算法设计与分析
最接近点对实验报告

编译环境： Microsoft Visual Studio 2019

Windows SDK **版本：** 10.0

平台工具集： Visual Studio 2019（v142）

语言： C++

一、基本原理：

采用分治法，先对平面中所有点按照 x 坐标进行排序，用 S 记排序后的点集，求出所有点 x 坐标值的中位数，记为 m，然后用直线 $l: x=m$ 把点集 S 划分为大致相等的两个子集 S_1 和 S_2，且 $S_1=\{p\in S\,|\,x(p)\le m\}$； $S_2=\{p\in S\,|\,x(p)>m\}$。于是原问题变为：

（1）求 S1 中最近距离的两个点；

（2）求 S2 中最近距离的两个点；

（3）由（1）、（2）得到 S 中的最接近点。

二、问题描述：

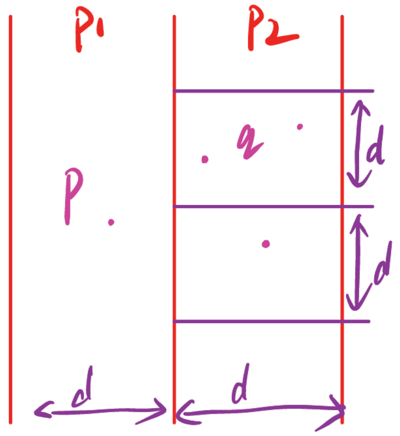
给定 n 个点(xi,yi) 1<=i<=n，找出其中距离最近的两个点，并简单讨论到 3 维情形下的推广。

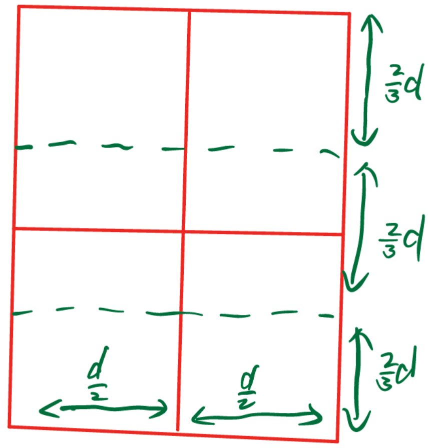
三、算法分析与实现：

Ps：图不好画，我在 ipad 上手写了部分，然后截图到了 word 中

（1）如果组成 S 中的最近距离点都在 S1 中，或都在 S2 中，可以通过递归很容易求解。

（2）如果组成 S 中的最近距离点分别在 S1 和 S2 中。设 d1 和 d2 分别为 S1 和 S2 中的最小距离，设 d=min{d1,d2}。在合并时，对于 S 中距离小于 d 的两点 p 和 q 必定满足：p 和 q 分别属于 S1 和 S2，在此设 p 属于 S1,q 属于 S2；令 P1 和 P2 分别表示距直线 l 的左边和右边的宽为 d 的两个区间，则 p 属于 P1,q 属于 P2，如下左图所示：





因此 P1 中的所有点和 P2 中的所有点在最坏情况下有 n^2/4 对最接近点对的候选者，但是 P1 和 P2 中的点具有稀疏性质，使得不必检查所有这 n^2/4 个候选者，而最多只需要检查 6*n/2=3n 个候选者。P1 和 P2 中的点的稀疏性表现在：对于 P1 中的任意一点 p，P2 中与其构成最接近点对的候选者的点必定落在一个 d*2d 的矩形中。利用鸽舍原理，这样的候选点最多只有 6 个，如上右图所示。6 个虚线矩形中每个矩形中最多只有一个候选点，在课本上已经证明。因此，将 P1 和 P2 和中所有中点按其坐标 y 排好序，则对 P1 中所有点最多只要检查 P2 中排好序的相继 6 个点。此时合并的计算复杂度需要 O(n)，所以二维情况下的最接近点对可以在的 O(n*logn)时间内求得。

算法效率：

由前面的问题描述可知，整个问题的求解过程需要两次排序，一次用于分治划分，一次用于合并时的点对扫描，所以用于排序的时间为 O(n*logn)。由于在合并过程中消耗时间为 O(n)，因此用分治法求解问题所耗费的时间 T(n)可以用下式表达：

$$T(n)=\begin{cases}O(1) & n<4\\ 2T(n/2)+O(n) & n\geq 4\end{cases}$$

计算得出 T(n)= O(n*logn)，结合排序时所消耗的时间，整个算法可以在 O(n*logn)时间内完成。

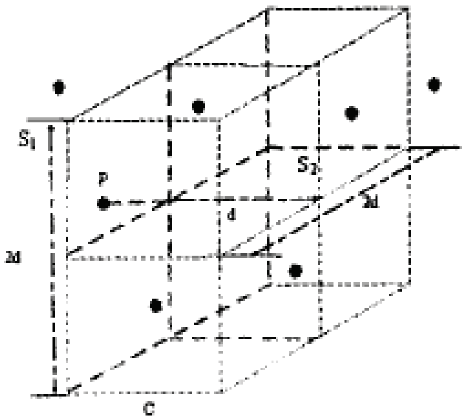
对于该问题推广到 3 维的情况，与 2 维类似：

先对三维空间中所有点按照 y 坐标进行排序，用 S 记排序后的点集。求出所有点 y 坐标值的中位数，记为 m，然后用平面 y=m 把点集 S 划分为大致相等的两个子集 S1 和 S2，且 S1={P 属于 S|y(p)<=m}；S2={P 属于 S|y(p)>m}。这样把求 S 中的最接近点对转变为分别求 S1,S2 中的最接近点对和其合并后的最接近点对。

递归地在 S1 和 S2 中求解最接近点对，设 d1 和 d2 分别为 S1 和 S2 中的最

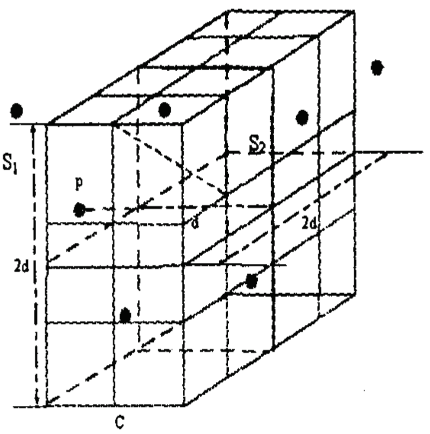
小距离，设 d=min(d1,d2)。

合并时，对于 S 中距离小于 d 的两点 p 和 1 必定满足：p 和 q 分别属于 S1 和 S2，在此设 p 属于 S1,q 属于 S2。那么 p 和 q 距平面的距离均小于 d。因此，若令 P1 和 P2 分别表示距平面的左边和右边的宽为 d 的两个长条形区间，则 p 属于 S1,q 属于 S2，如下图所示：



此时，P1 中的所有点和 P2 中的所有点在最坏情况下有 n^2/4 对最接近点对的候选者。但是 P1 和 P2 中的点与二维情况下具有类似的稀疏性质，使得不必检查所有这 n^2/4 个候选者。

P1 和 P2 中的点的稀疏性表现在：对于 P2 中的任意一点 p，P2 中与其构成最接近点对的候选者的点必定落在一个 d*2d*2d 的长方体中，如图 3 左所示；利用鸽巢原理，这样的候选点最多只有 24 个，如下图所示：



24 个小长方体中每个长方体中最多只有一个候选点。因为对于每一个小长方体(d/2)*(d/2)*(2d/3)中如果有 2 个以上 S 中的点，设 u 和 v 是这样的 2 个点，则 $(x(u)-x(v))^2+(y(u)-y(v))^2+(z(u)-z(v))^2=\frac{17}{18}d^2$

因此，distance(u,v)<d，这与前面 d 的意义相矛盾。也就是说长方体中最多只有 24 个 S 中的点。

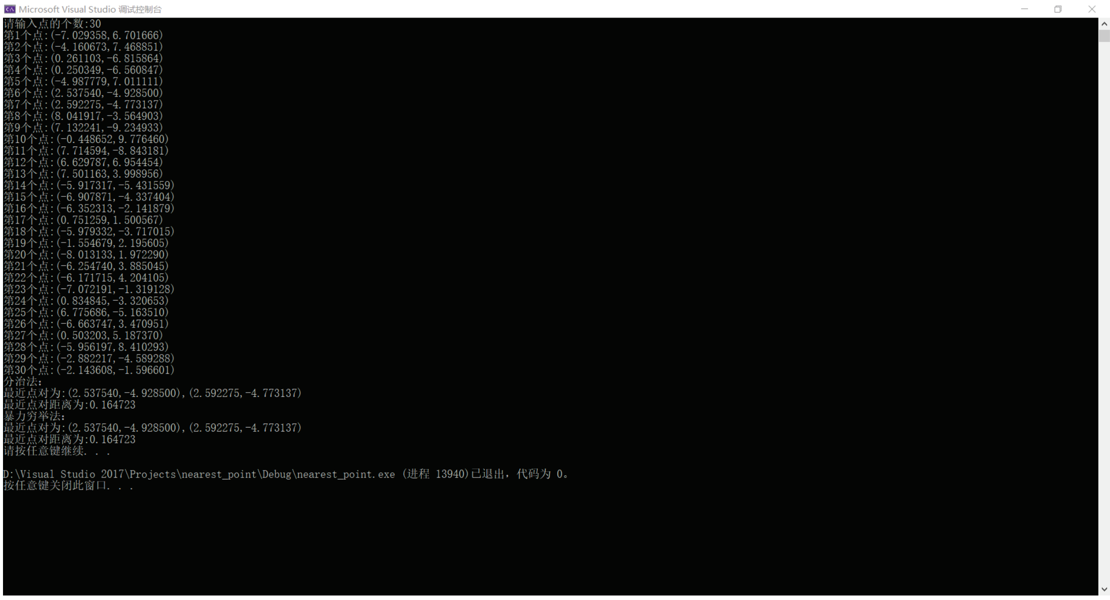
因此，在分治法的合并过程中，最多只需要检查 24*n/2=12n 个候选者，而不 n^2/4 个候选者。

四、测试样例与实验总结：

总结：

分治法可以将时间复杂度很高的问题拆分为几个小问题，从而降低时间复杂度。使用分治法的核心是问题可以分为和主问题同样的解法，只是规模变小的子问题，不断拆分，直到子问题可以非常简单的求解，最终合并子问题的解，就是主问题的解，主要要用到递归函数，求时间复杂度可以用主定理计算。

输出：



五、附录：

源码：

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<ctime>
#include<random>
using namespace std;

class point
{
```

```
public:
    double x;
    double y;
    point(double x, double y) :x(x), y(y) {}
    point() { return; }
};

bool compare_x(const point& A, const point& B)//比较x坐标，A的x小返回true，A和B相等或B
的x小返回false
{
    return A.x < B.x;
}

bool compare_y(const point& A, const point& B)//比较y坐标，A的y小返回true，A和B的y相等
或B的y小返回false
{
    return A.y < B.y;
}

double distance(const point& A, const point& B)//返回A和B之间的距离
{
    return sqrt(pow(A.x - B.x, 2) + pow(A.y - B.y, 2));
}

double merge(vector<point>& points_y, double min_d, int mid, pair<point, point>&
min_point)//合并函数，将左右区域距离的最小值与中间区域的6个点比较
{
            //points为点的集合；dis为左右的最近点距离；mid为x坐标排序后，点集合中中间
点的索引值；min_point为求得的最近点对
    vector<point> left, right;
    for (int i = 0; i < points_y.size(); i++)//搜集左右两边符合条件的点
    {
        if (points_y[i].x <= points_y[mid].x && points_y[i].x > points_y[mid].x -
min_d)
            left.push_back(points_y[i]);
        else if (points_y[i].x > points_y[mid].x && points_y[i].x < points_y[mid].x +
min_d)
            right.push_back(points_y[i]);
    }
    int j = 0;//右侧点的下界
    for (int i = 0; i < left.size(); i++)//遍历左边的点集合，与右边符合条件的计算距离
    {
        for (; j < right.size(); j++)
        {
```

```
            if (right[j].y >= left[i].y - min_d)
            {
                break;
            }
        }
        for (int l = 0; l < 6 && l + j < right.size(); l++)//遍历右边的6个点
        {
            if (distance(left[i], right[j+l]) < min_d)
            {
                min_d = distance(left[i], right[j+l]);
                min_point.first = left[i];
                min_point.second = right[j+l];
            }
        }
    }
    return min_d;
}

double closest_point(vector<point>& points, vector<point>& points_y,pair<point,point>
&min_point)//递归求解points中的最近点对，返回最近点对距离，min_point存储这个点对
{
    if (points.size() == 2)//两个点
    {
        min_point.first = points[0];
        min_point.second = points[1];
        return distance(points[0], points[1]);
    }
    if (points.size() == 3)//三个点
    {
        double min_distance = min(distance(points[0], points[1]),
min(distance(points[0], points[2]), distance(points[1], points[2])));
        if (distance(points[0], points[1])==min_distance)
        {
            min_point.first = points[0];
            min_point.second = points[1];
        }
        else if(distance(points[0], points[2]) == min_distance)
        {
            min_point.first = points[0];
            min_point.second = points[2];
        }
        else
        {
            min_point.first = points[1];
```

```
            min_point.second = points[2];
        }
        return min_distance;
    }
    pair<point, point> temp_min_point1, temp_min_point2;
    int mid = (points.size() >> 1) - 1;//size为偶数，mid为中点左边；size为奇数，mid为中点-1
    double d_left, d_right, min_d;
    vector<point> left(mid + 1), right(points.size() - mid - 1), left_y, right_y;//定义两个vector：left，right
    copy(points.begin(), points.begin() + mid + 1, left.begin());//复制左边区域点集合到left
    copy(points.begin() + mid + 1, points.end(), right.begin());//复制右边区域点集合到right
    for (int i = 0; i < points_y.size(); i++)//将按y排好序的数组以point[mid]分为两部分，分完后还是按y排好序的
    {
        if (points_y[i].x <= points[mid].x)
            left_y.push_back(points_y[i]);
        else
            right_y.push_back(points_y[i]);
    }
    d_left = closest_point(left, left_y, temp_min_point1);
    d_right = closest_point(right, right_y, temp_min_point2);
    min_d = min(d_left, d_right);
    if (d_left == min_d)
    {
        min_point = temp_min_point1;
    }
    else
    {
        min_point = temp_min_point2;
    }
    return merge(points_y, min_d, mid, min_point);
}

double closest_point2(vector<point> p, pair<point, point>& min_point)//暴力穷举法
{
    double minDistance = 99999;
    for (int i = 0; i < p.size(); i++)
    {
        for (int j = 0; j < p.size(); j++)
        {
            double d = distance(p[i], p[j]);
```

```
            if (d - 0 < 1e-4)
            {
                continue;
            }
            if (d < minDistance)
            {
                minDistance = d;
                min_point = make_pair(p[i], p[j]);
            }
        }
    }
    return minDistance;
}
int main()
{
    int count;
    double x, y, min_distance;
    vector<point> points;
    vector<point> points_y;
    pair<point, point> min_point;
    default_random_engine e(time(NULL));
    uniform_real_distribution<double> u(-10, 10);
    cout << "请输入点的个数:";
    cin >> count;
    for (int i = 0; i < count; i++)
    {
        printf("第%d个点:", i+1);
        point p(u(e), u(e));
        printf("(%lf,%lf)\n", p.x, p.y);
        points.push_back(p);
        points_y.push_back(p);
    }
    sort(points.begin(), points.end(), compare_x);//把所有的点按x从小到大排序
    sort(points_y.begin(), points_y.end(), compare_y);//把左边的点按y从小到大排序
    min_distance = closest_point(points, points_y, min_point);
    cout << "分治法： " << endl;
    printf("最近点对为:(%lf,%lf),(%lf,%lf)\n最近点对距离为:%lf\n", min_point.first.x,
min_point.first.y, min_point.second.x, min_point.second.y, min_distance);
    cout << "暴力穷举法： " << endl;
    min_distance = closest_point2(points, min_point);
    printf("最近点对为:(%lf,%lf),(%lf,%lf)\n最近点对距离为:%lf\n", min_point.first.x,
min_point.first.y, min_point.second.x, min_point.second.y, min_distance);
    system("pause");
    return 0;
```

```
}
```