《算法设计与分析》

课程实验报告

专业：计算机科学与技术

班级：2021211304

姓名：张梓良

学号：2021212484

算法设计与分析（第 2 章）：分治与递归
编程实验报告

张梓良　2021212484

日期：2023 年 11 月 25 日

1　数据预处理

使用以下 python 代码预处理所给的 xls 格式的数据，从中提取出所需数据并分别存入 data1.txt 和 data2.txt。

```
import xlrd

# 打开xls文件
workbook = xlrd.open_workbook('DATA/data.xls')
worksheet = workbook.sheet_by_index(0)
num_rows = worksheet.nrows

# 遍历每一行，读取K_DIST值并写入data.txt文件
with open('DATA/data1.txt', 'w') as file:
    for row_idx in range(1, num_rows):
        k_dist = worksheet.cell_value(row_idx, 3)
        file.write(str(k_dist) + '\n')

with open('DATA/data2.txt', 'w') as file:
    for row_idx in range(1, num_rows):
        ENODEBID, LONGITUDE, LATITUDE,= worksheet.cell_value(row_idx, 0),
            worksheet.cell_value(row_idx, 1), worksheet.cell_value(row_idx, 2)
        file.write(str(ENODEBID) + ' ' + str(LONGITUDE) + ' ' + str(LATITUDE) +
             '\n')

print('Finished writing data to file.')
```

2　线性时间选择

2.1　实验内容及要求

内容

采用线性时间选择算法，根据基站 k-dist 距离，挑选出

- k-dist 值最小的基站

- k-dist 第 5 小的基站
- k-dist 值第 50 小的基站
- k-dist 值最大的基站

要求

- 在排序主程序中设置全局变量，记录选择划分过程的递归层次
- 参照讲义 PPT，将教科书上的“一分为二”的子问题划分方法，改进为“一分为三”，比较这 2 种划分方式下，选择过程递归层次的差异

2.2 算法设计

一分为二

Algorithm 1 线性时间选择算法（一分为二）

function SELECT($a[], p, r, k, cur\_level$)
  if $cur\_level > level$ **then**
    $level \leftarrow cur\_level$
  end if
  if $r - p < 20$ **then**
    BUBBLESORT(a, p, r)
    return $a[p + k - 1]$
  end if
  for $i \leftarrow 0$ to $(r - p - 4)/5$ **do**
    $s \leftarrow p + 5 * i$
    $t \leftarrow s + 4$
    BUBBLESORT(a, s, t)
    SWAP($a[p + i], a[s + 2]$)
  end for
  $x \leftarrow$ SELECT($a, p, p + (r - p - 4)/5, (r - p + 6)/10, cur\_level + 1$)
  $i \leftarrow$ PARTITION(a, p, r, x)
  $j \leftarrow i - p + 1$
  if $k \leq j$ **then**
    return SELECT($a, p, i, k, cur\_level + 1$)
  else
    return SELECT($a, i + 1, r, k - j, cur\_level + 1$)
  end if
end function

一分为三

只需修改上述算法 18-22 行即可。

Algorithm 2 线性时间选择算法（一分为三）

```
1:  function SELECT(a[], p, r, k, cur_level)
2:      ... ...
3:      // 一分为三
4:      if k == j then
5:          return x
6:      else
7:          if k < j then
8:              return SELECT(a, p, i, k, cur_level + 1)
9:          else
10:             return SELECT(a, i + 1, r, k − j, cur_level + 1)
11:         end if
12:     end if
13: end function
```

2.3 算法性能分析

时间复杂度

当子问题的规模小于 20 时，算法使用冒泡排序来查找第 k 小的元素，时间复杂度为常数级。当子问题的规模大于等于 20 时，算法通过选择主元进行划分，并根据划分结果递归地处理子问题。在每一次递归中，算法都会将数组划分为两部分（或者三部分）。由于每次划分都会减少问题的规模，因此总体上，时间复杂度可以表示为 $T(n) = T(n/5) + T(7n/10) + O(n)$，其中 n 为问题规模。根据主定理，可以得到该算法的时间复杂度为 $O(n)$。

空间复杂度

该算法的空间复杂度取决于递归调用的深度。由于每次递归都会将问题规模减少至不超过原来的 $\frac{7}{10}$ 或 $\frac{1}{5}$，因此递归深度的上界为 $O(logn)$。在每一层递归中，算法需要使用一些额外的空间来存储中间结果和递归调用的参数。因此，总体上，该算法的空间复杂度为 $O(logn)$。

2.4 所遇问题

在编写Partition函数时最开始写成了以下错误版本：

```
1  int Partition(double a[], int p, int r, double x)
2  {
3      int i = p - 1, j = r + 1;
4      while (true)
5      {
6          while (a[++i] < x && i < r)
7              ;
```

```
            while (a[--j] > x)
                ;
            if (i >= j)
                break;
            swap(a[i], a[j]);
        }
        return j;
    }
```

该代码能够实现将数组 a[] 按照 x 分为两部分，左部分小于等于 x ，右部分大于等于x，但是最终 a[j] != x 。

修改为正确版本：

```
    int Partition(double a[], int p, int r, double x)
    {
        // 将 x 放到最左边
        swap(a[p + (r - p - 4) / 10], a[p]);

        int i = p, j = r + 1;
        while (true)
        {
            while (a[++i] < x && i < r)
                ;
            while (a[--j] > x)
                ;
            if (i >= j)
                break;
            swap(a[i], a[j]);
        }
        a[p] = a[j];
        a[j] = x;
        return j;
    }
```

2.5 结果展示及分析

一分为二

```
1033 rows read.
The 1th smallest element is 103.075
The recursion depth is 7
The 5th smallest element is 126.096
The recursion depth is 7
The 50th smallest element is 208.475
The recursion depth is 8
The 1033th smallest element is 2735.8
```

```
The recursion depth is 7
Verifying...
The 1th smallest element is 103.075
The 5th smallest element is 126.096
The 50th smallest element is 208.475
The 1033th smallest element is 2735.8
```

可以发现线性时间选择算法（一分为二）得到的结果和使用 C++ 自带的 `sort` 函数得到的验证结果相同，说明算法实现正确。

一分为三

```
1033 rows read.
The 1th smallest element is 103.075
The recursion depth is 7
The 5th smallest element is 126.096
The recursion depth is 7
The 50th smallest element is 208.475
The recursion depth is 7
The 1033th smallest element is 2735.8
The recursion depth is 7
Verifying...
The 1th smallest element is 103.075
The 5th smallest element is 126.096
The 50th smallest element is 208.475
The 1033th smallest element is 2735.8
```

可以发现线性时间选择算法（一分为三）得到的结果和使用 C++ 自带的 `sort` 函数得到的验证结果相同，说明算法实现正确。

同时可以发现一分为三的选择递归层次是小于等于一分为二的。

3 最近平面点对

3.1 实验内容及要求

内容

采用平面最近点对算法，根据基站经纬度，挑选出

- 距离非零、且最近的 2 个基站
- 距离非零、且次最近的 2 个基站

要求

- 返回最近/次最近的 2 个基站间距离
- 返回最近/次最近的 2 个基站点对（用基站 ENodeBID 表示）

3.2 算法设计

Algorithm 3 最近平面点对算法

```
1: function CLOSEST(X[], Y[], Z[], l, r, p1, p2, d)
2:     if r == l then
3:         p1 ← X[l]
4:         p2 ← X[l]
5:         d ← ∞
6:         return
7:     end if
8:     if r − l == 1 then
9:         p1 ← X[l]
10:        p2 ← X[r]
11:        d ← distance(p1, p2)
12:        if not cmp(p1, p2) then
13:            d ← ∞
14:        end if
15:        return
16:    end if
17:    if r − l == 2 then
18:        closest3(X, l, p1, p2, d)
19:        return
20:    end if
21:    m ← (l + r)/2
22:    while m < r and |X[m].LATITUDE − X[m + 1].LATITUDE| < 1e − 8 do
23:        m ← m + 1
24:    end while
25:    if m ≠ r then
26:        f ← l, g ← m + 1
27:        for i ← l to r do
28:            if Y[i].LATITUDE > X[m].LATITUDE then
29:                Z[g + +] ← Y[i]
30:            else
31:                Z[f + +] ← Y[i]
32:            end if
```

```
        end for
    else
        m ← (l + r)/2
        for i ← l to r do
            Z[i] ← X[i]
        end for
        merge_sort(Z, l, m, 1)
        merge_sort(Z, m + 1, r, 1)
    end if
    Closest(X, Z, Y, l, m, p1, p2, d)
    dr ← closest(X, Z, Y, m + 1, r, p3, p4)          ▷ 右部分的最近点对距离
    p3, p4 ← closest(X, Z, Y, m + 1, r)              ▷ 右部分的最近点对
    if dr < d then
        p1 ← p3
        p2 ← p4
        d ← dr
    end if
    merge(Z, Y, l, m, r, 1)                          ▷ 重构 Y[]
    k ← l
    for i ← l to r do
        if |Y[i].LATITUDE − X[m].LATITUDE| < d then
            Z[k++] ← Y[i]
        end if
    end for
    for i ← l to k do
        for j ← i + 1 to k do
            if Z[j].LONGITUDE − Z[i].LONGITUDE < d then
                d1 ← distance(Z[i], Z[j])
                if not cmp(Z[i], Z[j]) then
                    d1 ← ∞
                end if
                if d1 < d then
                    d ← d1
                    p1 ← Z[i]
                    p2 ← Z[j]
                end if
            end if
        end for
```

71: **end for**

72: **end function**

3.3 算法性能

时间复杂度

- 当 $n \leq 3$ 时，时间复杂度为 $O(1)$。
- 在递归的主要部分，算法将 `Y[]` 数组根据 `X[]` 数组的中位数进行分割，时间复杂度为 $O(n)$。然后，算法递归调用 `closest` 函数两次，每次处理的点数为 $\frac{n}{2}$。所以，递归部分的时间复杂度可以表示为 $T(n) = 2T(n/2) + O(n)$。
- 在重构 `Y[]` 数组的 `merge` 步骤中，算法调用 `merge` 函数，其时间复杂度为 $O(n)$。
- 在求解跨越两个子区域的最近点对时，算法使用了两层循环。外层循环遍历 k 次 $(k \leq n)$，内层循环最多遍历 6 次。所以该部分的时间复杂度为 $O(k)$。

综上所述，整个算法的时间复杂度可以表示为 $T(n) = 2T(n/2) + O(n) + O(k)$。根据主定理，可以得到该算法的时间复杂度为 $O(nlogn)$。

空间复杂度

该算法的空间复杂度取决于递归调用的深度。由于每次递归都会将问题规模减少至原来的 $\frac{1}{2}$，因此递归深度的上界为 $O(logn)$。在每一层递归中，算法需要使用一些额外的空间来存储中间结果和递归调用的参数。因此，总体上，该算法的空间复杂度为 $O(logn)$。

3.4 所遇问题

在编写最近平面点对算法中将 `Y[]` 以 `X[]` 的中位数为界分成两部分时写成了以下错误版本：

```
    int m = (l + r) / 2;
    int f = l, g = m + 1;
    for (int i = l; i <= r; i++)
        if (Y[i].LATITUDE > X[m].LATITUDE)
            Z[g++] = Y[i];
        else
            Z[f++] = Y[i];
```

当 X 的中位数右侧存在和它的 x 值相同的元素时，会产生错误：

```
X = a1, a3, a4, a5, a2, a7, a6 (a4.x = a5.x = a2.x)
Y = a3, a4, a1, a2, a7, a5, a6
-> Z = a3, a4, a1, a2, a5, a6
```

可以发现原本写在 `Z[4]` 的 a7 被后面写入的 a5 覆盖，产生错误。

修改为正确版本：

```
    int m = (l + r) / 2;
    while (m < r && fabs(X[m].LATITUDE - X[m + 1].LATITUDE) < 1e-8)
        m++;
    if (m != r)
    {
        int f = l, g = m + 1;
        for (int i = l; i <= r; i++)
            if (Y[i].LATITUDE > X[m].LATITUDE)
                Z[g++] = Y[i];
            else
                Z[f++] = Y[i];
    }
    else
    {
        m = (l + r) / 2;
        for (int i = l; i <= r; i++)
            Z[i] = X[i];
        merge_sort(Z, l, m, 1);
        merge_sort(Z, m + 1, r, 1);
    }
```

正确版本同时考虑了特殊的边界情况，当 X 退化到它的中位数右侧元素的 x 值都和它相同时，此时按照中位数划分问题规模不会减小，直接采用归并排序对中位数两侧按照 y 值排序。

3.5 结果展示及分析

```
1033 rows read.
The closest pair is: 567389 566803
Point 1: 102.741 25.0539
Point 2: 102.741 25.0539
The distance is: 1.27865 m
The second closest pair is: 566784 567222
Point 1: 102.791 25.0398
Point 2: 102.791 25.0397
The distance is: 1.67338 m
Verifying...
The minimum distance is: 1.27865 m
The second minimum distance is: 1.67338 m
```

可以发现最近平面点对算法得到的结果和使用暴力枚举计算得到的验证结果相同，说明算法实现正确。（特别说明：最近两点的经纬度打印出来相同是由于打印时精度丢失，但实际计算时两点的经纬度是不同的）

4 总结

通过本次编程实验，让我更深入地了解了线性时间选择算法和平面最近点对算法。在第一个实验中，我使用了线性时间选择算法来根据基站的 k-dist 距离选择基站。通过设置全局变量来记录选择划分过程的递归层次，我能够更清晰地追踪算法的执行过程。同时，我还比较了“一分为二”和“一分为三”两种划分方式下选择过程的递归层次的差异，这让我能够评估不同划分方式对算法性能的影响。

在第二个实验中，我采用了平面最近点对算法来根据基站的经纬度选择基站。该算法通过计算基站之间的距离来找到距离非零且最近的两个基站以及距离非零且次最近的两个基站。这种算法在处理平面上的点对时非常高效，能够快速给出结果。

总的来说，这两个实验为我提供了宝贵的学习经验。通过思考和实践，让我更好地理解算法的原理和应用，并能够将其运用到其他类似的问题中。

A select.cpp

```cpp
/**
 * @file select.cpp
 * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
 * @brief 线性时间选择算法
 * @date 2023-11-22
 */
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <algorithm>

using namespace std;

const int MAX_NUM = 2000; // 最大元素个数
int level = 0;            // 递归深度

void BubbleSort(double a[], int p, int r);
int Partition(double a[], int p, int r, double x);
double Select(double a[], int p, int r, int k, int cur_level);
void KthSmallest(double a[], int p, int r, int k);

int main()
{
    // 读入数据
    ifstream file("DATA/data1.txt", ios::in);
    if (!file.is_open())
    {
        cerr << "Failed to open file." << endl;
        return 1;
    }

    double k_dist[MAX_NUM];
    int row_idx = 0;
    string line;
    while (row_idx < MAX_NUM && getline(file, line))
    {
        k_dist[row_idx] = stod(line);
        row_idx++;
    }
    cout << row_idx << " rows read." << endl;

    file.close();

    // 选择第k小的元素
```

```
46          KthSmallest(k_dist, 0, row_idx - 1, 1);
47          KthSmallest(k_dist, 0, row_idx - 1, 5);
48          KthSmallest(k_dist, 0, row_idx - 1, 50);
49          KthSmallest(k_dist, 0, row_idx - 1, row_idx);
50
51          // 验证结果
52          cout << "Verifying..." << endl;
53          sort(k_dist, k_dist + row_idx);
54          cout << "The 1th smallest element is " << k_dist[0] << endl;
55          cout << "The 5th smallest element is " << k_dist[4] << endl;
56          cout << "The 50th smallest element is " << k_dist[49] << endl;
57          cout << "The " << row_idx << "th smallest element is " << k_dist[row_idx -
                1] << endl;
58
59          return 0;
60      }
61
62      // 冒泡排序
63      void BubbleSort(double a[], int p, int r)
64      {
65          for (int i = p; i < r; i++)
66              for (int j = r; j > i; j--)
67                  if (a[j] < a[j - 1])
68                      swap(a[j], a[j - 1]);
69      }
70
71      // 以 x 为主元划分
72      int Partition(double a[], int p, int r, double x)
73      {
74          // 将 x 放到最左边
75          swap(a[p + (r - p - 4) / 10], a[p]);
76
77          int i = p, j = r + 1;
78          while (true)
79          {
80              while (a[++i] < x && i < r)
81                  ;
82              while (a[--j] > x)
83                  ;
84              if (i >= j)
85                  break;
86              swap(a[i], a[j]);
87          }
88          a[p] = a[j];
89          a[j] = x;
90          return j;
91      }
```

```
92
93      // 递归查找第k小的元素
94      double Select(double a[], int p, int r, int k, int cur_level)
95      {
96          // 更新递归深度
97          if (cur_level > level)
98              level = cur_level;
99
100         // 规模小于20的子问题直接用冒泡排序查找
101         if (r - p < 20)
102         {
103             BubbleSort(a, p, r);
104             return a[p + k - 1];
105         };
106
107         // 分成 n/5 组并找到每组的中位数，若最后一段不足5个元素则不考虑
108         for (int i = 0; i <= (r - p - 4) / 5; i++)
109         {
110             int s = p + 5 * i;
111             int t = s + 4;
112             BubbleSort(a, s, t);
113             swap(a[p + i], a[s + 2]);
114         }
115
116         // 找到所有中位数的中位数
117         double x = Select(a, p, p + (r - p - 4) / 5, (r - p + 6) / 10, cur_level +
                1);
118
119         // 以中位数的中位数为主元划分
120         int i = Partition(a, p, r, x);
121         int j = i - p + 1; // 左部元素个数
122
123         // 一分为二
124         if (k <= j)
125             return Select(a, p, i, k, cur_level + 1);
126         else
127             return Select(a, i + 1, r, k - j, cur_level + 1);
128
129         // 一分为三
130         // if (k == j)
131         //     return x;
132         // else
133         // {
134         //     if (k < j)
135         //         return Select(a, p, i, k, cur_level + 1);
136         //     else
137         //         return Select(a, i + 1, r, k - j, cur_level + 1);
```

```
        // }
    }

    // 找到第k小的元素
    void KthSmallest(double a[], int p, int r, int k)
    {
        level = 0;
        double kth_smallest = Select(a, p, r, k, 1);
        cout << "The " << k << "th smallest element is " << kth_smallest << endl;
        cout << "The recursion depth is " << level << endl;
    }
```

B cpair.cpp

```
    /**
     * @file cpair.cpp
     * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
     * @brief 平面最近点对算法
     * @date 2023-11-23
     */
    #include <iostream>
    #include <cmath>
    #include <fstream>
    #include <string>

    using namespace std;

    const int INF = 0x3f3f3f3f;
    const int MAX_NUM = 2000; // 最大元素个数
    const double PI = 3.141592653589793238626;
    const double R = 6378.137 * 1e3; // 地球半径

    typedef struct
    {
        int ENODEBID;
        double LATITUDE;
        double LONGITUDE;
    } Point;

    Point tmp[MAX_NUM];

    double distance(Point p1, Point p2);
    void merge(Point a[], Point b[], int l, int m, int r, int mode);
    void merge_sort(Point a[], int l, int r, int mode);
    bool cmp(Point p1, Point p2);
    void closest3(Point X[], int l, Point &p1, Point &p2, double &d);
```

```
33      void closest(Point X[], Point Y[], Point Z[], int l, int r, Point &p1, Point &
            p2, double &d);
34
35      int main()
36      {
37          // 读入数据
38          ifstream file("DATA/data2.txt", ios::in);
39          if (!file.is_open())
40          {
41              cerr << "Failed to open file." << endl;
42              return 1;
43          }
44
45          Point points[MAX_NUM];
46          int row_idx = 0;
47          string line;
48          while (row_idx < MAX_NUM && getline(file, line))
49          {
50              int pos = line.find(' ');
51              points[row_idx].ENODEBID = stoi(line.substr(0, pos));
52              line = line.substr(pos + 1);
53              pos = line.find(' ');
54              points[row_idx].LATITUDE = stod(line.substr(0, pos));
55              line = line.substr(pos + 1);
56              points[row_idx].LONGITUDE = stod(line);
57              row_idx++;
58          }
59          cout << row_idx << " rows read." << endl;
60
61          file.close();
62
63          Point X[MAX_NUM], Y[MAX_NUM], Z[MAX_NUM];
64          merge_sort(points, 0, row_idx - 1, 0); // 按照纬度排序
65          for (int i = 0; i < row_idx; i++)
66              X[i] = points[i];
67          merge_sort(points, 0, row_idx - 1, 1); // 按照经度排序
68          for (int i = 0; i < row_idx; i++)
69              Y[i] = points[i];
70
71          // 求解最近点对
72          Point p1, p2;
73          double d;
74          closest(X, Y, Z, 0, row_idx - 1, p1, p2, d);
75          cout << "The closest pair is: " << p1.ENODEBID << " " << p2.ENODEBID <<
                endl;
76          cout << "Point 1: " << p1.LATITUDE << " " << p1.LONGITUDE << endl;
77          cout << "Point 2: " << p2.LATITUDE << " " << p2.LONGITUDE << endl;
```

```
78          cout << "The distance is: " << d << " m" << endl;
79
80          // 求解次近点对
81          for (int i = 0; i < row_idx; i++)
82          {
83              if (p1.ENODEBID == X[i].ENODEBID)
84              {
85                  X[i].LATITUDE = 0;
86                  X[i].LONGITUDE = 0;
87              }
88              if (p1.ENODEBID == Y[i].ENODEBID)
89              {
90                  Y[i].LATITUDE = 0;
91                  Y[i].LONGITUDE = 0;
92              }
93          }
94          closest(X, Y, Z, 0, row_idx - 1, p1, p2, d);
95          cout << "The second closest pair is: " << p1.ENODEBID << " " << p2.ENODEBID
                << endl;
96          cout << "Point 1: " << p1.LATITUDE << " " << p1.LONGITUDE << endl;
97          cout << "Point 2: " << p2.LATITUDE << " " << p2.LONGITUDE << endl;
98          cout << "The distance is: " << d << " m" << endl;
99
100         // 验证结果
101         cout << "Verifying..." << endl;
102         double min_dist = INF;
103         for (int i = 0; i < row_idx; i++)
104             for (int j = i + 1; j < row_idx; j++)
105             {
106                 double dist = distance(points[i], points[j]);
107                 if (dist < min_dist && cmp(points[i], points[j]))
108                     min_dist = dist;
109             }
110         cout << "The minimum distance is: " << min_dist << " m" << endl;
111         double min_dist2 = INF;
112         for (int i = 0; i < row_idx; i++)
113             for (int j = i + 1; j < row_idx; j++)
114             {
115                 double dist = distance(points[i], points[j]);
116                 if (dist < min_dist2 && cmp(points[i], points[j]) && dist !=
                        min_dist)
117                     min_dist2 = dist;
118             }
119         cout << "The second minimum distance is: " << min_dist2 << " m" << endl;
120     }
121
122     // 计算两点之间的距离
```

```
123     double distance(Point p1, Point p2)
124     {
125         double radLat1 = p1.LATITUDE * PI / 180.0;
126         double radLat2 = p2.LATITUDE * PI / 180.0;
127         double radLon1 = p1.LONGITUDE * PI / 180.0;
128         double radLon2 = p2.LONGITUDE * PI / 180.0;
129         return R * acos(cos(radLat1) * cos(radLat2) * cos(radLon1 - radLon2) + sin(
                radLat1) * sin(radLat2));
130     }
131
132     // 归并排序
133     void merge_sort(Point a[], int l, int r, int mode)
134     {
135         if (l >= r)
136             return;
137         int m = (l + r) / 2;
138         merge_sort(a, l, m, mode);
139         merge_sort(a, m + 1, r, mode);
140         merge(a, tmp, l, m, r, mode);
141         for (int i = l, j = l; i <= r; i++, j++)
142             a[i] = tmp[j];
143     }
144
145     // 归并
146     void merge(Point a[], Point b[], int l, int m, int r, int mode)
147     {
148         int i = l, j = m + 1, k = l;
149         while (i <= m && j <= r)
150             if ((a[i].LONGITUDE <= a[j].LONGITUDE && mode) || (a[i].LATITUDE <= a[j
                    ].LATITUDE && !mode))
151                 b[k++] = a[i++];
152             else
153                 b[k++] = a[j++];
154         while (i <= m)
155             b[k++] = a[i++];
156         while (j <= r)
157             b[k++] = a[j++];
158     }
159
160     // 比较两个点位置是否相同
161     bool cmp(Point p1, Point p2)
162     {
163         if ((fabs(p1.LATITUDE - p2.LATITUDE) < 1e-8) && (fabs(p1.LONGITUDE - p2.
                LONGITUDE) < 1e-8))
164             return false;
165         return true;
166     }
```

```
167
168     // 求解三个点中的最近点对
169     void closest3(Point X[], int l, Point &p1, Point &p2, double &d)
170     {
171         double d1 = distance(X[l], X[l + 1]);
172         double d2 = distance(X[l], X[l + 2]);
173         double d3 = distance(X[l + 1], X[l + 2]);
174         if (!cmp(X[l], X[l + 1]))
175             d1 = INF;
176         if (!cmp(X[l], X[l + 2]))
177             d2 = INF;
178         if (!cmp(X[l + 1], X[l + 2]))
179             d3 = INF;
180         if (d1 <= d2 && d1 <= d3)
181         {
182             p1 = X[l];
183             p2 = X[l + 1];
184             d = d1;
185         }
186         else if (d2 <= d1 && d2 <= d3)
187         {
188             p1 = X[l];
189             p2 = X[l + 2];
190             d = d2;
191         }
192         else
193         {
194             p1 = X[l + 1];
195             p2 = X[l + 2];
196             d = d3;
197         }
198     }
199
200     /**
201      * @brief 递归求解最近点对
202      * @param X 按照x坐标排序的点集
203      * @param Y 按照y坐标排序的点集
204      * @param Z 跨越两个子区域的点集
205      * @param l 左边界
206      * @param r 右边界
207      * @param p1 最近点对中的一个点
208      * @param p2 最近点对中的另一个点
209      * @param d 最近点对的距离
210      */
211     void closest(Point X[], Point Y[], Point Z[], int l, int r, Point &p1, Point &
         p2, double &d)
212     {
```

```
        // 只有一个点
        if (r == l)
        {
            p1 = X[l];
            p2 = X[l];
            d = INF;
            return;
        }

        // 只有两个点
        if (r - l == 1)
        {
            p1 = X[l];
            p2 = X[r];
            d = distance(p1, p2);
            if (!cmp(p1, p2))
                d = INF;
            return;
        }

        // 只有三个点
        if (r - l == 2)
        {
            closest3(X, l, p1, p2, d);
            return;
        }

        // 将Y[]以X[]的中位数为界分成两部分
        int m = (l + r) / 2;
        while (m < r && fabs(X[m].LATITUDE - X[m + 1].LATITUDE) < 1e-8)
            m++;
        if (m != r)
        {
            int f = l, g = m + 1;
            for (int i = l; i <= r; i++)
                if (Y[i].LATITUDE > X[m].LATITUDE)
                    Z[g++] = Y[i];
                else
                    Z[f++] = Y[i];
        }
        else
        {
            m = (l + r) / 2;
            for (int i = l; i <= r; i++)
                Z[i] = X[i];
            merge_sort(Z, l, m, 1);
            merge_sort(Z, m + 1, r, 1);
```

```
260         }
261
262         closest(X, Z, Y, l, m, p1, p2, d);
263         double dr;     // 右部分的最近点对距离
264         Point p3, p4; // 右部分的最近点对
265         closest(X, Z, Y, m + 1, r, p3, p4, dr);
266         if (dr < d)
267         {
268             p1 = p3;
269             p2 = p4;
270             d = dr;
271         }
272
273         merge(Z, Y, l, m, r, 1); // 重构Y[]
274
275         int k = l;
276         for (int i = l; i <= r; i++)
277             if (fabs(Y[i].LATITUDE - X[m].LATITUDE) < d)
278                 Z[k++] = Y[i];
279
280         // 求解跨越两个子区域的最近点对
281         for (int i = l; i < k; i++)
282         {
283             for (int j = i + 1; j < k && Z[j].LONGITUDE - Z[i].LONGITUDE < d; j++)
284             {
285                 double d1 = distance(Z[i], Z[j]);
286                 if (!cmp(Z[i], Z[j]))
287                     d1 = INF;
288                 if (d1 < d)
289                 {
290                     d = d1;
291                     p1 = Z[i];
292                     p2 = Z[j];
293                 }
294             }
295         }
296     }
```