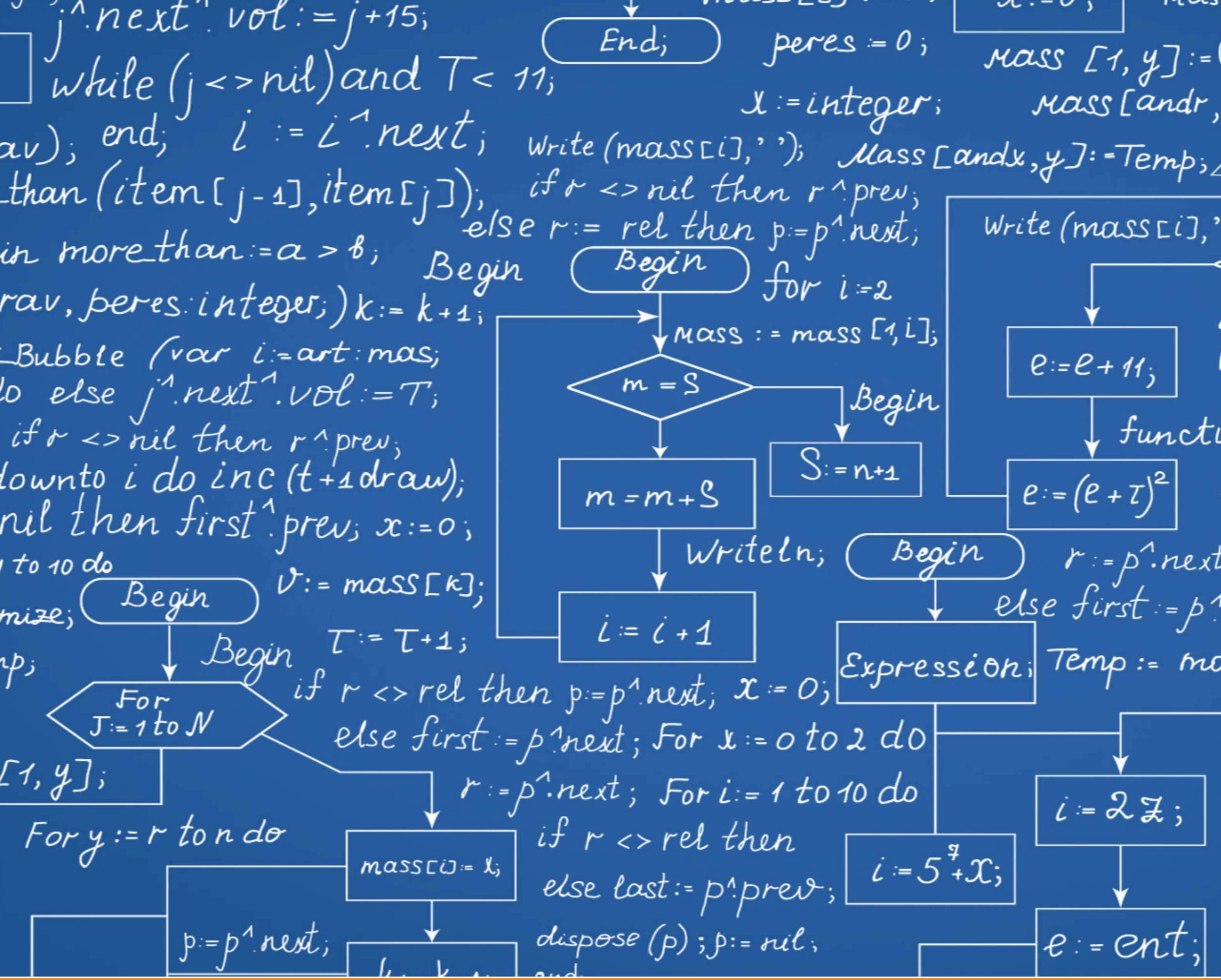

何老师算法课笔记

还没有想好的副标题

作者：计卓 1801 全体

组织：华中科技大学

时间：2020 年 11 月 11 日

版本：1.0.0

目录

第 1 章 分治算法之平面最近点对问题

内容提要

- 平面最近点对问题定义
- 分治算法时间复杂度分析
- 分治算法设计
- 伪代码

1.1 平面最近点对问题定义

给定二维平面上的 $n(n \geq 2)$ 个不同的点 p 组成点集 $P = \{p_i | 1 \leq i \leq n\}$，设计算法寻找欧式距离最近的点对 (A, B)。

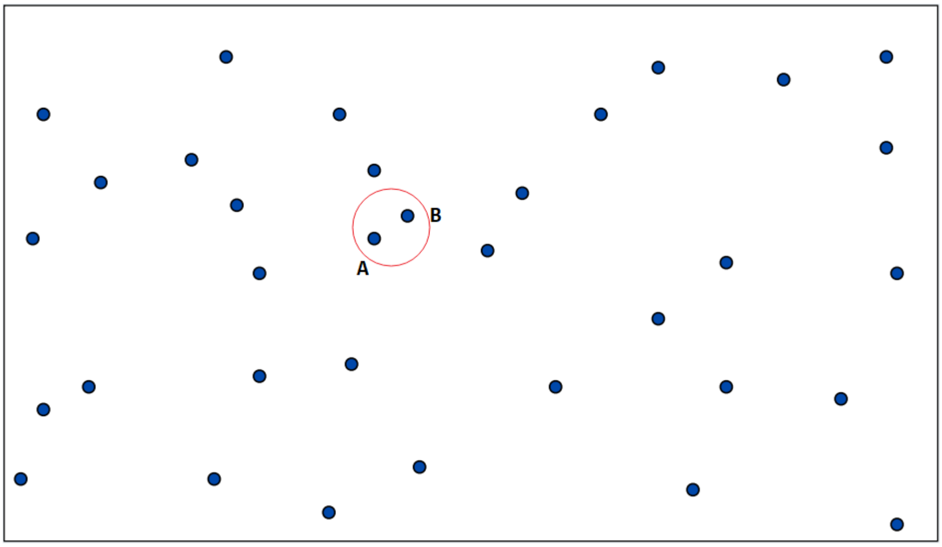

图 1.1: 问题定义图例

如上图Figure 1.1中点对 (A, B) 即为问题的答案。

1.2 分治算法设计

对于这样一个问题，我们很直接地可以使用 BF (Brute Force) 算法进行暴力求解，即二重循环计算所有点之间的距离，从而获得最小距离，显然该算法的时间复杂度为 $O(n^2)$。那么有没有更快的算法呢？本章我们使用经典的算法思想——分治，设计一个 $O(n \log n)$ 的算法。

1.2.1 分治问题

遵循分治思想，我们首先要考虑如何分治问题使得问题规模约减。

我们使用 X 坐标作为第一关键字、Y 坐标作为第二关键字，对点集 P 进行排序，并以点 $p_{\lfloor \frac{n}{2} \rfloor}$ 作为分治点，获得如下两个点集：

$$P_1 = \{p_i \mid 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$$

$$P_2 = \{p_i \mid \lfloor \frac{n}{2} \rfloor < i \leq n\}$$

这样就将当前问题约减为两个规模为 $\frac{n}{2}$ 的子问题分治过程如Figure 1.2中所示。

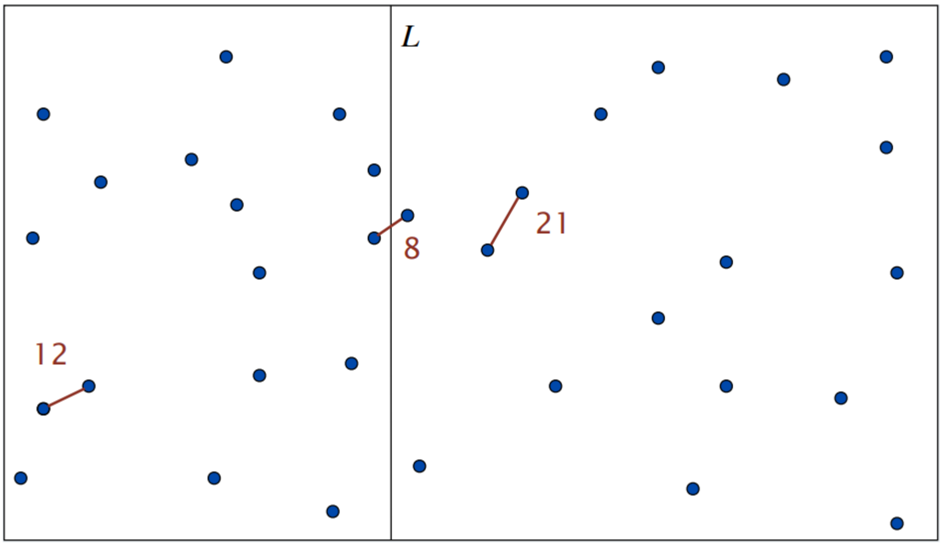

图 1.2: 分治过程图例

如此递归下去，我们可以求得两个点集相对应的最近点对距离 δ_1, δ_2，取其中较小值记为 $\delta = \min\{\delta_1, \delta_2\}$。

当分治到点集大小为 2 个或 3 个时，可以在常数时间内计算出子问题的解。

1.2.2 合并结果

接着，我们需要考虑如何合并子问题的解。

上述的 δ 一定是正确的合并结果嘛？显然不是，我们并没有考虑，一端在 P_1，一端在 P_2 的线段。因此，在合并阶段，我们要将这种情况考虑在内。

这里，我们将所有横坐标与分治点 $p_{\lfloor \frac{n}{2} \rfloor}$ 的横坐标 $x_{\lfloor \frac{n}{2} \rfloor}$ 差值小于 δ 的点组成集合 B，即

$$B = \{p_i \mid \left| x_i - x_{\lfloor \frac{n}{2} \rfloor} \right| \leq \delta,\ 1 \leq i \leq n\}$$

因为只有 B 集合中的点之间的距离才有可能小于 δ。B 集合如下图Figure 1.3中阴影部分所示:

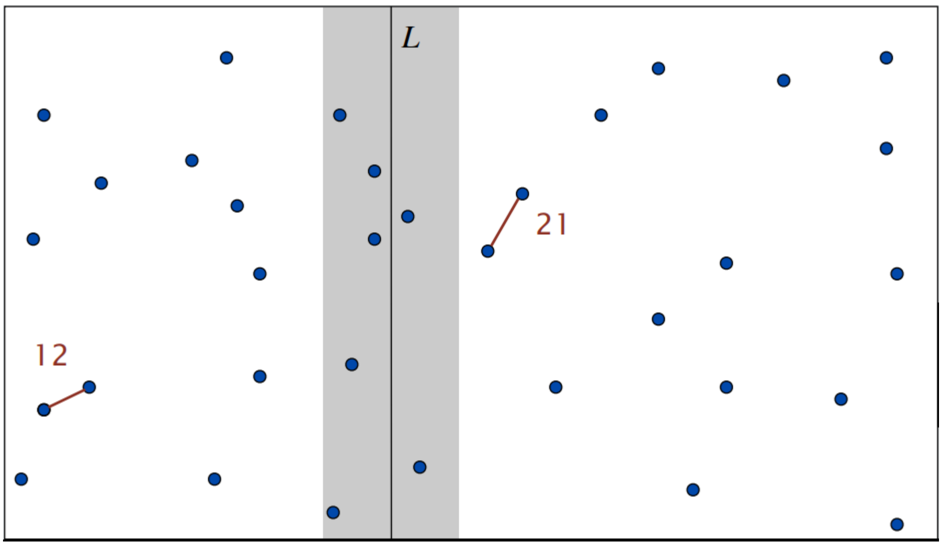

图 1.3: 合并过程图例

进一步，我们的目标是检验在 B 集合中是否存在距离比 δ 更近的点对，以此更新当前问题

的解。因此，对于每个 $p_i = (x_i, y_i) \in B$ 遍历所有在其之下竖直距离不超过 δ 的点，即遍历集合

$$C(p_i) = \{p_j \mid y_i - \delta \le y_j \le y_i, p_j \in B\}$$

为了方便遍历，我们可能会想到对 B 集合中的点，以 Y 坐标为第一关键字，X 坐标为第二关键字，进行排序。但是如此一来，每一次合并的时间复杂度为 $O(n\log n)$，徒增时间消耗，因此我们采取合并策略，即按照 Y 坐标为关键字，进行 P_1, P_2 的归并来直接获得排序后的集合 B，这样只需要 $O(n)$ 的时间。

考虑到 $C(p_i)$ 会因为归并操作而维持在 $O(n)$ 数量级，其实不然，该集合的大小不会超过 7。下面给出证明。

根据定义，$C(p_i)$ 中的点的纵坐标均处于 $(y_i - \delta, y_i]$ 范围内，且其中的所有点的横坐标均处于 $(x_m - \delta, x_m + \delta)$ 范围内。这样便构成了一个 $2\delta \times \delta$ 的矩形。如下图Figure 1.4所示 。

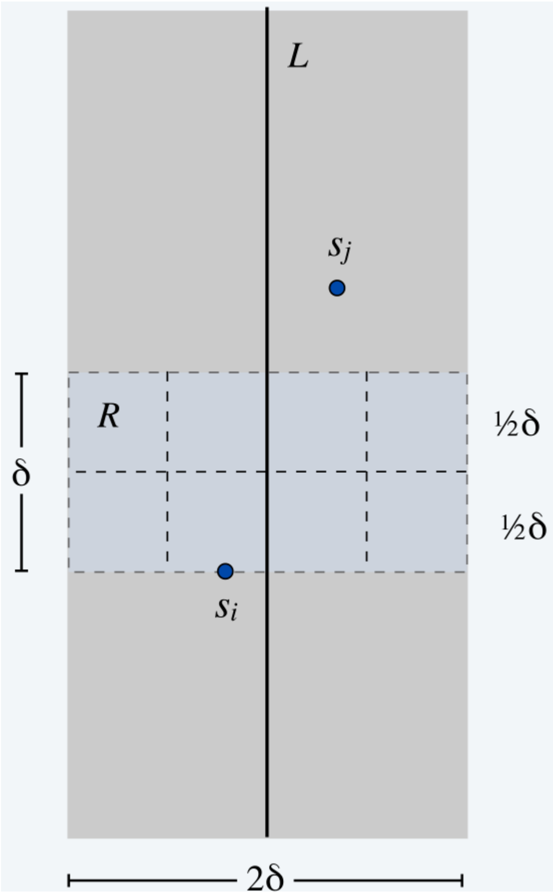

图 1.4: $C(p_i)$

接着，我们将这个矩形分拆成左右两个 $\delta \times \delta$ 的正方形，左侧正方形的点集为 $C(p_i) \cap P_1$，右侧正方形的点集为 $C(p_i) \cap P_2$，从上述的分治过程可知，这两个点集内的点之间的距离一定不小于 δ。

进一步，我们将 $\delta \times \delta$ 正方形，分拆成四个 $\frac{\delta}{2} \times \frac{\delta}{2}$ 小正方形，因为这个小正方形的对角线为 $\frac{\delta}{\sqrt{2}} < \delta$，所以小正方形中最多只有一个点，而总共有 8 个小正方形，最多有 8 个点，除去 p_i，则最多只有 7 个点。

至此，我们完成了父问题的分治与子问题的合并。

1.3 分治算法的时间复杂度分析

首先，第一次排序可以使用时间复杂度为 $O(n\log n)$ 的排序算法，如快速排序或者归并排序。

接着，我们考虑分治过程，即通过分治，我们将规模为 n 的父问题，分为两个规模为 $\frac{n}{2}$ 的子问题。

最后，归并过程中，根据采用的合并策略以及上述对更新操作的证明，我们需要 $O(n)$ 级别的时间完成。

综上，给出递推式如下：

$$T(n) = \begin{cases} O(1) & 2 \le n \le 3 \\ 2T(\frac{n}{2}) + O(n) & n > 3 \end{cases}$$

推导如下：

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n) \\ &= 2^2T(\frac{n}{2^2}) + 2O(\frac{n}{2}) + O(n) \\ &= 2^2T(\frac{n}{2^2}) + 2O(n) \\ &\vdots \\ &= 2^kT(\frac{n}{2^k}) + kO(n) \ \ (n = 2^k) \\ &= O(n) + O(n\log n) \\ &= O(n\log n) \end{aligned}$$

1.4 伪代码

Algorithm 1: Nearest-Pair

Data: Point List $P = \{p_i \mid 1 \le i \le n, p_i = (x_i, y_i)\}$
P should be sorted by x-coordinate in descending order.
Result: the minimum distance δ

```
begin
    if |P| <= 3 then
        Return the minimum Euclidean-Distance between each pair of points.
    m ← ⌊n/2⌋
    δ1 ← Nearest-Pair(P[1, ..., m])
    δ2 ← Nearest-Pair(P[m+1, ..., n])
    δ ← min{δ1, δ2}
    B ← MergeByY(P1, P2)
    foreach p_i ∈ B do
        foreach p_j ∈ C(p_i) do
            δ ← min{δ, Euclidean-Distance(p_i, p_j)}
    Return δ
```

第 2 章 分治法之大数乘法

内容提要

- ❑ 问题背景
- ❑ 直接分治法
- ❑ 改进分治法

2.1 问题描述

给定两个大数 A 和 B, 试计算 $A \times B$. 其中 A 和 B 分别表示为 $A = a_n a_{n-1} a_{n-2} \dots a_2 a_1$, $B = b_n b_{n-1} b_{n-2} \dots b_2 b_1$. 根据已学知识，给出如下引理。

引理 2.1

直接计算 $A+B$，其复杂度为 $O(n)$, 其中 n 为 A 和 B 的十进制位数。 ♡

直接计算 $A \times B$ 时，我们将 A 与 B 的各位相乘，在将各中间结果相加，得到最终结果。不难得出，这一过程需要进行 n 次基本乘法与 $n+1$ 次加法。根据引理2.1，有：

定理 2.1

直接计算 $A \times B$ 的时间复杂度为 $O(n^2)$. ♡

由定理2.1和引理2.1可知，如果我们直接相乘两个大数，其时间复杂度相比加法运算高出一个量级。由于乘法在计算机中大量存在，我们希望找到更好的算法来降低乘法计算的时间复杂度，以提升计算机的性能。分治法为我们提供了一条途径。

2.2 直接分治法

2.2.1 算法描述

这是一种简单的分治方法，将两个大数分为前后两部分，进行相乘。不失一般性，这里假设 n 为偶数。将 A 与 B 分割为 A_2, A_1, B_2, B_1, 即：

$$
\begin{aligned}
A_2 &= a_n a_{n-1} \dots a_{\frac{n}{2}+2} a_{\frac{n}{2}+1} \\
A_1 &= a_{\frac{n}{2}} a_{\frac{n}{2}-1} \dots a_2 a_1 \\
B_2 &= b_n b_{n-1} \dots b_{\frac{n}{2}+2} b_{\frac{n}{2}+1} \\
B_1 &= b_{\frac{n}{2}} b_{\frac{n}{2}-1} \dots b_2 b_1
\end{aligned}
$$

则 A 可以写为 $A = A_2 \times 2^{\frac{n}{2}} + A_1$. B 可以写为 $B = B_2 \times 2^{\frac{n}{2}} + B_1$. 计算 $A \times B$ 的问题在进行上述转换后表示为:

$$
\begin{aligned}
A \times B &= (A_2 \times 2^{\frac{n}{2}} + A_1) \times (B_2 \times 2^{\frac{n}{2}} + B_1) \\
&= A_2 B_2 \times 2^n + (A_2 B_1 + A_1 B_2) \times 2^{\frac{n}{2}} + A_1 B_1
\end{aligned}
$$

此时将两个大数相乘的问题转化为 4 个乘法子问题和 3 个加法子问题。显然，分治策略还可以对子问题使用，继续减小问题的规模。

2.2.2 伪代码

Algorithm 2: DirectDAC

```
Input: Two large numbers A, B, which both have n decimal digits
Result: A × B
begin
    n ← Number of Decimal Digits of A and B
    if n ≠ 1 then
        Divide A, B into A_2, A_1, B_2 and B_1
        C_3 ← DirectDAC(A_2, B_2)
        C_2 ← DirectDAC(A_2, B_1)
        C_1 ← DirectDAC(A_1, B_2)
        C_0 ← DirectDAC(A_1, B_1)
        return C_3 ≪ n + (C_2 + C_1) ≪ (n − 1) + C_0
    else
        return A × B
```

2.2.3 复杂度分析

由上述的算法描述可知，算法的主要开销来自于每次分支带来的 4 个乘法子问题和 3 个加法子问题，由于移位可在机器中由一个简单的指令完成，我们忽略这个操作的时间。
假设 $T(n)$ 表示两个 n 位大数相乘所需的时间开销，则在直接分治法中：

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + 3n \\ &= 4T(\frac{n}{2}) + O(n) \end{aligned}$$

根据主方法，$\log_2 4 = 2 > 1$, 推出如下定理：

定理 2.2
用直接分治法计算 $A \times B$ 的时间复杂度为 $O(n^2)$.

根据定理2.2, 直接分治法的性能是令人失望的，因为其并不能提供时间上优于直接相乘的性能。但分治策略提示我们，这个算法的性能与乘法子问题的数目强相关。我们如果能够用一些其他的开销换取更少的乘法子问题数目，也许能得到更好的算法。

2.3 改进分治法

2.3.1 改进思路

在直接分治法中，通过对大数进行分割，我们有：

$$A \times B = A_2B_2 \times 2^n + (A_2B_1 + A_1B_2) \times 2^{\frac{n}{2}} + A_1B_1$$

这个过程中，引入了 4 次乘法运算；在上一节中提到，分治策略和主定理提示我们尽可能减少乘法的次数。但换取更低的乘法子问题数，需要其他的开销。一种想法是，由于加法的复杂度为 $O(n)$，我们也许可以用略多的加法子问题，来减少乘法子问题数。基于此想法，我们对直接分治法作出一些改进。首先将直接分治法中的计算式修改为：

$$\begin{aligned}A \times B &= A_2B_2 \times 2^n + (A_2B_1 + A_1B_2) \times 2^{\frac{n}{2}} + A_1B_1 \\ &= A_2B_2 \times 2^n + ((A_2 + A_1) \times (B_2 + B_1) - A_2B_2 - (A_1B_1)) \times 2^{\frac{n}{2}} + A_1B_1\end{aligned}$$

观察上式，我们只需要做 3 次乘法，即计算 $A_2B_2, A_1B_1, (A_2 + A_1) \times (B_2 + B_1)$, 以及 4 次加法，2 次减法。考虑到加法和减法本质上等同，我们成功地将这一问题转化为了 3 个乘法子问题和 6 个加法子问题。相比于直接分治法，我们降低了乘法的数量。

下面给出该算法的伪代码及复杂度分析。

2.3.2 伪代码

Algorithm 3: ModifiedDAC

```
Input: Two large numbers A, B, which both have n decimal digits
Result: A × B
begin
    n ← Number of Decimal Digits of A and B
    if n ≠ 1 then
        Divide A, B into A2, A1, B2 and B1
        C2 ← DirectDAC(A2, B2)
        C1 ← DirectDAC(A1, B1)
        C0 ← DirectDAC(A2 + A1, B2 + B1)
        return C2 ≪ n + (C0 − C2 − C1) ≪ (n − 1) + C1
    else
        return A × B
```

2.3.3 复杂度分析

同上节的复杂度分析，我们此处也忽略移位操作带来的开销。改进分治法中，我们将问题分解为 3 个乘法子问题与 6 个加法子问题。因此有：

$$\begin{aligned}T(n) &= 3T(\frac{n}{2}) + 6n \\ &= 3T(\frac{n}{2}) + O(n)\end{aligned}$$

根据主方法，$\log_2 3 > 1$. 推出如下定理：

定理 2.3

用改进分治法计算 $A \times B$ 的时间复杂度为 $O(n^{\log_2 3}) \approx O(n^{1.585})$.

参考文献