求平面上 n 个顶点的最近点对问题
实验报告

一、实验目的

通过实践，加深理解在 n≥ 2 个点的集合 Q 中寻找最近点对的问题，提高编程能力。

二、实验内容

编程实现求平面上 n 个顶点的最近点对问题。

三、实验步骤

解决本问题可以通过分治算法递归实现。

1.分治算法的输入及“预排序”

算法每一次递归调用的输入为子集 P∈Q 以及数组 X 和 Y，每个数组均包含输入子集 P 的所有点，对数组 X 中的点排序，使其 x 坐标单调递增，类似地，对数组 Y 中的点排序，使其 y 坐标单调递增。我们对数组 X 和 Y 进行“预排序”，这样就无需在每次递归调用中都排序。

预排序的实现如下所示：

```
vector<double> X;
vector<double> Y;
for(int i=0; i<P.size(); i++)
{
    X.push_back(P[i].first);
    Y.push_back(P[i].second);
}
sort(X.begin(), X.end());
sort(Y.begin(), Y.end());
Point_Sorting(P);    //对输入点对排序
```

2.分治算法的具体实现

（1）递归出口

输入为 P、X、Y 的递归调用首先检查是否有|P|≤3 成立。如果有，则通过暴力搜索实现：对所有$\binom{P}{2}$个点对进行检查，并返回最近点对。

```
if(P.size()<=3) //递归出口
{

    Point_Pair pp;
    pair<double, double> p1 = P[0], p2 = P[1], p3 = P[2];
    double dist1 = sqrt(pow(p1.first-p2.first,2) + pow(p1.second-p2.second,2));
    double dist2 = sqrt(pow(p1.first-p3.first,2) + pow(p1.second-p3.second,2));
    double dist3 = sqrt(pow(p2.first-p3.first,2) + pow(p2.second-p3.second,2));
    double min_dist;
    min_dist = (dist1<dist2) ? dist1 : dist2;
    min_dist = (min_dist<dist3) ? min_dist : dist3;
    if(min_dist==dist1)
    {
        pp.first = p1;
        pp.second = p2;
    }
    else if(min_dist==dist2)
    {
        pp.first = p1;
        pp.second = p3;
    }
    else{
        pp.first = p2;
        pp.second = p3;
    }

    result.first = pp;
    result.second = min_dist;
}
```

（2）分治算法的分解

当|P|>3 时，找出一条垂直线 l，它把点集 P 对分为满足下列条件的两个集合P_L和P_R：使得$P_L = \lceil P/2 \rceil$，$P_R = \lfloor P/2 \rfloor$，$P_L$中的所有点都在直线 l 上或在 l 的左侧，$P_R$中的所有点都在直线 l 上或在 l 的右侧。数组 X 被划分为两个数组X_L和X_R，分别包含P_L和P_R中的点，并按 x 坐标单调递增的顺序排好序。对数组 Y 的处理与数组 X 类似。

```
else{
    //将输入点集P划分为两个子点集P_L和P_R
    vector<pair<double, double> > P_L;
    vector<pair<double, double> > P_R;
    int middle = X.size()/2;
    if(X.size()%2==1)
        middle += 1;
    for(int i=0; i<middle; i++)  //生成P_L
    {
        P_L.push_back(P[i]);
    }
    for(int i=middle; i<P.size(); i++)  //生成P_R
    {
        P_R.push_back(P[i]);
    }
    vector<double> X_L;
    vector<double> X_R;
    vector<double> Y_L;
    vector<double> Y_R;

    for(int i=0; i<P_L.size(); i++)  //生成X_L和Y_L
    {
        X_L.push_back(P_L[i].first);
        Y_L.push_back(P_L[i].second);
    }
    for(int i=0; i<P_R.size(); i++)  //生成X_R和Y_R
    {
        X_R.push_back(P_R[i].first);
        Y_R.push_back(P_R[i].second);
    }
```

(3)子问题的解决

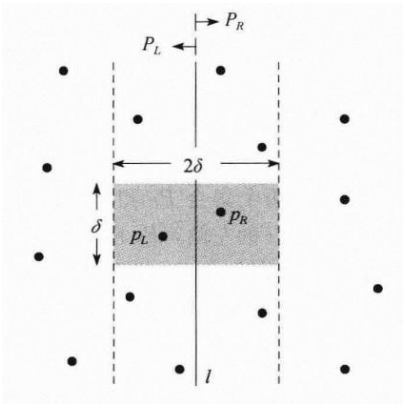
对第(2)步中划分出的P_L和P_R各自进行一次递归调用，一次找出P_L中的最近点对，另一次找出P_R中的最近点对。第一次调用的输入为子集P_L、数组X_L和Y_L，第二次调用的输入为子集P_R、数组X_R和Y_R。令P_L和P_R返回的最近点对的距离分别为δ_L和δ_R，并且置$\delta=\min(\delta_L,\delta_R)$。

```
Result result_L = Find_Most_Close_Point_Pair(P_L, X_L, Y_L); //对P_L递归
Result result_R = Find_Most_Close_Point_Pair(P_R, X_R, Y_R); //对P_R递归
double sigma1 = result_L.second;
double sigma2 = result_R.second;
double sigma = min(sigma1, sigma2);
result = (sigma==sigma1) ? result_L : result_R;
```

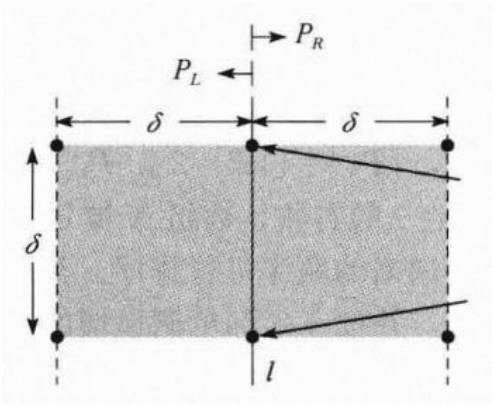
（4）分治算法的合并

最近点对要么是某次递归调用找出的距离为δ的点对，要么是P_L中的

一个点与P_R中的一个点组成的点对。算法确定是否存在距离小于δ的一个点对，一个点位于P_L中，一个点位于P_R中。注意，如果存在这样的一个点对，则点对中的两个点与直线 l 的距离必定都在δ单位之内。因此，它们必定都处于以直线 l 为中心、宽度为 2δ的垂直带形区域内。



(a)带形区域



(b)P 中至多有 8 个点位于δ × 2δ的矩形区域内

为了找出这样的点对(如果存在)，算法要做如下工作：

①建立一个数组Y'，它是把数组 Y 中所有不在宽度为 2δ的垂直带形区域内的点去掉后得到的数组。数组Y'与 Y 一样，是按 y 坐标排序的。

②对数组Y'中的每个点 p，算法试图找出Y'中距离 p 在δ单位内的点。在Y'中仅需考虑紧随 p 后的 7 个点。算法计算出从 p 到这 7 个点的距离，并记录下Y'中的所有点对中最近点对的距离δ'。

③如果$\delta' < \delta$,则垂直带形区域内的确包含比根据递归调用所找出的最近距离更近的点对，于是返回该点对及其距离δ'。否则，返回函数的递归调用中发现的最近点对及其距离。

```
double left_end, right_end;  //设置带形区域的左右端点
if(middle%2==1)  //如middle = 3  -> 0 1 2 3 4
{
    left_end = X[middle-1] - 2*sigma;
    right_end = X[middle-1] + 2*sigma;
}
else
{
    left_end = (X[middle-1]+X[middle])/2 - 2*sigma;
    right_end = (X[middle-1]+X[middle])/2 + 2*sigma;
}

for(int i=0; i<P.size(); i++)  //找Y'中的点
{
    if(P[i].first>=left_end&&P[i].first<=right_end)
    {
        P2.push_back(P[i]);
    }
}

for(int i=0; i<P2.size(); i++) //一个点在P_L中，一个点在P_R中
{
    for(int j=i+1; j<P2.size()&&j<i+8; j++)
    {
        double dist = sqrt(pow(P2[i].first-P2[j].first,2) + pow(P2[i].second-P2[j].second,2));
        if(dist<result.second) //更新最近点对
        {
            result.first.first = P[i];
            result.first.second = P2[j];
            result.second = dist;
        }
    }
}
```

3. 使用暴力搜索算法对分治法的结果进行验证：

暴力搜索算法主要过程如下：

```
Result result;
result.first.first = P[0];
result.first.second = P[1];
result.second = 100;
for(int i=0; i<P.size(); i++)
{
    for(int j=i+1; j<P.size(); j++)
    {
        double dist = sqrt(pow(P[i].first-P[j].first,2) + pow(P[i].second-P[j].second,2));
        if(dist<result.second)
        {
            result.first.first = P[i];
            result.first.second = P[j];
            result.second = dist;
        }
    }
}
```

四、实验结果

输入点集 1：

```
vector<pair<double, double> > P(8, pair<double,double>(0,0));
P[0] = make_pair(1.1, 2.0);
P[1] = make_pair(3.1, 1.0);
P[2] = make_pair(4.4, 5.3);
P[3] = make_pair(2.6, 3.2);
P[4] = make_pair(2.6, 4.6);
P[5] = make_pair(5.3, 5.2);
P[6] = make_pair(3.2, 4.5);
P[7] = make_pair(5.3, 4.7);
```

分治法结果：

```
The shortest-distance Point-Pair is:
(5.3,5.2) and (5.3,4.7)
The shortest dist is:0.5
```

暴力搜索结果：

```
brutal search:
The shortest-distance Point-Pair is:
(5.3,5.2) and (5.3,4.7)
The shortest dist is:0.5
```

输入点集 2：

```
vector<pair<double, double> > P(12, pair<double,double>(0,0));
P[0] = make_pair(1.1, 2.0);
P[1] = make_pair(3.1, 1.0);
P[2] = make_pair(4.4, 5.3);
P[3] = make_pair(2.6, 3.2);
P[4] = make_pair(2.6, 4.6);
P[5] = make_pair(5.3, 5.2);
P[6] = make_pair(3.2, 4.5);
P[7] = make_pair(5.3, 4.7);
P[8] = make_pair(4, 6.6);
P[9] = make_pair(8, 7.9);
P[10] = make_pair(6, 3.8);
P[11] = make_pair(4, 6.9);
```

分治法结果：

```
The shortest-distance Point-Pair is:
(2.6,4.6) and (4,6.9)
The shortest dist is:0.3
```

暴力搜索结果：

```
brutal search:
The shortest-distance Point-Pair is:
(4,6.6) and (4,6.9)
The shortest dist is:0.3
```

五、实验心得

有时候对算法的输入进行一些“预处理”能够给算法的执行效率带来一些不小得提升，例如在上述算法中，对分治算法的每个输入数组X_L和Y_L都要按递增排序(X_R和Y_R一样)，但是我们在分治算法的第一次输入前就对数组 X 和 Y 分别执行了一次“预排序”，使得以后在每次递归调用中子数组X_L和Y_L、X_R和Y_R均为按递增排序，而不需要我们每次都为其排序，给算法执行效率带来了提升。