斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：1、1、2、3、5、8、13、21、34、……这个数列从第3项开始，每一项都等于前两项之和。在数学上，斐波纳契数列以如下被以递归的方法定义：

### 一、直接递归实现

使用公式f(n)=f(n-1)+f(n-2)，依次递归计算，递归结束条件是f(1)=1，f(2)=1。

代码如下：

**public** **static** Long recursiveCompute(**int** n) {

**if** (n <= 2)

**return** 1L;

**else**

**return** *recursiveCompute*(n - 1) + *recursiveCompute*(n - 2);

}

这种方式可以以非常少的代码实现第N+1项的计算，但是与之相对的，这种算法的代价也非常的高，计算机每次调用函数时，为了保证函数执行完毕，都必须要将函数调用时的程序现场入栈保存，递归存在大量的自身调用，势必会产生非常庞大的函数现场栈来记录数据

我们以求解f(10)作为例子来分析递归求解的过程。要求得f(10)，需要求得f(9)和f(8)。同样，要求得f(9)，要先求得f(8)和f(7)……我们用树形结构来表示这种依赖关系

                  f(10)  
               /        \  
            f(9)         f(8)  
          /     \       /    \  
       f(8)     f(7)  f(7)   f(6)  
      /   \     /   \   
   f(7)  f(6)  f(6) f(5)

我们不难发现在这棵树中有很多结点会重复的，而且重复的结点数会随着n的增大而急剧增加。这意味这计算量会随着n的增大而急剧增大。用递归方法计算的时间复杂度是以n的指数的方式递增的。

CPU为i5 2.5GHZ，8G内存，windows7 64位操作系统，使用System.nanoTime()来计算消耗时间，一轮计算十次取平均耗时，三轮的平均耗时如下：

（后续方法相同，不再赘述）

|  |  |
| --- | --- |
| 计算值 | 纳秒 |
| f(10) | 67615 |
| f(40) | 577718143 |
| f(80) | 大于8小时 |

当计算f(80)时，超过8小时未计算出结果。

### 二、自底向上实现

上述直接递归算法，为了求解f(n)，需要同时递归求解f(n - 1)和f(n - 2)，显然这样就做了大量的重复工作。采用自底向上的算法即可避免这样的冗余。要计算f(n)，则依次计算f0，f1，f2...fn，这时计算fn只需要利用前两个结果即可，这样算法效率提高到了O(n)：

**public** **static** Long arraylist\_cache(**int** n) {

**if** (n <= 2) **return** 1L;

List<Long> results = **new** ArrayList<Long>();

results.add(1L);

results.add(1L);

**int** length = -1;

**while** ((length = results.size()) < n)

results.add(results.get(length - 2) + results.get(length - 1));

**return** results.get(n - 1);

如上使用的是List实现，也可使用数组实现。注意到缓存空间仅用于两次计算，因此可以使用新计算结果覆盖原有空间，节省空间。

核心算法如下：

**while** (index < n ) {

l = l2;

l2 = l1 + l2;

l1 = l;

index++;

}

消耗时间：

|  |  |  |  |
| --- | --- | --- | --- |
| 计算值 | List | array | swap |
| f(40) | 76620 | 1464 | 2463 |
| f(80) | 104838 | 2395 | 3216 |
| f(92) | 111393 | 2723 | 3448 |

对比简单递归实现，计算时间减少了不少，而且随着计算值增大，节约时间越明显。

不同的实现方式耗时不同，时间复杂度与实际执行时间并不是正比关系。

### 三、矩阵实现

f(n) =1\*f(n-1)+1\*f(n-2)

f(n-1)=1\*f(n-1)+0\*f(n-2)

可以转换为矩阵表达：

因此，成为了矩阵的乘方问题.

若A为n×k矩阵，B为k×m矩阵，则它们的乘积AB(有时记做A·B)将是一个n×m矩阵。前一个矩阵的列数应该等于后一个矩阵的行数，得出的矩阵行数等于前一个矩阵的行数，列数等于后一个矩阵的行数。

其乘积矩阵AB的第i行第j列的元素为：

通过分治法或快速幂，算法效率提高到了O(logn)。所谓分治法就是：

快速幂就是利用结合律快速计算幂次的方法，应用到矩阵即可：

**public** **static** **long**[][] matrix\_pow(**int** n) {

**long** [][] res = { { 1, 1 }, { 1, 1 } };

**long** [][] a=***UNIT***;

**while**(n>0){

// *matrixMultiply是矩阵乘法函数*

**if**((n&1)==1) res = *matrixMultiply*(res,a);

a=*matrixMultiply*(a,a);

n=(n>>1);

}

**return** res;

}

当n越大，相比较其他算法会节省更多时间。

核心代码为矩阵乘法与分治法，此处不细述

### 四、公式实现

f(n)=f(n-1)+f(n-2)线性递推数列的特征方程为：

解得,

则

∵f(1)=f(2)=1  
∴

解得 ，

∴

核心代码为

**double** temp = Math.*sqrt*(5.0);

**return** (**long**) ( (1/temp)\*(Math.*pow*((1+temp)/2,n)-Math.*pow*((1-temp)/2, n)) );

代码复杂度为O(1)，由于double类型的精度还不够，所以程序算出来的结果会有误差

代码实际执行时间：

|  |  |  |
| --- | --- | --- |
| 计算值 | Matrix | Formula |
| f(40) | 6569 | 2038 |
| f(80) | 7348 | 2422 |
| f(92) | 10687 | 2627 |

以上使用基本类型long，当n>92时，会超出long的表达范围，因此结果不正确，此时需要用BigInteger解决溢出问题，但会比long多一些耗时。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 计算值 | List | array | swap | Matrix | Formula |
| f(80) | 82997 | 48689 | 42518 | 100103 | 1207930 |
| f(800) | 506591 | 306645 | 383252 | 238591 | 7871161 |
| f(8000) | 5446817 | 3900261 | 4076532 | 1218439 | 107777421 |

可以看到计算值从f(80)到f(800)到f(8000)，10倍的增长，但矩阵法耗时并没有随着线性增长，性能最佳。

由于公式法小数需要使用BigDecimal，效率降低了很多，f(800)结果达到了，公式法精度存在问题，取整后结果已然不对，保留10000位小数时仅能保证前13位正确，f(8000)结果更是达到了的数量级。