**21120491: 高级数据结构与算法分析** **2023-2024 学年春夏学期**

# Lecture 7: 分治法

教师：刘金飞，助教：吴一航 日期: 2024 年 4 月 9 日

## 7.1 主定理

分治法我想诸位已经非常熟悉，因此我们也不再谈论一些基础的话题，直接开始讨论如何分析分治法的时间复杂度。通常而言，分治法递归的时间复杂度递推公式具有如下形式：

$$T(n) = aT(n/b) + f(n),$$

其中 $a$ 是分了多少个子问题，$b$ 与子问题划分方式有关，是子问题输入长度的收缩因子，$f(n)$ 则是合并各个子问题的解需要的时间。当然我们的练习中会看到不这么寻常的结构，这些我们留在讨论题中供诸位思考。

### 7.1.1 代入法

美其名曰代入法，其实就是猜证，显得有些耍无赖，但面对考试中都是选择判断的情况，这种方法显得尤为便捷。代入法一般分为两步：

1. 猜测解的形式；
2. 用数学归纳法求出解中的常数，并证明解是正确的。

这一方法在 PPT 的 8-9 页给出了例子，第 8 页关于归纳基础的讨论，希望表达的意思是如果常数取的足够大，那么在 $N$ 小的情况下不管什么复杂度都是正确的，因为都是单纯的数值计算，所以我们无需关心归纳基础。第 9 页的例子特别需要强调这里的常数 $c$ 是要保持的，否则常数也会与 $N$ 有关（$N$ 增加，常数增加），那就不再是常数了。至于如何猜测一个好的界，可以回忆一些熟悉的结论，例如归并排序这类，也可以猜测比较松的上下界，然后往中间逼近最紧的。

回忆熟悉的结论这一点引出了一个技巧，即换元法，我们来看下面这个问题：

【讨论 1:】$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$。

为方便起见，我们不必担心值的舍入误差问题，只考虑 $\sqrt{n}$ 是整数的情形即可。令 $m = \log n$，得到

$$T(2^m) = 2T(2^{m/2}) + m,$$

然后我们令 $S(m) = T(2^m)$，则有 $S(m) = 2S(m/2) + m$，这就是我们熟悉的归并排序结果，因为 $S(m) = O(m \log m)$，因此 $T(n) = \log n \log\log n$。

下面是一个从算法导论截取的技巧，对于一些情况会有所帮助：

> **微妙的细节**
>
> 有时你可能正确猜出了递归式解的渐近界，但莫名其妙地在归纳证明时失败了。问题常常出在归纳假设不够强，无法证出准确的界。当遇到这种障碍时，如果修改猜测，将它减去一个低阶的项，数学证明常常能顺利进行。
>
> 考虑如下递归式：
>
> $$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$
>
> 我们猜测解为 $T(n)=O(n)$，并尝试证明对某个恰当选出的常数 $c$，$T(n)\leqslant cn$ 成立。将我们的猜测代入递归式，得到
>
> $$T(n) \leqslant c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1$$
>
> 这并不意味着对任意 $c$ 都有 $T(n)\leqslant cn$。我们可能忍不住尝试猜测一个更大的界，比如 $T(n)=O(n^2)$。虽然从这个猜测也能推出结果，但原来的猜测 $T(n)=O(n)$ 是正确的。然而为了证明它是正确的，我们必须做出更强的归纳假设。
>
> 直觉上，我们的猜测是接近正确的：只差一个常数 1，一个低阶项。但是，除非我们证明与归纳假设严格一致的形式，否则数学归纳法还是会失败。克服这个困难的方法是从先前的猜测中减去一个低阶项。新的猜测为 $T(n)\leqslant cn-d$，$d$ 是大于等于 0 的一个常数。我们现在有
>
> $$\begin{aligned} T(n) &\leqslant (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \leqslant cn - d \end{aligned}$$
>
> 只要 $d\geqslant 1$，此式就成立。与以前一样，我们必须选择足够大的 $c$ 来处理边界条件。
>
> 你可能发现减去一个低阶项的想法与直觉是相悖的。毕竟，如果证明上界失败了，就应该将猜测增加而不是减少，更松的界难道不是更容易证明吗？不一定！当利用归纳法证明一个上界时，实际上证明一个更弱的上界可能会更困难一些，因为为了证明一个更弱的上界，我们在归纳证明中也必须使用同样更弱的界。在当前的例子中，当递归式包含超过一个递归项时，将猜测的界减去一个低阶项意味着每次对每个递归项都减去一个低阶项。在上例中，我们减去常数 $d$ 两次，一次是对 $T(\lfloor n/2 \rfloor)$项，另一次是对 $T(\lceil n/2 \rceil)$项。我们以不等式 $T(n)\leqslant cn-2d+1$ 结束，可以很容易地找到一个 $d$ 值，使得 $cn-2d+1$ 小于等于 $cn-d$。

【讨论 2:】$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$。

$n$ 很大的时候，17 是完全可以忽略的，因此我们直接猜测 $O(n \log n)$ 然后验证即可，注意这里可能需要用到上面的技巧。

### 7.1.2 递归树方法

最直白且最自然的方法，相信直接看出来树怎么画（注意树每一层的复杂度就是合并复杂度，每个叶子的复杂度就是 base case 的复杂度，显然为常数），直到基本的数列求和公式，这个方法对于能用得上的场景都是很简单的，这里也就不再赘述了（PPT 第 10-11 页），更具体的描述可以阅读算法导论。

【注:】PPT 第 11 页的例子可以用更更高级的定理解决，详情请看Akra–Bazzi 定理的维基百科。

需要注意的是，第 10 页的例子与接下来的主定理有很大的关联，我们看到在第 10 页的例子中，我们最后求和分为两个部分，第一个部分是每层的时间，其实就是每次递归后合并解需要的时间和（注意 PPT 的图上每层右边的大小是从下一层合并到该层需要的时间，而非该层合并到上一层需要的时间），第二个部分则是递归到 base case 的子问题个数，每个需要常数的时间操作然后 return 然后开始合并。这两个部分谁更大，谁就 master 了整个时间复杂度（这一例子中是每层合并时间 master 了），这也是接下来主定理（master theorem）的思想。除此之外，通过画递归树得到的结果需要用替代法验证一下正确性。

【讨论 3:】给定一个整数 $M$，

$$T(n) = \begin{cases} 8T(n/2) + 1 & n^2 > M \\ M & otherwise \end{cases}.$$

有 $\log_2 n/\sqrt{M}$ 层，每层 1 个单位时间，且有 $8^{\log_2 n/\sqrt{M}}$ 个叶子，每个叶子 $M$ 个单位时间，故总时间为

$$O(M \cdot 8^{\log_2 n/\sqrt{M}} + \log_2 n/\sqrt{M}) = O(n^3/\sqrt{M}).$$

事实上考试中也出现过类似的问题:

Consider the following pseudo-code.

strange($a_1, \ldots, a_n$):
1. if $n \leq 2022$ then return
2. strange($a_1, \ldots, a_{\lfloor n/2 \rfloor}$)
3. for $i = 1$ to $n$:
4. for $j = 1$ to $\lfloor \sqrt{n} \rfloor$:
5. print($a_i + a_j$)
6. strange($a_{\lfloor n/2+1 \rfloor}, \ldots, a_n$)

What is the running time of this pseudo-code? Your answer should be as tight as possible. (You may assume that $n$ is a power of 2.)

- A. $O(n^2)$
- B. $O(n^{1.5}) \log n$
- C. None of the other options is correct
- D. $O(n^{1.5})$

我们可以将这一问题转化为上面的分段表达式，然后求解。

### 7.1.3 主定理

主定理的证明属实有点老太太裹脚布，感兴趣的同学可以阅读算法导论，这里没必要重复这些纯技术性的过程。

考虑如下形式：

$$T(n) = aT(n/b) + f(n),\ a \geqslant 1, b \geqslant 2$$

**定理 7.1** *1. 若对于某个 $\varepsilon$ 有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$；*

*2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \log n)$；*

*3. 若对于某个 $\varepsilon$ 有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，且对某个常数 $c < 1$ 和所有足够大的 $n$ 有 $af(n/b) \leqslant cf(n)$，则 $T(n) = \Theta(f(n))$.*

注意，这三种情况并未覆盖 $f(n)$ 的所有可能性。情况 1 和情况 2 之间有一定间隙，情况 2 和情况 3 之间也有一定间隙。如果函数 $f(n)$ 落在这两个间隙中，或者情况 3 中要求的正则条件不成立（关于正则条件可以看后面的主定理第二种形式的推导理解，也可以直接看证明理解），就不能使用主方法来求解递归式。我们来看一个不符合正则条件的例子：

**例 7.2** $T(n) = T(n/2) + n(2 - \cos n)$，此时 $a = 1, b = 2$，且我们知道 cos 函数的取值范围是 $-1$ 到 1，因此满足上述第三种情况的第一个条件，然而不满足第三种情况的正则条件。因为如果满足正则条件，我们代入有

$$\frac{n}{2}(2 - \cos\frac{n}{2}) \leqslant c(2 - \cos n)$$

对某个常数 $c < 1$ 和充分大的 $n$ 都成立。然而，当 $n$ 很大时，不等号左边可以任意大，右边只能小于 3（注意 $c$ 和余弦函数的取值范围），故不可能，因此这时不能用主定理。

实际上主定理并不需要死记硬背，无论什么形式的主定理（后面会介绍其它形式的），其实关键都在于比较 $n^{\log_b a}$ 和 $f(n)$ 之间的关联，如果前者大，则前者“掌控了”整个时间复杂度，所以时间复杂度就是 $T(n) = aT(n/b)$ 对应的复杂度，这也很符合直观，因为此时 $a$ 比较大，分叉比较多，树比较大（结合前面递归树的例子），所以更大的复杂度会落在叶子上；反之后者大则每一层的复杂度“掌控了”整个时间复杂度，故整体时间就是 $f(n)$ 级别的。这也就是“主定理”这一名字的含义，十分形象，就是看前后两半谁 master 了整体时间复杂度，具体为什么是比较这两个函数则是具体证明中可以很容易看出的。

【讨论 4:】

1. $T(n) = 4T(n/2) + n/\log n$；
2. $T(n) = 2T(n/2) + n/\log n$.

第一个直接使用主定理即可，结果为 $\Theta(n^2)$；第二个不能使用主定理，因此用递归树，递归树我们有 $\log_2 n$ 层，第 $i$ 层时间复杂度 $n/\log(n/2^i)$，叶子有 $n$ 个，故整体复杂度为

$$\sum_{i=0}^{\log_2 n - 1} \frac{n}{\log(n/2^i)} + \Theta(n) = \sum_{i=0}^{\log n - 1} \frac{n}{\log_2 n - i} + \Theta(n) = \sum_{j=1}^{\log_2 n} \frac{n}{j} + \Theta(n) = O(n \log\log n).$$

最后代入检验也没问题。注意最后一步用了调和级数的估计结论，如果大家还能回忆起微积分中的欧拉常数，那么这个估计也是很熟悉的。

在这一问题中我们给出了一个主定理不适用的场景，实际上还有其它场景，例如递推式中 $a$ 不是常数，$a$ 和 $b$ 取值不在规定范围内等，做题时需要多加小心（但事实上本题可以通过本节最后给出的主定理推广形式解决）。

下面我们介绍主定理相对比较简单的两种形式，第一种实际上是前面原始版本的推论：

**定理 7.3** *1. 若对于某个常数 $c > 1$ 有 $af(n/b) = cf(n)$，则 $T(n) = \Theta(n^{\log_b a})$；*

*2. 若 $af(n/b) = f(n)$，则 $T(n) = \Theta(n^{\log_b a} \log n)$；*

*3. 若对于某个常数 $c < 1$ 有 $af(n/b) = cf(n)$，则 $T(n) = \Theta(f(n))$.*

不难看出是推论，例如对于第一种情况，$af(n/b) = cf(n)$ 表明 $f(n) = a/cf(n/b)$，故大致可以递推得到

$$f(n) = \frac{a^{\log_b n}}{c^{\log_b n}} = \frac{n^{\log_b a}}{n^{\log_b c}} = O(n^{\log_b a - \varepsilon}),$$

正好适用于原始方法的第一种情况。第二、三种情况事实上也是同样的推导。从这里我们可以看出，对某个常数 $c < 1$, 正则条件 $af(n/b) \leqslant cf(n)$ 成立本身就意味着存在常数 $\epsilon > 0$ 使得 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，因此前面的正则条件的意义在这里也可以看出。

第二种则在某些情况下比之前介绍的原始形式更强，我们首先介绍这一结论，简要说明其证明，然后给出一个经典的例子：

**定理 7.4** *对于递推式 $T(n) = aT(n/b) + \Theta(n^k \log^p n), a \geqslant 1, b > 1, k \geqslant 0, p \geqslant 0$,*

*1. 若 $a > b^k$，则 $T(n) = \Theta(n^{\log_b a})$；*

*2. 若 $a = b^k$，则 $T(n) = \Theta(n^k \log^{p+1} n)$；*

*3. 若 $a < b^k$，则 $T(n) = \Theta(n^k \log^p n)$.*

这里的第一种情况实际上就可以直接对应到主定理原始形式的第一种情况，第三种情况也对应原始形式第三种情况（包括正则条件），重点是第二种情况，相比于原先的形式更加强大，我们可以看下面这个例子：

**例 7.5** *考虑 $T(n) = 2T(n/2) + n \log n$，则原始版本和这里的第一种都是不可行的，但最后一种形式可以使用。*

至于第二种情况的证明，相对而言较为复杂。我们使用递归树进行分析，并且为了简化讨论，假设 $n$ 是 $b$ 的幂次。则我们很容易利用递归树得到（不妨设 $T(n) = aT(n/b) + cn^k \log^p n$）：

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n - 1} a^i c \left(\frac{n}{b^i}\right)^k \log^p \frac{n}{b^i} + a^{\log_b n} T(1) \\ &= cn^k \sum_{i=0}^{\log_b n - 1} \log^p \frac{n}{b^i} + n^k (\text{注意} a = b^k), \end{aligned}$$

所以现在的关键问题就是求出对数和 $\sum_{i=0}^{\log_b n - 1} \log^p \frac{n}{b^i}$，我们有一个简单的观察

$$\log^p \frac{n}{d} = (\log n - \log d)^p = \log^k n + O(\log^k n),$$

所以我们有

$$\begin{aligned} \sum_{i=0}^{\log_b n - 1} \log^p \frac{n}{b^i} &= \sum_{i=0}^{\log_b n - 1} (\log^p n + O(\log^p n)) \\ &= \log_b n \log^p n + \log_b n \cdot O(\log^p n) \\ &= \Theta(\log_b n \log^p n) = \Theta(\log^{p+1} n), \end{aligned}$$

代入前面的推导，可以得到 $T(n) = \Theta(n^k \log^{p+1} n)$ 成立。

【注：】上一主定理的形式还有更强的表达：

**定理 7.6** *对于递推式 $T(n) = aT(n/b) + \Theta(n^k \log^p n), a \geqslant 1, b > 1, k \geqslant 0$，而 $p$ 为任意实数，*

1. *若 $a > b^k$，则 $T(n) = \Theta(n^{\log_b a})$.*
2. *若 $a = b^k$，则*
   (a) *若 $p > -1$，$T(n) = \Theta(n^k \log^{p+1} n)$；*
   (b) *若 $p = -1$，$T(n) = \Theta(n^k \log\log n)$；*
   (c) *若 $p < -1$，$T(n) = \Theta(n^k)$.*
3. *若 $a < b^k$，则*
   (a) *若 $p \geqslant 0$，$T(n) = \Theta(n^k \log^p n)$；*
   (b) *若 $p < 0$，$T(n) = \Theta(n^k)$.*

在这一形式下，讨论 4 的第二问实际上也在推广的主定理的讨论范围内了。更多的关于这一推广的描述与可以或不可以使用主定理的例子见这个链接，部分证明感兴趣的同学可以参考这个文档。

## 7.2 分治法应用

### 7.2.1 曾经学过的问题

在学习数据结构基础甚至普通 C 程时我们便已经多次接触分而治之的思想，这一节我们就先来回顾一下我们曾经学过的那些问题，并用主定理给出一些基本的分析。

#### 7.2.1.1 最大子序列和问题

相信大家对这个例子并不陌生——这是数据结构基础第一节课的内容！按照规律第一节课的内容应该是学习得比较认真的..... 我们用这个例子来回顾一下基本思想。

假定我们要寻找子数组 $A[low, high]$ 的最大子序列和，我们首先找到数组中央位置 $mid$，分治法很自然地将这个问题的解 $A[i, j]$ 变为以下三种情况之一：

1. 最大子序列和完全在 $A[low, mid]$ 中，即 $low \leqslant i \leqslant j \leqslant mid$；
2. 最大子序列和完全在 $A[mid + 1, high]$ 中，即 $mid + 1 \leqslant i \leqslant j \leqslant high$；
3. 最大子序列和跨越 $mid$ 两边，即 $low \leqslant i \leqslant mid < j \leqslant high$。

所以我们的想法就是：

1. 分治：将原数组平分为两半 $A[low, mid]$ 和 $A[mid + 1, high]$，然后分别对这两半求解最大子序列和；一定不能忘记递归有 base case，这里的 base case 就是数组只剩下一个元素，那就什么都不用操作，直接返回进入下一步合并阶段；
2. 合并：首先我们要计算跨越 $mid$ 两边的最大子序列和，然后和左右两半的结果比较，选择最大的作为最终结果。关键在于计算跨越 $mid$ 两边的最大子序列和，这其实是线性的，为什么？因为这里的子序列都必须跨越 $mid$，所以最大子序列和就是最大的 $A[i, mid]$ 加上最大的 $A[mid + 1, j]$，这个只需要对 $i, j$ 做遍历就行，所以是线性的。

那么我们得到的递推公式就是很简单的 $T(n) = 2T(n/2) + O(n)$，所以就是 $O(n \log n)$ 的复杂度。

#### 7.2.1.2 归并排序与快速排序

这个例子出现的时间应该比最大子序列和更早——学习 C 语言的时候应该就有所了解了！归并排序的思想相当简单，分治直接将数组分成两半然后分别排序即可，base case 就是分到数组长度为 1 了，此时什么都不用做直接返回进入合并阶段，而合并就是将两个排序好的小数组合并成一个大数组，这一步显然是线性的复杂度，这也就很轻松地得到 $O(n \log n)$ 的复杂度。

至于快速排序，我们是选择一个 pivot，然后首先将小于 pivot 的都放左边，大于的都放右边，这一步只需线性时间，然后对 pivot 左右的数组递归继续排序。所以这里的流程和之前的问题不太一样，先做了额外操作后才进行分治。在数据结构基础的讨论中我们知道快速排序最差可能到平方级别的运行时间，在 ADS 中我们将会在随机算法一节中引入随机性证明平均时间复杂度为 $O(n\log n)$。

#### 7.2.1.3 逆序对计数

逆序对在定义行列式时已经见到（当然有的同学不一定学习了这种定义方式，但只需简单搜索就可以知道逆序对的含义），显然分治法计算逆序对就是分成三类：一是全在左半边的逆序对个数，二是全在右半边的，三是跨越中点的。看起来很正常，但这个跨越中点的逆序对个数似乎最多有 $n/2\times n/2=n^2/4$ 个，这样分治法就失去了时间上的优势。我们的想法是站在归并排序的肩膀上，如果我们每次计算逆序对后也进行了排序，那么合并过程中计算跨越中点的逆序对个数就只需要线性时间了：在合并两个已排序的子序列的过程中我们就很容易完成逆序对的计算，相信读者只需稍加思考就可以理解这一点，如果实在不理解相信网上有大量资料能说明白这一点。

### 7.2.2 最近点对问题

这是我们 PPT 上的例子，也是相当经典的应用。同样分为三个部分，左最近点对、右最近点对和分离最近点对，关键在于线性时间找到分离最近点对。这是一个相当聪明的算法，我们首先取 $x$ 坐标来看的中点，其 $x$ 坐标记为 $\overline{x}$，然后考虑 $[\overline{x}-\delta,\overline{x}+\delta]$（$\delta$ 是左右两半中最近点对距离）之间的所有点 $q_1,q_2,\cdots,q_l$，它们按 $y$ 坐标排序。设 $q_i$ 的 $y$ 坐标为 $y_i$，那么我们只需要对 $q_i$ 检查 $x$ 坐标在 $[\overline{x}-\delta,\overline{x}+\delta]$ 之间，$y$ 坐标在 $[y_i,y_i+\delta]$ 之间构成的长方形区域中的所有点是否有更近的点对即可（所有点都往上找就行，往下和下面的点往上找重合）。我们将这个长方形区域分为平均分割为 8 块，则每块内最多出现一个点，否则每块内两点距离不超过 $\frac{\sqrt{2}}{2}\delta$，并且每块都不跨中点，这与 $\delta$ 是左右两半中最近点对距离矛盾，因为我们可以找到距离更短的两点。所以对于每个 $q_i$，我们只需找向上 7 个点即可，所以找分离最近点对的时间复杂度是线性的。

但是，7 个点看起来可能有点太多了。实际上当前研究的点所在的一侧的所有点都完全可以不考虑，例如当前我们循环到了一个在左半边的点，那么整个左半边的点都不需要考虑，只需要考虑右半边 4 个格子最多的 4 个点。但要注意的是这种情况你需要提前维护好左右两边各自的按 $y$ 坐标排序的点列，然后要维护 $y$ 坐标紧跟着左半边的每个点的四个右半边的点，对称的右半边也要维护，这并不复杂。

然而实际上我们还可以进一步将 4 这个数字降为 3。因为我们每在右半边放一个点，那么在右半边，以这个点为圆心半径为 $\delta$ 的圆内不可能再有另一个点，事实上最差的情况就是有四个这样的圆心，此时四个圆心在右半边区域的四个角上（也就是 $(0,0),(0,\delta),(\delta,0),(\delta,\delta)$），那这时其实找三个也就够了（其实就是 $(0,0)$ 最好），而其它情况不可能有四个圆心（因为四个圆心互相的最近距离都只能是 $\delta$，最极端的情况也就是一个正方形），最多三个圆心那么找三个也就足够了。

当然，其实三个点还是有点多，两个点也是可以说明的，诸位可以自己思考这一点，实际上如果理解了 3 个点的算法这一点也并不难。除此之外，这一问题还有一个经典的线性随机算法，我们在随机算法一节中可能会简单介绍。

### 7.2.3 从整数乘法到矩阵乘法

提到分治法，不得不提的就是 Strassen 矩阵乘法。这是一个看起来有点魔法，能将矩阵乘法从最简单的三次方算法复杂度下降的方法。为了使这一方法看起来不那么魔法，我们可以先介绍一个 Karatsuba 整数乘法算法作为基础。

我们考虑两个长度为 $n$ 的整数做乘法（长度不同则短的高位补 0 补齐长度即可），如果不做优化，用传统的竖式乘法显然复杂度为 $O(n^2)$。如果用分治法，想法很简单，就是将两个整数分为两半分别计算乘法，然后组合起来即可。假设 $n$ 是偶数（奇数就是不能平分，记号上麻烦一点，但本质是一样的），则可以设第一个乘数为 $a = 10^{n/2}a_1 + a_2$，第二个乘数为 $b = 10^{n/2}b_1 + b_2$，那么二者相乘的结果就是

$$ab = 10^n a_1 b_1 + 10^{n/2}(a_1 b_2 + a_2 b_1) + a_2 b_2.$$

因此我们在分治阶段需要计算 $a_1b_1, a_1b_2, a_2b_1, a_2b_2$ 四个长度为 $n/2$ 的乘法，合并阶段只需要简单移位和加法就行，复杂度是线性的，所以整体时间复杂度递推公式为

$$T(n) = 4T(n/2) + O(n).$$

尴尬的事情出现了，根据主定理我们知道这个时间复杂度是 $O(n^2)$，似乎我们的优化失败了。其实关键就在于这里我们分治要计算四个乘法，如果能计算更少的乘法呢？我们注意到

$$a_1b_2 + a_2b_1 = (a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2,$$

而 $a_1b_1$ 和 $a_2b_2$ 是递归本来就要计算的，所以事实上我们分治时只要计算 $(a_1 + a_2)(b_1 + b_2)$、$a_1b_1$ 和 $a_2b_2$ 三个乘法即可算出最后的乘法，然后合并时计算多了一步计算 $(a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2$，其实就是减法，所以复杂度还是线性的，故我们的递推式变为了

$$T(n) = 3T(n/2) + O(n),$$

很高兴，我们的复杂度变为了 $O(n^{\log_2 3})$，这相比平方有了一定改进。

有了上面的基础，我们来看矩阵乘法。分治的想法显然是使用分块矩阵，我们考虑简单的情况，即矩阵是方阵的情况，设矩阵阶数 $n$ 为偶数（同理奇数只是记号复杂，偶数不失一般性），我们将两个矩阵进行如下分块：

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix},\ B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix},$$

设结果矩阵

$$C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix},$$

则有

$$C_1 = A_1B_1 + A_2B_3$$
$$C_2 = A_1B_2 + A_2B_4$$
$$C_3 = A_3B_1 + A_4B_3$$
$$C_4 = A_3B_2 + A_4B_4$$

于是我们分治需要计算 8 个阶数为 $n/2$ 的矩阵乘法，然后用平方时间做矩阵加法即可，即时间复杂度递推公式为

$$T(n) = 8T(n/2) + O(n^2).$$

很遗憾，时间复杂度仍然是 $O(n^3)$。所以类似于 Karatsuba 整数乘法，我们希望用更少的乘法来表达出 8 个乘法。天才的 Strassen 在 1969 年给出了这一算法，他构造了如下七个矩阵

$$M_1 = (A_1 + A_4)(B_1 + B_4)$$
$$M_2 = (A_3 + A_4)B_1$$
$$M_3 = A_1(B_2 - B_4)$$
$$M_4 = A_4(B_3 - B_1)$$
$$M_5 = (A_1 + A_2)B_4$$
$$M_6 = (A_3 - A_1)(B_1 + B_2)$$
$$M_7 = (A_2 - A_4)(B_3 + B_4)$$

通过组合上述 7 个新矩阵得到：

$$C_1 = M_1 + M_4 - M_5 + M_7$$
$$C_2 = M_3 + M_5$$
$$C_3 = M_2 + M_4$$
$$C_4 = M_1 - M_2 + M_3 + M_6$$

于是我们只需要线性时间计算加法和减法，然后分治只需要计算 7 个乘法即可，因此时间复杂度递推公式为

$$T(n) = 7T(n/2) + O(n^2),$$

因此复杂度变为了 $O(n^{\log_2 7})$，这相比三次方有了一定改进。

一个有趣但无解的问题是，Strassen 是怎么想出这 7 个矩阵的——他的论文也没给出解释，我们且当这是智慧与尝试的结晶吧。有些地方给了基于代数几何和表示论的解释，表明这样的组合一定存在，从更高的视角下给这个答案一个合理的解释。