斐波那契数列计算方法比较实验报告

软件 13 杨楠 2021010711

摘要：本报告对比分析了求斐波那契数列的第 n 项的几种算法，着重分析算法复杂度，并通过编写代码，测试比较运行时间。递归法的时间复杂度最大，为指数级。递推法的时间复杂度为线性。代入公式计算法和矩阵乘法的时间复杂度均为对数级，但是前者的计算会存在误差。

1 实验环境

使用 C++11 语言在 VSCode 平台编写代码，在 Windows 环境编译运行。

2 算法分析

斐波那契数列F_n的递归式定义如下：

$$F_n=\begin{cases}0, n=0\\1, n=1\\F_{n-1}+F_{n-2}, n\geq 2\end{cases} \tag{1}$$

由递归式，可以用特征方程法或者数学归纳法，解得F_n的通项公式为

$$F_n=\frac{1}{\sqrt{5}}(\phi^n-\psi^n) \tag{2}$$

其中

$$\phi=\frac{1+\sqrt{5}}{2}, \varphi=\frac{1-\sqrt{5}}{2}$$

更具体地，由于$|\varphi|<1$，从而

$$\frac{|\varphi^n|}{\sqrt{5}}<\frac{1}{\sqrt{5}}<\frac{1}{2}$$

所以[1]

$$F_n=\left\lfloor\frac{\phi^n}{\sqrt{5}}+\frac{1}{2}\right\rfloor \tag{3}$$

2.1 递归法

直接利用递归式(1)编写递归函数：欲求 F(n)，需调用自身函数 F(n-1)和 F(n-2)，直至递归终点（即 n==1 和 n==0）。

递归法的 T(n)满足(1)式，即

$$T(n)=T(n-1)+T(n-2)$$

相应地可以求出

$$T(n)=\Theta(\phi^n)$$

为指数复杂度。

2.2 递推法

由递归式(1)，也可直接用递推法进行循环迭代，依次求出 F(0)，F(1)，…，F(n)
每次循环内进行有限次操作，从而

$$T(n) = \Theta(n)$$

为线性复杂度。

2.3 公式法

由(2)式或者(3)式，可以直接将 n 代入计算，得出结果。
其中的乘方计算，如果用逐次相乘，乘 n 次，那么时间复杂度为$\Theta(n)$。

而如果使用快速幂的方法，即：欲求ϕ^n，先求出$\phi^{\frac{n}{2}}$，以此递归下去，那么有

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

使用主定理，可算得$T(n) = \Theta(lgn)$

2.4 矩阵乘法

可以用数学归纳法证明

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \tag{4}$$

证明如下。当$n = 1$时，上式成立，即

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

假设$n = p$时，上式成立，即$\begin{bmatrix} F_{p+1} & F_p \\ F_p & F_{p-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p$

那么当$n = p + 1$时，有

$$\begin{bmatrix} F_{p+1} & F_p \\ F_p & F_{p-1} \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{p+1} + F_p & F_{p+1} \\ F_p + F_{p-1} & F_p \end{bmatrix} = \begin{bmatrix} F_{p+2} & F_{p+1} \\ F_{p+1} & F_p \end{bmatrix}$$

又有

$$\begin{bmatrix} F_{p+1} & F_p \\ F_p & F_{p-1} \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{p+1}$$

从而$\begin{bmatrix} F_{p+2} & F_{p+1} \\ F_{p+1} & F_p \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{p+1}$成立。综上，(4)式成立。

由此可以使用矩阵乘法，算出$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$从而得到$F_n$的值。矩阵乘法如果使用逐次相乘计算，时间复杂度为$\Theta(n)$，如果用快速幂，类似于之前的公式法，可算得$T(n) = \Theta(lgn)$

3 实验设计思路

计算斐波那契具体项的函数，都命名为 F(n)。

```
long long F(long long n)
```

使用 long long 类型，是便于后续数据大的情况下的调用。

测试数据过大时，计算结果可能溢出。可以通过取模的方式规避，但是本实验主要是比较算法所用时间，所以对此不作过多的处理。

分别在 demo1.cpp，demo2.cpp，demo3-1.cpp，demo3-2.cpp，demo4.cpp 中测试这几种算法。由于不同算法的时间复杂度差距较大，所以不直接同时运行，而是分开测试。
具体测试时，直接调用函数 test()，可以根据需要打印出运行时间（单位是微秒）等其他数据。

```
void test(long long n)
{
    cout << n;
    cout << " ";

    auto start = steady_clock::now();
    long long result = F(n);
    auto end = steady_clock::now();

    duration<double, micro> diff = end - start;
    cout << diff.count();
    cout << " ";

    cout << result;
    cout << " ";

    cout << "\n";
}
```

使用了<chrono>库，用于计时。调用 F()之前获取当前时间 start=steady_clock::now()，调用完毕后再次获取当前时间 end，两者作差得到 diff。

3.1 递归法

函数如下。

```
long long F(long long n)
{
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return F(n-1) + F(n-2);
}
```

3.2 递推法

函数如下。

```
long long F(long long n)
{
    long long a = 0, b = 1;
    if(n == 0)
        return a;
    else
    {
        for(long long i = 2; i <= n; i++)
        {
            b = a + b;
            a = b - a;
        }
        return b;
    }
}
```

每次循环，更新 a 和 b，最终返回 b 的值。

3.3 公式法

函数如下。

```
long long F(long long n)
{
    double phi = (sqrt5 + 1) / 2;
    long long result = round(power(phi, n) / sqrt5);
    return result;
}
```

其中，调用了<cmath>库中的 round()函数，用于获取距离该数最近的整数（即四舍五入）。在 power()函数中使用快速幂的方式。常量 sqrt5 的值为 sqrt(5)。

3.4 矩阵乘法

函数如下。

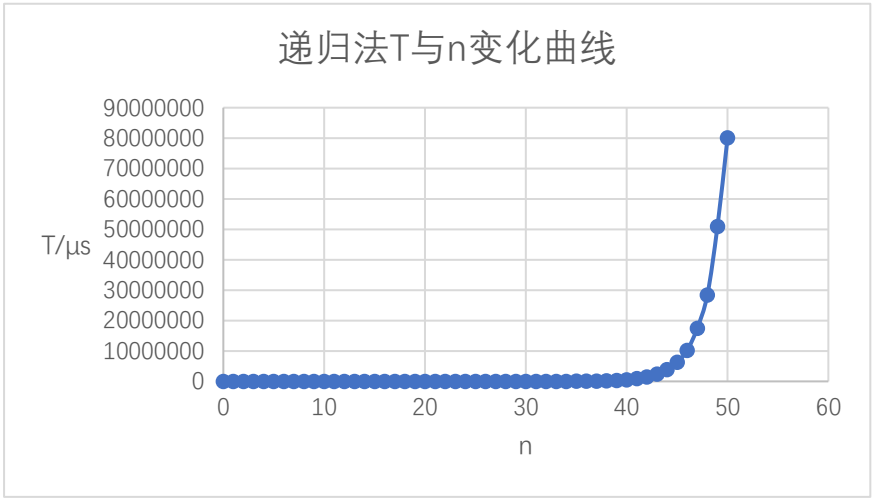
```
long long F(long long n)
{
    matrix m{1, 1, 1, 0};
    matrix result = power(m, n);
    return result.a12;
}
```

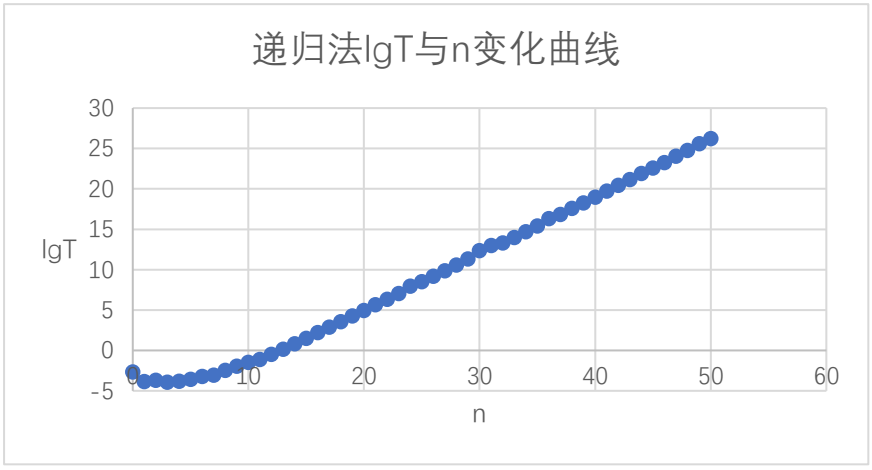
其中 matrix 是一个结构体，效果是一个 2*2 矩阵。用 power()函数实现矩阵乘方运算。

4 结果分析

4.1 递归法

编译 demo1.cpp 后运行，将输出结果整理成表后，绘图如下。





可以看出，当 n 比较大时，lgT 与 n 大致是线性关系，从而可知 T 确实是呈指数增长。注意到当 n>=40 后，T 有了很明显的上升，在 n>=45 后，T 基本是在 1 秒甚至 10 秒以上。可以预见，当 n 更大的时候，T 将会增长地更快。

4.2 递推法

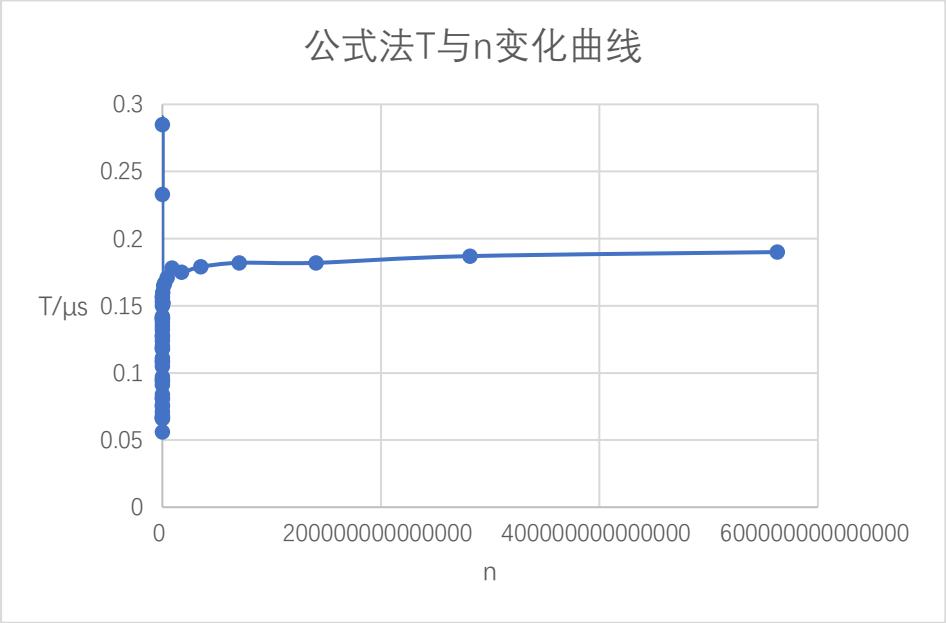
编译 demo2.cpp 后运行，将输出结果整理成表后，绘图如下。

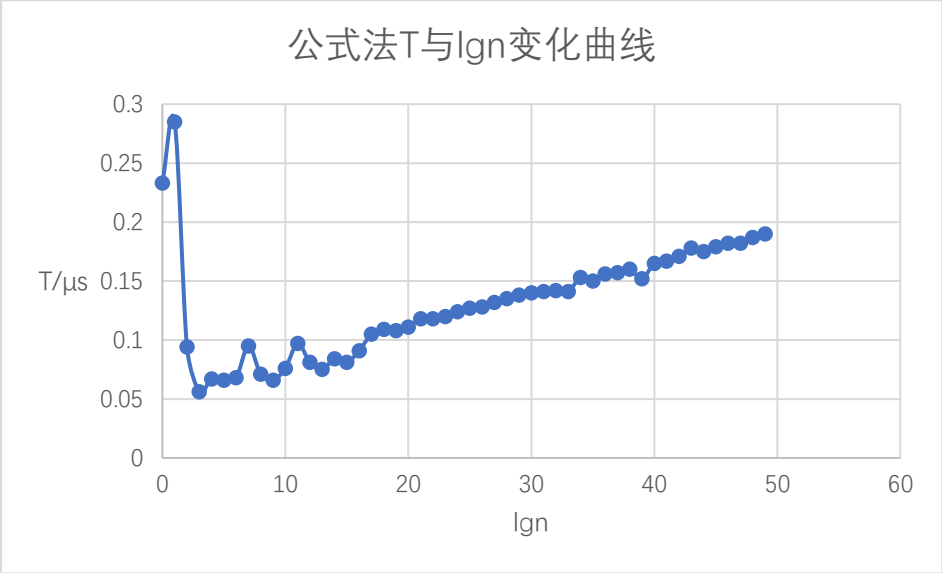
可以看到 T 与 n 大致呈线性关系，从而验证$T(n) = \Theta(n)$成立。

4.3 公式法

4.3.1 运行时间

编译 demo3-1.cpp 后运行，将输出结果整理成表后，绘图如下。





可以看到当 n 较大时，T 与 lgn 大致呈线性关系，从而验证$T(n) = \Theta(lgn)$成立。

4.3.2 计算误差

由于 C++对于无理数是用浮点数 double 进行存储的，在计算过程中又进行了乘法运算，计算结果可能会出现误差。

编译 demo3-2.cpp 后运行，将输出结果整理成表格如下。

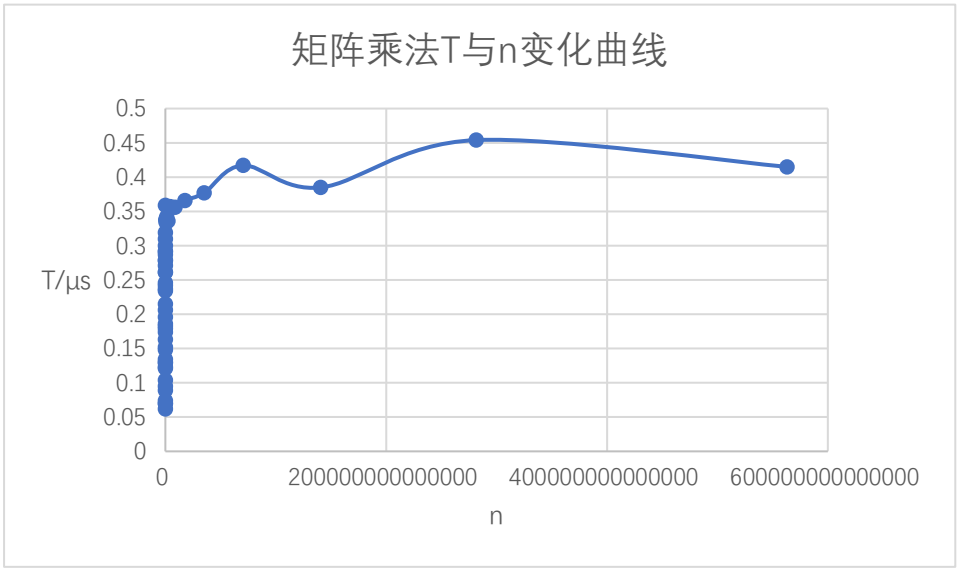
n	F(n)	F’(n)	F’(n)-F(n)
0	0	0	0
1	1	1	0
2	1	1	0
3	2	2	0
4	3	3	0
…	…	…	…
70	190392490709135	190392490709135	0
71	308061521170129	308061521170129	0
72	498454011879264	498454011879264	0
73	806515533049393	806515533049393	0
74	1304969544928657	1304969544928657	0
75	2111485077978050	2111485077978050	0
76	3416454622906707	3416454622906708	1
77	5527939700884757	5527939700884758	1
78	8944394323791464	8944394323791466	2
79	14472334024676221	14472334024676226	5
80	23416728348467685	23416728348467700	15
81	37889062373143906	37889062373143928	22
82	61305790721611591	61305790721611624	33
83	99194853094755497	99194853094755552	55
84	160500643816367088	160500643816367232	144
85	259695496911122585	259695496911122816	231
86	420196140727489673	420196140727490112	439

87	679891637638612258	679891637638612992	734
88	1100087778366101931	1100087778366103296	1365
89	1779979416004714189	1779979416004716288	2099
90	2880067194370816120	2880067194370819584	3464
91	4660046610375530309	4660046610375535616	5307
92	7540113804746346429	7540113804746356736	10307

用 F(n)表示理论值，F’(n)表示计算值。可以看到，在 n<=75 的时候，两者相等，但 n>=76 时，计算值与理论值开始有偏差，计算值偏大些。并且随 n 增大而增大。说明用这种方法计算，确实会存在计算误差。

4.4 矩阵乘法

编译 demo4.cpp 后运行，将输出结果整理成表后，绘图如下。



可以看到当 n 较大时，T 与 lgn 基本呈线性关系，从而验证$T(n) = \Theta(lgn)$成立。

5 结论

比较以上四种算法可得：递归法尽管思路非常自然，但是时间复杂度是指数级，运行效率非常有限，在 n>=40 之后的计算所耗费的时间就很高。递推法尽管是递归法的一个逆运算，但是时间复杂度降到了线性。实际上，如果将递归函数进行改进，比如使用数组来保存已经求得的前几项的值，下一次调用的时候直接使用而，则也可将时间复杂度降为线性。

公式法和矩阵乘法的时间复杂度都是对数级，但是由于前者设计根号的乘方运算，会产生误差，其准确性不如后者。综合来看，使用矩阵乘法快速幂，在准确性和效率上，是这几种方法中较高的。

本实验着重比较不同算法的时间复杂度，当数据较大时（比如 n>=100000），计算结果会溢出。尽管可以使用大数加法（高精度加法）等方式进行优化，但这些方式本身也会带来时间复杂度，本实验在此方面不做过多考察。

参考文献

[1] CLRS, Introduction to Algorithms (3rd edition), (2009), The MIT Press