算法分析与设计-作业二

王宸昊 2019214541

2019 年 10 月 7 日

1 最近点对

1.1 算法简介

最近点对问题是指在一个点对的集合里面寻找一对欧几里得距离最小的点对，根据解决问题的思想的不同，可以采用两种不同的算法，一种是暴力法，计算每个点对其他点的距离，找到一个最小值，复杂度为 $O(n^2)$，另一种方法是采用分治的思想，将一个规模为 n 的子问题拆解为两个规模为 n/2 的问题，复杂度为 $O(nlgn)$。

1.2 算法分析与设计

暴力法：

对于集合中的每一个点，都计算一遍与其他点的距离，然后选择一个最小的进行记录。由以上的算法思想可以得知，暴力法总体的复杂度为 $O(n^2)$。再进一步的考虑优化，在遍历每个点的时候，每个点和其他点只需要计算一次距离，所以在遍历的时候，只需要计算位于其后面的点。

算法的实现在 *closet_points* 文件夹下的 *hw2_brute.py* 文件下。首先根据 *rand*() 函数随机生成若干个点。然后作为参数传递给 *BruteSolution*() 函数中，该函数中包括两个 for 循环：对于第 i 个点，以此遍历第 $i+1$ 后面的点，通过 *GetDistance*() 函数计算距离，记录下来最近的距离返回。

分治法：

分治算法的思想将从 n 个点对中寻找最近点对转化为从 2 个 n/2 规模的子问题，再从中寻找最近的点对，因此算法的递归式可以写为 $T(n) = 2T(n/2) + O(n)$，根据主方法，可以得到整体算法的复杂度为 O(nlgn)。

具体算法的实现在 *closet_points* 文件夹下的 *hw2_recursive.py* 文件下。同理暴力法，首先随机产生若干点对，首先对点对根据 X 和 Y 进行从小到大的排序，得到两个有序的点对集：SortedA_X 和 SortedA_Y，传入 RecursiveSolution() 中进行递归求解。
递归的过程：当 n<3 时，采用暴力法，n>3 时，根据 n 的大小将点对分为左右两个子集，对左右两个子集继续求解最小点对，并记录下来左右两个子集中更小的点对，以及其距离 δ。
接下来需要处理最近点对一个在左子集，一个在右子集的情况，根据书上的证明可知, 只需要考虑 X 坐标在 $[mid - \delta, mid + \delta]$ 范围内的点即可。对于范围内的每个点，只需要考虑按 Y 排序后，后面 5 个点的距离即可，最后与之前的最小值相比，取最小值。复杂度为 O(n)，这也解释了递推式后面合并解的复杂度为 O(n)。同时只进行了 2 次预排序。

1.3 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。最多 n 可以达到百万级。
在选择实验语言方面，选择 Python 主要是由于 Python 支持的 rand() 函数可以产生任意大小的随机函数，而且由于 Python 的特殊设计，其支持整型数据不会溢出。

如表 (1) 当中所示，在相同的输入规模下，对比了暴力法和分治法的执行时间，可以从表中明显的看出，暴力法的整体执行时间高于分治法几个数量级，并且随着输入规模的增大，暴力法的增长速度明显高于分治法的增长速度。

表 1: 同等输入规模下暴力法和分治法执行时间对比

点对个数	暴力法执行时间 (ms)	分治法执行时间 (ms)
1000	673.589	21.493
2000	2682.150	47.543
5000	16315.519	130.447
10000	68268.161	276.948
15000	153055.579	435.02
20000	273476.506	588.759

在实际的测试中，为了测试极限情况下算法的执行性能，设计了如下实验。从 1000 个随机点开始，以 20000 为步长，依次执行算法，测量算法的性能，得到的结果如图 (1) 所示. 可以看到暴力法对应的 Y 坐标为左侧的坐标轴，算法的执行时间随规模增大增长迅速，在 60000 个点对的时候，算法执行时间已经高达 2378667ms。而分治算法随着输入规模的增大增长比较稳定，在达到极限 100 万个点的时候，算法的执行时间为 48747ms，明显优于暴力法。

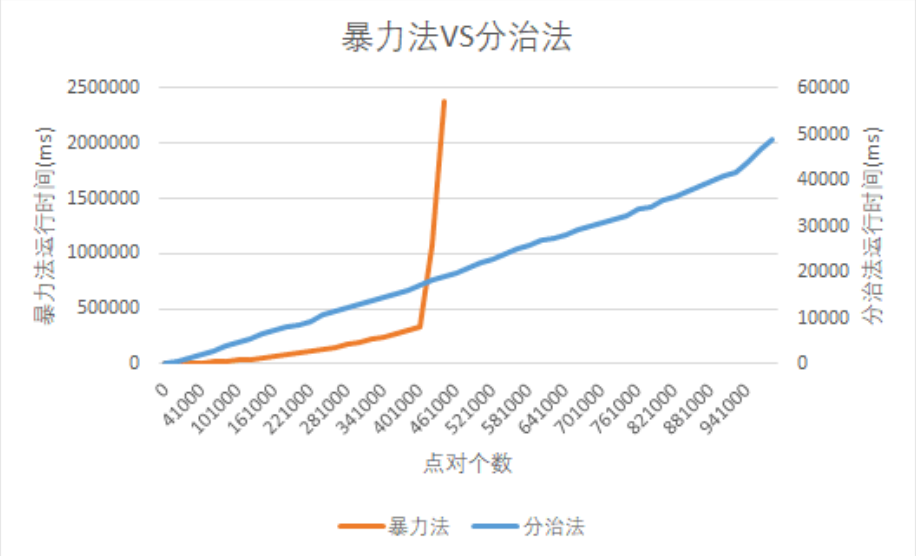

图 1: 暴力法 VS 分治法

1.4 进一步改进

考虑到算法的进一步改进。在计算距离的函数 GetDistance() 中，其作用在与比较不同点之间的距离的大小，所以并没有必要在其中计算欧式距离的时候开根号，只需要在最后返回的时候。对最小距离开一次平方就可以了，这样在每一次的迭代中，都能节省不少开方带来的计算开销。

实验效果如图 (2) 和图 (3) 所示。在减少了开平方操作以后，暴力法和分治法都在运行时间上有不同程度上的减少，在输入的规模大了以后，减少的效果更加明显，在 231000 个点的时候，减少了将近 3000ms，效果比较明显。

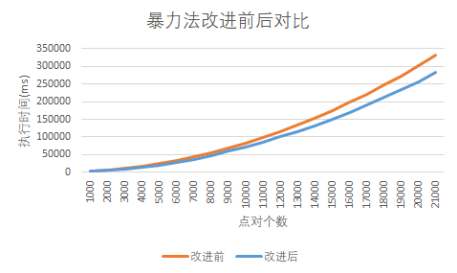

图 2: 暴力法改进前后对比

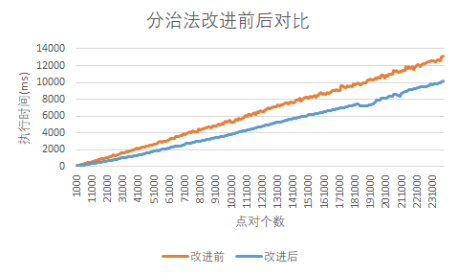

图 3: 分治法改进前后对比

1.5 界面展示

界面展示采用 Python 实验，GUI 的展示使用 Python 自带的 tkinter 模块，此外，使用 matplotlib 绘制散点图。代码的实现在 closest_points 文件夹下的 hw2_GUI.py, 命令行下执行 python hw2_GUI.py 即可执行。

首先在代码中封装了 From 类，在构造函数中初始化了界面的不同的组件的相对位置：在上侧是两个 label 进行提示，两个 Entry 一个负责输入产生随机点对的个数，一个负责输出最近距离。再下面是三个 Button，一个负责退出，一个负责执行算法，另一个负责选择完点之后，执行鼠标随机选点的寻找对近点对。在下方是一张 canvas 的画布，将 matplotlib 输出的图形进行展示，此外还保留了工具栏，有放大、保存等功能。在后面的逻辑中，根据输入框的数字，传递给算法的逻辑，产生相应的点对，调用分治法计算出最近点对和距离，最后在图上画出。

随机生成：

在输入框中输入要随机生成的点对个数，点击 Run 即可得到最近点对。实际项目运行效果如图 (4)：

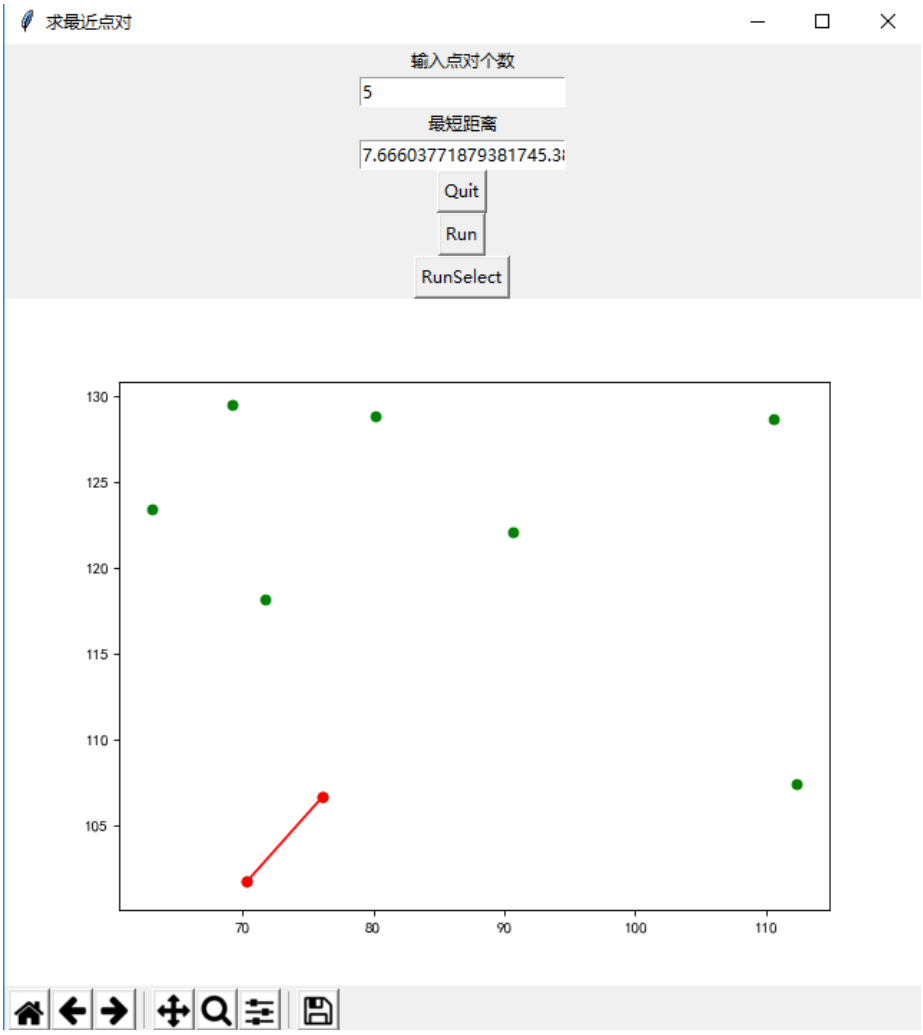

图 4: 随机生成点运行效果

鼠标选点：

在下方的画布中移动鼠标，右下角对显示鼠标悬停的位置的坐标值，点击鼠标可以进行选择，选择的点会在控制台输出 (注意不要选到坐标轴以外的点)，如图 (6)。选点结束后点击第三个 Button 进行计算，实际演示如图 (5)

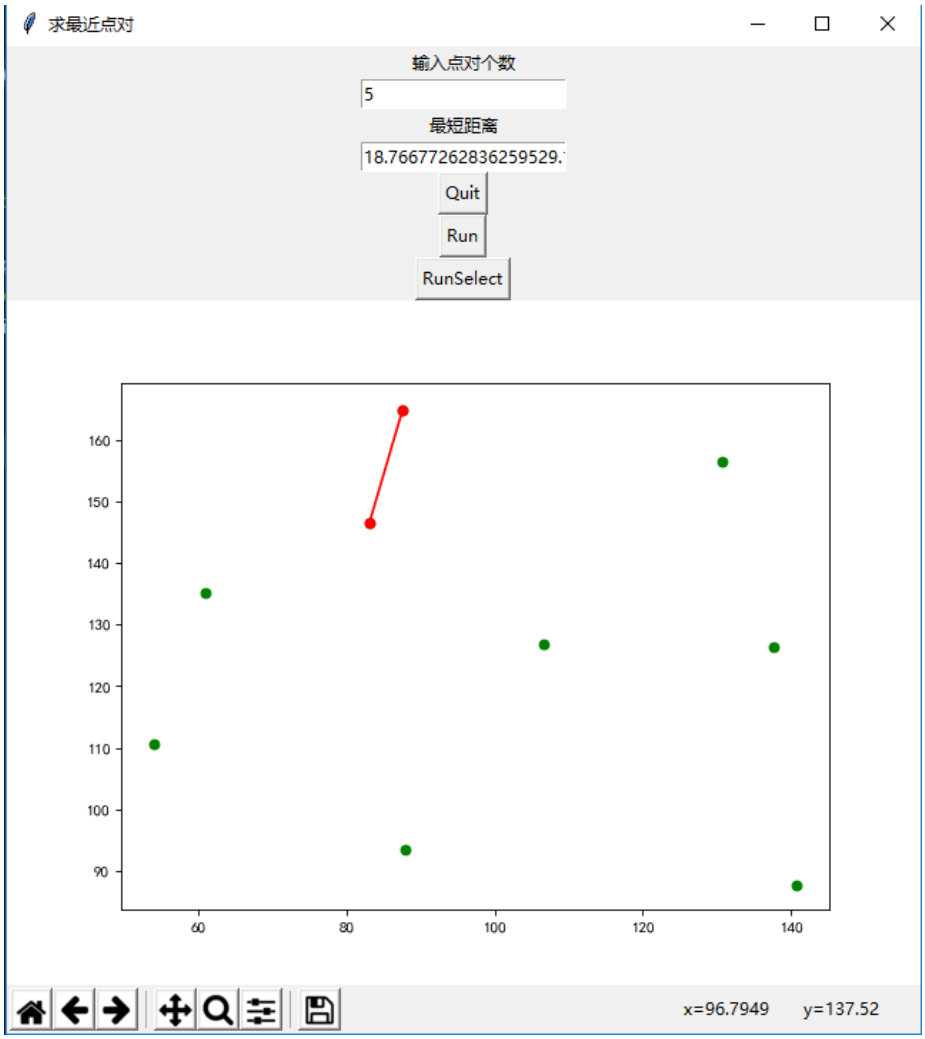
求最近点对
输入点对个数
5
最短距离
18.76677262836259529.
Quit
Run
RunSelect
160
150
140
130
120
110
100
90
60
80
100
120
140
x=96.7949
y=137.52

图 5: 鼠标选点运行效果

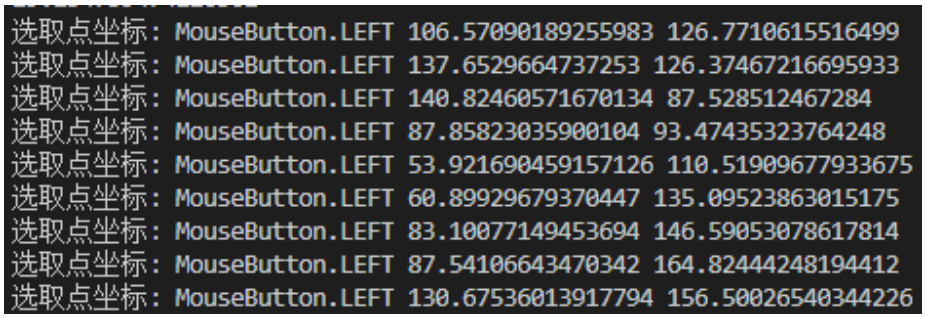

图 6: 命令行输出选点坐标

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 (7) 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 7: 递归法运行时间

矩阵法的结果如图 (8) 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。

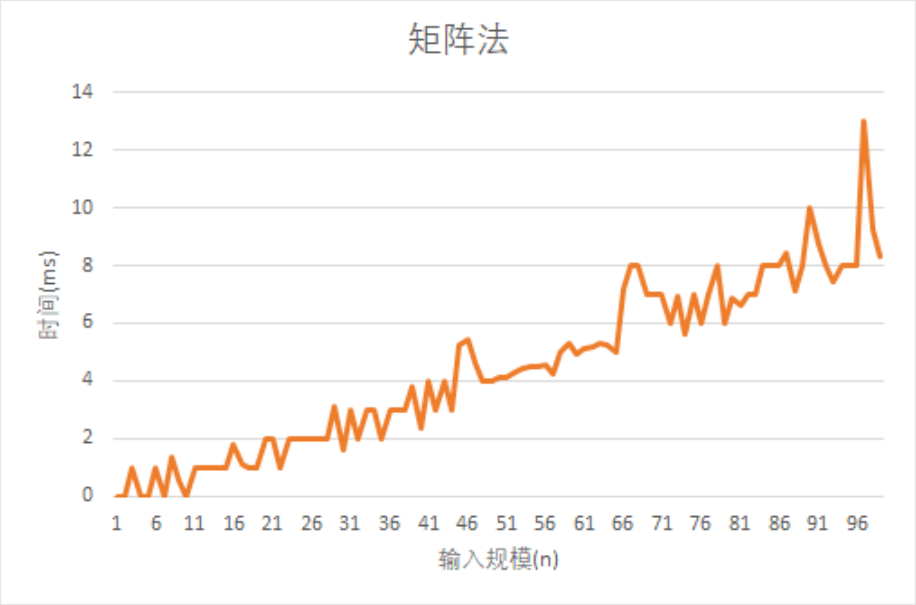

图 8: 矩阵法运行时间

迭代法的结果如图 (9) 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。

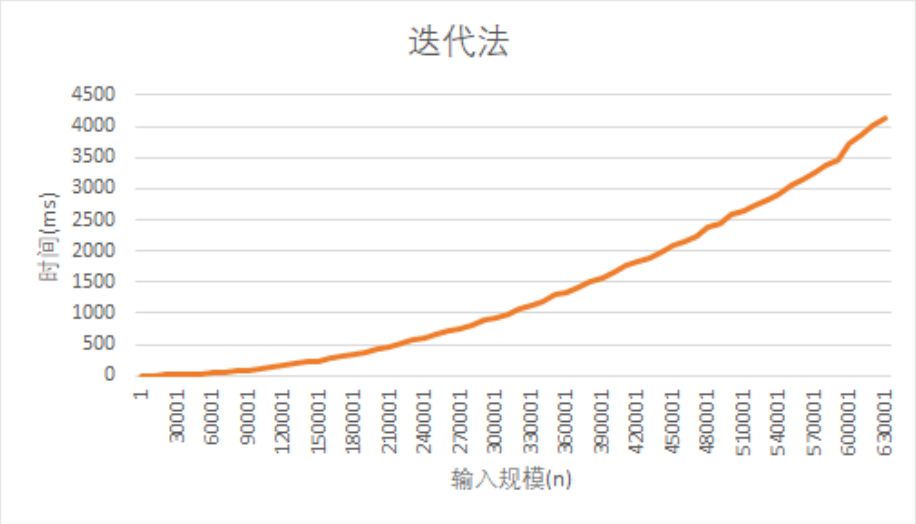

图 9: 迭代法运行时间

算法的运行时间对比如图 (10) 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。

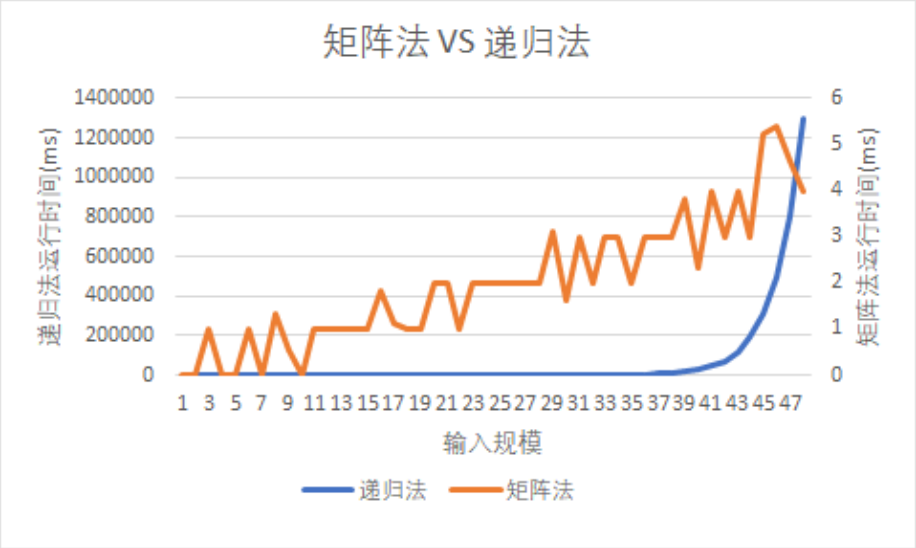

图 10: 矩阵法 VS 递归法

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。

假设 $A[i]$ 出现在 $B[j]$ 的位置的概率为

$$\begin{aligned} P(ij) &= P(i < j)P(offset = j - i) + P(j > i)P(offset = j - i + n) \\ &= \frac{n-i}{n} \cdot \frac{1}{n} + \frac{i}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n} \end{aligned}$$

(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：

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