第三章

1. 根据表达式 $a^{2k} = a^k \cdot a^k$ 和 $a^{2k+1} = a^k \cdot a^k \cdot a$ 设计分治 (或递归) 算法求解下列问题, 并分析算法的时间复杂度。
(a) 输入实数 a 和自然数 n, 输出实数 a^n；
(b) 输入实数矩阵 A 和自然数 n, 输出实数矩阵 A^n。
解：
(a) 对于这个问题，采取分治算法将原问题分解为两个规模相同的子问题进行求解，算法设计如下：

```
CALCULATE-POWER(a, n)
1  if n == 1
2     then return a
3  if n%2 == 0
4     then c = CALCULATE-POWER(a, n/2)
5          return c × c
6     else
7          c = CALCULATE-POWER(a, (n−1)/2)
8          return c × c × a
```

算法分析： 我们可以列出算法的递归表达式:

$$T(n) = T(\lfloor n/2 \rfloor) + O(1)$$

由 master 主定理可以很容易得到解：$T(n) = \Theta(\log n)$。

(b) 我们可以采用相同的方法来分析这个问题，只是在分治算法的 merge 步骤的时候会不同。

```
CALCULATE-MATPOWER(A, n)
1  if n == 1
2     then return A
3  if n%2 == 0
4     then c = CALCULATE-POWER(A, n/2)
5          return MATRIX-MULT(c, c)
6     else
7          c = CALCULATE-POWER(A, (n−1)/2)
8          m = MATRIX-MULT(c, c)
9          return MATRIX-MULT(m, A)
```

其中，MATRIX-MULT(c, c) 这个函数表示两个矩阵相乘，由题目可知，问题中的矩阵都是方阵，设输入的矩阵 A 为 $m \times m$ 的矩阵，我们可以列出递归表达式：

$$T(n) = T(\lfloor n/2 \rfloor) + m^3$$

采用迭代法求解上面的递归表达式可得：$T(n) = \Theta(m^3 \times \log n)$。

2. 斐波那契数列满足递归方程 $F(n+2)=F(n+1)+F(n)$，其中 $F(0)=F(1)=1$。

(a) 分别用数学归纳法和第 2 章例 19 的结论, 证明:

$(1)F(n+2)=1+\sum_{i=0}^{n}F(n)$; $(2)F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$;

(b) 设计算法根据递归方程计算 $F(n)$, 将时间复杂度表达成 n 的函数;

(c) 根据第 2 章例 19 中 $F(n)$ 的解析表达式, 设计算法计算 $F(n)$, 将时间复杂度复杂性表达成 n 的函数, 并指明计算机运行该算法时可能遇到的问题。

(d) 根据表达式 $\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}F(n+1)&F(n)\\F(n)&F(n-1)\end{pmatrix}=\begin{pmatrix}F(n+2)&F(n+1)\\F(n+1)&F(n)\end{pmatrix}$ 和习题 3.1(b) 的算法, 设计算法计算 $F(n)$, 将时间复杂度复杂性表达成 n 的函数。

(e) 比较 (b),(c),(d) 得到的算法。

解：

(a)

1. 当 $n=0$ 时，$F(0+2)=1+F(0)=2$，满足表达式。

当 $n=1$ 时，$F(1+2)=1+F(0)+F(1)=3$，满足表达式。

假设当 $n<N$ 时，$F(n+2)=1+\sum_{i=0}^{n}F(n)$ 成立；

当 $n=N$ 时，$F(N+2)=F(N+1)+F(N)=1+\sum_{i=0}^{N-1}F(n)+F(N)=1+\sum_{i=0}^{N}F(n)$，表达式成立。

综上所述，$F(n+2)=1+\sum_{i=0}^{n}F(n)$。

2. 当 $n=0$ 时，$F(0+2)=2>0$，满足表达式。

当 $n=1$ 时，$F(1+2)=3>\left(\frac{1+\sqrt{5}}{2}\right)$，满足表达式。

假设当 $n<N$ 时，$F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$ 成立；

当 $n=N$ 时，$F(N+2)=F(N+1)+F(N)>\left(\frac{1+\sqrt{5}}{2}\right)^{N-1}+\left(\frac{1+\sqrt{5}}{2}\right)^{N-2}=\left(\frac{1+\sqrt{5}}{2}\right)^{N}$，表达式成立。

综上所述，$F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$。

(b) 我们根据递推方程来求解 $F(n)$，列出时间复杂度的递推方程：

$$T(n)=T(n-1)+T(n-2)+O(1)$$

用代入法可以求解：$T(n)=T(n-1)++T(n-2)+O(1)=T(n-2)+T(n-3)+T(n-3)+T(n-4)+O(2)+O(1)=\cdots=\sum_{i=0}^{n-1}2^i=\Theta(2^n)$

(c) 我们可以得到 $F(n)$ 的表达式：$F(n)=\frac{1}{\sqrt{5}}\cdot\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

利用公式求解就能将原问题转换为求解 $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 和 $\left(\frac{1-\sqrt{5}}{2}\right)^n$ 两个问题。

我们可以用习题 3.1(a) 的算法来解决这两个问题，由第一题的结论可知，该问题的时间复杂度为：$T(n)=\Theta(2\log n)$。

但该算法在实际执行的时候会出现一些问题，因为 $\sqrt{5}$ 是无理数，而计算机存储位数有限，在计算的过程中，会丢弃掉无理数的一部分，而误差会随着 n 的增加而增加，最后会影响我们对斐波那契数列的求解。

(d) 我们通过对表达式 $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{pmatrix}=\begin{pmatrix}F(n+2) & F(n+1)\\ F(n+1) & F(n)\end{pmatrix}$ 进行迭代，可以得到一个递推表达式：

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n\begin{pmatrix}F(2) & F(1)\\ F(1) & F(0)\end{pmatrix}=\begin{pmatrix}F(n+2) & F(n+1)\\ F(n+1) & F(n)\end{pmatrix}$$

我们就将原问题转换为了求解 $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n$，我们采用习题 3.1(b) 的算法，对问题进行求解，由习题 1 的分析可得，时间复杂度为：$T(n)=\Theta(8\log n)$。

(e) 下面对以上的三种方法进行总结：

(b) 中的算法简单，容易实现，但是时间复杂度太高，可以用动态规划的方法，将时间复杂度降到线性时间。

(c) 中的算法时间复杂度最低，但是因为要对无理数进行计算，会产生让人不愉快的误差。

(d) 中的算法时间复杂度和 (c) 中算法的时间复杂度时同阶的，且避免了对无理数的操作，计算很快，我认为时最理想的方法。

3. 给定平面上 n 个点构成的集合 $S=\{p_1,\cdots,p_n\}$。如果存在边平行于坐标轴的矩形仅包含 S 中的两个点 p_i 和 p_j $(1\le i,j\le n)$, 则称 p_i 和 p_j 为“友谊点对”。试设计一个分治算法统计 S 中友谊点对的个数。

解：

使用分治算法来解决这道题。首先将点的坐标离散化，将所有 x 坐标相同的点视为一个整体来进行分治。

Preprocessing:

1. 如果 $n\le 2$ 时，则算法结束。
2. 将所有的位于 S 上的点按照 x-坐标和 y-坐标进行排序。

Divide:

1. 假设 S 中最小的 x-坐标为 l，最大的 x-坐标为 r，我们要处理的问题为 $\textsc{Solve}(l,r)$ 我们令 $mid=\lfloor(l+r)/2\rfloor$。
2. 将点集分成左右两个点集 S_L和S_R，分别递归处理 $n_L=\textsc{Solve}(l,mid)$ 和 $n_R=\textsc{Solve}(mid+1,r)$。

Merge:

1. 对于位于左边区域的点 $a\in S_L$，我们现在需要找出 a 的可视区域 $[y_l,y_r]$，可视区间的意思是指在区域 $[y_l,y_r]$ 中，左边区域不存在点位于这个可视区域中。
2. 对每个位于左边区域的点 $a\in S_L$，只需在右边区域和可视区域的交集区域进行搜索就行了，记录每个点的友谊点数 n_a。
3. 返回求解的总的数目：$n_L+n_R+\sum_{a\in S_L}n_a$。

说明：

(1) 对于 Merge 的第一步，计算方法如下：对于左边区域的点，从右往左处理，每次处理一批 x-坐标相等的点。首先将这些点的 y-坐标全部插入平衡树。插入完毕之后一个一个的检查，如果一个点的 y-坐标仅在平衡树中出现了一次，那么求出它的前驱的 y 值 y_{prev} 和后继的由值 y_{succ}，区间 $[y_{prev}-1,y_{succ}-1]$ 就是我们所求的可视区间，特别要说明的一点，如果在第一次处理的时候得到的点只有一个，则设它的 $y_{prev}=-\infty$ 和 $y_{succ}=\infty$。

(2) 对于 Merge 的第二步，计算方法如下：先对于右边区域中的点，查找在可视区域范围内是否存在点，如果没有点的话，则返回 0，否则对于在可视区域的点，从左往右处理，找到该 x-坐标下位于可视区域中点的最大 y-坐标值 y_{max} 和最小 y-坐标值 y_{min}，更新可视区域为 $[y_{min}-1, y_{max}-1]$，再反复执行上面操作。
下面给出伪代码：
先将所有的位于 S 上的点按照 x-坐标和 y-坐标进行排序。

CALCULATE-FRINDLYPOINT(n, S)

```
1  if n == 2
2     then return 1
3  if n ≤ 1
4     then return 0
5  l ← min(x)，r ← max(y)
6  mid ← ⌊(l + r)/2⌋
7  n_L ← CALCULATE-FRINDLYPOINT(l, mid, S_L)
8  n_R ← CALCULATE-FRINDLYPOINT(mid + 1, r, S_R)
9  [y_prev, y_succ] ← CALCULATE-VIEWAREA(S_L)
10 for a ∈ S_L
11       do prev ← y_prev[a], succ ← y_succ[a], i ← 1, n_a ← 0
12          while FIND-NUM(prev, succ) ≠ 0
13                do x_now ← FIND-RIGHTX(i)
14                   [y_min, y_max] ← FIND-MAXMIN(x_now)
15                   [prev, succ] ← [y_min, y_max]
16                   if prev ≠ y_min
17                      then n_a ← n_a + 1
18                   if succ ≠ y_max
19                      then n_a ← n_a + 1
20                   i ← i + 1
21 return n_L + n_R + Σ_{a∈S_L} n_a
```

最后，我们来分析算法的时间复杂性：
(1) 在 Preprocessing 阶段，排序所需要的时间复杂性为 $O(n\log n)$。
(2) 在 Divide 阶段，分割集合所需要的时间复杂性为 $O(n)$。
(3) 在 Merge 阶段的第一步，维护和搜索平衡树的时间为 $n\log n$。
(4) 在 Merge 阶段的第二步，看似时二层循环，但是因为第二层循环每进行一次必会缩小可视区域，而可视区域是常数数量级，所以第二层循环能在常数级别的时间内结束，而在有序的数组中进行查找操作的时间复杂度为 $\log n$，所以总的时间复杂度为 $n\log n$。
可以给出分治算法的迭代公式：

$$T(n) = 2T(n/2) + O(n\log n)$$

我们可以很容易解出：$T(n) = O(n\log^2 n)$。

4. 给定平面上 n 个点构成的集合 S, 试设计一个分治算法输出 S 的三个点, 使得以这三个点为顶点的三角形的周长达到最小值。(提示: 模仿最邻近点的分

治过程)。
解：
我们采用分治算法的思想来求解：
Preprocessing:
1. 如果 $n \leq 2$，则算法结束。
2. 把 S 中的点分别按 x-坐标值和 y-坐标值排序。
Divide:
1. 计算 S 中各点 x-坐标的中位数 m。
2. 用垂线 $L : x = m$ 把 S 划分为两个大小相等的子集 S_L 和 S_R，集合 S_L 中的点在直线 L 左边集合 S_R 中的点在直线 L 右边。
3. 递归的在子集 S_L 和 S_R 上找出能构成三角形并且周长最小的三个点：$(p_1, p_2, p_3) \in S_L$ $(q_1, q_2, q_3) \in S_R$。
4. 记 $d = min\{Dis(p_1, p_2, p_3), Dis(q_1, q_2, q_3)\}$。
Merge:
1. 在临界区查找距离小于 d 的三个点，并且这三个点不在同一个子集中。
2. 如果找到，则新找到的三个点为所求的点，否则，就是 (p_1, p_2, p_3) 和 (q_1, q_2, q_3) 中距离最小者为周长最小的三角形顶点。
说明：
(1) 对于上述问题的描述，判断三点 $(p_1, p_2, p_3) \in S$ 是否能构成三角形，也就是判断这三点是否在一条直线上，即判断三点坐标满不满足条件 $\left|\frac{x_{p_1}-x_{p_2}}{y_{p_1}-y_{p_2}}\right| = \left|\frac{x_{p_2}-x_{p_3}}{y_{p_2}-y_{p_3}}\right|$，如果不满足上式的话，就说明可以构成三角形。
(2) 算法的关键在于 Merge 中第一步的求解，我们将搜索的区域限定在垂线 $L : x = m$ 两侧，垂线 $L_1 : x = m - \frac{d}{2}$ 和垂线 $L_2 : x = m + \frac{d}{2}$ 之间。
因为当三角形中一条边长度大于 $\frac{d}{2}$ 时，由三角形定义，两边之和大于第三边可以得到，三角形的周长将大于 d。
对于 S_L 集合中位于搜索区域的点，搜寻另外两点都位于 S_R 中，使得所构成三角形周长小于 d，同理，对于 S_R 集合中位于搜索区域的点，搜寻另外两点都位于 S_L 中，使得所构成三角形周长小于 d。
而对于位于搜索区域的点 $P \in S_L$，我们只需要在它的 P-右邻域中搜寻两点，它的右邻域为 $d \times \frac{d}{2}$ 大小的矩形。
我们将证明，这个邻域中最多存在 16 个点。将 $d \times \frac{d}{2}$ 的矩形分成 8 个 $\frac{d}{4} \times \frac{d}{4}$ 的矩形，每个矩形中最多有两个点，如果存在 3 个点在 $\frac{d}{4} \times \frac{d}{4}$ 矩形中，则三个点构成的三角形的周长将小于矩形的周长 $L < 4 \times \frac{d}{4} d = d$，与条件矛盾。
给出算法的伪代码:
先将 S 中的点分别按 x-坐标值和 y-坐标值排序。

```
CALCULATE-MINPERIMETER(S, n)
1  if n ≤ 2
2      then return 0
3  计算 S 中各点 x-坐标的中位数 m。
4  用垂线 L : x = m 把 S 划分为两个大小相等的子集 S_L 和 S_R。
5  a ← CALCULATE-MINPERIMETER(S_L, n/2)
6  b ← CALCULATE-MINPERIMETER(S_R, n/2)
7  d = min(a, b)
8  return MERGE(S_L, S_R, d)
```

最后分析算法的时间复杂性：

$$T(n) = 2T(n/2) + O(n)$$

用 Master 定理求解 $T(n) = O(n \log n)$。

5. 证明求解凸包问题的蛮力算法的正确性。
解：先给出求解凸包问题的蛮力算法的伪代码：

BruteForceCH(Q)
1 **for** $\forall A, B, C, D \in Q$
2 **do if** 其中一点位于其它三点构成的三角形内
3 **then** 从 Q 中删除该点
4 $A \leftarrow Q$ 中横坐标最大的点
5 $B \leftarrow Q$ 中横坐标最小的点
6 $S_L \leftarrow \{P | P \in Q$且$g(A, B, P) < 0\}$
7 $S_U \leftarrow \{P | P \in Q$且$g(A, B, P) > 0\}$
8 排序 S_L, S_U
9 输出 $A, S_L, B,$ 逆序S_U

蛮力算法的思想就是对点集的任意 4 个点，判断是否有点位于其它三个构成的三角形内，有的话就删除该点，外层循环是对点集所有的点的 4 层循环遍历，所以蛮力算法的时间复杂性为 $O(n^4)$，我们将采用循环不变量来证明算法的正确性。
设置循环不变量为：当前点集的凸包和原始点集的凸包是一样的。
初始化： 在循环开始前点集 Q 中间的点和原始点集中的点一样，所以凸包也时一样的，循环不变式成立。
保持： 经过一次循环迭代之后，如果不存在其中一点位于其它三点构成的三角形内，则点集跟循环前的点集一样，所以凸包也是一样的，当存在其中一点位于其它三点构成的三角形内，则删除掉该点。
现在我们来证明引理 1：给定平面点集 S，如果 $P, P_i, P_j, P_k \in S$ 是四个不同的点，且 P 位于三角形 $\Delta P_i P_j P_k$ 的内部或边界上，则 P 不是 S 的凸包顶点。
我们再给出引理 2：给定平面点集 S，如果 P 点是点集凸包上的顶点，则其余的点都在 P 与其在凸包上相邻的顶点 P_l 构成的直线 PP_l 的一侧。
现在来证明引理 2 的正确性：
因为凸包会包含点集上所有的点，所以我们只需要证明凸包的顶点点都位于直线的一侧。
如果存在凸包上的顶点 P_1, P_2 分别位于直线 PP_l 的两侧，则 ΔPP_lP_1 是凸包内的一部分，并且形 ΔPP_lP_2 也是凸包内的一部分，而两个三角形共用一条边 PP_l，因此，构成的四边形 $PP_1P_lP_2$ 位于凸包的内部，所以直线 PP_l 位于凸包内部，与假设矛盾，所以引理 2 成立。
再来证明引理 1：
假设 P 是凸包的顶点，则它与其在凸包上相邻的顶点会构成一条直线，而 $P_i, P_j, P_k \in S$，所以 P_i, P_j, P_k 将位于直线的一侧。
当 P 位于三角形 $\Delta P_i P_j P_k$ 的内部时，不可能存在一条直线通过 P 使得

P_i, P_j, P_k 位于直线的一侧，与引理 2 矛盾。
P 位于三角形 $\Delta P_i P_j P_k$ 的边界上，不失一般性，设 P 位于边 $P_i P_j$ 上，则通过 P 的直线并且满足 P_i, P_j, P_k 位于直线一侧，该直线必经过 $P_i P_j$，所以 P_i，P_j 也为凸包的顶点，并且 P 不能称之为凸包的顶点，矛盾。
综上所述，定理 1 成立，循环不变式成立。
终止：因为点集中点的个数时有限的，最终循环会停止，只留下部分点，这些点就构成了凸包的顶点，循环不变式成立。

6. (选做) 给定平面上 n 个白点和 n 个黑点, 试设计一个分治算法将每个白点与一个黑点相连, 使得所有连线互不相交, 分析算法的时间复杂度。(提示: 划分时类似于 GrahamScan 算法考虑极角, 确保子问题比较均匀)
解：
下面给出一个分治算法：
Preprocessing:
1. 如果 $n == 1$，说明平面上只有一个白点和一个黑点，就将白点和黑点连接就可以满足题意。
2. 将所有的点，根据 y-坐标进行排序。
3. 对于所有的点，考虑 y-坐标最小的点 (最低点)。如果存在多个这样的点，就考虑 x-坐标最小的点 (最左边的点)，以这点为原点，对剩余的点建立极坐标，且所有的点的极角都在范围 $[\ 0, \pi\ )$ 内。
4. 这一点可能是白点，也可能是黑点，这个不影响我们接下来的说明，不妨设该点为白点，按照极角从小到大的顺序排序。
Divide:
1. 按照极角排序后的顺序搜索，搜索直到得到黑点和白点的数目一样多为止。
2. 根据搜索的结果将平面上的点集的集合分成两个集合，递归求解就行。
Merge:
1. 将分解的方案合并，就能完成对问题的求解。
说明：
1. 对于 Divide 阶段的第一步，我们已经假设原点为白点，如果搜索的第一点为黑点，则我们就可以让原点和该点配对，剩下的点继续搜索。如果搜索的第一点为白点，则必会存在搜索到 $2k$ 个点时，包括 k 个白点和 k 个黑点。因为搜索到第一个点时，白点数量比黑点数量多一个 (第一个点为白点)，而搜索到最后一个点时黑点的数量比白点数量多一个 (最后的时候是 n 个黑点，n-1 个白点)，而每次搜索黑点数目和白点数目最多只能加 1，所以必有一个时刻，黑点和白点的数目时一样的。
2. 分成两个集合之后在递归的重复上面的划分过程进行求解。
下面给出伪代码：
先将白点和黑点按照 y-坐标值排序。

```
CALCULATE-BLACKWHITE(n, B, W)
 1  if n == 1
 2     then return (B, W)
 3  [s_0, label_0] ← FIND-LOWPOINT(B, W)
 4  [S_polar, Label] ← CACULATE-POLAR(B, W, s_0)
 5  [S_polar, Label] ← SORT(S_polar, Label)
 6  num_b ← 0, num_w ← 0, B_1 ← Null, B_1 ← Null
 7  if Label[1] ≠ label_0
 8     then B_1 ← B − {s_0}, W_1 ← W − {s_1}
 9          num_b ← 1, num_w ← 1
10     else
11          while num_b == num_w&&num_b ≠ 0
12               do s_i ∈ S_polar
13                  if Label[i] == label_0
14                     then num_b ← num_b + 1
15                          B_1 ← B_1 ⋃ s_i
16                     else  num_w ← num_w + 1
17                          W_1 ← W_1 ⋃ s_i
18  CALCULATE-BLACKWHITE(num_b, B_1, W_1)
19  CALCULATE-BLACKWHITE(n − num_b, B − B_1, w − W_1)
```

最后进行时间复杂度分析：
该算法主要是划分并不稳定，但是平均时间复杂度为 $O(n\log^2 n)$。

7. 给定凸多边形 $p_1, p_2, \cdots, p_n$ (边界逆时针顺序) 和 n 个点 $q_1, q_2, \cdots, q_n$, 试设计一个分治算法计算 $q_1, q_2, \cdots, q_n$ 中位于凸多边形 $p_1, p_2, \cdots, p_n$ 内部的点的个数, 使其时间复杂度是 n^2 的严格低阶函数。

解：

我们设计一个分治算法来解决这个问题：

Preprocessing:

1. 找出 $p_1, p_2, \cdots, p_n$ 中最大的 x-坐标值 x_{max} 最小的 x-坐标值 x_{min}，最大的 y-坐标值 y_{max} 最小的 y-坐标值 y_{min}。
2. 在 $q_1, q_2, \cdots, q_n$ 中去掉满足 $x < x_{min}$，$x > x_{max}$，$y < y_{min}$，$y > y_{min}$ 条件的点，对剩下的点进行 x-坐标值排序。
3. 当 $n == 1$ 时，选择 $p_1, p_2, \cdots, p_n$ 中 x 坐标最小的点 p_0，如有多个，则选择其中 y 坐标最小的那一个点，从这点的左侧点开始，计算其它点到这一点的极角值，并且按照原来逆时针的顺序排列，将这一点与 S 中的点也连接起来算极角值，将该极角值插入到上面的序列中，得到前后两个点 p_l, p_r ，判断该点是否在三角形 $\Delta p_0 p_l p_r$ 中，如果在返回 1，否则返回 0。

Divide:

1. 计算 S 中各点 x-坐标的中位数 x_m 和 Y-坐标的中位数 y_m。
2. 用垂线 $L_1 : x = x_m$ 和水平线 $L_2 : y = y_m$ 把 S 划分为四个大小相等的子集 $S_{LT}, S_{RT}, S_{LB}, S_{RB}$。
3. 递归的在子集 $S_{LT}, S_{RT}, S_{LB}, S_{RB}$ 中寻找满足条件的点的个数 $n_{LT}, n_{RT}, n_{LB}, n_{RB}$。

Merge:

1. 返回 $n_{LT} + n_{RT} + n_{LB} + n_{RB}$。

接下来给出伪代码：

```
CALCULATE-INNERNUM(P, Q, n)
 1  x_max ← FIND-MAXX(P)
 2  x_min ← FIND-MINX(P)
 3  y_max ← FIND-MAXY(P)
 4  y_min ← FIND-MINY(P)
 5  WIPEOUT(Q, x_max, x_min, y_max, y_min)
 6  if n == 1
 7     then [p_0, P_polar, q_polar] ← POLAR-ANGLE(P, Q)
 8          [p_l, p_r] ← FIND-POINT(P_polar, q_polar)
 9          if IS-INTRIANGLE(Q, p_0, p_l, p_r)
10             then return 1
11             else  return 0
12  计算 S 中各点 x-坐标的中位数 x_m 和 Y-坐标的中位数 y_m
13  用垂线 L_1 : x = x_m 和水平线 L_2 : y = y_m 把 S 划分为四个大小相等的子集
14  a ← CALCULATE-INNERNUM(S_LT, Q, n/4)
15  b ← CALCULATE-INNERNUM(S_RT, Q, n/4)
16  c ← CALCULATE-INNERNUM(S_LB, Q, n/4)
17  d ← CALCULATE-INNERNUM(S_RB, Q, n/4)
18  return a + b + c + d
```

最后给出算法的时间复杂度：$O(n\log n)$。

8. 输入含有 n 个顶点的加权树 T 和实数 τ , 树 T 中每条边的权值均非负, 树中顶点 x, y 的距离 $dis(x, y)$ 定义为从 x 到 y 的路径上各边权值之和。试设计一个分治算法输出满足 $dis(x, y) \leq \tau$ 的顶点对的个数。

解：

我们将距离分为两部分计算，第一部分是两个节点在同一棵子树中，并且距离小于 τ，第二部分是两个节点在不同的子树中，并且距离小于 τ。设计一个分治算法：

Preprocessing:

1. 如果 $n \leq 2$ 时，则算法结束。

Divide:

1. 计算得到加权树中的一点 t_m，使得删除这个点后最大的子树是最小的。
2. 设这点的子节点个数为 k，删除这点，把 T 分成 k 个子树。
3. 递归在子树上找出满足条件的点对数目 $S_1, S_2, \cdots, S_k$。

Merge:

1. 求出每个节点到根节点 t_m 的距离 $A = \{a_1, a_2, \cdots, a_{n-1}\}$。
2. 将集合 A 中元素排序。
3. 求解满足条件 $a_i + a_j \leq \tau (1 \leq i < j \leq n-1)$ 的数目 n_a。
4. 求解满足条件 $a_i + a_j \leq \tau (i, j \in T_l 且 1 \leq l \leq k)$ 的数目 $n_1, n_2, \cdots, n_k$。
5. 返回总的数目：$\sum_{i=1}^{k} S_k + \left(n_a - \sum_{j=1}^{k} n_j\right)$。

说明：

(1) 在 Divide 阶段中，找到 t_m 的算法先可以采用动态规划的方法对树进行深度优先搜索，得到每个节点分割之后，最大子树的节点数，再通过在 n 个数中寻找最大的数的方法，得到 t_m。

(2) 在 Merge 阶段中，第一步采用深度遍历算法，得到距离；因为在在 Divide 阶段中已经对点集进行了划分，所以第三步和第四步所求解的问题时相同的，我们转换为求解满足条件 $a_i + a_j \leq \tau (1 \leq i < j \leq n)$ 的数目 k。而对于这类问题，我们可以先对距离序列排序，然后找头尾两个数，如果符合情况，算中间的个数，然后从头的下一个开始算，如果不符合情况说明太大，要从尾的前一个开始计算，直到头等于尾。
下面给出伪代码：

CALCULATE-TREE(n, T, τ)

if $n == 2$
  then return $dis(x, y) < \tau ? 1 : 0$
if $n \leq 1$
  then return 0
$(t_m, k) \leftarrow$ CALCULATE-TREEHEART(T)
for $i \leftarrow 1$ **to** k
  then CALCULATE-TREE(n_i, T_i, τ)
求出每个节点到根节点 t_m 的距离 $A = \{a_1, a_2, \cdots, a_{n-1}\}$
$A \leftarrow$ SORT(A)
$n_a \leftarrow$ CALCULATE-NUM(A, τ)
for $j \leftarrow 1$ **to** k
  then $n_j \leftarrow$ CALCULATE-NUM(A_j, τ)
return $\sum_{i=1}^{k} S_k + \left(n_a - \sum_{j=1}^{k} n_j\right)$

接下来分析算法的时间复杂度：
(1) 在 Divide 阶段中，寻找 t_m 的时间复杂度为 $O(n)$
(2) 在 Merge 阶段中，第一步的时间复杂度为 $O(n)$；第二步的时间复杂度为 $O(n \log n)$；第三步和第四步的时间复杂度均为 $O(n)$。
综上所述，每一次总的时间复杂度为 $O(n \log n)$，而因为我们每次迭代寻找的是最优的 t_m，总的次数为 $O(\log n)$，所以算法总的时间复杂度为 $O(n \log^2 n)$。

9. 设 $X[0 : n-1]$ 和 $Y[0 : n-1]$ 为两个数组, 每个数组中的 n 个均已经排好序, 试设计一个 $O(\log n)$ 的算法, 找出 X 和 Y 中 $2n$ 个数的中位数, 并进行复杂性分析。
解：
设计分治算法来实现：
Preprocessing:
1. 将 $X[0 : n-1]$ 和 $Y[0 : n-1]$ 两个数组按从小到大的顺序排序，记为 $A[0 : n-1]$ 和 $B[0 : n-1]$。
2. 如果 $n \leq 2$，问题很容易解决。
Divide:
1. 找到两个数组的中位数 mid_a, mid_b。
2. 将两个数组分别平均分成两部分 $A[0 : mid_a], A[mid_a : n-1]$ 和 $B[0 : mid_b], B[mid_b : n-1]$。
3. 将 $A[0 : mid_a]$ 和 $B[mid_b : n-1]$ 分为一组构成新问题的子问题，求出该子问题的中位数 ans_1；将 $A[mid_a : n-1]$ 和 $B[0 : mid_a]$ 分为一组构成新问题的另一个子问题，求出该子问题的中位数 ans_2。

Merge:
1. 比较两个中位数的大小，如果 $mid_a < mid_b$，则 ans_2 为原问题的解，如果 $mid_a > mid_b$，则 ans_1 为原问题的解，如果 $mid_a = mid_b$，则 mid_a 和 mid_b 为原问题的解。

说明：
(1) 对于 Divide 阶段的第一步，寻找两个数组的中位数，为了保证得到的子问题中 A 数组和 B 数组中数的个数一样，如果 n 为偶数，则对 A 数组取上中位数 $\lfloor n/2 \rfloor$，对 B 数组取下中位数 $\lceil n/2 \rceil$，如果 n 为奇数，则就取中位数 $n/2$。
(2) 对于 Merge 阶段的第一步，如果 $mid_a < mid_b$，则说明对于数组 A 中比 mid_a 小的数在数组 A 中小于 $n/2$ 个数，在数组 B 中小于 $n/2$ 个数，所以这些数肯定不是中位数。同理，对于数组 B 中比 mid_b 大的数也不是中位数；如果 $mid_a > mid_b$，则说明对于数组 A 中比 mid_a 大的数在数组 A 中大于 $n/2$ 个数，在数组 B 中大于 $n/2$ 个数，所以这些数肯定不是中位数。同理，对于数组 B 中比 mid_b 小的数也不是中位数；如果 $mid_a = mid_b$，则说明 mid_a 比 A 数组中 $n/2$ 个数大，比 B 数组中 $n/2$ 个数大，所以 mid_a 和 mid_b 为中位数。

FIND-MIDNUM$(A, l_1, r_1, B, l_2, r_2)$

if $(r_1 - l_1 + 1)\%2 = 0$
  then $mid_1 \leftarrow (l_1 + r_1)/2 + 1$; $//A$ 取上中位数
    $mid_2 \leftarrow (l_2 + r_2)/2$; $//B$ 取下中位数
  else
    $mid1 \leftarrow (l_1 + r_1)/2$;
    $mid2 \leftarrow (l_2 + r_2)/2$;
if $l_1 = r_1 \,\&\&\, l_2 = r_2$ //n=1 的情况
  then return $(double)(A[l_1] + B[l_2])/2$;
if $r_1 - l_1 = 1 \,\&\&\, r_2 - l_2 = 1$ //n=2 的情况
  then return $(\max\{A[r_1], B[r_1]\} + \min\{A[r_2], B[r_2]\})/2$;
if $A[mid_1] = B[mid_2]$
  then return $A[mid_1]$;
elseif $A[mid_1] > B[mid_2]$
  then return FIND-MIDNUM$(A, l_1, mid_1, B, mid_2, r_2)$;
else return FIND-MIDNUM$(A, mid_1, r_1, B, l_2, mid_2)$;

算法复杂性分析:
Preprocessing:$O(1)$；
Divide 阶段：寻找两个有序数组的中位数，并进行比较，需要 $O(1)$ 时间；
Conquer 阶段：需要 $T(n/2)$ 时间；
Merge 阶段：需要 $O(1)$ 时间。
递归方程：

$$T(n) = \begin{cases} O(1), & n = 1, 2 \\ T(n/2) + O(1), & n \geq 3 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(\log(n))$。

10. 设 $A[1:n]$ 是由不同实数组成的数组, 如果 $i<j$ 且 $A[i]>A[j]$, 则称实数对 $(A[i],A[j])$ 是该数组的一个反序。如, 若 $A=[3,5,2,4]$, 则该数组存在 3 个反序 (3,2)、(5,2) 和 (5,4)。反序的个数可以用来衡量一个数组的无序程度。设计一个分治算法 (要求时间复杂度严格低于 n^2), 计算给定数组的反序个数。

解：

在这个问题上采用类似于归并排序的算法, 只是每次在 merge 的时候每次组间比较都根据比较结果对反序对计数量进行加一操作即可。

Preprocessing: 如果数组 A 中仅有一个数，算法结束；

Divide:

(1) 用数组 A 的中间位置的数将 A 分成两个长度相当的数组 $A_1=A[1:\lfloor n/2\rfloor]$ 和 $A_2=A[\lfloor n/2\rfloor+1:n]$；

(2) 递归地计算 A_1 和 A_2 各自的反序数 a_1 和 a_2；

Merge: 将数组 A_1 和 A_2 进行归并排序，同时得到两个数组之间的反序数 m，则 a_1+a_2+m 为 A 的反序数。

FIND-INVERSIONCOUPLES(A)

```
1  n ← length(A);
2  if n = 1
3     then return 0;
   //递归地求子问题的反序数;
4  a1 ←FIND-INVERSIONCOUPLES(A[1 : ⌊n/2⌋])
5  a2 ←FIND-INVERSIONCOUPLES(A[⌊n/2⌋ + 1 : n]);
   //求两组元素之间的反序数;
6  num ← 0;
7  i1 ← 1; i2 ← ⌊n/2⌋ + 1;
8  k ← 1;
9  while i1 ≤ ⌊n/2⌋ && i2 ≤ n
10       do
11          if A[i1] > A[i2]
12             then num + +;
13                  new_A[k + +] ← A[i2 + +];
14             else new_A[k + +] ← A[i1 + +];
15 while i1 ≤ ⌊n/2⌋
16       do new_A[k + +] = A[i + +];
17 while i2 ≤ ⌊n/2⌋
18       do new_A[k + +] = A[i + +];
19 return a1 + a2 + m;
```

算法复杂性分析: 该算法和归并排序过程完全类似，很容易给出递归方程：

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

用 Master 定理求解 $T(n)$, 得 $T(n)=O(n\log(n))$。

11. 给定一个由 n 个实数构成的集合 S 和另一个实数 x, 判断 S 中是否有两个元素的和为 x。试设计一个分治算法求解上述问题, 并分析算法的时间复杂度。

解：
Preprocessing:
1. 将集合 S 按从小到大的顺序排序。
2. 将 S 中去掉相同的元素，如果存在多个等于 $x/2$ 的元素，则直接返回 True。
Divide:
1. 得到一个新的集合 $S_1 = \{y|y = x - a; a \in S\}$，并且容易得知，$S_1$ 是从大到小排列的。
2. 将 S_1 从小到大排列。
Merge:
1. 通过将 S 和 S_1 两个有序的数列合并，如果存在相同的数，则返回 True，否则返回 False。
下面给出伪代码：

IsEqual(S, x)
1 $S \leftarrow$ Sort(S)
2 **if** Find$(x/2) \geq 2$
3 **then return** True
4 $S \leftarrow$ WarpOut(S)
5 $S_1 \leftarrow x - S$
6 $S_1 \leftarrow$ Sort(S_1)
7 $i \leftarrow 0, j \leftarrow 0$
8 $n \leftarrow length(S)$
9 **while** $i \leq n$ && $j \leq n$
10 **do**
11 **if** $S[i] > S_1[j]$
12 **then** $j++$
13 **elseif** $S[i] < S_1[j]$
14 **then** $i++$
15 **else return** True
16 **return** False

时间复杂度为 $O(n \log n)$。

12. 设单调递增有序数组 A 中的元素被循环右移了 k 个位置, 如 $< 35; 42; 5; 15; 27; 29 >$ 被循环右移两个位置 $(k = 2)$ 得到 $< 27; 29; 35; 42; 5; 15 >$。
(1). 假设 k 已知, 给出一个时间复杂度为 $O(1)$ 的算法找出 A 中的最大元素。
(2). 假设 k 未知, 设计一个时间复杂度为 $O(logn)$ 的算法找出 A 中的最大元素。
解：
考虑数组元素严格单调递增的情况。
(1) 算法伪代码如下：

FIND-MAX(A, k)

```
1  n ← A.length;
2  index ← k%n;
3  if index == 0
4      then return A[n − 1];
5  else return A[index − 1];
```

算法的时间复杂度显然为 $O(1)$.
(2)
数组 A 在初始阶段是单调递增的，当 $k \geq n$ 时，与 $k \leftarrow k\%n$ 的情况一致，我们在这里只考虑 $k < n$ 的情况。
由第一问可知，左移 $k > 0$ 位之后，最大数在第 $k-1$ 位，这样就将数组 A 分为两部分，其中 $A[0 : K-1]$ 和 $A[k : n-1]$ 都是单调递增的并且 $A[0 : K-1]$ 中的元素都要比 $A[k : n-1]$ 中的元素大。这就给我们设计分治算法提供来办法。
算法伪代码如下

FIND-MAX(A)

```
1  n ← A.length;
2  if n ≤ 3
3      then return max(A)
4  mid ← n/2 − 1;
5  if A[0] < A[mid] && A[mid] < A[n − 1]
6      then return A[n − 1];
7  elseif A[0] > A[mid]
8      then return FIND-MAX(A[0 : mid]);
9  else return FIND-MAX(A[mid : n − 1]);
```

递归方程：

$$T(n) \begin{cases} = O(1), & n = 2, 3 \\ \leq T(n/2) + O(1), & n \geq 4 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(\log n)$.

13. 给定非负数组 $A[1 : n]$，$B[1 : m]$ 和整数 k。试设计一个算法计算集合 $\{A[i] \cdot B[j] | 1 \leq i \leq n, 1 \leq j \leq m\}$ 中的第 k 小的元素。
解：
先将数组 A 和数组 B 排好序。

```
FIND-KMIN(A, B, k)
 1  m ← A.Length, n ← B.length
 2  mid_a ← A[m/2], mid_b ← B[n/2]
 3  mid ← mid_a · mid_b
 4  cnt ← JUDGE(mid, A, B)
 5  if cnt < k
 6     then FIND-KMIN(A[mid_a : m − 1], B[mid_b : n − 1], k)
 7          FIND-KMIN(A[0 : mid_a], B[mid_b : n − 1], k)
 8          FIND-KMIN(A[mid_a : m − 1], B[0 : mid_b], k)
 9  elseif cnt > k
10     then FIND-KMIN(A[0 : mid_a], B[0 : mid_b], k)
11          FIND-KMIN(A[0 : mid_a], B[mid_b : n − 1], k)
12          FIND-KMIN(A[mid_a : m − 1], B[0 : mid_b], k)
13  else return mid
```

```
JUDGE(mid, A, B)
 1  m ← A.Length, n ← B.length
 2  num ← m/2 × n/2
 3  for i ← 0 to m/2
 4       do
 5          now ← mid/A[i]
 6          cnt ← BINARY-SEARCH(B[n/2 : n − 1], now)
 7          num ← num + cnt
 8  for j ← 0 to n/2
 9       do
10          now ← mid/B[j]
11          cnt ← BINARY-SEARCH(A[m/2 : m − 1], now)
12          num ← num + cnt
13  return num
```

上面出现的函数 BINARY-SEARCH(A, now) 表示的不是二分查找的算法，而是查找 now 在 A 中插入的位置，返回查找的位置前面有多少个数。
递归方程：

$$T(n) = \begin{cases} O(1), & m, n = 1 \\ 3T(n/2) + O(n \log n), & m, n \geq 2 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(n^{\log_2 3})$.

14. 设 M 是一个 $m * n$ 的矩阵，其中每行的元素从左到右单增有序，每列的元素从上到下单增有序。给出一个分治算法计算出给定元素 x 在 M 中的位置或者表明 x 不在 M 中。分析算法的时间复杂性。
解：

```
FIND-X(M, x, i, j)
    /* 返回给定元素 x 在 M 中的位置 (i, j),(−1, −1) 表明 x 不在 M 中。*/
 1  m ← M.Rownum, n ← Columnnum
 2  if M[[m/2] + 1, [n/2] + 1] = x
 3     then return ([m/2] + 1, [n/2] + 1);
 4  if m == 1&&n == 1
 5     then if M[0, 0] == x
 6             then return (i, j);
 7             else  return (−1, −1);
 8  if M[m/2, n/2] > x
 9     then if (a1, a2) ← Find − X(M[0 : m/2, 0 : n/2], x, i, j) ≠ (−1, −1)
10             then return (a1, a2);
11          elseif (a1, a2) ← Find − X(M[0 : m/2, n/2 : n − 1], x, i, j + n/2) ≠ (−1, −1)
12             then return (a1, a2);
13          elseif (a1, a2) ← Find − X(M[m/2 : m − 1, 0 : n/2], x, i + m/2, j) ≠ (−1, −1)
14             then return (a1, a2);
15          else return (−1, −1);
16  elseif M[m/2, n/2] < x
17     then if (a1, a2) ← Find − X(M[m/2 : m − 1, n/2 : n − 1], x, i + m/2, j + n/2) ≠ (−1,
18             then return (a1, a2);
19          elseif (a1, a2) ← Find − X(M[0 : m/2, n/2 : n − 1], x, i, j + n/2) ≠ (−1, −1)
20             then return (a1, a2);
21          elseif (a1, a2) ← Find − X(M[m/2 : m − 1, 0 : n/2], x, i + m/2, j) ≠ (−1, −1)
22             then return (a1, a2) + ([m/2] + 1, 0);
23          else return (−1, −1);
24  else return (i, j)
```

算法复杂性分析:
Preprocessing: $O(1)$;
Divide: $O(1)$;
Conquer: $3T(n/4)$;
Merge: $O(1)$.
递归方程:

$$T(n) = \begin{cases} O(1), & m, n = 1 \\ 3T(n/4) + O(1), & m, n \geq 2 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(n^{\log_4 3})$.