计算机学院《算法设计与分析》第一次作业

20373363 李子涵

September 25, 2022

1 计算尽可能紧凑的渐进上界并予以说明

主定理法公式:
对形如 $T(n)=aT(\frac{n}{b})+f(n)$ 的递归式:

$$T(n)=\begin{cases}O(f(n)) & if\ f(n)=\Omega(n^{log_ba+\epsilon})\\ O(n^{log_ba}logn) & if\ f(n)=\Theta(n^{log_ba})\\ O(n^{log_ba}) & if\ f(n)=O(n^{log_ba-\epsilon})\end{cases}$$

对形如 $T(n)=aT(\frac{n}{b})+n^k$ 的递归式:

$$T(n)=\begin{cases}O(n^k) & if\ k>log_ba\\ O(n^klogn) & if\ k=log_ba\\ O(n^{log_ba}) & if\ k<log_ba\end{cases}$$

1.1

$$T(n)=\begin{cases}1\ ,\ n=1,2\\ T(n-2)+1\ ,\ n>2\end{cases}$$

$T(n)=T(n-2)+1=T(n-4)+2=T(n-6)+3=...=\lceil\frac{n}{2}\rceil$
由此可知，原式渐进上界为 $O(n)$。

1.2

$$T(n)=\begin{cases}1\ ,\ n=1\\ T(n/2)+1\ ,\ n>1\end{cases}$$

使用主定理法计算渐进上界，由上式可知 $a=1,b=2,log_ba=log_21=0$。
使用 $T(n)=aT(\frac{n}{b})+n^k$ 的递归式，有 $n^k=1=n^0,k=0$。
由 $k=log_ba$，可得上式的渐进上界为 $O(n^klogn)=O(logn)$。

1.3

$$T(n)=\begin{cases}1\ ,\ n=1\\ T(n/2)+n\ ,\ n>1\end{cases}$$

使用主定理法计算渐进上界，由上式可知 $a=1,b=2,log_ba=log_21=0$。
使用 $T(n)=aT(\frac{n}{b})+n^k$ 的递归式，有 $n^k=n=n^1,k=1$。
由 $k>log_ba$，可得上式的渐进上界为 $O(n^k)=O(n)$。

1.4

$$T(n) = \begin{cases} 1\ ,\ n = 1 \\ 2T(n/2) + 1\ ,\ n > 1 \end{cases}$$

使用主定理法计算渐进上界，由上式可知 $a = 2, b = 2, log_b a = log_2 2 = 1$。

使用 $T(n) = aT(\frac{n}{b}) + n^k$ 的递归式，有 $n^k = 1 = n^0, k = 0$。

由 $k < log_b a$，可得上式的渐进上界为 $O(n^{log_b a}) = O(n)$。

1.5

$$T(n) = \begin{cases} 1\ ,\ n = 1 \\ 4T(n/2) + 1\ ,\ n > 1 \end{cases}$$

使用主定理法计算渐进上界，由上式可知 $a = 4, b = 2, log_b a = log_2 4 = 2$。

使用 $T(n) = aT(\frac{n}{b}) + n^k$ 的递归式，有 $n^k = 1 = n^0, k = 0$。

由 $k < log_b a$，可得上式的渐进上界为 $O(n^{log_b a}) = O(n^2)$。

1.6

$$T(n) = \begin{cases} 1\ ,\ n = 1 \\ T(n/2) + logn\ ,\ n > 1 \end{cases}$$

若使用主定理法计算渐进上界，由上式可知 $a = 1, b = 2, log_b a = log_2 1 = 0$。

使用 $T(n) = aT(\frac{n}{b}) + f(n)$ 的递归式，然而对 $\forall \epsilon > 0, logn$ 渐进小于 n^ϵ，故 $\nexists \epsilon > 0$ 使 $f(n) = \Omega(n^{log_b a + \epsilon})$。

该情况落入 (1)(2) 之间，不能使用主定理。

引入主定理的扩展形式：对形如 $T(n) = aT(\frac{n}{b}) + f(n)$ 的递归式

$$T(n) = \begin{cases} O(f(n)) & if\ f(n) = \Omega(n^{log_b a + \epsilon}) \\ O(n^{log_b a} log^{k+1} n) & if\ f(n) = \Theta(n^{log_b a} log^k n), k \geq 0 \\ O(n^{log_b a}) & if\ f(n) = O(n^{log_b a - \epsilon}) \end{cases}$$

情况 (2) 的三种拓展：

$$T(n) = \begin{cases} O(n^{log_b a} log^{k+1} n) & k > -1 \\ O(n^{log_b a} loglogn) & k = -1 \\ O(n^{log_b a}) & k < -1 \end{cases}$$

$k = 1, f(n) = logn = \Theta(n^{log_b a} log^k n)$, 属于情况 (2), 可得上式的渐进上界为 $T(n) = O(n^{log_b a} log^{k+1} n) = O(log^2 n)$。

1.7

$$T(n) = \begin{cases} 1\ ,\ n = 1 \\ 3T(n/2) + n^2\ ,\ n > 1 \end{cases}$$

使用主定理法计算渐进上界，由上式可知 $a = 3, b = 2, log_b a = log_2 3$。

使用 $T(n) = aT(\frac{n}{b}) + n^k$ 的递归式，有 $n^k = n^2, k = 2$。

由 $k > log_b a$，可得上式的渐进上界为 $O(n^k) = O(n^2)$。

2 寻找中位数问题

一组数据 $x=(x_1,x_2,...,x_n)$ 从小到大排序后的序列为 $x'_1,...,x'_n$，则这组数据的中位数 M 为

$$M=\begin{cases} x'_{\frac{n+1}{2}} & ，若\ n\ 为奇数; \\ \frac{1}{2}(x'_{\frac{n}{2}}+x'_{\frac{n}{2}+1}) & ，若\ n\ 为偶数。 \end{cases}$$

给定两个长度分别为 n,m 的有序数组 $A[1..n],B[1..m]$（A 与 B 均已按从小到大排序)。请设计尽可能高效的算法，求出这两个数组中所有数据的中位数，并分析其时间复杂度。

2.1 算法思想

采用朴素的**双指针**遍历法寻找两个有序数组的中位数，时间复杂度为 $O(m+n)$。

以下考虑使用**二分法**设计更加高效的算法。求两个有序数组的中位数，由定义得:

求两个有序数组中**第 k 小**的数 K，$k=\begin{cases} \frac{m+n+1}{2} & ，若\ m+n\ 为奇数; \\ \frac{m+n}{2}\ 和\ \frac{m+n}{2}+1 & ，若\ m+n\ 为偶数。 \end{cases}$

每次二分时令 $t=\begin{cases} \frac{k-1}{2} & ，若\ k\ 为奇数; \\ \frac{k}{2} & ，若\ k\ 为偶数。 \end{cases}$，即保证从两个数组分别取前 t 个元素时，总个数不超过 k, 用于后续二分法剪枝。

首先判断**边界情况**:

1. 如果 $t=0$，表示返回当前两个数组的最小值;
2. 如果其中一个数组的剩余元素个数为 0，则 K 在另一数组中，返回 K;

此时分别计算二分法的中点下标 a,b。取 $A[a]$ 和 $B[b]$:

1. 此时假设 $A[a]\le B[b]$，则一定有 $A[a]<K$。否则假设 $A[a]>K$，则 $B[b]\ge A[a]>K$，即在不超过 $2t(\le k)$ 个较小数中，出现 $B[b]>K$，矛盾。
   故 $A[1,a]$ 的值**均小于** K，可**排除** a **个元素**，继续在 $k-a$ 个剩余元素中求 K。
2. $A[a]>B[b]$ 同理。

2.2 伪代码

Algorithm 1: 求两个数组中所有数据的中位数

Input: 正整数 n，为有序数组 A 的长度；正整数 m，为有序数组 B 的长度；
长度为 n 的数组 $A[1..n]$，长度为 m 的数组 $B[1..m]$

Output: 浮点数 ans，为两个数组中所有数据的中位数

```
function median(n, m, A, B):
    if (m + n)%2 then
        return getK((m+n+1)/2);
    end
    K1 ← getK(n, m, A[1..n], B[1..m], (m+n)/2);
    K2 ← getK(n, m, A[1..n], B[1..m], (m+n)/2 + 1);
    return (K1+K2)/2;
end
```

Algorithm 2: 求两个数组中第 k 小的数

Input: 正整数 n，为有序数组 A 的长度；正整数 m，为有序数组 B 的长度；
长度为 n 的数组 $A[1..n]$，长度为 m 的数组 $B[1..m]$; 第 k 小的数的顺序 k

Output: 整数 K，为两个数组中第 k 小的数

```
1  function getK(n, m, A, B, k):
2      k' ← k;
       // 下一次循环寻找第k'小的元素
3      a, b ← 1, 1;
       // 从数组A, B的下标a, b继续寻找第k'小的数K
4      while True do
           // 二分结束，返回当前两个数组的首个较小的元素
5          t ← (k' − 1)/2;
6          if t = 0 then
7              return min(A[a], B[b]);
8          end
           // 一个数组的所有元素均小于第k小的数K
9          else if a > n then
10             return B[k − n];
11         end
12         else if b > m then
13             return A[k − m];
14         end
           // 当前能取到的二分中点或者数组的最大下标a', b'
15         a' ← min(a + t, n);
16         b' ← min(b + t, m);
           // 舍弃较小的部分元素，更新k', a或b
17         if A[a'] > B[b'] then
18             k' = k' − b' + b − 1;
19             b = b' + 1;
20         end
21         else
22             k' = k' − a' + a − 1;
23             a = a' + 1;
24         end
25     end
26 end
```

2.3 复杂度分析

k 初始取值 $(m+n)/2$。每次分治都对当前查找的 k 二分，每层以因子 $b=2$ 的速度下降。

仅根据下标值获取数组值进行比较，可**去除一半的数组元素**, 每层共 $a=1$ 个分支。得到的递归式为 $T(m+n)=T((m+n)/2)+O(1)$。

解得 $T(m+n)$ 的时间复杂度为 $O(log(m+n))$。

3 双调序列最大值问题

若一个序列 $a[1..n]$，满足以下两个性质之一，则该序列称为双调序列：

性质 1：若存在 $k \in [1,n]$，使得 $a_1 \le a_2 \le ... \le a_{k-1} \le a_k \ge a_{k+1} \ge ... \ge a_{n-1} \ge a_n$；

性质 2：序列经循环移位后满足性质 1。

例如，序列 $[1,2,4,8,7,5,3]$ 为双调序列（满足性质 1），序列 $[5,3,1,2,4,8,7]$ 也为双调序列（循环移位后可得序列 $[1,2,4,8,7,5,3]$，满足性质 2）。

给定一个双调序列 $a[1...n]$，请设计一个算法来求出这个序列中的最大值，并对该算法的复杂度进行分析。

3.1 算法思想

双调序列求最大值，直接遍历的复杂度为 $O(n)$。求最大值可以使用分治思想转换为求极大值，进一步分析此题相较于二分法求最大值的复杂之处：

1. 考虑求**极大值的基本图形 (b)**：$a_1 < a_2 < ... < a_{k-1} > a_k > a_{k+1} > ... > a_{n-1} > a_n$。此时二分法可以求得极大值所对的下标：每次找到非端点的中点 x，通过比较 $a[x-1], a[x], a[x+1]$ 的大小，判断**单调性**，即可逐步缩小极大值所在的区间，进而找到极大值所对的下标 x'。

2. 由于性质 2，序列经过**循环移位**后，可能出现上图的 4 种情况。此时如图 (c)(d)，仅靠中点附近点的**增减趋势**，**无法唯一地确定**极大值所在区间。

解决思路：**综合**左右端点值的大小、中点值与左右端点值大小的相对关系、中点值附近点的增减趋势，可以**唯一地确定二分后最大值**所在的区间，从而优化到 $O(log(n))$ 的复杂度。

假设数组中元素互不相等，具体算法如下：记二分法的区间端点的下标分别为 $l, r(l < r)$, 取得的中点（非端点）下标为 x。记区间端点的值分别为 $a[l], a[r]$, 区间中点的值为 $a[x]$。

Step1: 首先比较 $a[x], a[l], a[r]$ 的相对大小。如果 $a[x] < a[l]$ and $a[x] < a[r]$，则比较区间端点值 $a[l], a[r]$ 的大小，选择**端点值较大值的一侧**，即为下一次二分的区间。

Step2: 再取 $a[x-1], a[x], a[x+1]$ 的值，比较中点附近的**增减性**。

1. 增：选择右侧继续二分
2. 减：选择左侧继续二分
3. 极大值：返回此极大值，即为双调序列最大值
4. 极小值的情况已被 Step1 包含，返回区间端点值较大的一侧继续二分即可。

3.2 二分法伪代码

Algorithm 3: 双调序列最大值

Input: 双调序列 $a[1...n]$，数组元素个数 n

Output: 整数 max, 双调序列最大值

```
1  global a[1...n];
2  function main(a[1...n], n):
3      max = findMax(1,n);
4  end
5  function findMax(l, r):
6      m ← (l + r)/2;
       // 左右端点相等，仅剩余1个元素，返回
7      if l == r then
8          return a[l];
9      end
       // 左端点与中点相等，仅剩余2个元素，返回较大值
10     else if m == l then
11         return max {a[l], a[r]};
12     end
       // Step 1 中点小于两端点，较大端点一侧继续二分
13     if a[x] < a[l] and a[x] < a[r] then
14         if a[l] < a[r] then
15             return findMax(m + 1, r);
16         end
17         return findMax(l, m − 1);
18     end
       // Step 2 判断中点附近单调性
       // 增，中点右侧继续二分
19     if a[m − 1] < a[m] and a[m] < a[m + 1] then
20         return findMax(m + 1, r);
21     end
       // 减，中点左侧继续二分
22     else if a[m − 1] > a[m] and a[m] > a[m + 1] then
23         return findMax(l, m − 1);
24     end
       // 极大值，即为单调序列最大值
25     else if a[m − 1] < a[m] and a[m] > a[m + 1] then
26         return a[m];
27     end
       // 极小值
28     if a[l] < a[r] then
29         return findMax(m + 1, r);
30     end
31     return findMax(l, m − 1);
32 end
```

3.3 复杂度分析

每次分治时，都对当前的数组 a 二分，每层以因子 $b=2$ 的速度下降。

仅根据下标值获取数组值进行比较，即可选择最大值所在的分支，去除一半的数组元素后继续二分。每层共 $a=1$ 个分支。

得到的递归式为 $T(n)=T(n/2)+O(1)$，解得 $T(n)$ 的时间复杂度为 $O(log(n))$。

4 字符串等价关系判定问题

给定两个字符串长度为 n 的字符串 A 和 B，若称 A 与 B 是等价的，当且仅当满足如下关系之一：

1.A 和 B 完全相同；

2. 若把 A 分成长度相等的两段 A_1 和 A_2，也将 B 分成长度相等的两端段 B_1 和 B_2。且他们之间满足如下两种关系之一：

a.A_1 和 B_1 等价且 A_2 和 B_2 等价，

b.A_1 和 B_2 等价且 A_2 和 B_1 等价。

请你设计一个高效的算法来判断两个字符串是否等价并分析你的算法的时间复杂度。

4.1 算法思想

如果采用朴素的分治思想，拆分成形如 A_1A_2 与 B_1B_2 的字符串后，每层以 $n/2$ 的速度下降，并考虑：

$$\text{每一层中的分支数}=\begin{cases}\text{返回等价} & \text{若 } A_1A_2==B_1B_2\\ \text{不相等的两个字符串 } A_* \text{ 和 } B_* \text{ 继续分治} & \text{若 } A \text{ 与 } B \text{ 存在完全相同的子串}\\ A_* \text{ 和 } B_* \text{ 共四条分支继续分治} & \text{若 } A \text{ 与 } B \text{ 不存在完全相同的子串}\end{cases}$$

此时最坏情况下每一层有**四个**分支，复杂度较高。下考虑优化方法：

从分支生成的根节点角度考虑，由于**无法**在**自顶向下**拆分时，将生成的子串按**从左到右**的顺序匹配，造成最坏情况下一层需要匹配四个分支。

当将所有子串拆分成最小串（长度为奇数）时，从叶节点的角度考虑，此时的生成树每一个父节点有两个儿子节点。按照等价的关系，在**自底向上合并**时，可以自由地**交换两个叶节点的顺序**，从而生成一个新的字符串 A'。由于对叶节点进行的交换**符合等价关系变换**，产生的新的父节点也是与原父节点等价的。所以**自底向上**的两棵**同级子树的交换**，均为对与 A 字符串的一种**等价变换**。的所以易知 A 与 A' 等价。

考虑**按字典序做等价变换**后结果的**唯一性**：作为一组的两个叶节点，他们的相对顺序仅能在对这两个叶节点的交换中改变。**改变父节点的顺序，不影响这两棵子树的相对顺序**。故易知，在**自底向上按字典序排序**后，**等价**的字符串树的**生成树唯一**。

考虑字符串 B，如果字符串 B 与 A 等价，那么 B 也可经过此类等价变化生成 A，或者与 A 等价的 A'。

对 A 和 B 分别进行拆分后的自底向上合并时，按照一定的顺序例如字典序交换叶节点，即可生成分别与 A 和 B 等价的字符串 A' 和 B'。判断 **A 与 B 的等价性**，只需判断 **A' 与 B' 的相等性**即可。

此种算法的分治，每层以 $n/2$ 的速度下降，且满足**每层 2 个分支**，复杂度低于方案一的自顶向下拆分时判断等价性。

4.2 按字典序进行等价变换做法的伪代码

Algorithm 4: 按字典序进行等价变换

Input: 字符串 $S[1..n]$

Output: 字符串 $S'[1..n]$，为对 $S[1..n]$ 按字典序进行等价变换后的字符串

```
function orderS(S[1..n]):
    if n%2 then
        return S;
    end
    // 自顶向下二分
    S1 ← orderS(S[1..n/2]);
    S2 ← orderS(S[n/2 + 1, n]);
    // 自底向上合并
    if S1 ≤ S2 then
        return S1 + S2;
    end
    return S2 + S1;
end
```

Algorithm 5: 判断两字符串是否等价

Input: 字符串 $A[1..n]$ $B[1..n]$

Output: Bool*equal* 是否等价

```
function equal(A[1..n] B[1..n]):
    A' = orderS(A);
    B' = orderS(B);
    // 等价变换后只需判断相等性
    return (A' == B');
end
```

4.3 按字典序进行等价变换做法的复杂度分析

每次分治将当前字符串二分，每层共 $a = 2$ 个分支。每层以因子 $b = 2$ 的速度下降。

每次合并分支时需要进行两个字符串的比较大小，时间复杂度 $O(n)$。

得到的递归式为 $T(n) = 2T(n/2) + O(n)$。解得**分治**的 $T(n)$ 的时间复杂度为 $O(nlogn)$。

最后**比较**排序后的字符串 A, B 是否相等，时间复杂度 $O(n)$。

故最终的时间复杂度为 $O(nlogn)$。

5 向量的最小和问题

给定 n 个二维向量 $v1, v2, ..., v_n$。每一个向量 $v_i = (x_i, y_i)$，都可以变换为如下四种形式：

$1.v_i^1 = (x_i, y_i)$

$2.v_i^2 = (-x_i, y_i)$

$3.v_i^3 = (x_i, -y_i)$

$4.v_i^4 = (-x_i, -y_i)$

请你设计一个高效的算法从 n 个向量中找出两个向量，使得他们以某种形式相加后的模长最小。换言之，请找出两个向量 $v_i, v_j (1 \le i, j \le n$ 且 $i \ne j)$，以及两个整数 $k_1, k_2 (1 \le k_1, k_2 \le 4)$，使得 $||v_i^{k_1} + v_j^{k_2}||_2$ 最小。此外，请分析该算法的时间复杂度。

5.1 算法思想

首先对题目所求**向量和的最小模长作化简**：设两向量夹角为 θ, 由于每个向量存在四种变换形式，故**变换后的两向量夹角**的取值为 θ 或 $(180° - \theta)$。由余弦公式知，两变换后的向量和取得最小模长时，两变换后的向量夹角取值为 $[0°, 90°]$。

故问题可以简化为：将所有向量**做变换至第一象限**，求两变换后的**向量的最小距离**。

对于平面的最近点对问题，采用朴素的两两枚举求向量和并维护最小值，时间复杂度为 $O(n^2)$。以下考虑使用分治思想设计更加高效的算法:

首先把所有向量变换至第一象限，时间复杂度 $O(n)$。

利用**归并排序**将所有的点按**横坐标**从小到大排序，如果横坐标相同比较纵坐标，形成数组 V_X。时间复杂度 $O(nlogn)$。

再利用**归并排序**将所有的点按**纵坐标**从小到大排序，如果纵坐标相同比较横坐标, 形成数组 V_Y。时间复杂度 $O(nlogn)$。

然后把按**横坐标排序**后的点集 V_X **二分**，并把按纵坐标排序的 V_Y 按横坐标，**分成左右两个按纵坐标排序**的数组，时间复杂度 $O(n)$。然后分别求出左右两点集的最近距离 d_1, d_2，取此时最小距离 $d = min(d_1, d_2)$。

再考虑是否存在点对的两点 (x_l, x_r) 分别取在左右点集并且使得 $|x_l x_r| < d$ 的情况。

以二分时取的点集中点的横坐标 x_m 为中点，则上述 (x_l, x_r) 一定为横坐标介于 $[x_m - d, x_m + d]$ 之间的点。否则若一个点的横坐标在此区间外，则两点距离大于 d。

再遍历按纵坐标排序的数组 V_Y，仅保留横坐标位于 $[x_m - d, x_m + d]$ 之间的点，即可得到**此区间内按纵坐标排序**的点集，时间复杂度 $O(n)$。只需考察**每个点下方至多 7 个点**。原因如下:

假设考察 $[x_m - d, x_m + d]$ 之间的点 X, 则 X 下方与它距离小于等于 d 的点必然在纵坐标为 $[y_m, y_m - d]$ 的区间内。对于属于左侧 $d \times d$ 的区域内，最多有四个点分别处在顶点位置，使其距离不小于 d。故在 $d \times 2d$ 的区域内，最多同时存在 8 个点。所以每个点只需考察其下方与其纵坐标之差不大于 d 的**至多 7 个点**。

此时遍历 $[x_m - d, x_m + d]$ 区间内的点求最小距离，时间复杂度为 $O(7 * n)$。

5.2 伪代码

Algorithm 6: 向量和的最小模长

Input: 个数为 n 的二维数组 V，代表 n 个向量坐标

Output: 浮点数 d，为向量和的最小模长

```
1  global d ← INF, V_X[1..n], V_Y[1..n];
2  function main(V[1..n], n):
3      for i ← 1 to n do
4          V[i][0], V[i][1] ← abs(V[i][0], abs(V[i][1]));
5      end
6      let V_X[1..n], V_Y[1..n] be new arrays copied from array V[1..n];
7      mergeSort V_X[1..n] by X-axis ;
8      mergeSort V_Y[1..n] by Y-axis ;
9      d = minDis(Y[1..n], 1, n);
10 end
```

Algorithm 7: 平面最近点对距离的伪代码

Input: 二维数组 $Y[1..Y.length]$，$V_X[l,r]$ 按纵坐标排序的点集; 点集的左右下标 l,r

Output: 浮点数 ans，为平面最近点对距离

```
function minDis(Y[1..Y.length], l, r):
    if l = r then
        return INF;
    end
    m = (l + r)/2;
    // 把已按纵坐标排序的数组Y按横坐标分成两个有序数组Y_L, Y_R
    let Y_L[1..Y_L.length] and Y_R[1..Y_R.length] be new arrays;
    Y_L.length, Y_R.length ← 0, 0;
    for i ← 1 to Y_L.length do
        if Y[i][0] ≤ mor(Y[i][0] == mandY[i][1] ≤ y_m) then
            Y_L.length = Y_L.length + 1;
            Y_L[Y_L.length] = Y[i];
        end
        else
            Y_R.length = Y_R.length + 1;
            Y_R[Y_R.length] = Y[i];
        end
    end
    // 左右两个点集内的最小距离d
    d = min(minDis(Y_L[1..Y_L.length], l, m), minDis(Y_R[1..Y_R.length], m + 1, r));
    x_l ← m − d;
    x_r ← m + d;
    // 遍历按纵坐标排好序的Y[1..Y.length]，把横坐标[x − d, x +
       d]的点集选出来形成新的点集V_new
    let V_new[1..V_new.length] be new arrays;
    V_new ← 0;
    for i ← 1 to Y.length do
        if Y[i][0] ≥ x_l and Y[i][0] ≤ x_r then
            V_new.length = V_new.length + 1;
            V_new[V_new.length] = V[i];
        end
    end
    // 遍历按纵坐标排序的点集V_new，每个点判断其下方至多 7 个点是否形成最小距离d
    for i ← 2 to V_new.length do
        for j ← max(1, i − 7) to i − 1 do
            d = min(d, dis(Y[i], Y[j]));
        end
    end
    return d;
end
```

Algorithm 8: 归并排序 $mergeSort(A, p, r)$

Input: 数组 $A[p..r]$

Output: 有序数组 $A[p..r]$

```
function mergeSort(A, p, r):
    if p < r then
        q = ⌊(p + r)/2⌋;
        mergeSort(A, p, q);
        mergeSort(A, q + 1, r);
        merge(A, p, q, r);
    end
end
```

Algorithm 9: 归并 $merge(A, p, q, r)$

Input: 数组 A，数组下标 p, q, r，满足 $p \le q < r$。且子数组 $A[p..q]$ 和 $A[q+1..r]$ 都已经排好序。

Output: 数组 A，且子数组 $A[p..r]$ 排好序

```
function merge(A, p, q, r):
    n1, n2 ← q − p + 1, r − q;
    let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
    for i ← 1 to n_l do
        L[i] ← A[p + i − 1];
    end
    for j ← 1 to n2 do
        R[j] ← A[q + j];
    end
    L[n1 + 1], R[n2 + 1] ← ∞, ∞;
    i, j ← 1, 1;
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i];
            i ← i + 1;
        end
        else
            A[k] ← R[j];
            j ← j + 1;
        end
    end
end
```

5.3 复杂度分析

由算法思想部分的分析，对点集的**预处理**中，遍历点集转换为第一象限的时间复杂度 $O(n)$。分别按横坐标和纵坐标归并排序的时间复杂度 $O(nlogn)$。

每次**分治**时，将当前点集二分，每层共 $a = 2$ 个分支。每层以因子 $b = 2$ 的速度下降。

每次**合并分支**时需要对按纵坐标排好序的点集遍历并选出横跨左右区间的点集，进一步判断最小距离。两步的时间复杂度为 $O(n), O(7n)$，即为 $O(n)$。

得到的递归式为 $T(n) = 2T(n/2) + O(n)$。解得 $T(n)$ 的时间复杂度为 $O(nlogn)$。