计算斐波那契数列 实验报告

吴佳龙 2018013418

摘要

本次实验结合理论分析和程序设计，运用了不同的方法求解斐波那契数列的第 n 项，具体地，这些方法为：暴力递归法、通项公式法、递推法、矩阵快速幂方法，并比较了他们的结果正确性和计算时间，实验结果与理论分析相符。

1 问题

比较计算斐波那契数列的各种方法。

说明：为了避免高精度整数对于计算时间的影响，本次实验中的结果统一使用 **unsigned long long** 类型存储，即计算结果为斐波那契数列对于 $2^w = 2^{64}$ 自然溢出的结果。

2 实验环境

操作系统：Windows 10
IDE：Visual Studio 2017
处理器：3.1 GHz 双核 Intel Core i5

3 算法分析

本次实验共实现五种算法，分别在以下 4 个小节中进行算法描述与分析。

3.1 暴力递归

函数原型 ull fibo_brute_recursive(**int** n);

算法描述 递归地调用

```
fibo_brute_recursive(n−2) +
    fibo_brute_recursive(n−1)
```

边界条件 n==0 和 n==1

时间复杂度分析 递归式

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

的解为

$$T(n) = \Theta(Fib(n)) = \Theta((\frac{1+\sqrt{5}}{2})^n)$$

这是因为

$$\begin{aligned} &c_1 Fib(n-2) \le T(n-2) \le c_2 Fib(n-2), \\ &c_1 Fib(n-1) \le T(n-1) \le c_2 Fib(n-1) \\ &\implies c_1 Fib(n) \le T(n) \le c_2 Fib(n) \end{aligned}$$

空间复杂度分析 栈深度 $\Theta(n)$

3.2 通项公式

函数原型 ull fibo_formula(**int** n);

算法原理 斐波那契数列的通项公式为

$$Fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

当 n 足够大时，可忽略后一项，因此

$$Fib(n) = \text{round}(\frac{1}{\sqrt{5}}\phi^n), \forall n \ge 0, \phi = \frac{1+\sqrt{5}}{2}$$

算法描述 求 ϕ^n 可采用快速幂算法，具体地，有递归和非递归两种形式。

递归地，$\phi^n = (\phi^{\lfloor \frac{n}{2} \rfloor})^2$ 或 $\phi^n = (\phi^{\lfloor \frac{n}{2} \rfloor})^2 \times \phi$。

非递归地，记 $n = \sum_{i=1}^{w} b_i 2^i$，有

$$\phi^n = \prod_{b_i=1} \phi^{2^i}$$

ϕ^{2^i} 可递推求得。

本次实验中，实现了非递归的形式。

时间复杂度分析 递归地，

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n)$$

非递归地，

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

空间复杂度分析 $\Theta(1)$

误差分析 浮点数存储存在误差，若 x_1, x_2 的两个近似值 x_1^*, x_2^* 的误差为 $\varepsilon(x_1^*), \varepsilon(x_2^*)$ 有

$$\varepsilon\left(x_1^* \cdot x_2^*\right) \approx \left|x_1^*\right| \cdot \varepsilon\left(x_2^*\right) + \left|x_2^*\right| \cdot \varepsilon\left(x_1^*\right)$$

在计算 ϕ^n 时，误差

$$\begin{aligned} E(n) &\approx 2\phi^{\frac{n}{2}} E(\frac{n}{2}) \\ &\approx n\phi^n \varepsilon(\phi) \approx nFib(n)\varepsilon(\phi) \end{aligned}$$

当误差积累到接近 1 时，该算法计算结果可能出现错误。

3.3 递推

函数原型 `ull fibo_brute_recursive(int n);`

算法描述 从初始值出发，运用递推公式

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

时间复杂度分析 为求 $Fib(n)$ 共递推 n 次，

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

空间复杂度分析 $\Theta(1)$

3.4 矩阵快速幂

函数原型 `ull fibo_mat_recursive(int n);`
`ull fibo_mat_nonrecursive(int n);`

算法原理

Theorem 1.

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

算法描述 对斐波那契矩阵 $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ 应用快速幂算法。

在本次实验中，实现了递归和非递归的两种形式，试图比较他们的计算时间差异。

时间复杂度分析 与实数形式的快速幂算法类似，计算两个 2×2 矩阵的乘积的复杂度是 $\Theta(1)$ 的，因此递归和非递归版本都有

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

空间复杂度分析 $\Theta(1)$

4 结果分析

4.1 结果正确性

有符号 64 位整数的精确表示保证了暴力递归法、递推法和矩阵快速幂方法的结果的正确性。

对于通项公式法，调用不同大小的 n ，部分计算结果和误差见表 4.1。

Table 1: 通项公式法的实际误差

n	通项公式法结果	真实值	误差
75	2111485077978050	2111485077978050	0
76	3416454622906706	3416454622906707	1
77	5527939700884755	5527939700884757	2
78	8944394323791463	8944394323791464	1
79	14472334024676218	14472334024676221	3
80	23416728348467676	23416728348467685	9
81	37889062373143896	37889062373143906	10
82	61305790721611584	61305790721611591	7
83	99194853094755488	99194853094755497	9
84	160500643816367040	160500643816367088	48

结果分析 在本次实验的环境下，通项公式法在 $n > 75$ 的情况下出现误差，这与双精度浮点数约有 15 或 16 位有效数字，以及误差分析中通项公式法的误差积累约为 $nFib(n)\varepsilon(\phi)$ 相符合。

4.2 计算时间

在不同尺度的输入 n 的情况下，测得不同算法的计算时间如图 1, 2, 3

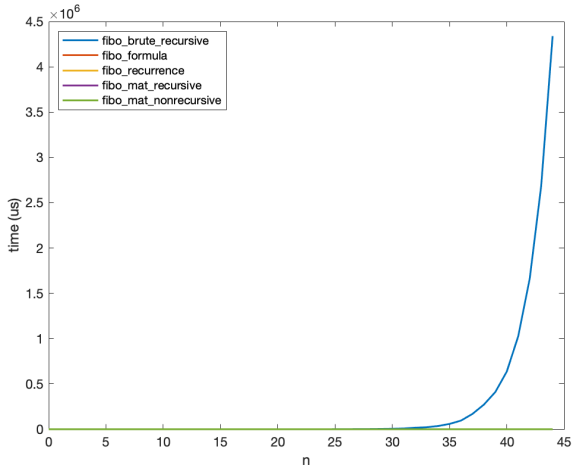

Figure 1: $n \in [0, 44]$，不同算法计算时间（单位 us）

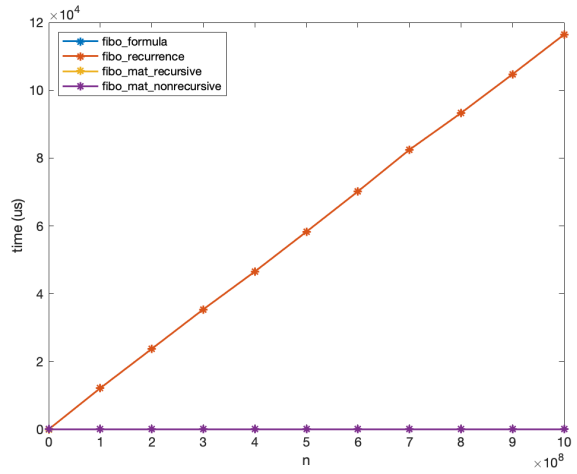

Figure 2: $n \in [0, 10^8]$，不同算法计算时间（单位 us）

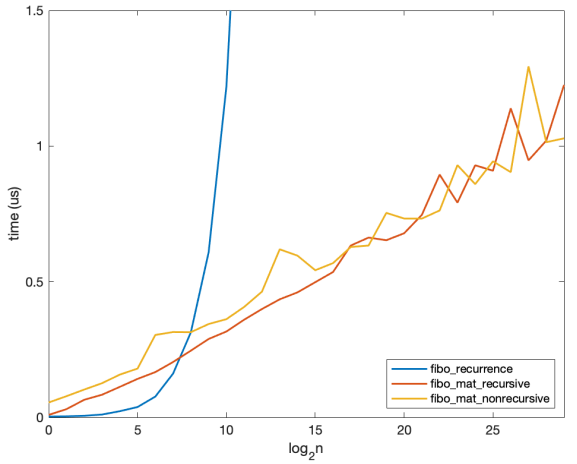

Figure 3: $n \in [2^0, 2^{30}]$，不同算法计算时间（单位 us）

结果分析 从图 1 可以看到，暴力递归法的计算时间呈指数级增长，符合上文对其时间复杂度的分析，其余算法远远快于暴力递归法。

从图 2 可以看到，递推法的计算时间呈线性增长，符合其时间复杂度，矩阵快速幂方法远远快于递推法。需要注意的是，此时 n 已经远远超过通项公式法的误差能够承受的范围。

从图 3 可以看到，矩阵快速幂方法的计算时间随着 n 指数增长时线性增长，而递推法指数增长，再次印证了时间复杂度的分析。但是由于 $\lg n$ 仍然过小，递归调用对于计算时间的影响仍不显著，递归与非递归形式地矩阵快速幂速度相当。

5 总结：不同方法的比较

暴力递归 复杂度是指数级，不能接受。

如果修改成**记忆化** 的形式，即：通过一维辅助数组存储第一次调用 $Fib(n)$ 时计算的值，在之后调用相同的 n 的时候直接返回存储的值，则可以把时间复杂度降到 $\Theta(n)$，但是空间复杂度仍然劣于递推法。

通项公式法 能够快速地（$T(n) = \Theta(\lg n)$）求出第 n 项的值，但是由于浮点数误差的积累，当 n 过大时，计算结果不准确。

递推 相比通项公式法和快速幂法，如果设立一个一维数组存储临时计算值，则在一次计算之后能够得到 $0 \sim n$ 所有项的值。

矩阵快速幂 能够快速地（$T(n) = \Theta(\lg n)$）求出第 n 项的值，且不会产生通项公式法中的误差积累。在仅求一项的情况下是一种非常合适的算法。