# 算法分析与设计-作业二

王宸昊 2019214541

2019 年 10 月 6 日

## 1 最近点对

### 1.1 算法简介

**Stoichiometry** The relationship between the relative quantities of substances taking part in a reaction or forming a compound, typically a ratio of whole integers.

**Atomic mass** The mass of an atom of a chemical element expressed in atomic mass units. It is approximately equivalent to the number of protons and neutrons in the atom (the mass number) or to the average number allowing for the relative abundances of different isotopes.

### 1.2 算法伪代码

### 1.3 实验效果

### 1.4 进一步改进

### 1.5 界面展示

## 2 三种不同的方法求 Fibonacci 数

### 2.1 算法简介

斐波那契数列 (Fibonacci Sequence) 是指 1,1,2,3,5,8... 这样的数列, 在数学上使用一下递推的方式定义：

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n) = F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

在本次实验中，采用三种不同的算法对 Fibonacci 数列进行计算，分别是递归法、迭代法和矩阵法。对于不同的算法采用的策略不同，算法的运行时间和结果的精度都不尽相同，在下面的实验中，会对不同的算法的性能进行对比。

**递归法** 递归法利用分治的思想，将规模为 n 问题分解为 n-1 和 n-2 规模的问题，借助自身函数调用解决问题。但是递归法也存在着很多的问题，例如递归调用栈不能太深，否则程序无法正常运行；递归法中存在着大量重复的计算，极大影响了算法执行的效率。

**迭代法** 迭代法的算法思想更加直观，不断利用前两个元素的值累加迭代得到下一个结果，与递归法相比，减少了很多重复的计算，而且计算的复杂度也是线性的。

**矩阵法** 矩阵法是利用了分治算法的思想，将求解斐波那契数列的问题转化为等价的矩阵运算，利用分治法的算法思想，将矩阵运算分解为规模更小的问题进行求解。其算法复杂度为 $lg(n)$, 在实际的实验中，其算法的执行时间比较优秀，但是由于涉及到矩阵乘法，存在着算数精度的问题。

## 2.2 算法伪代码

递归法：

**Algorithm 1** FIB_RECUR($n$)

```
1: if n ≤ 1 then
2:     return n
3: end if
4: return FIB_RECUR(n-1) + FIB_RECUR(n-2)
```

迭代法：

**Algorithm 2** FIB_LOOP($n$)

```
1: a = 0
2: b = 1
3: for i = 0 to n do
4:     a = b
5:     b = a + b
6: end for
7: return a
```

矩阵法：

**Algorithm 3** FIB_MATRIX($n$)

```
1: matrix = [1, 1; 1, 0]
2: return pow(matrix, n)
```

### 2.3 实验效果

首先介绍实验的实验环境。编译环境为 Python3.6,操作系统的版本为 Win10。CPU 为 AMD Ryzen 7 1700(3.0 GHz)，RAM 8G。
测试环境下，使用 Python 中 time 模块进行测量算法执行时间，最高精度达到微秒级别。此外还额外使用 numpy 模块进行高精度矩阵运算。
在实验方法上，将输入规模 n 逐渐增大，为了降低随机因素对执行效果的影响，对于每一个 n 进行 10 次计算取平均值。

不同方法的算法执行时间如下：
递归法的结果如图 (1) 所示，递归法的算法执行时间增长的速度非常快，在计算前 20 项时，运行时间基本在 1ms 以下，但是当 n 继续增大时，算法的执行之间明显上升，在 n 达到 30 左右，算法执行时间大约为 350ms，而当 n 达到 40 时，所需的时间到达了 44810ms，当 n 到 47 以后，算法再继续计算新的结果。由此可见，在递归的情况下，当递归层数达到 40 以上，很难计算出结果。

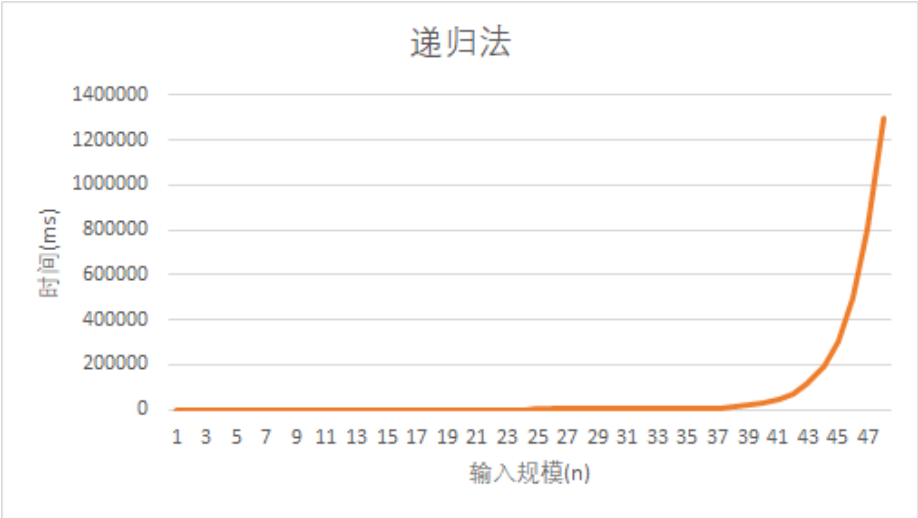


图 1: 递归法运行时间

矩阵法的结果如图 (2) 所示，矩阵法的执行效果要优于递归法。虽有波动，但是整体上算法增长的速度比较慢，n 在 100 以下的时候，基本上都在 10ms 以下能计算出结果。但是矩阵法存在巨大的精度问题，在 Python 中，int 类型是不会溢出的，但是在实际的计算中，由于使用 numpy 的矩阵乘法，其数据长度对多支持 64 位，在实际的实验中可以观察到，所以当 n 大于 93 时，在计算矩阵乘法时，会产生溢出，导致得到的斐波那契数出现负数。

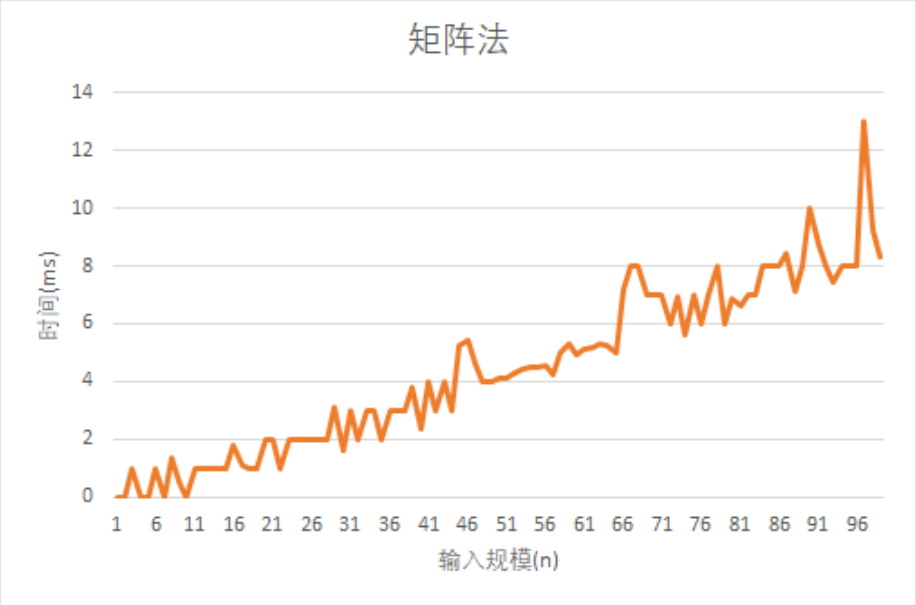


图 2: 矩阵法运行时间

迭代法的结果如图 (3) 所示，综合三个算法，迭代法是最稳定的算法，在大规模的输入规模的情况下仍可以保证算法的执行时间，并且不存在溢出的问题。可以观察到，在输入规模 10000 的情况下，在 100ms 内仍可以得到结果，在实际的测试中，输入的 n 最多可以达到 60 万，结果可以在 4000ms 左右得到。在精度的上，由于迭代法只有加法运算，在 Python 中对 int 类型做了特殊的处理，保证在整型的情况下，不会出现溢出的情况。

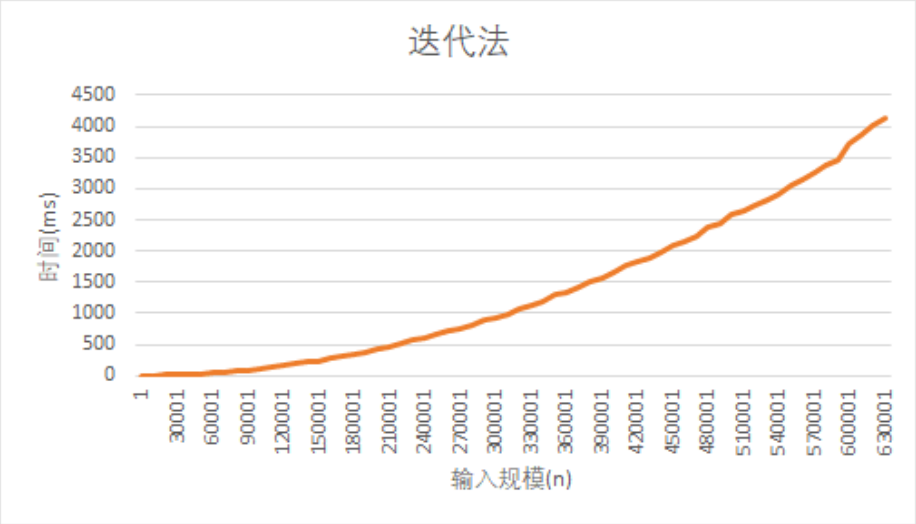


图 3: 迭代法运行时间

算法的运行时间对比如图 (4) 所示。由于迭代法的输入规模明显大于其他两种方法，效果也明显优于其他两种方法，因此并没有在图上画出。从图上可以看出，蓝色线代表递归法的运行时间，对应时间刻度为左侧的坐标轴；橘色代表的是矩阵法的运行时间，对应时间刻度为右侧的坐标轴。通过对比可以看出，在相同的输入 n 下，递归法的运行时间比矩阵法的运行时间要高很大的数量级。

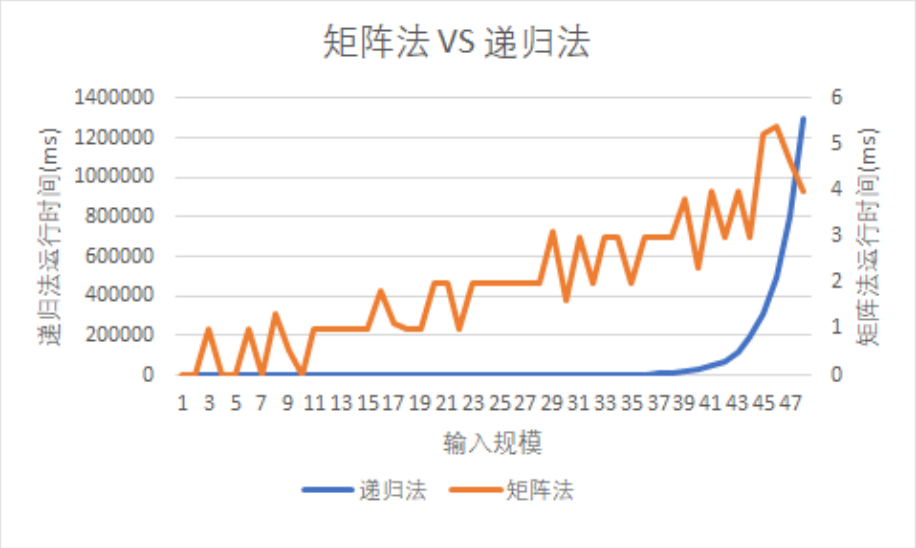


图 4: 矩阵法 VS 递归法

综合以上实验结果来看，迭代法效果无论从精度还是时间上，都明显优于其他两种方法。矩阵法在小规模输入下精度和执行时间也比较优秀，递归法的算法性能最差。

# 3 CLRS, Page, 72 5.3-4

证明：

(1) 根据题意得，当 dest = j 时，表示 $A[i]$ 出现在 $B[j]$ 的位置，即当 $i + offset = j$ 或 $i + offset - n = j$。根据 offset 的含义可知，offset 只是在原 A[i] 的数组的基础上循环移位 offset 位，对于每个 i, 映射到 B 中的唯一位置 j 中。∴ offset 取不同值 B[j] 的取值不同，A[i] 出现在 B[j] 的概为 $\frac{1}{n}$

(2) 此算法没有改变数组元素之间的相对位置，并不能产生均匀随机排列。由上一问可知，此算法最多产生 n 种排列，而均匀随机排列应有 n！种。例如假设 $A[] = [1, 2, 3]$。随机排列应有 $3! = 6$ 种，根据此算法的执行过程，当 offset=1 时，B[]=[3,1,2]; 当 offset=2 时，B[]=[2,3,1]; 当 offset=3 时，B[]=[1,2,3]。只能得到 3 种排列，无法产生类似 [2, 1, 3] 类似的排列。

# 4 CLRS, Page, 73 5.3-5

证明：

设 $X_i$ 表示第 i 个位置元素唯一，共有 $n^3$ 个元素。则每个位置的元素都唯一的概率为：

$$
\begin{aligned}
P(\text{所有元素都唯一}) &= P(X_1 \bigcap X_2 \bigcap X_3 \cdots \bigcap X_n) \\
&= 1(1-\frac{1}{n^3})(1-\frac{2}{n^3})(1-\frac{3}{n^3})\cdots(1-\frac{n}{n^3}) \\
&\geq 1(1-\frac{n}{n^3})(1-\frac{n}{n^3})(1-\frac{n}{n^3})\cdots(1-\frac{n}{n^3}) \\
&\geq (1-\frac{1}{n^2})^n \\
&\geq (1-\frac{1}{n})
\end{aligned}
$$