深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验名称：分治法求最近点对问题

学院：计算机与软件学院 专业：计科

报告人：欧阳宇杰 学号：2021150143 班级：计科 2 班

同组人：欧阳宇杰

指导教师：杜智华

实验时间：2023 年 3 月 7 日~2023 年 3 月 21 日

实验报告提交时间：2023 年 3 月 21 日星期二

教务处制

一、实验目的：

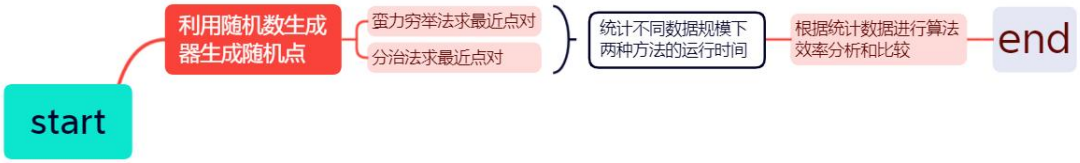
（1） 掌握分治法思想。
（2） 学会最近点对问题求解方法。

二、实验要求内容：

1) 对于平面上给定的N个点，给出所有点对的最短距离，即，输入是平面上的N个点，输出是N点中具有最短距离的两点。
2) 要求随机生成N个点的平面坐标，应用蛮力法编程计算出所有点对的最短距离。
3) 要求随机生成N个点的平面坐标，应用分治法编程计算出所有点对的最短距离。
4) 分别对N=100000—1000000，统计算法运行时间，比较理论效率与实测效率的差异，同时对蛮力法和分治法的算法效率进行分析和比较。
5) 如果能将算法执行过程利用图形界面输出，可获加分。

三、实验步骤与过程

实验流程如下图：



A. 前期工作

1) 二维点定义如下：

```
struct pi
{
    double x, y;
    pi() {}
    pi(int xx, int yy) :x(xx), y(yy) {}
};
```

图 1　　二维点的定义

2) 我使用随机数生成器 std::uniform_int_distribution 生成所需要的随机点，该随机数生成器的作用是：随机生成一个整数 i ，根据分布概率函数 P（i| a,b) = 1/（b-a+1）均匀分布在一个闭区间 [a，b]。

3) 点集去重

生成的随机点很难避免会有重复点，故需要对重复的点进行去除。

我使用 STL 中的 *unordered_set* 来去除重复点。

unordered_set 的类模板定义如下：

```
template < class Key,                    //容器中存储元素的类型
           class Hash = hash<Key>,       //确定元素存储位置所用的哈希函数
           class Pred = equal_to<Key>,   //判断各个元素是否相等所用的函数
           class Alloc = allocator<Key>  //指定分配器对象的类型
           > class unordered_set;
```

因为 *unordered_set* 容器中存储的元素为我自己定义的数据类型 pi，故默认的哈希函数 *hash<key>* 以及比较函数 *equal_to<key>* 将不再适用，只能自己设计适用该类型的哈希函数和比较函数。

下面是自己重载的 hash 函数和比较函数：

```
struct hashfunc //哈希函数
{
    size_t operator()(const pi& P) const
    {
        return size_t(P.x * 202311 + P.y);
      //return : 根据点坐标得到的哈希值
    }
};

struct eqfunc //比较函数
{
    bool operator()(const pi& p1, const pi& p2) const
    {
        return ((p1.x == p2.x) && (p1.y == p2.y));
    }
};
```

图 2　自己重载的 hash 函数和比较函数

完成上面的两个函数后就可以用 *unordered_set* 去重了

```
unordered_set<pi, hashfunc, eqfunc> hash;
for (int i = 0; i < n; i++)
{
    p[i].x = dist(& _Eng: rd);
    p[i].y = dist(& _Eng: rd);
    if (hash.find(_Keyval: p[i]) == hash.end())
        hash.insert(_Val: p[i]);
    else
        i--;
}
```

图 3　生成随机点和去除重复点的代码

代码就像上图，生成一个随机点后，用 STL 的 find 函数看 hash 里能不能找到，若是找不到，说明该点不重复，那么该点保留，并将该点加入 hash 中。若是该点在 hash 里找到了，说明该点重复，则 i--，重新生成该点的横纵坐标，再进行上面的判断。

B. 蛮力穷举法

算法设计原理：

依次遍历计算每两个点之间的距离，如果当前记录的最短距离大于此刻计算所得的两点间距离，则更新最短距离为当前计算的两点间距离，实现代码如下：

```
double bruteForce(vector<pi>& p1) {
    double ans = INF;
    for (int i = 0; i < p1.size(); i++) {
        for (int j = i + 1; j < p1.size(); j++) {
            ans = min(_Left: ans, _Right: dis(p1[i], p2: p1[j]));
        }
    }
    return ans;
}
```

图 4　蛮力穷举法求最近点对距离函数

其中 dis 是求两点间距离的函数，实现如下：

```
double dis(const pi& p1, const pi& p2) {
    return sqrt(_X: (double)(p1.x - p2.x) * (p1.x - p2.x) + (double)(p1.y - p2.y) * (p1.y - p2.y));
}
```

图 5　求两点间距离的函数

时间复杂度分析：

从上述蛮力穷举法求解最近点对问题的代码中，可以通过计算得到该算法时间复杂度为$T(n) = O(n^2)$，具体计算过程如下式所示。

$$T(n) = (n-1) + (n-2) + \ldots\ldots + 2 + 1 = \frac{n^2}{2} - \frac{1}{2} \qquad T(n) = O(n^2)$$

C. 分治法

分治法基本思想：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如何用分治法求最近点对？

按照分治法的常用分法，将图中所有的点分为大致相等的两部分，也就是做一条直线 L 将 S 分割为两个点集——左点集与右点集。

分析子问题：

问题是选取两个点：p1，p2，使得他们距离最短，子问题存在以下可能：

i. p1，p2 同时位于左边点集

ii. p1，p2 同时位于右边点集

iii. p1，p2 一个在左侧一个在右侧

对于子问题 i 和 ii，我们可以通过递归解决，难点就在于解决子问题 iii，即两点在两边的情况

探索子问题 iii 的求解：

我们可以通过递归得到左右两边点子集的最近点对距离。分别设为d_l和d_r，取$d = min(d_l, d_r)$，我们可以借助 d 来缩小我们查找的空间。

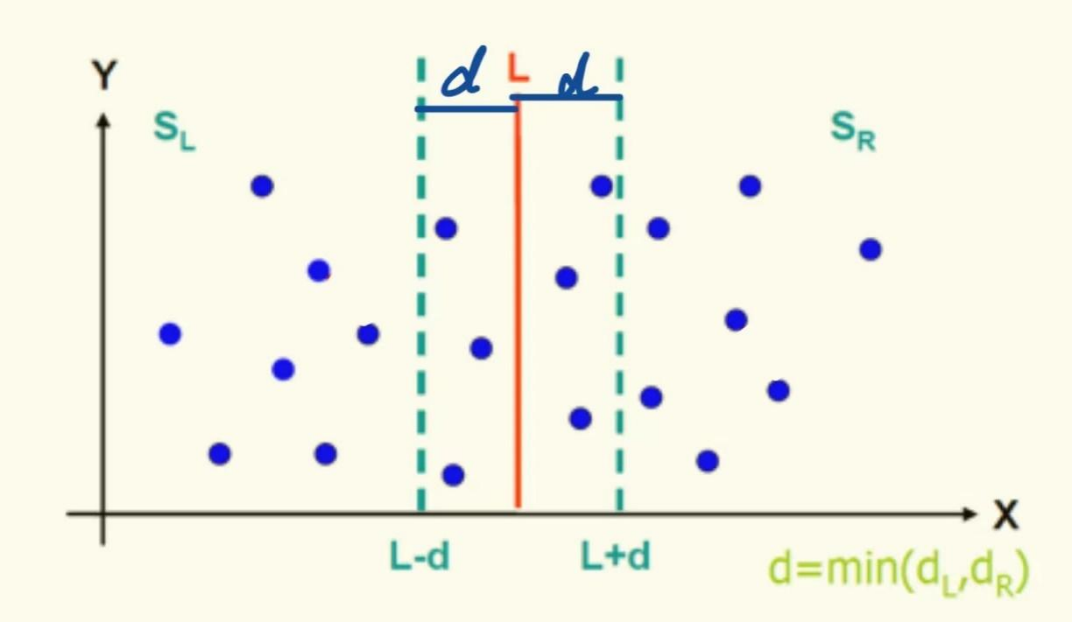


图 6　借助 d 得到的绿色矩形示意图

容易得到跨越中间线 L 的最近点对只可能存在于绿色矩形内，所以仅需要对绿色矩形内的点进行研究。

对于绿色矩形内的点，若仍然采用蛮力穷举法一一配对求距离，并不能实质性地降低时间复杂度，因为可能查找空间和原来一样。但我们可以根据鸽笼定理来对算法进行优化。

如下图，当对[L － d, L + d]中的点进行归并后，这些点在数组中关于 y 坐标已经有序。不妨假设点 p(x,y)是集合 P1(绿色矩形左边)和 P2（绿色矩形右边）中 y 坐标最小的点，p 即可能在 P1 中也可能在 P2 中。现在需要找出的是和点 p 的距离小于 d 的点，显然，这些点肯定位于[y, y+d]之间，如下图所示，即位于红色矩形中。因为根据分析，当且仅当点位于该红色矩形内时，两点间 y 坐标之差小于 d，其他不在该范围内的点 y 坐标之差必定大于 d，y 坐标之差都大于 d 了，则点对距离必定大于 d。

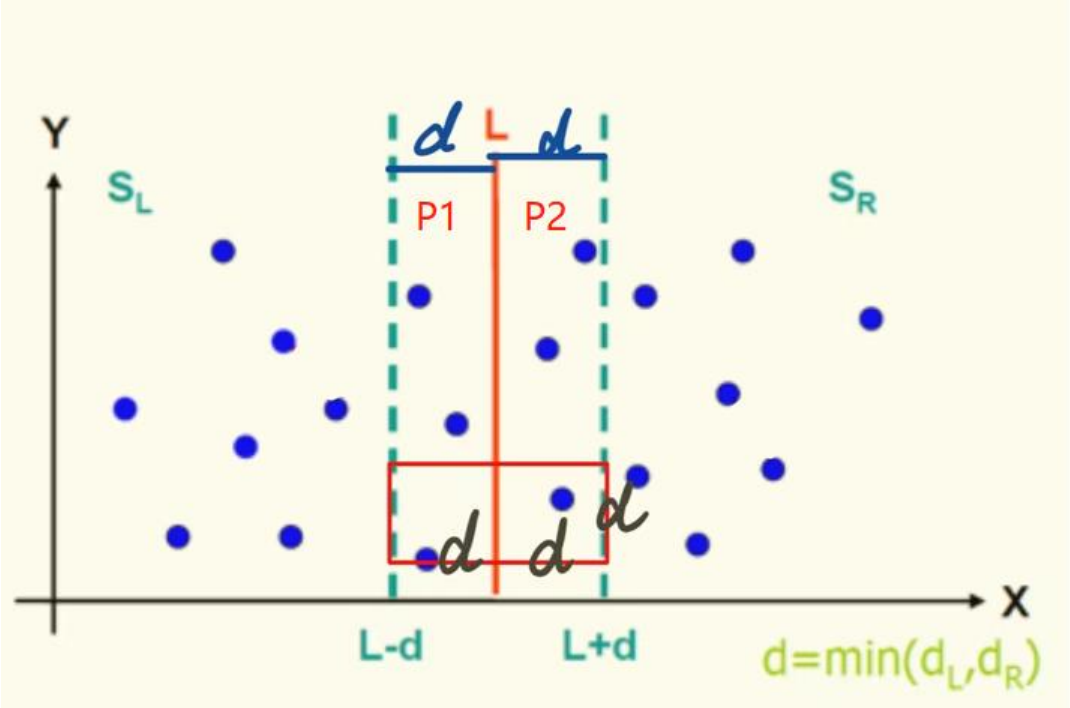


图 7　红色矩形示意图

此时，对于下图中红色矩形左半边的 d×d 正方形，因为 P1 中所有点对之间最小距离至少为 d，那么根据鸽笼原理，该正方形内最多有 4 个点（见下图）。同理知右边的 d×d 正方形内也最多有 4 个点。又因两个正方形有一条边重合，所以有两个点为同一点，下图即展示了点最多的情形。那么和点 p 的距离小于 d 的点不会超过 5 个，因为必须满足在左点集和右点集中两点间点对距离至少为 d。

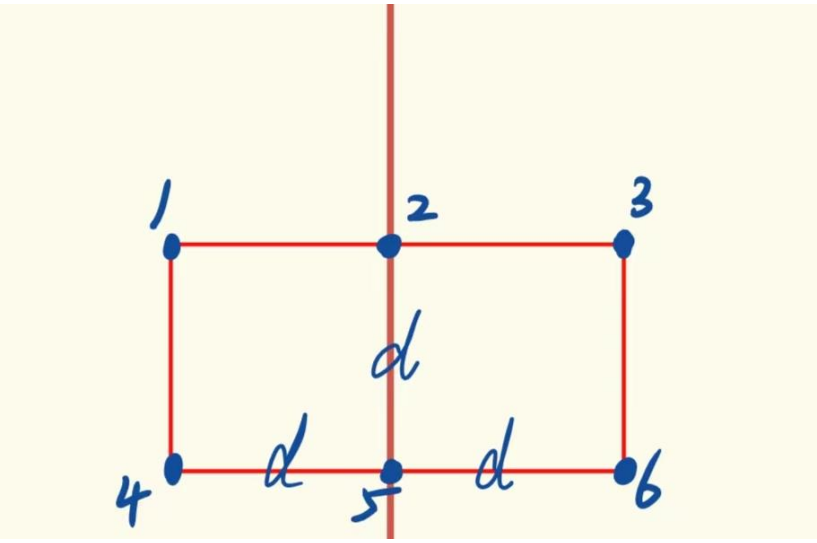


图 8　　红色矩形点最多的情形

又由于[L - d, L + d]中的点已经根据 y 升序排好了序。所以我们在判断时只需要让点 p(x,y)与其之后的五个点也就是 y 坐标在它之上的五个点进行距离计算，如果当前计算的距离小于最小距离，则更新最小距离即可。然后 p 变为原 p 点的上面一个点，进行类似操作，以此类推。

时间复杂度分析：

a) 由于分治函数中当递归至仅有两个点时根据 y 坐标排序后返回有序数组，故后面每次合并时待合并的两个小数组arr_l和arr_r都是 y 坐标有序的。因此合并这两个小数组的操作很简单，和归并排序中的合并操作很相似，只不过这里比较的是 y 坐标。
因此归并操作的时间复杂度为 O(n)。

b) 又由于之前的优化，对于绿色矩形内的每个点，仅需要计算 y 坐标在它之上的五个点与其的距离，故对于每个点仅需要常数级的时间，即时间复杂度为 O(1)，故对绿色矩形左边的所有点都进行这样的枚举的时间复杂度为 O(n)。

综上所述以上两点，对于每层递归，消耗的时间为 O(n)。

则有递推式如下：T(n) = 2T(n/2) + n

运用主定理法进行算法时间复杂度的计算，可得算法的时间复杂度为 $O(nlogn)$。运算过程如下图所示。

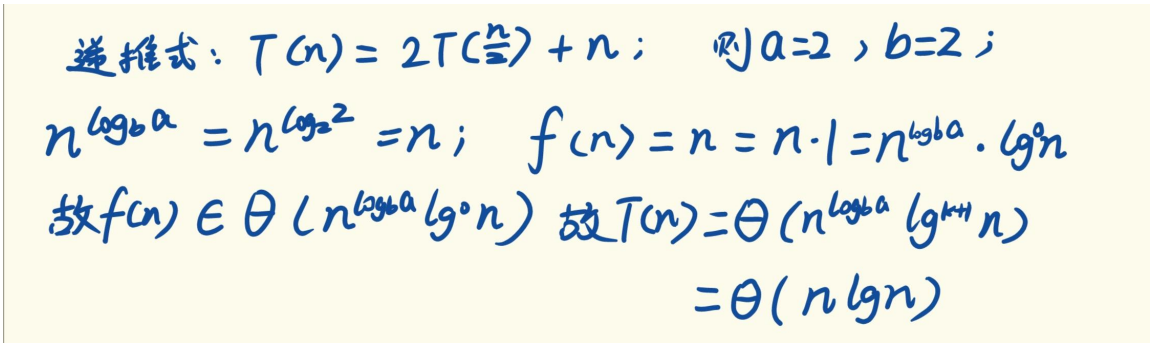


图 9　主定理法计算算法时间复杂度过程

下面我将展示如何编程实现运用分治法求最近点对。

运用分治法求最近点对的关键函数：

1．函数声明部分。

```
double getmin(int l, int r)
```

2．仅有一点时，返回无穷大。

```
if (l == r)
    return INF;
```

3．当有两个点时，将两点根据 y 升序排序，再直接返回两点间距离。

```
if (r - l == 1) {
    temp1 = p[l];
    temp2 = p[r];
    p[l] = temp1.y < temp2.y ? temp1 : temp2;
    p[r] = temp1.y > temp2.y ? temp1 : temp2;
    return dis(p1: p[l], p2: p[r]);
}
```

4．当点的个数多于 2 时，分治。

首先根据数组左右端点求得分治的中点，以此点将点集分为左点集和右点集。

```
int mid = (l + r) >> 1;
```

5．递归求得左点集的最近点对距离d_l和右点集的最近点对距离d_r，并求得两者中的最小值d。另外，递归之后数组的左半边和右半边都已有序，故接着的就是合并操作。

```
double l_min = getmin(l, r: mid);
double r_min = getmin(l: mid + 1, r);
double base = min(_Left: l_min, _Right: r_min);
Merge(left: l, mid, right: r);
```

6．将绿色矩形内的点装进点集 v 中。

```
vector<pi> v;
for (int i = l; i <= r; i++) {
    if (p[mid].x - p[i].x < base || p[i].x - p[mid].x < base) {
        v.push_back(_Val: p[i]);
    }
}
```

7．对于绿色矩形内的每个点，计算 y 坐标在它之上的五个点与其的距离。

如果两个点的 y 坐标之差已经大于当前最小点对距离，又因为点是按 y 坐标升序排序的，之后的差距只会增大，故之后的点都不符合条件，故直接 break 出内层循环。

如果计算得到的两点间距离小于当前最小点对距离，则更新当前最小点对距离。

```
int size = v.size();
for (int i = 0; i < size - 1; i++)
{
    for (int j = i + 1; j < size && j < i + 6; j++)
    {
        if ((v[j].y - v[i].y) > base)
            break;
        base = min(_Left: base, _Right: dis(p1: v[i], p2: v[j]));
    }
}
return base;
```

8. 关于合并操作，即关于 Merge 函数。

首先完成两侧点的深复制，将左点集复制到 L 数组中，将右点集复制到 R 数组中，接着利用类似归并排序的方法将点根据 y 升序排序，如果左点集或右点集有剩余元素，就将剩余元素直接放入即可。

```
void Merge(int left, int mid, int right) {
    int n1 = mid - left + 1;//左侧点集大小
    int n2 = right - mid;//右侧点集大小
    pi* L = new pi[n1];
    pi* R = new pi[n2];
    //完成两侧点集的复制
    for (int i = 0; i < n1; i++)
        L[i] = p[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = p[mid + 1 + i];
    //算法利用分治法，利用类似归并排序的方法将点根据y升序排序
    int i = 0, j = 0, k;
    for (k = left; i < n1 && j < n2; k++) {
        if (L[i].y < R[j].y)//根据y升序来排
            p[k] = L[i++];
        else
            p[k] = R[j++];
    }
    //如果左点集或右点集有剩余元素，就将剩余元素直接放入即可。
    while (i < n1)
        p[k++] = L[i++];
    while (j < n2)
        p[k++] = R[j++];
}
```

D. 算法效率分析与比较

正确性检验：

- 经过我多次实践检验，通过蛮力穷举法和分治法求得的最短距离均相同，互相验证了这两个算法的正确性。

1. 蛮力穷举法

对于蛮力穷举法，选取数据规模为：10、20、30、40、50、60、70，单位为万的数据规模进行算法运行时间测试。对于每一个数据规模，都随机产生 10 组测试样本来测试，取 10 次的运行时间平均值来作为每一个数据规模下，算法实际执行所需的时间值。

我以输入规模为 400000 的数据运行时间为基准点，通过 $\frac{n_1^2}{n_2^2} = \frac{time1}{time2}$ 的比例关系，计算得到各实验数据中算法执行时间的理论值。

其中差别比例为（实际值-理论值）/实际值。

数据规模/个	100000	200000	300000	400000	500000	600000	700000
实际时间/s	11.512	45.735	102.6588	182.9248	286.6695	412.8073	559.403
理论时间/s	11.4328	45.7312	102.8952	182.9248	285.82	411.5808	560.2072
差别比例	0.69%	0.008%	-0.23%	0%	0.296%	0.297%	-0.144%

下面为蛮力穷举法的理论效率曲线和实测效率曲线对比图：

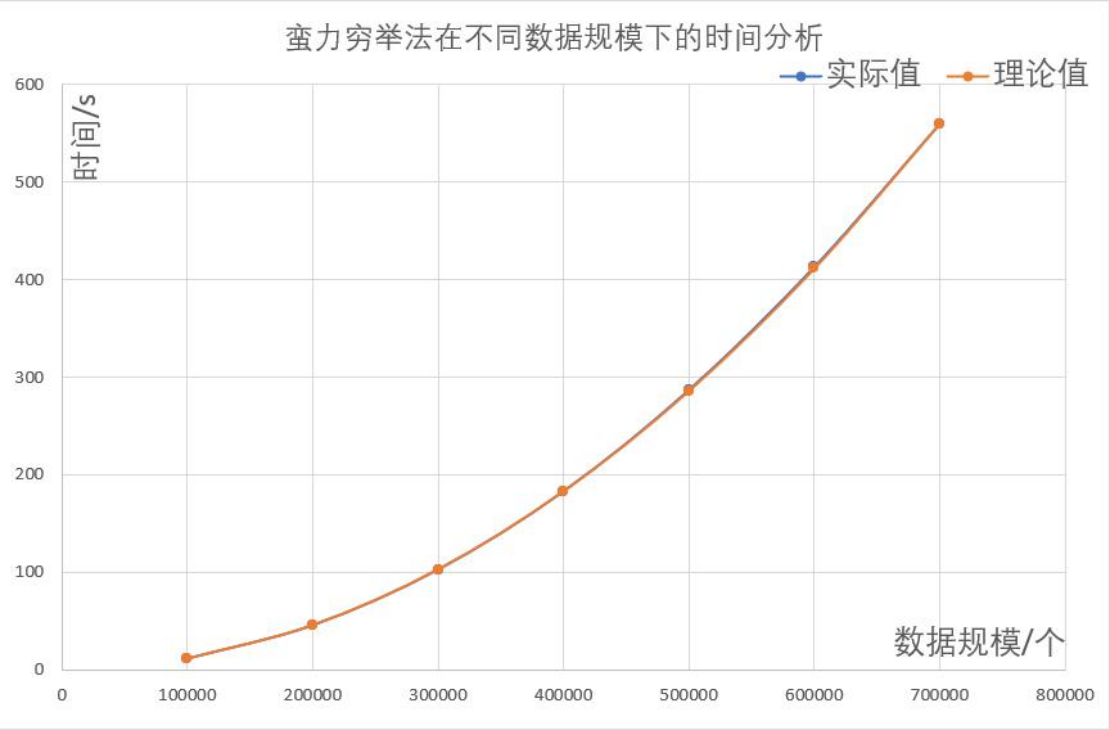


图 10　蛮力穷举法的理论效率曲线和实测效率曲线对比图

通过上面的表格和曲线图可知，蛮力穷举法的实际值曲线和理论值曲线的整体拟合效果非常好，这也表明了蛮力穷举法的时间复杂度就是 $O(n^2)$。

2. 分治法

对于分治法选取数据规模为：10、20、30、40、50、60、70、80、90、100，单位为万的数据规模进行算法运行时间测试。对于每一个数据规模，都随机产生 10 组测试样本来测试，取 10 次的运行时间平均值来作为每一个数据规模下，算法实际执行所需的时间值。

我以输入规模为 500000 的数据运行时间为基准点，通过 $\frac{n_1 logn_1}{n_2 logn_2} = \frac{time1}{time2}$ 的比例关系，计算得到各实验数据中算法执行时间的理论值。

其中差别比例为（实际值-理论值）/实际值

数据规模/个	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
实际时间/s	0.0564	0.1176	0.1688	0.2524	0.3106	0.3628	0.456	0.5114	0.58	0.6396
理论时间/s	0.0545	0.1156	0.1791	0.2443	0.3106	0.3779	0.446	0.5148	0.5841	0.654
差别比例	3.37%	1.70%	-6.10%	3.21%	0%	-4.16%	2.19%	-0.66%	-0.71%	-2.25%

下面为分治法的理论效率曲线和实测效率曲线对比图：

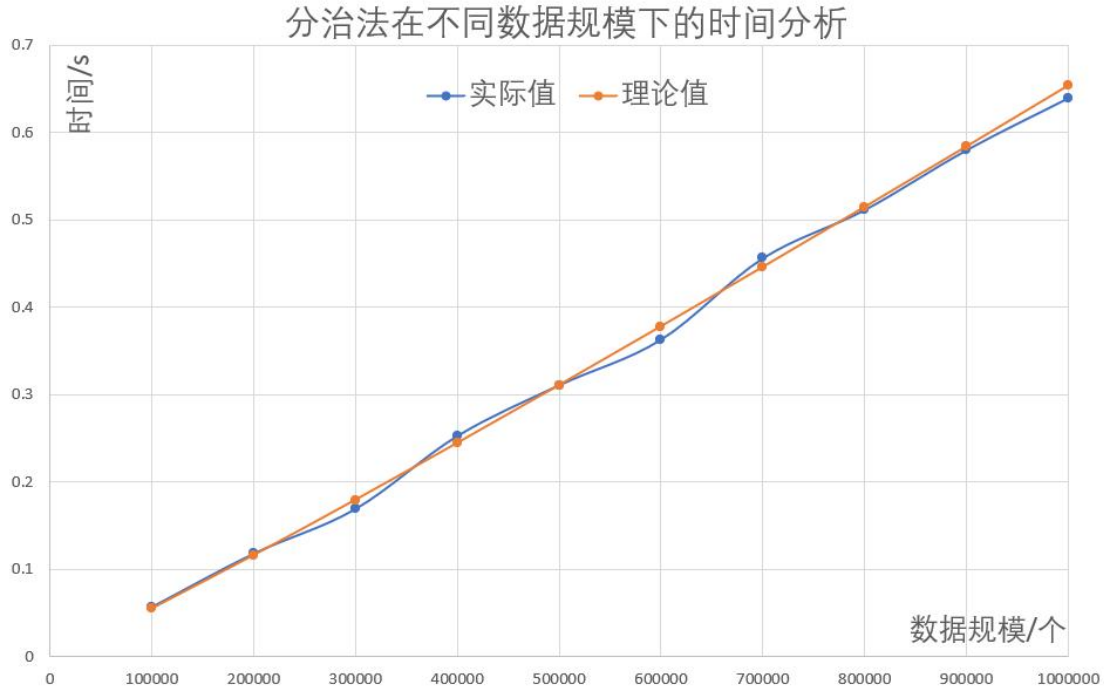


图 11　分治法的理论效率曲线和实测效率曲线对比图

通过上面的表格和曲线图可知，分治法的实际值曲线和理论值曲线的整体拟合效果挺不错的，在有的数据规模下略有差别我觉得是测得的时间难免有波动的原因。总的来说基本拟合，这也表明了分治法的时间复杂度就是 $O(nlogn)$。

两种算法的效率分析及比较：

由于蛮力穷举法的最大数据规模为 70 万，故我下面的对比图的最大数据规模也为 70 万。

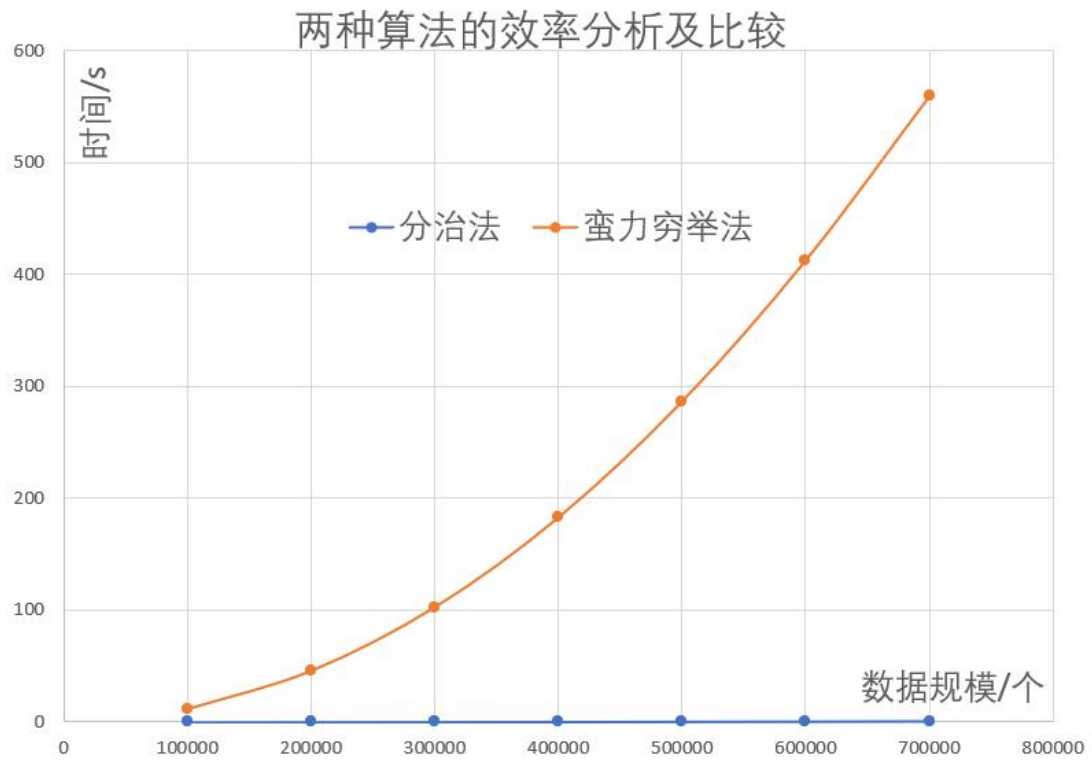


图 12　两种算法的效率比较图

可以看到，从实验要求的起始数据规模 10 万开始直到 70 万，分治法的效率都远远优于蛮力穷举法。一直到 70 万的数据规模，分治法的时间消耗都未超过 1 秒，可蛮力穷举法的时间消耗已经超过了 500 秒。较蛮力穷举法的时间消耗，分治法的时间消耗都可以忽略不计。

四、实验总结与体会

A. 验证更优算法的正确性可以根据较差算法的结果。具体到本题中，可以用蛮力穷举法算出正确结果，再看分治法是否得出了与蛮力穷举法相同的正确答案来验证写的分治算法的正确性。

B. 遇到问题的时要积极思考，想想可不可以运用数学或逻辑思维与推理，来解决问题或者简化问题。如在此题中，运用了数学逻辑思维将要分析的点局限在了 2d*d 的绿色矩形内，并根据鸽笼原理，使得绿色矩形内的每个点，仅需要计算 y 坐标在它之上的五个点与其的距离，成功地将算法的时间复杂度降低了。

C. 对于许多问题，都可以经过数学或逻辑分析后根据分治法进行求解，从而有效地降低问题的时间复杂度，提高解决问题的效率。如：Strassen 矩阵乘法，最大子数组，合并排序等。

D. 通过此次实验，我掌握了分治法的思想，并学会了如何将它应用到最近点对问题的求解中。同时，我也深刻理解了算法优化的重要性，通过调整算法的实现方式和数据结构，可以大幅提高算法的效率和性能。

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。