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：

        /  0                      n=0  
f(n)=      1                      n=1  
        \  f(n-1)+f(n-2)          n>1

### 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。

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，即算法复杂度：

空间复杂度：

消耗时间：

用y表示f(n+1), x表示f(n)  
(1) y >= x>=1, n>=0  
(2) y <= 2x, n>=1，即从n=1开始，Fibonacci数列的上限是一个等比为2的数列。  
(3) 由(1)(2)可以推算出，(x+y)/y = 1 + x/y >= 1 + 1/2 = 1.5, n>=1，即y >= 1.5x, n>=2。又有f(2)/f(1) >= 1.5，所以不等式也适用于n=1，即从n=1开始，Fibonacci数列的下限是一个等比为1.5的数列。  
(4) 由(1)(3)可以推算出，(x+y)/y = 1 + x/y <= 1 + 1/1.5 = 5/3，即从某个数开始，Fibonacci数列的上限是一个等比为5/3的数列。  
(5) (3)(4)可以反复进行下去，上下限的比例逐渐靠拢。最终收敛在黄金比例1.6180339887

### 二、非递归cache算法

上述方法之所以慢是因为重复的计算太多，只要避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，如果下次需要计算的时候我们先查找一下，如果前面已经计算过了就不用再次计算了。

算法复杂度：

空间复杂度：

消耗时间：

### 三、新算法

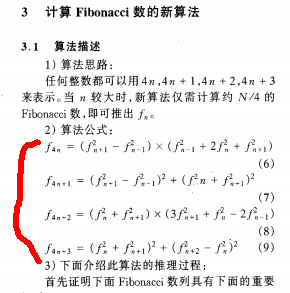
使用公式f[n]=f[n-1]+f[n-2]，依次递归计算，递归结束条件是f[1]=1，f[2]=1。

算法复杂度：

空间复杂度：

消耗时间：

新算法：



### 四、迭代swap

更简单的办法是从下往上计算，首先根据f(0)和f(1)算出f(2)，在根据f(1)和f(2)算出f(3)……依此类推就可以算出第n项了。很容易理解，这种思路的时间复杂度是O(n)。

算法复杂度：O(n)

空间复杂度：

消耗时间：

时间复杂度是0(n)，空间复杂度是0(1)。

### 五、数组

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。

算法复杂度：

空间复杂度：

消耗时间：

空间复杂度和时间复杂度都是0(n)，效率一般，比递归来得快。

int Fibonacci(int n,int\* f)

{

  f[0]=1;

  f[1]=1;

  for(int i=2;i<n;i++)

     f[n]=f[n-1]+f[n-2];

  return f[n-1];

  }

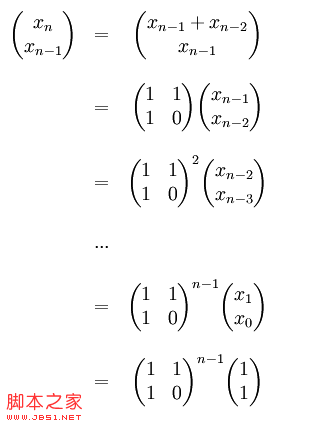
### 六、矩阵

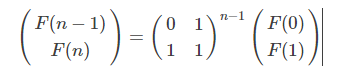
使用公式f[n]=f[n-1]+f[n-2]，依次递归计算，递归结束条件是f[1]=1，f[2]=1。

算法复杂度：O(log n)

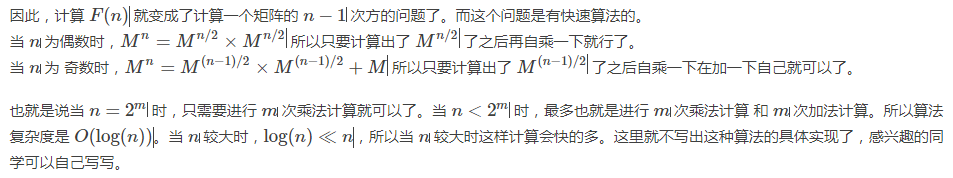
空间复杂度：

消耗时间：





分治法



如上图，Fibonacci 数列中任何一项可以用矩阵幂算出，而n次幂是可以在logn的时间内算出的。

如上见Fibonacci2和Fibonacci7

### 七、公式

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。

算法复杂度：

空间复杂度：

消耗时间：

因为公式使用浮点数求的，精度显然不够

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：  
　　 https://gss3.bdstatic.com/-Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D75/sign=ce6769f7aa51f3dec7b2bb6194ee3932/d833c895d143ad4b8262e7fd87025aafa40f06ba.jpg

　　解得  
　　 https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D87/sign=4d3d3ace7c310a55c024d3f3b745504e/242dd42a2834349bb930ceb8ccea15ce36d3bedd.jpg ， https://gss0.bdstatic.com/-4o3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D86/sign=ad56328bcc177f3e1434f10b71cf6edc/8601a18b87d6277fe2b192052d381f30e924fc68.jpg

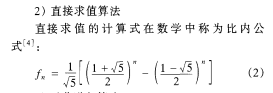
则 https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D146/sign=4445051eb57eca8016053de3a7239712/9c16fdfaaf51f3de42db1a5991eef01f3a2979a6.jpg

∵ https://gss0.bdstatic.com/94o3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D108/sign=e9f55ab26981800a6ae58d0e893433d6/8694a4c27d1ed21b36598dd8a86eddc451da3f7e.jpg   
　　∴ https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D230/sign=dc5833024efbfbedd859317c48f1f78e/6f061d950a7b0208fd039ddc67d9f2d3562cc8c1.jpg

　　解得 https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D139/sign=bd6a62ffb7119313c343fbb35c380c10/e61190ef76c6a7ef2032041bf8faaf51f3de66f3.jpg

https://gss0.bdstatic.com/-4o3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D275/sign=5cae971bbfa1cd1101b675278c13c8b0/ac4bd11373f08202fedde2144efbfbedab641b49.jpg

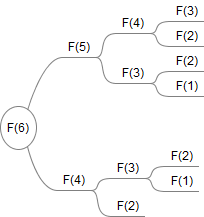
**六：公式实现**百度的时候，发现原来斐波那契数列有公式的，所以可以使用公式来计算的。  
由于double类型的精度还不够，所以程序算出来的结果会有误差，如果把公式展开计算，得出的结果就是正确的。



递归低效：

这段代码应该算是短小精悍（执行代码只有一行），直观清晰，而且非常符合许多程序员的代码美学，许多人在面试时写出这样的代码可能心里还会暗爽。但是如果用这段代码试试计算Fib(1000)我想就再也爽不起来了，它的运行时间也许会让你抓狂。

看来好看的代码未必中用，如果程序在效率不能接受那美观神马的就都是浮云了。如果简单分析一下程序的执行流，就会发现问题在哪，以计算Fibonacci(5)为例：



可以看到 F(4),F(3) 都重复计算了好几遍。这个问题随着规模的增加会愈发凸显，以至于Fib(1000)已经无法再可接受的时间内算出。也就是使用公式 fib(n) = fib(n-1) + fib(n-2)。这样的想法是很容易想到的，可是仔细分析一下我们发现，当调用fib(n-1)的时候，还要调用fib(n-2)，也就是说fib(n-2)调用了两次，同样的道理，调用f(n-2)时f(n-3)也调用了两次，而这些冗余的调用是完全没有必要的。可以计算这个算法的复杂度是指数级的。其实，只要将计算过的F(m) 保存下来，下次用时直接读取就好了，这样就省去了反复计算 F(m) 的问题

#### 递归与迭代的效率比较

我们知道，递归调用实际上是函数自己在调用自己，而函数的调用开销是很大的，系统要为每次函数调用分配存储空间，并将调用点压栈予以记录。而在函数调用结束后，还要释放空间，弹栈恢复断点。所以说，函数调用不仅浪费空间，还浪费时间。

这样，我们发现，同一个问题，如果递归解决方案的复杂度不明显优于其它解决方案的话，那么使用递归是不划算的。因为它的很多时间浪费在对函数调用的处理上，其效率上的差异是很显著的。

改进递归：尾递归

除了这种递归方式还有另外一种实现递归的方式，同样是上面的斐波那契数作为例子，这次我们不按照斐波那契的定义入手，我们从正常产生数列的过程入手来实现，0，1，的情况很简单可以直接返回，之后的计算过程就是累加，我们在递归的过程中要保持状态，这个状态要保持三个数，也就是上两个数和迭代的步数，所以我们定义的方法为：

function fibonacci (n , ac1 = 1 , ac2 = 1) {  
if( n <= 1 ) {return ac1};  
return fibonacci (n-1 , ac2 , ac1 + ac2);  
}

Fib(n,b1=1,b2=1,c=3):

   if n <= 2:

 return 1

else:

if n==c:

return b1+b2

else:

return Fib(n,b1=b2,b2=b1+b2,c=c+1)

这种方法我们在每一次递归的过程中保持了上一次计算的状态，所以称之为“线性迭代过程”，也就是俗称的尾递归。由于每一步计算都保持了状态所以消除了冗余计算，所以这种方式的效率明显高于前一种，其计算过程如下：

1 fib(6)

2 fib 0,0,1

3 fib 0,1,2

4 fib 1,2,3

5 fib 2,3,4

6 fib 3,5,5

7 fib 5,8,6

这两种递归方式之间是可以转换的，凡是可以通过固定数量状态来描述中间计算过程的递归过程都可以通过线性迭代来表示。

这样得到的算法复杂度是O(n)的。已经是线性的了。它的效率已经可以与迭代算法的效率相比了，但由于还是要反复的进行函数调用，还是不够经济。

递归算法和循环算法总结：

1. 一般递归调用可以处理的算法，也通过循环去解决常需要额外的低效处理。

2. 现在的编译器在优化后，对于多次调用的函数处理会有非常好的效率优化，效率未必低于循环。

3.递归和循环两者完全可以互换。如果用到递归的地方可以很方便使用循环替换，而不影响程序的阅读，那么替换成递归往往是好的。（例如：求阶乘的递归实现与循环实现。）