踪图

实质上该过程的功能就是，通过对G()的n次调用，实现对n个2的累乘。其中，函数F()共计出现过n + 1个递归实例（在图中以紫色矩形示意），各自需要$O(1)$时间。

在整个计算过程中，函数G()的递归实例可以分为n组，各组的起始实例所对应的入口参数m依次从2^{n-1}开始不断折半，直至$2^0 = 1$。这些起始实例，在图中以蓝色矩形示意；为简洁起见，由其引发的后续实例则未予标出。

根据以上分析，G()属于单分支的线性递归，递归深度取决于入口参数m。因此，各组递归实例的数目也从2^{n-1}开始反复折半，直至最终的$2^0 = 1$。根据几何级数的特性，其总和应与最高项渐进地同阶，为$O(2^n)$。

综合考虑F()和G()两类递归实例，总体时间复杂度取决于G()，亦为$O(2^n)$。

也可采用递推方程法。首先考查g()。根据以上分析，可以得出如下边界条件和递推关系：

$$g(n, 0) = O(1)$$

$$g(n, m) = g(n, m - 1) + O(1)$$

两式联合求解，即得：

$$g(n, m) = O(m)$$

至于f()，根据以上分析，也可以得出如下边界条件和递推关系：

$$f(0) = O(1)$$

$$f(n) = f(n - 1) + g(2, 2^{n-1}) = f(n - 1) + O(2^{n-1})$$

两式联合求解，即得：

$$f(n) = O(2^n)$$

同样地请注意，G()最大的递归深度为2^m。这就意味着，递归深度将随着m的扩大急剧增加，故在实际运行时，此类代码极其容易因递归过深而导致存储空间的溢出。因此，在设计和实现算法时，应尽力避免这类形式的递归。

```
15) int F(int n) { return (n > 3) ? F(n >> 1) + F(n >> 2) : n; }
```

【解答】

该F(n)函数属于典型的二分递归，以下采用递推方程法，对其做一分析。

根据该函数的定义，可以得出如下边界条件和递推关系：

$$f(0) = f(1) = f(2) = f(3) = 1$$

$$f(4) = 3$$

$$f(n) = f(n/2) + f(n/4) + 1$$

若令

$$s(m) = f(2^m)$$

则以上方程可等价地转换为：

$$s(0) = 1$$

$$s(1) = 1$$

$$s(2) = 3$$

$$s(m) = s(m - 1) + s(m - 2) + 1$$

再令

$$t(m) = (s(m) + 1)/2$$

则可进一步转换为：

$$t(0) = 1$$

$$t(1) = 1$$

$$t(2) = 2$$

$$t(m) = t(m - 1) + t(m - 2)$$

与Fibonacci数列做一对比，即可知有：

$$t(m) = fib(m + 1) = \mathcal{O}(\Phi^m)$$

其中，$\Phi = (1 + \sqrt{5})/2 = 1.618$

于是有：

$$\begin{aligned} f(n) &= 2 \cdot t(\log n) - 1 \\ &= 2 \cdot \mathcal{O}(\Phi^{\log n}) - 1 \\ &= \mathcal{O}(\Phi^{\log n}) \\ &= \mathcal{O}(n^{\log \Phi}) \\ &= \mathcal{O}(n^{0.694}) \end{aligned}$$

饶有趣味的是，尽管该函数的形式属于二分递归，但经以上分析可见，其计算过程中出现的递归实例却远不足$\mathcal{O}(n)$个，其复杂度亦远低于线性。

从分治策略的角度看，该算法模式意味着，每个规模为n的问题，均可在$\mathcal{O}(1)$时间内分解为规模分别为n/2和n/4的两个子问题。实际上其复杂度之所以仅为$\omega(n)$，关键在于两个子问题的规模总和（3n/4）已经严格地小于原问题的规模（n）。

```
16) void F(int n) {
       for (int i = n; 0 < i; i --)
          if (!(rand() % i))
             for (int j = 0; j < n; j ++);
    }
```

【解答】

这是一种典型的随机算法（randomized algorithm）模式，其中通过随机数rand()决定程序执行的去向，因此通常需要从概率期望的角度来界定其运行时间。

以下不妨基于随机均匀分布的假定条件（即rand()在整数范围内取任意值的概率均等），来分析该程序的平均运行时间。

这里的外循环共迭代n步。在每一步中，只有当随机数rand()整除外循环的控制变量i时，方可执行内循环。内循环的长度与变量i无关，共执行n步。若内循环执行，则其对总体时间复杂度的贡献即为n；否则，贡献为0。

既然假定属于均匀随机分布，故rand()能够整除变量i的概率应为1/i。这就意味着，与每个变量i相对应的内循环被执行的概率为：

```
1/i
```

反过来，内循环不予执行的概率即为：

```
(i - 1)/i
```

故就概率期望的角度而言，对应于变量i的内循环平均迭代n/i步。

于是根据期望值的线性律（linearity of expectation），整个程序执行过程中内循环迭代步数的期望值，应该等于在每一步外循环中内循环迭代步数期望值的总和，亦即：

```
f(n)  = (1/n + 1/(n - 1) + ... + 1/3 + 1/2 + 1) × n
      = expected-O(nlogn)
```

这里同样需要借助关于调和级数的以下性质：

$$\sum_{i=1}^{n}\frac{1}{i}=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}=\ln n+\gamma+\Theta(\frac{1}{2n})$$

其中，$\gamma\approx 0.577216$为欧拉常数。

即便再计入n步外循环本身所需的O(n)时间，总体的渐进复杂度亦是如此。