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、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

简单递归算法，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]=f[n-1]+f[n-2]可以转换为矩阵表达：

因此，成为了矩阵的乘方问题，通过分治法，算法效率提高到了O(logn)。所谓分治法就是：

所以当n较大时，计算会快很多。

核心代码为矩阵乘法与分治法，此处不细述

### 四、公式实现

线性[递推数列](https://baike.baidu.com/item/%E9%80%92%E6%8E%A8%E6%95%B0%E5%88%97)的[特征方程](https://baike.baidu.com/item/%E7%89%B9%E5%BE%81%E6%96%B9%E7%A8%8B)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 | 2093 |
| F(92) | 10687 | 2025 |

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，不