Fibonacci 数计算中的两个思维盲点及其扩展数列的通用高效解法

http://www.cppblog.com/flyinghearts

http://www.cnblogs.com/flyinghearts

http://blog.csdn.net/flyinghearts

㈠ Fibonacci 数

刚接触 Fibonacci 数的时候，在网上看到“矩阵法”，看到要先实现一个矩阵乘法，感觉太麻烦了。后来仔细观察 Fibonacci 数列，发现有下面的规律：

F(n) = F(k)*F(n+1-k) + F(k-1)*F(n-k) =>

F(2*n) = F(n+1) * F(n) + F(n) * F(n - 1)

F(2*n+1) = F(n+1) * F(n+1) + F(n) * F(n)

根据该公式：要计算 F(n)，只需先计算出 F(n/2)和 F(n/2+1)，于是得出一个数的 O(log n)解法。（例如：计算 F(13) => 计算 F(6)、F(7) => 计算 F(3)、F(4) => 计算 F(1)、F(2)。）再后来无意间发现，“矩阵法”根本就不必实现一个矩阵，网上广为流传的糟糕的做法，掩盖了“矩阵法”的优美。

先回顾下 Fibonacci 数列的矩阵法：

$$\begin{pmatrix} F(n+2) & F(n+1) \\ F(n+1) & F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} \begin{pmatrix} F(2) & F(1) \\ F(1) & F(0) \end{pmatrix} \quad (1)$$

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} \begin{pmatrix} F(1) \\ F(0) \end{pmatrix} \quad (2)$$

上式中，对系数矩阵 A 求 n 次方，有 O(log n)解法，因而整个算法是 O(log n)。

某些介绍矩阵法的文章，会“偷懒”采用上面的第二种写法，而不是第一种写法。偷懒的结果，总是要付出代价的。对上面矩阵法的实现，存在两个盲点，也正由于这两个盲点，使“矩阵法”的实现代码看起来很复杂，失去了简洁之美。

盲点之一：对系数矩阵 A 求 n 次方，可以不采用矩阵乘法来实现。

将 F(1) = F(2) = 1， F(0) = 0 代入上面的公式 1，得到：

$$\begin{pmatrix} F(n+2) & F(n+1) \\ F(n+1) & F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} \quad =>$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix} = \begin{pmatrix} a1+a2 & a1 \\ a1 & a2 \end{pmatrix} (\text{其中} a1 = F(n),\ \ a2 = F(n-1))$$

上式，对任意 n >=1 都成立，也就是说 **A 的任意 n 次方，只要用两个变量表示，根本没必要去实现矩阵乘法。**

另外，由 A^n = A^k * A^(n-k)，结合上式，很容易就得到前面提到的公式：

F(n) = F(k)*F(n+1-k) + F(k-1)*F(n-k)，

盲点之二： A 的 n 次方计算方法。

计算一个数 m 的 n 次方，

若采用迭代法的话，一般是将 m^n，拆分成 m、m^2、m^4、m^8…中的几个的乘积。

若采用递归的话，则是将 m^n 拆分成计算 m^(n/2)

```
//迭代法：
int pow1(int m, unsigned n)
{
   int result = 1;
   int factor = m;
   while (n) {
     if (n & 1) { result *= factor; }
     factor *= factor;
     n /= 2u;
   }
   return result;
}

//递归法
int pow2(int m, unsigned n)
{
  if (n == 0) return 1;
  int square_root = pow2(m, n / 2);
  int result = square_root * square_root;
  if (n & 1) result *= m;
  return result;
}
```

对于计算一个整数的 n 次方，显然第一种解法效率高，但对计算矩阵的 n 次方，第二种解法（递归法）则更简单。该递归算法也可写成迭代形式：

```
int pow3(int m, unsigned n)
{
  if (n == 0) return 1;
  unsigned flag = n;                //小等于n的最大的2的k次幂
  for (unsigned value = n; value &= (value - 1); ) flag = value;

  int result = m;
  while (flag >>= 1) {
    result *= result;
    if (n & flag) result *= m;
  }
  return result;
}
```

（求小等于 n 的最大的 2 的 k 次幂(或求二进制表示中的最高/左位 1),有两种**不通用**的 O(1)方法：一种是使用位扫描汇编指令、另外一种是利用浮点数的二进制表示。）

```
unsigned extract_leftmost_one(unsigned num)
{
  union {
    unsigned i;
    float f;
  } u;
  u.f = (float)num;
  return u.i >> 23;
}
```

最后可得到如下代码：

① 采用一般迭代法计算A^n

```
static inline void matrix_multiply(uint& b1, uint& b2, uint a1, uint a2)
{
  const uint r1 =   a1 * b1 + a1 * b2 + a2 * b1;
  const uint r2 =   a1 * b1 + a2 * b2;
  b1 = r1;
  b2 = r2;
}

uint fib_matrix(uint num)
{
  uint b1 = 0, b2 = 1;
  uint a1 = 1, a2 = 0;
  for (; num != 0; num >>= 1) {
    if (num & 1) matrix_multiply(b1, b2, a1, a2);
    matrix_multiply(a1, a2, a1, a2);
  }
  return b1;
}
```

② 采用新的迭代法计算 A^n

```
typedef unsigned uint;

uint fibonacci(uint num)
{
  if (num == 0) return 0;
  uint flag = num;              //extract_leftmost_one
  for (uint value = num; value &= value - 1; ) flag = value;

  uint a1 = 1, a2 = 0;
```

```
  while (flag >>= 1) {
    const uint r1 = a1 * a1 + 2 * a1 * a2;
    const uint r2 = a1 * a1 +     a2 * a2;
    a1 = r1;
    a2 = r2;
    if (num & flag) {
      a1 = r1 + r2;
      a2 = r1;
    }
  }
  return a1;
}
```

上面提到的方法，很容易扩展到三阶矩阵，下面是《编程之美》书上的一道扩展题的解法：（具体分析见下一节）

假设：A(0)=1, A(1)=2, A(2)=2，对 n>2 都有 A(n)=A(n-1)+A(n-2)+A(n-3)，
1. 对于任何一个给定的 n，如何计算出 A(n)？
2. 对于 n 非常大的情况，如 n=2^60 的时候，如何计算 A(n) mod M （M < 100000)呢？

```
typedef unsigned uint;
typedef unsigned long long uint64;

uint fib_ex(uint64 num, uint M)
{
  assert(M != 0);
  const uint g0 = 1, g1 = 2, g2 = 2;
  if (num == 0) return g0;
  uint64 flag = num;
  for (uint64 value = num; value &= value - 1; ) flag = value;

  uint64 a1 = 0, a2 = 1, a3 = 0;
  while (flag >>= 1) {
    const uint64 r1 = 2 * (a1 + a2 + a3) * a1  +      a2 * a2;
    const uint64 r2 = 2 * (a1 + a2)      * a1  +  2 * a2 * a3;
    const uint64 r3 =     (a1 + 2 * a2)  * a1  +      a3 * a3;
    a1 = r1;
    a2 = r2;
    a3 = r3;
    if (num & flag) {
      a1 = r1 + r2;
      a2 = r1 + r3;
      a3 = r1;
```

```
    }
    a1 %= M;
    a2 %= M;
    a3 %= M;
  }
  return (a1 * g2 + a2 * g1 + a3 * g0) % M;
}
```

㈡ 扩展数列的通解：

下面将前面的结果扩展到任意 m 阶数列：

已知通项公式：$g(n)=\sum_{k=1}^{m} g(n-k)*f_k$（$f_k$为常系数），及初始值$g(i)$（$i=0,1\ldots m-1$），

对任意给定的n，求$g(n)$

采用矩阵法：

$$\begin{pmatrix} g(n+2*m-2) & \cdots & g(n+m) & g(n+m-1) \\ g(n+2*m-3) & \cdots & g(n+m-1) & g(n+m-2) \\ & \cdots & & \\ g(n+m) & \ldots & g(n+2) & g(n+1) \\ g(n+m-1) & \ldots & g(n+1) & g(n) \end{pmatrix} = \begin{pmatrix} f_1 & f_2 & \cdots & f_{m-1} & f_m \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & & \cdots & & \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} g(n+2*m-3) & \cdots & g(n+m-1) & g(n+m-2) \\ g(n+2*m-4) & \cdots & g(n+m-2) & g(n+m-3) \\ & \cdots & & \\ g(n+m-1) & \cdots & g(n+1) & g(n) \\ g(n+m-2) & \cdots & g(n) & g(n-1) \end{pmatrix}$$

$$= \cdots = \begin{pmatrix} f_1 & f_2 & \cdots & f_{m-1} & f_m \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}^n \begin{pmatrix} g(2*m-2) & \cdots & g(m) & g(m-1) \\ \ldots & & \ldots & \ldots \\ g(m+1) & \ldots & g(3) & g(2) \\ g(m) & \ldots & g(2) & g(1) \\ g(m-1) & \ldots & g(1) & g(0) \end{pmatrix}$$

设系数矩阵$A=\begin{pmatrix} f_1 & f_2 & \cdots & f_{m-1} & f_m \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$

可以证明：$A^m=\sum_{k=1}^{m} A^{m-k}*f_k$，进而可以得到：$A^n=\sum_{k=1}^{m} A^{m-k}*F_k$（$F_k$为常数），

而$(F_1\ F_2\ \cdots\ F_{m-1}\ F_m)$恰恰是m阶矩阵的$A^n$的最后一行。

（对第一个等式的证明有两种方法，

一种方法是只利用矩阵A的特殊性质，任意一个矩阵B左乘矩阵A得到的新矩阵C，第一行与其它行是线性相关的，且新矩阵的第k行等于矩阵B的第$k-1$行($k>=2$)）

另一种方法，取初使值$g(i)=f_{i+1}$，将通项公式代入矩阵中，即可证明两边相等。）

若A^n的第k行为$(L_1\ L_2\ \cdots\ L_{m-1}\ L_m)$，（利用右乘矩阵$A$的性质）可证明：

A^{n+1}的第$k-1$行等于：$(L_2\ L_3\ \cdots\ L_{m-1}\ L_m\ 0)\ +\ L_1*(f_1\ f_2\ \cdots\ f_{m-1}\ f_m)$

即等于：$(f_1*L_1+L_2\ \ f_2*L_1+L_3\ \cdots\ f_{m-1}*L_1+L_m\ \ f_m*L_1)$

例子：

① **m=2： g(n)＝f1 * g(n-1)＋f2 * g(n-2)， 初始值为：g0＝g(0)， g1=g(1)**

设系数矩阵为 A，A^n的最后一行为(a1　a2)，则

倒数第二行为：(f1*a1＋a2　f2*a1)

即：

系数矩阵 A		A^n	
f1	f2	f1*a1＋a2	f2*a1
1	0	a1	a2

```
typedef unsigned uint;

uint fib_matrix2(uint num)
{
  if (num == 0) return g0;
  uint flag = num;
  for (uint value = num; value &= value - 1; ) flag = value;
  /*
     A              A^n
    f1  f2   f1*a1 + a2    f2*a1
    1   0       a1         a2
  */
  uint a1 = 1, a2 = 0;  // 0 0 ... 1 0
  while (flag >>= 1) {
    const uint r1 = f1 * a1 * a1 +  2 * a1 * a2;
    const uint r2 = f2 * a1 * a1 +      a2 * a2;
    a1 = r1;
    a2 = r2;
    if (num & flag) {
      a1 = f1 * r1 + r2;
      a2 = f2 * r1;
    }
  }
  return a1 * g1 + a2 * g0;
}
```

② **m=3： g(n)＝f1 * g(n-1)＋f2 * g(n-2)＋f3*g(n-3)，初始值为：g0＝g(0)，g1=g(1),g2=g(2)**

设系数矩阵为 A，A^n的最后一行为(a1　a2　a3)，则

倒数第二行为：(f1*a1＋a2　f2*a1＋a3　f3*a1)

倒数第三行为：((f1*f1+f2)*a1＋f1*a2＋a3　(f1*f2+f3)*a1＋f2*a2　f1*f3*a1＋f3*a2)

即：

系数矩阵 A			A^n		
f1	f2	f3	(f1*f1+f2)*a1＋f1*a2＋a3	(f1*f2+f3)*a1＋f2*a2	f1*f3*a1＋f3*a2
1	0	0	f1*a1＋a2	f2*a1＋a3	f3*a1

0 1 0 a1 a2 a3

```
typedef unsigned uint;

uint fib_matrix3(uint num)
{
  if (num == 0) return g0;
  uint flag = num;
  for (uint value = num; value &= value - 1; ) flag = value;
  /*
       A                                        A^n
    f1  f2  f3   (f1*f1+f2)*a1 + f1*a2 + a3  (f1*f2+f3)*a1 + f2*a2   f1*f3*a1 + f3*a2
    1   0   0            f1*a1 + a2                  f2*a1 + a3         f3*a1
    0   1   0               a1                          a2                 a3
  */
  uint a1 = 0, a2 = 1, a3 = 0;  // 0 0 ... 1 0
  while (flag >>= 1) {
    const uint r1 = (f1 * f1 + f2) * a1 * a1 +  2 * f1 * a1 * a2  + 2 * a1 * a3  + a2 * a2;
    const uint r2 = (f1 * f2 + f3) * a1 * a1 +  2 * f2 * a1 * a2  + 2 * a2 * a3;
    const uint r3 = (f1 * f3)      * a1 * a1 +  2 * f3 * a1 * a2  +     a3 * a3;
    a1 = r1;
    a2 = r2;
    a3 = r3;
    if (num & flag) {
      a1 = f1 * r1 + r2;
      a2 = f2 * r1 + r3;
      a3 = f3 * r1;
    }
  }
  return a1 * g2 + a2 * g1 + a3 * g0;
}
```