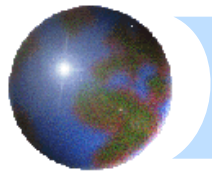


# 多项式乘法

张家琳

复旦大学附属中学



# 引言

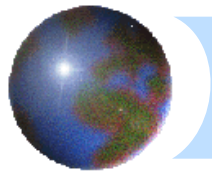
多项式是最基本的数学工具之一，由于其形式简单，且易于用计算机对其进行各种计算，在当今的社会中应用越来越广。不仅在像Maple这样的数学软件中有着举足轻重的作用，在工程、信息等诸多领域中都有着广阔的应用。

下面我们给出几个多项式逼近的例子：

$$\ln(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots + (-1)^{n-1} \frac{x^n}{n} + \cdots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots$$



# 多项式的基本运算

✚ 加法运算

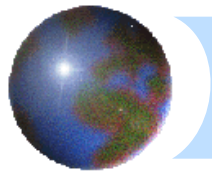
✚ 求值运算

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

$$= a_0 + x^*(a_1 + x^*(a_2 + \cdots + x^*(a_{n-1} + x^*a_n)\cdots))$$

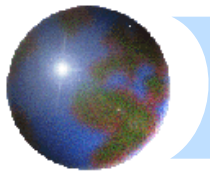
✚ 乘法运算





# 普通的多项式乘法运算

- ❖ 多项式乘法是一个很常见的问题，在通常的算法中，两个  $n$  次多项式的乘法需要用  $O(n^2)$  的时间才能完成。
- ❖ 让我们先来看一看这样的算法是如何进行的：



$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

$$* \quad b_n x^n + b_{n-1} x^{n-1} + \dots + b_1 x + b_0$$

---


$$a_n b_0 x^n + a_{n-1} b_0 x^{n-1} + \dots + a_1 b_0 x + a_0 b_0$$

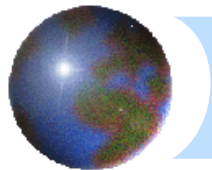
$$a_n b_1 x^{n+1} + a_{n-1} b_1 x^n + \dots + a_1 b_1 x^2 + a_1 b_0 x$$

.....

$$a_n b_n x^{2n} + \dots + a_1 b_n x^{n+1} + a_0 b_n x^n$$

---

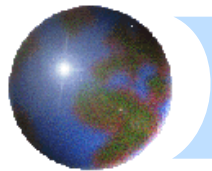

$$a_n b_n x^{2n} + \dots + \sum_{k=1}^n a_{n+1-k} b_k x^{n+1} + \sum_{k=0}^n a_{n-k} b_k x^n + \dots + (a_1 b_0 + a_0 b_1) x + a_0 b_0$$



✚ 在上面的过程中，似乎我们觉得这个算法并没有做任何多余的事情，因为我们必须考虑每一组  $b_i$ ，它们的乘积对最终的结果都会产生影响。而且我们不能通过调整计算顺序从根本上降低算法时间复杂度，至多只能使其常数因子稍小一些。

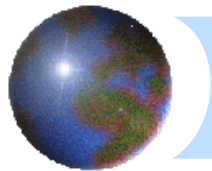
✚ 如果我们不能跳出以上思维的局限，我们就不可能在根本上降低算法的时间复杂度。

✚ 让我们首先换一个角度来考察多项式。



# 多项式的点值表示法

- 首先让我们来看多项式的另一种表示方法：点值表示法，即用  $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1})$  (其中， $f(x_i) = y_i, x_i$  互不相同) 来表示一个不超过  $n-1$  次多项式。
- 给定一个多项式，要给出  $n$  组点值对最简单的方法是任选  $n$  个互不相同的  $x_i$ ，依次求出多项式在这  $n$  个点的值。
- 用  $n$  个点值对也可以唯一确定一个不超过  $n-1$  次多项式，这个过程称之为**插值**。

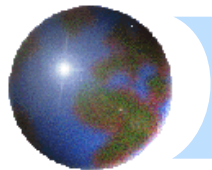


## 引理1（多项式插值的唯一性）

✚ 对于任意 $n$ 个点值对组成的集合,  
 $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ 其中  $x_0, x_1, \dots, x_{n-1}$  互不相同, 则存在唯一的次数不超过 $n-1$ 的多项式  $A(x)$ , 满足  $y_k = A(x_k) \quad k = 0, 1, \dots, n-1$

[continue](#)






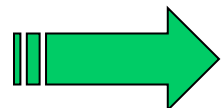
## 存在性:

已知  $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$

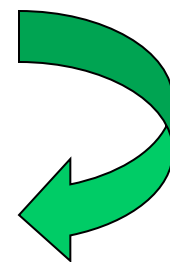
$$A(y_k) = x_k, (k = 0, 1, \dots, n-1)$$

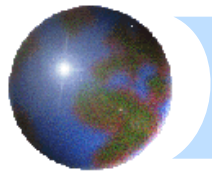

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

不妨记为  $XA=Y$


$$A = X^{-1}Y \quad X^{-1}?$$

X是范德蒙矩阵，利用行列式的变换可得  
该矩阵的行列式的值为  $\prod_{j < k} (x_k - x_j) \neq 0$   
因此X有逆矩阵  $X^{-1}$ 。

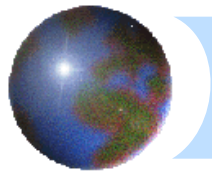




## 唯一性:

- 若两个函数次数不超过 $n-1$ 次的多项式  $f(x), g(x)$  均符合题意，则多项式  $r(x) = f(x) - g(x)$  有  $n$  个根，且  $r(x)$  为不超过  $n-1$  次的多项式，所以  $r(x) \equiv 0$ ，即  $f(x) = g(x)$ 。

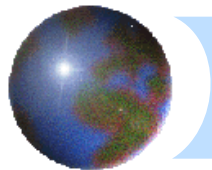
[back](#)



## 点值多项式的乘法

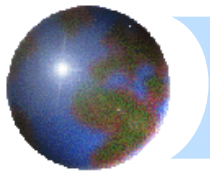
$$\begin{array}{ccc} r(x) = f(x) * g(x) & \xrightarrow{\text{green arrow}} & r(x_i) = f(x_i) * g(x_i) \\ \downarrow \quad \downarrow \quad \downarrow & & \downarrow \text{green arrow} \\ (x_i, r_i) \quad (x_i, f_i) \quad (x_i, g_i) & & r_i = f_i * g_i \end{array}$$

因此： 适当的利用点值表示可以使  
多项式的乘法可以在线性时间内完成！



✚ 因为 $f(x)g(x)$ 的次数为 $n-1$ ,  $r(x)$ 的次数为 $2n-2$ , 因此确定 $r(x)$ 需要 $2n-1$ 个点值对, 而现在我们只有 $n$ 个点值对。

✚ 我们可以通过对 $f(x)$ 与 $g(x)$ 的点值对个数的 扩充来解决这个问题, 即将 $f(x), g(x)$ 的点值对在一开始就取为 $2n-1$ 。



# 利用点值表示法改善多项式系数表示法的乘法

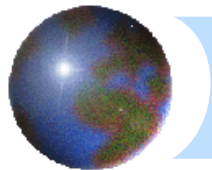
下面让我们来看一看我们是否能够利用点值表示法在计算多项式乘法时的线性时间来提高系数表示法的多项式乘法的速度。

为了做到这一点，我们需要做：

- ❖ 将多项式由系数表示法转化为点值表示法（点值过程）
- ❖ 利用点值表示法完成多项式乘法
- ❖ 将点值表示法再转化为系数表示法（插值过程）

其中第二步只需要线性时间。问题的关键转化为第一步第三步。

[continue1](#)   [continue2](#)   [continue3](#)



# 1. 由系数表示法转化为点值表示法。 (点值过程)

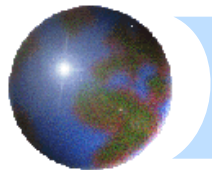
$$x_i \rightarrow f(x_i) \quad (i = 0, 1, \dots, n-1)$$

$$f(x_i) = (((\dots(a_{n-1}x_i + a_{n-2}) * x_i + a_{n-3}) \dots + a_1) * x_i + a_0$$

✚ 注意！ $x_0, x_1, \dots, x_{n-1}$ 是由我们自己选择的，我们可以充分利用这一点通过适当的选择使转化过程降为 $O(n \log n)$ 。

✚ 这里我们选择1的 $n$ 次单位根作为 $x_0, x_1, \dots, x_{n-1}$ ，即

$$\varepsilon_n^0, \varepsilon_n^1, \dots, \varepsilon_n^{n-1}, \text{ 其中 } \varepsilon_n^k = e^{i \frac{2\pi k}{n}} = \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n}.$$

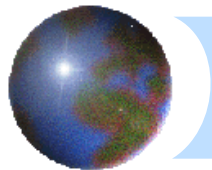


引理2： 对任何整数 $n \geq 0, k \geq 0, j > 0$ , 成立： $\varepsilon_{2n}^{2k} = \varepsilon_n^k$

证 明：

$$\varepsilon_{2n}^{2k} = \cos \frac{2\pi * 2k}{2n} + i \sin \frac{2\pi * 2k}{2n} = \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n} = \varepsilon_n^k$$

折半定理



**问题：** 求  $f(\varepsilon_n^k) = a_0 + a_1 \varepsilon_n^k + \dots + a_{n-1} (\varepsilon_n^k)^{n-1} \quad k = 0, 1, \dots, n-1$

设  $n$  为偶数（否则可以通过添加高次零项使  $n$  化为偶数）。

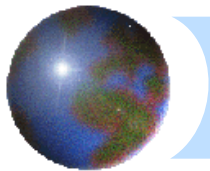
并记  $f^{[0]}(x) = a_0 + a_2 x + \dots + a_{n-2} x^{\frac{n}{2}-1} \quad f^{[1]}(x) = a_1 + a_3 x + \dots + a_{n-1} x^{\frac{n}{2}-1}$

注意到， $f^{[0]}$  中包含了  $f$  中所有偶下标的系数，而  $f^{[1]}$  中包含了  $f$  中所有奇下标的系数。

➡ 
$$f(x) = f^{[0]}(x^2) + x * f^{[1]}(x^2) \quad !$$

➡ 求  $f^{[0]}(x)$  与  $f^{[1]}(x)$  在点  $(\varepsilon_n^0)^2, (\varepsilon_n^1)^2, \dots, (\varepsilon_n^{n-1})^2$  的值。



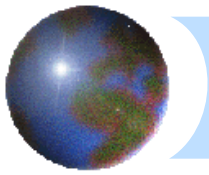


另一方面，由折半定理：

$$\begin{aligned}(\varepsilon_n^0)^2, (\varepsilon_n^1)^2, \dots, (\varepsilon_n^{n-1})^2 &= \varepsilon_{n/2}^0, \varepsilon_{n/2}^1, \dots, \varepsilon_{n/2}^{n-1} \\ &= \varepsilon_{n/2}^0, \varepsilon_{n/2}^1, \dots, \varepsilon_{n/2}^{n/2-1}, \varepsilon_{n/2}^0, \dots, \varepsilon_{n/2}^{n/2-1}\end{aligned}$$

➡  $(\varepsilon_n^0)^2, (\varepsilon_n^1)^2, \dots, (\varepsilon_n^{n-1})^2$  并不是  $n$  个不同的数，而是仅由 1 的  $n/2$  次单位根组成，每个根恰好出现 2 次。

➡ 由此可以看到子问题与原问题形式相同，但规模缩小一半，这启示我们可以利用分治的思想通过递归来解决这个问题。



# 递归算法的具体实现过程

[back](#)

Function transform(a:atype):y:ytype;

递归边界条件      if n=1 then return  $a_0$

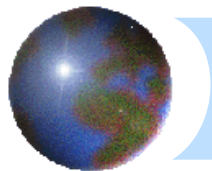
递归预处理      if odd(n) then  
                    begin inc(n);  $a_{n-1} := 0$ ; end;

递归过程       $a^{[0]} := (a_0, a_2, \dots, a_{n-2}); a^{[1]} := (a_1, a_3, \dots, a_{n-1});$   
 $y^{[0]} := \text{transform}(a^{[0]}); y^{[1]} := \text{transform}(a^{[1]});$

For k:=0 to n-1 do 利用  $f(x) = f^{[0]}(x^2) + x * f^{[1]}(x^2)$

计算  $y(\varepsilon_n^k) = y^{[0]}(\varepsilon_{n/2}^k) + \varepsilon_n^k * y^{[1]}(\varepsilon_{n/2}^k)$

可以证明，以上计算方法的时间复杂度为  $O(n \log n)$ 。



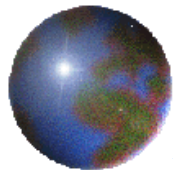
## 2. 将点值表示法再转化为系数表示法。

### （插值过程）

插值过程是点值过程的逆运算。这个问题比前一个问题看起来更复杂，但事实上，通过适当的转化可以把这个问题转化为前一个问题。

点值过程所解决的问题可以等效为一个矩阵方程：

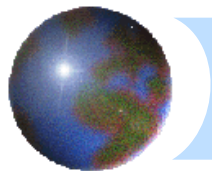
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \varepsilon_n & \varepsilon_n^2 & \varepsilon_n^3 & \cdots & \varepsilon_n^{n-1} \\ 1 & \varepsilon_n^2 & \varepsilon_n^4 & \varepsilon_n^6 & \cdots & \varepsilon_n^{2(n-1)} \\ 1 & \varepsilon_n^3 & \varepsilon_n^6 & \varepsilon_n^9 & \cdots & \varepsilon_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \varepsilon_n^{n-1} & \varepsilon_n^{2(n-1)} & \varepsilon_n^{3(n-1)} & \cdots & \varepsilon_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad \text{记为 } Y = V_n A$$



$$\begin{array}{ccc} & \text{点值过程 } Y = V_n A & \\ Y & \longleftrightarrow & A \\ & \text{插值过程} & \end{array}$$

如果  $V_n^{-1}$  存在   $A = V_n^{-1} Y$

$$V_n^{-1} \text{ ? }$$

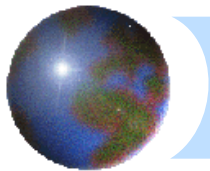


# 引理3

$$V_n = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \varepsilon_n & \varepsilon_n^2 & \varepsilon_n^3 & \cdots & \varepsilon_n^{n-1} \\ 1 & \varepsilon_n^2 & \varepsilon_n^4 & \varepsilon_n^6 & \cdots & \varepsilon_n^{2(n-1)} \\ 1 & \varepsilon_n^3 & \varepsilon_n^6 & \varepsilon_n^9 & \cdots & \varepsilon_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \varepsilon_n^{n-1} & \varepsilon_n^{2(n-1)} & \varepsilon_n^{3(n-1)} & \cdots & \varepsilon_n^{(n-1)(n-1)} \end{bmatrix}$$



$$V_n^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \varepsilon_n^{-1} & \varepsilon_n^{-2} & \varepsilon_n^{-3} & \cdots & \varepsilon_n^{-(n-1)} \\ 1 & \varepsilon_n^{-2} & \varepsilon_n^{-4} & \varepsilon_n^{-6} & \cdots & \varepsilon_n^{-2(n-1)} \\ 1 & \varepsilon_n^{-3} & \varepsilon_n^{-6} & \varepsilon_n^{-9} & \cdots & \varepsilon_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \varepsilon_n^{-(n-1)} & \varepsilon_n^{-2(n-1)} & \varepsilon_n^{-3(n-1)} & \cdots & \varepsilon_n^{-(n-1)(n-1)} \end{bmatrix}$$

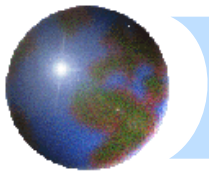


利用引理3，我们就可以很容易的解决插值的问题。

$\mathbf{Y} \longrightarrow \mathbf{A}$  可等效为矩阵方程：

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \varepsilon_n^{-1} & \varepsilon_n^{-2} & \varepsilon_n^{-3} & \cdots & \varepsilon_n^{-(n-1)} \\ 1 & \varepsilon_n^{-2} & \varepsilon_n^{-4} & \varepsilon_n^{-6} & \cdots & \varepsilon_n^{-2(n-1)} \\ 1 & \varepsilon_n^{-3} & \varepsilon_n^{-6} & \varepsilon_n^{-9} & \cdots & \varepsilon_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \varepsilon_n^{-(n-1)} & \varepsilon_n^{-2(n-1)} & \varepsilon_n^{-3(n-1)} & \cdots & \varepsilon_n^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

因此，我们可以充分利用点值过程的方法求出多项式的系数表达。



Function transform( $y:ytype$       $a:atype$

[back](#)

$a:atype$ );  $y:ytype$ ;  
递归边界条件     if  $n=1$  then return  $y_0$

递归预处理     if odd( $n$ ) then  
                    begin inc( $n$ );  $y_{n-1}:=0$ ; end;

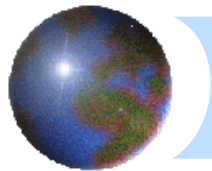
递归过程      $y^{[0]} := (y_0, y_2, \dots, y_{n-2}); y^{[1]} := (y_1, y_3, \dots, y_{n-1});$

$a^{[0]} := transform(y^{[0]}); a^{[1]} := transform(y^{[1]});$

For  $k:=0$  to  $n-1$  do 利用  $f(x) = f^{[0]}(x^2) + x * f^{[1]}(x^2)$

计算  $a(\varepsilon_n^{-k}) = a^{[0]}(\varepsilon_{n/2}^{-k}) + \varepsilon_n^{-k} * a^{[1]}(\varepsilon_{n/2}^{-k})$

递归结束后再将 $a$ 中每一个数除以 $n$ 。



# 多项式乘法的算法流程

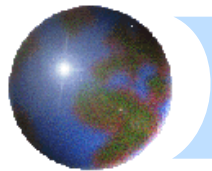
**问题：**  $f(x)$ ,  $g(x)$  是两个  $n-1$  次的多项式，已知  $f(x)g(x)$  的系数表示，求出  $r(x)=f(x)*g(x)$  的系数表示。

**算法流程：**

1. 预处理：通过加入  $n-1$  个值为0的高价次数，使  $f(x)g(x)$  的次数增加到  $2n-2$ 。这是为了使点值对的个数足够能够唯一确定  $r(x)$ 。
2. 点值：利用分治的方法，通过函数 `transform` 求出  $f(x)$  与  $g(x)$  在1的  $2n-1$  次单位根处的值。
3. 点乘：将  $f(x)$ ,  $g(x)$  在各点的值逐点相乘，计算出  $r(x)$  在各点的值。
4. 插值：互换  $a$  与  $y$  的作用，再利用函数 `transform` 求  $r(x)$  的系数表示，并注意要将结果除以次数作为最后结果。

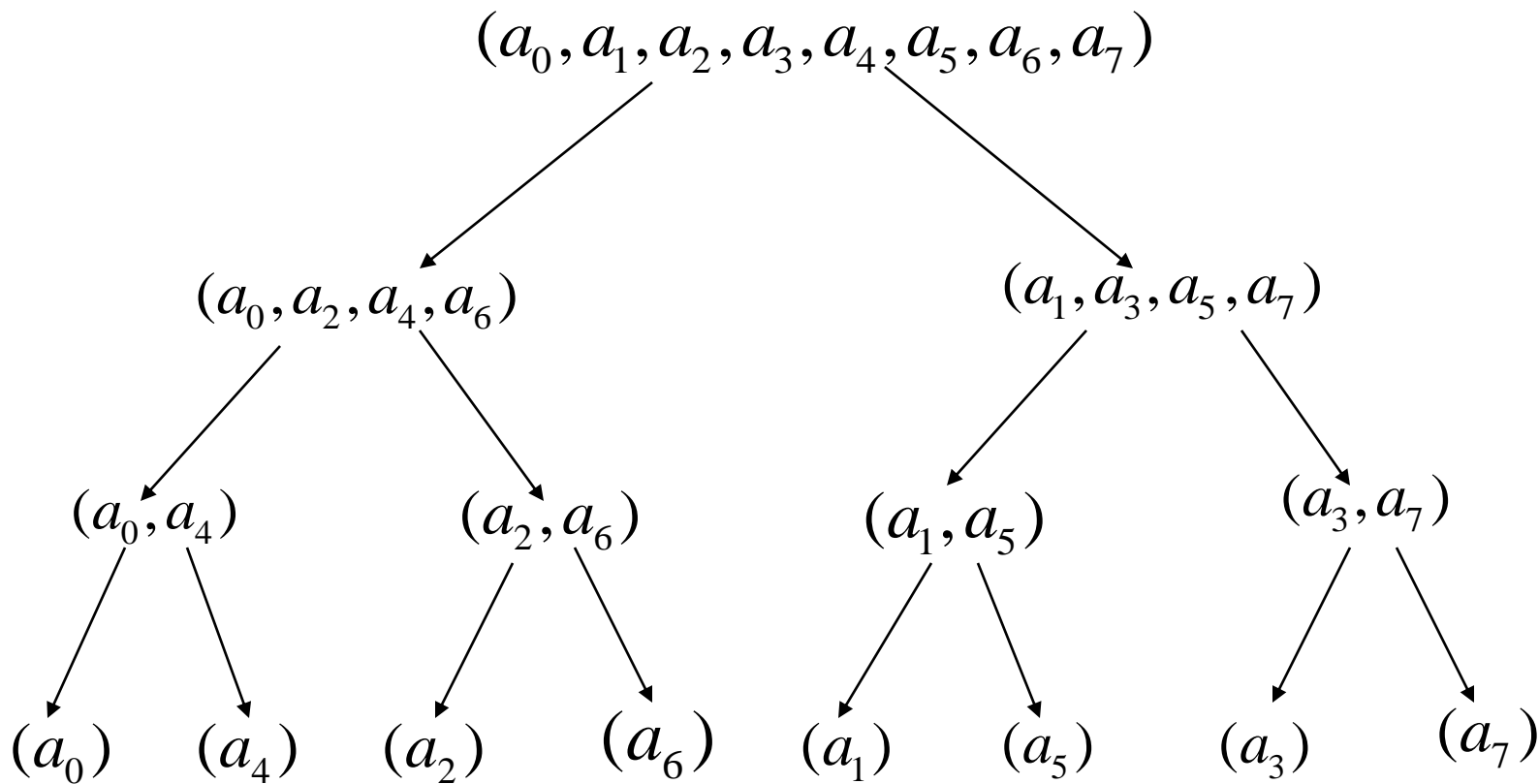
以上第1、第3步的执行时间都是  $O(n)$ ，第2、第4步的执行时间都是  $O(n \log n)$ 。

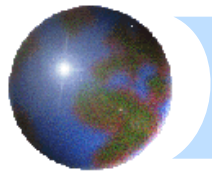




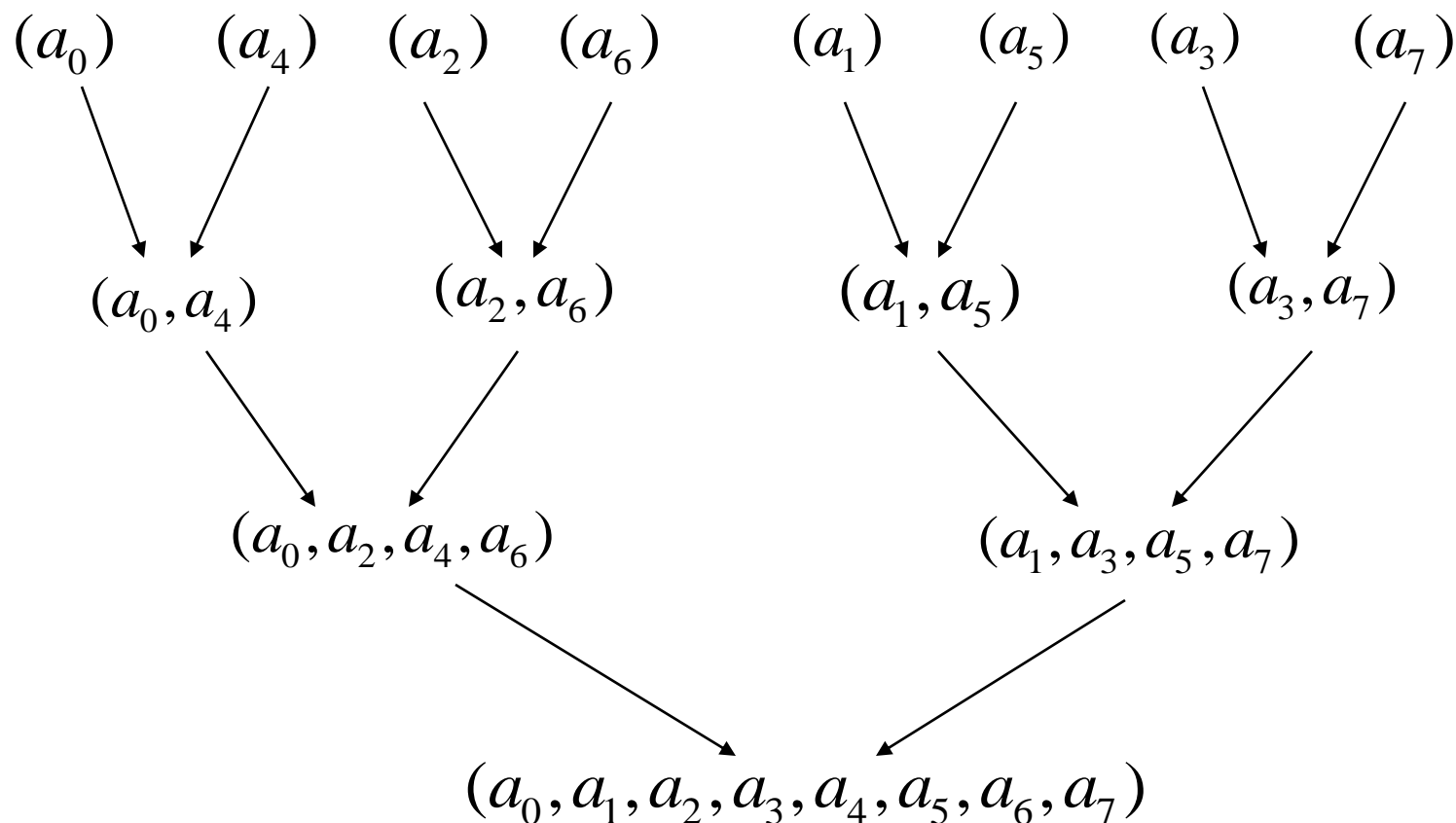
# 算法改进

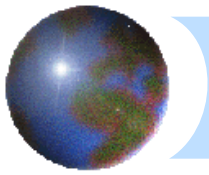
在函数transform中，我们是用递归的方式来求解，我们对n=8的情况来具体演示一下递归调用的过程。





但是递归的方法对空间的要求很高，从函数transform中可以看到每次递归调用时都需要新的系数数组传入递归过程内部。而通过刚才的演示，我们发现我们也可以用从底向上迭代的方法来进行。





# 迭代算法的具体实现过程

**Function transform\_better (a:atype):y:ytype;**

预处理

通过增加高次零项使将多项式的次数增加到2的幂次。

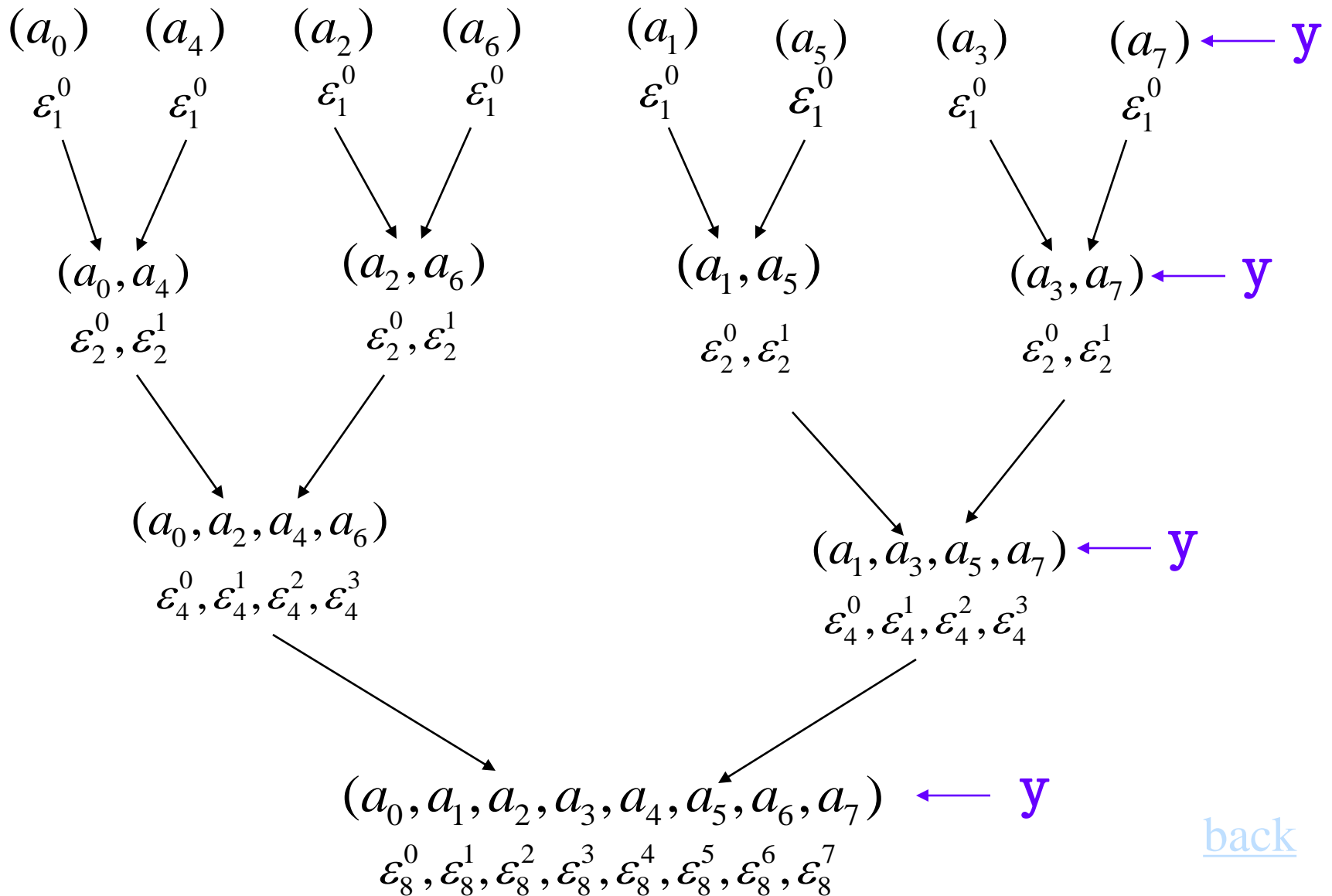
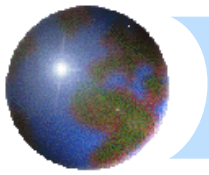
初始化

**y:=a;**

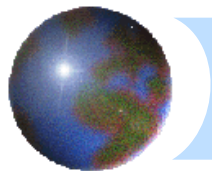
迭代过程

For k:=1 to lg n do

对数组进行恰当的合并并将结果放到数组恰当的位置。



[back](#)



下面我们将本文介绍的方法与普通的多项式乘法做一个比较。

测试环境: PIII 500 128M RAM    FreePascal 1.0.4

	多项式次数	n=100	n=1000	n=10000	n=20000	n=30000
普通方法	时间(s)	0.02	0.07	5.02	18.06	>30
点值方法	时间(s)	0.05	0.50	2.46	2.88	3.36
	精度(real)	10~11位	10~11位	9~10位	8~9位	8~9位