1

```
function findMedianSortedArrays(X, Y):
    n = length(X)
    if n != length(Y):
        return "两个数组必须具有相同的长度"

    low = 0
    high = n

    while low <= high:
        i = (low + high) / 2          // X 的分割位置
        j = (n + n + 1) / 2 - i       // Y 的分割位置

        // 处理 X 的边界情况
        X_left_max = (i == 0) ? -∞ : X[i - 1]
        X_right_min = (i == n) ? +∞ : X[i]

        // 处理 Y 的边界情况
        Y_left_max = (j == 0) ? -∞ : Y[j - 1]
        Y_right_min = (j == n) ? +∞ : Y[j]

        // 检查是否找到正确的分割位置
        if X_left_max <= Y_right_min && Y_left_max <= X_right_min:
            // 找到了正确的分割位置
            if (n + n) % 2 == 0:
                return (max(X_left_max, Y_left_max) + min(X_right_min, Y_right_min)) / 2
            else:
                return max(X_left_max, Y_left_max)

        // 在 X 数组中移动分割位置
        else if X_left_max > Y_right_min:
            high = i - 1   // 向左移动分割位置
        else:
            low = i + 1    // 向右移动分割位置

    return "输入的数组未排序或无效"
```

1. 建立递归方程

对于上文中讨论的中位数查找算法，可以定义如下：

1. **Divide（分割）**：将两个数组分割成子数组。这个操作通常是常数时间，所以可以表示为 $D(n) = O(1)$ 。
2. **Conquer（解决子问题）**：对 a 个规模为 $\frac{n}{b}$ 的子问题进行递归求解。对于这个中位数查找算法，通常情况下$a = 2$ 和$b = 2$（每个数组规模减半）。
3. **Combine（合并）**：合并结果。对于中位数查找，合并的操作也是常数时间，即 $C(n) = O(1)$。

结合这些，可以建立如下的递归方程：

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(1)$

2. Master定理的条件

Master定理适用于特定形式的递归关系 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$，其中：

- $a \geq 1$ 是子问题的数量。
- $b > 1$ 是子问题规模的缩小比例。
- $f(n)$ 是合并步骤的复杂度。

在这种情况下，$f(n) = O(1)$，这是常数时间。因此，需要检查 $f(n)$ 与 $n^{\log_b a}$ 之间的关系。

3. 计算 $n^{\log_b a}$

对于递归关系：

- $a = 2$
- $b = 2$

因此，

$$\log_b a = \log_2 2 = 1$$

这意味着 $n^{\log_b a} = n$。

4. 使用Master定理的限制

根据Master定理的第一种情况：

- 如果 $f(n)$ 的增长速度比 $n^{\log_b a}$ 要小（即 $f(n) = O(n^{\log_b a - \epsilon})$ 对于某个 $\epsilon > 0$），则 $T(n)$ 的复杂度为 $\Theta(n^{\log_b a})$。

在这里，$f(n) = O(1)$ 是常数时间，明显小于 n，所以符合条件，得到 $T(n) = \Theta(n)$。

5. 实际算法时间复杂度

虽然Master定理得出的 $T(n) = \Theta(n)$ 在形式上是正确的，但实际上，算法采用的是二分法，时间复杂度应为 $O(\log n)$，而不是线性复杂度。

这是因为Master定理并未考虑算法的具体实现细节。该算法利用的是特定的搜索策略，通过不断缩小问题规模，以对数时间找到中位数，而不是简单的合并操作。

2

```
function findMedianSortedArrays(X, Y):
    m = length(X)  // 数组 X 的长度
    n = length(Y)  // 数组 Y 的长度

    if m > n:
        return findMedianSortedArrays(Y, X)  // 确保 X 是较小的数组

    low = 0
    high = m
    half_len = (m + n + 1) / 2  // 总长度的一半

    while low <= high:
        i = (low + high) / 2          // 在 X 中的分割点
        j = half_len - i              // 在 Y 中的分割点

        // 处理边界情况
        maxX = (i == 0) ? -∞ : X[i - 1]  // X 左侧的最大值
        minX = (i == m) ? ∞ : X[i]      // X 右侧的最小值
        maxY = (j == 0) ? -∞ : Y[j - 1]  // Y 左侧的最大值
        minY = (j == n) ? ∞ : Y[j]      // Y 右侧的最小值

        if maxX <= minY and maxY <= minX:
            // 找到正确的分割
            if (m + n) % 2 == 0:
                return (max(maxX, maxY) + min(minX, minY)) / 2  // 偶数情况
            else:
                return max(maxX, maxY)  // 奇数情况
        elif maxX > minY:
            // 在 X 中向左移动分割点
            high = i - 1
        else:
            // 在 X 中向右移动分割点
            low = i + 1

    // 如果到达这里，说明输入数组未排序
    throw new Exception("输入数组未排序")
```

好的，下面是将伪代码的注释改为中文的版本，以及算法时间复杂度的证明。

时间复杂度证明

1. **选择较小的数组**:
   始终保证 X 是较小的数组（长度 m 小于等于长度 n）。这样，二分查找的复杂度将仅依赖于较小的数组 X 的长度。
2. **二分查找的次数**:
   在每次迭代中，我们根据当前的分割点 i 来调整搜索范围（通过修改 `low` 和 `high`）。由于每次都将搜索空间缩小一半，因此最多进行 $O(\log m)$ 次迭代。
3. **边界条件处理**:
   在每次迭代中，计算 `maxX`，`minX`，`maxY`，`minY` 的操作是常数时间复杂度 $O(1)$。
4. **综合时间复杂度**:
   因此，整个算法的时间复杂度为 $O(\log m)$，由于保证 $m \leq n$，所以最终的时间复杂度为:

$$O(\log(\min(m, n))) \subseteq O(\log(m + n))$$