Computer Algebra 2

October 11, 2018

1 Notations and conventions

Unless otherwise mentioned, we use the following notations:

- k, K, $\mathbb{K}$ are (commutative) fields
- R is a (commutative, with 1) ring

Given a ring R, R^* is the group of its invertible elements.

We assume that algebraic computations (sum, inverse, test of 0, test of 1, inverse where applicable) can be performed.

For a vector v in a vector space V of dimension n, we denote its coordinates by $(v_1, \dots, v_{n-1})$. If f is a polynomial of degree $\deg(f) = d$, its coefficients are denoted $f_0, \dots, f_d$, such that

$$f(X) = f_0 + f_1 X + \dots + f_d X^d = \sum_{i=0}^{d} f_i X^i.$$

In order to simplify notations, we may at times use the convention that $f_i = 0$ if $i < 0$ or $i > \deg(f)$, so that

$$f = \sum_{i \in \mathbb{Z}} f_i X^i.$$

By convention, the degree of the 0 polynomial is $-\infty$.

The logarithm log, without a base, is in base 2.

Definition 1. Given two functions $f, g : \mathbb{N} \to \mathbb{R}_{>0}$

$$\begin{aligned} f = O(g) &\iff \frac{f(n)}{g(n)} \text{ is bounded when } n \to \infty \\ &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq c g(n); \end{aligned}$$

$$f = \tilde{O}(g) \iff \exists l \in \mathbb{N}, f = O(g \log(g)^l).$$

1.1 Exercises

Exercise 1.1. Show that the “when $n \to \infty$” clause in the definition of O can be left out. In other words, given $f, g : \mathbb{N} \to \mathbb{R}_{>0}$, show that

$$\begin{aligned} f = O(g) &\iff \frac{f(n)}{g(n)} \text{ is bounded} \\ &\iff \exists c \in \mathbb{R}_{>0}, \forall n \in \mathbb{N}, f(n) \leq cg(n) \end{aligned}$$

2 Semi-fast multiplication

In this chapter, let R be any ring.

Given $f, g \in R[X]$ with degree less than n, we want to compute the coefficients of $h = f \cdot g$.

The complexity of the algorithm will be evaluated in number of multiplications and additions in R. Typically, multiplications are more expensive!

2.1 Naive algorithm

Each coefficient h_k $(0 \leq k < 2n)$ can be computed with

$$h_k = \sum_{i=0}^{k} f_i g_{k-i},$$

each costing $O(n)$ multiplications and additions.

The total complexity of the naive algorithm is $O(n^2)$ multiplications and $O(n^2)$ additions.

2.2 Karatsuba's algorithm

Remark 2. Linear polynomials can be multiplied using 3 multiplications instead of 4 :

$$(a + bX)(c + dX) = ac + (ad + bc)X + bdX^2$$

with

$$ad + bc = ad + bc + ac + bd - ac - bd = (a + b)(c + d) - ac - bd.$$

This can be used recursively to compute polynomial multiplication faster.

Algorithm 1 Karatsuba

Input: $f = f_0 + \cdots + f_{n-1}X^{n-1}$, $g = g_0 + \cdots + g_{n-1}X^{n-1}$

Output: $h = h_0 + \cdots + h_{2n-1}X^{2n-1}$ such that $h = fg$

1. If $n = 1$, then return $f_0 g_0$
2. Write $f = A + BX^{\lceil n/2 \rceil}$, $g = C + DX^{\lceil n/2 \rceil}$ where all of A, B, C, D have degree $< \lceil \frac{n}{2} \rceil$.
3. Compute recursively:
   - $P = AC$
   - $Q = BD$
   - $R = (A + B)(C + D)$
4. Return $P + (R - P - Q)X^{\lceil n/2 \rceil} + RX^{2\lceil n/2 \rceil}$

Theorem 3. *Karatsuba's algorithm multiplies polynomials with $O(n^{\log_2(3)}) = O(n^{1.585})$ multiplications and additions.*

Proof. Let $M(n)$ (resp. $A(n)$) be the number of multiplications (resp. additions) in a run of Algo. 1 on an input with size n. Then:

$$M(n) = 3M(n/2)$$

and

$$A(n) = 3A(n/2) + O(n)$$

so $M(n) = O(n^{\log_2(3)})$ and $A(n) = O(n^{\log_2(3)})$. □

Remark 4. Karatsuba's algorithm hides an evaluation/interpolation mechanism:

$$\begin{aligned} a &= (a + bX)_{X=0} \\ a + b &= (a + bX)_{X=1} \\ b &= \left(\frac{a + bX}{X}\right)_{X=\infty} \end{aligned}$$

and for two linear polynomials f, g, if $fg = h = h_0 + h_1X + h_2X^2$, we have

$$\begin{aligned} &f(0)g(0) = h(0) = h_0 \\ &f(1)g(1) = h(X = 1) = h_0 + h_1 + h_2 \\ &\left(\frac{f}{X}\right)_{X=\infty}\left(\frac{g}{X}\right)_{X=\infty} = \left(\frac{h}{X^2}\right)_{X=\infty} = h_2 \end{aligned}$$

2.3 Toom-k algorithm

For the remainder of this section, assume that the ring R is an infinite field.

In general the coefficients of h can be obtained as a linear combination of $f(i)g(i)$ for $i \in \{0, \ldots, 2n-1\}$ via

$$\begin{pmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}^{-1} \left[\begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \end{pmatrix} \right]$$

where $\odot$ is the component-wise multiplication of two vectors.

This suggests the following generalization of Algo. 1 for any fixed $k \geq 2$. First, let $V = \left(i^j\right)_{i,j=0}^{2k-1}$ (Vandermonde matrix), and precompute V^{-1}.

Algorithm 2 Toom-k

Input: $f = f_0 + \cdots + f_{n-1}X^{n-1}$, $g = g_0 + \cdots + g_{n-1}X^{n-1}$

Output: $h = h_0 + \cdots + h_{2n-1}X^{2n-1}$ such that $h = fg$

1. If $n < \max(k, 16)$, compute h naively and stop # Forget the "16" until Sec. 2.4
2. Write $f = F_0 + F_1X^{\lceil n/k \rceil} + \cdots + F_{k-1}X^{(k-1)\lceil n/k \rceil}$ and $g = G_0 + G_1X^{\lceil n/k \rceil} + \cdots + G_{k-1}X^{(k-1)\lceil n/k \rceil}$ where $\deg(F_i)$ and $\deg(G_i) < n/k$
3. Compute $\bar{f} = V\begin{pmatrix} F_0 \\ F_1 \\ \vdots \end{pmatrix}$ and $\bar{g} = V\begin{pmatrix} G_0 \\ G_1 \\ \vdots \end{pmatrix}$
4. Compute $\bar{h} = \bar{f} \odot \bar{g}$ recursively
5. Return $V^{-1}\bar{h}$

Remark 5. If we write $F_i = f_0^{(i)} + f_d^{(i)}X^d$ for $i \in \{0, \ldots, k-1\}$, one can compute the product $V \cdot (F_i)$ as

$$\begin{aligned} V \cdot \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{k-1} \end{pmatrix} &= V \cdot \left[\begin{pmatrix} f_0^0 \\ f_0^{(1)} \\ \vdots \\ f_0^{(k-1)} \end{pmatrix} + \begin{pmatrix} f_1^0 \\ f_1^{(1)} \\ \vdots \\ f_1^{(k-1)} \end{pmatrix} X + \cdots + \begin{pmatrix} f_d^0 \\ f_d^{(1)} \\ \vdots \\ f_d^{(k-1)} \end{pmatrix} X^d \right] \\ &= V \cdot \begin{pmatrix} f_0^0 \\ f_0^{(1)} \\ \vdots \\ f_0^{(k-1)} \end{pmatrix} + V \cdot \begin{pmatrix} f_1^0 \\ f_1^{(1)} \\ \vdots \\ f_1^{(k-1)} \end{pmatrix} X + \cdots + V \cdot \begin{pmatrix} f_d^0 \\ f_d^{(1)} \\ \vdots \\ f_d^{(k-1)} \end{pmatrix} X^d \end{aligned}$$

so the cost of computing that product is $O(dk^2)$.

Theorem 6. *A run of Algorithm 2 requires $O(n^{\log_k(2k-1)})$ operations. In particular, for any fixed $\epsilon > 0$, there exists a multiplication algorithm for $R[X]$ which requires $O(n^{1+\epsilon})$ operations in R.*

Proof. See Exercise 2.2. □

Remark 7. For fixed k, the cost of precomputing V and V^{-1} can be neglected, since it is a fixed cost of $O(k^2)$ and $O(k^3)$ respectively.

2.4 Toom-Cook algorithm

Theorem 8 (Toom-Cook)**.** *There exists a multiplication algorithm for $R[X]$ that requires $O(n^{1+2/\sqrt{\log(n)}})$ operations in R. This algorithm is obtained by adapting Algo. 2 to choose at each recursion level $k = \left\lfloor 2^{2\sqrt{\log(n)}} \right\rfloor$.*

Proof. See Exercise 2.3. □

Remark 9. This complexity is better than that of Toom-k, since it is better than $O(2^{1+\epsilon})$ for *all* $\epsilon > 0$.

Remark 10. Strassen's algorithm for matrix multiplication is based on the same idea as Karatsuba's algorithm, and runs in time $O(n^{\log_2(7)}) \le O(n^{2.82})$. Is there a Toom-Cook style algorithm for matrix multiplication, with complexity better than $O(2^{2+\epsilon})$ for all $\epsilon > 0$?

For even k, we can multiply $k \times k$ matrices with $\frac{1}{3}k^3 + 6k^2 - \frac{4}{3}k$ operations, so there are matrix multiplication algorithms with complexity $O(n^{\log_k(\frac{1}{3}k^3+6k^2-\frac{4}{3}k)})$. But $\log_k(\frac{1}{3}k^3 + 6k^2 - \frac{4}{3}k)$ tends to 3 when k tends to ∞. Its minimum (over $2\mathbb{N}$) is reached at $k = 70$, leading to a complexity $O(n^{2.796})$ (Pan's algorithm).

The current record is $O(n^{2.3728639})$ (Le Gall 2014), and yes, that many decimal points are necessary! It is conjectured that a complexity of $O(2^{1+\epsilon})$ for all ϵ is realizable.

Remark 11. It is conjectured that polynomial multiplication in $O(n)$ operations is not possible.

2.5 Exercises

Exercise 2.1. Is it possible to use the ideas of the Algorithm of Toom-k with evaluation at $\{0, 1, \ldots, k-2, \infty\}$? Describe the matrices V and V^{-1}.

Exercise 2.2. Prove Theorem 6.

Exercise 2.3. Prove Theorem 8.

Exercise 2.4. Show that there is no algorithm which can multiply two linear polynomials (over any ring) in 2 multiplications.

3 Fast multiplication in $\bar{k}[X]$

In this chapter, let k be an *algebraically closed* field. The problem to solve is the same as previously, but this time, we assume that $\deg(f) + \deg(g) < n$.

We will be considering evaluation/interpolation methods.

Algorithm 3 Evaluation/interpolation

Input: $f = f_0 + \cdots + f_{k-1}X^k$, $g = g_0 + \cdots + g_{l-1}X^l$ with $k + l < n$

Output: $h = h_0 + \cdots + h_{n-1}X^{n-1}$ such that $h = fg$

1. Fix $(x_0, \ldots, x_{n-1}) \in k^n$
2. Compute $f(x_i)$, $g(x_i)$ for $i = 0, \ldots, n-1$
3. Compute $h(x_i) = f(x_i)g(x_i)$ for $i = 0, \ldots, n-1$
4. Compute h by interpolating $h(x_i)$ for $i = 0, \ldots, n-1$

Remark 12. In general, Algo. 3 requires $O(n^2) + O(n) + O(n^2) = O(n^2)$ operations in k, like the classical algorithm. The idea is to choose specific values of $x_0, \ldots, x_{n-1}$ so that steps 2 and 4 can be done faster.

3.1 Roots of unity and discrete Fourier transform

Definition 13. An element $\omega \in k$ is called a *n'th root of unity* if $\omega^n = 1$. It is a *primitive* n'th root of unity if additionally $\omega^i \neq 1$ for $0 < i < n$.

Example 14. In $\mathbb{C}$, -1 is a primitive second root of unity. i is a primitive 4th root of unity.

In $\mathbb{F}_{17}$, 2 is a primitive 8th root of unity.

Definition 15. The matrix

$$\mathrm{DFT}_n := \mathrm{DFT}_n^{(\omega)} := \left(\omega^{ij}\right)_{i,j=0}^{n-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \in k^{n \times n}$$

is called the *discrete Fourier transform* (wrt ω).

Example 16. In $\mathbb{C}$, the discrete Fourier transform wrt i is

$$\mathrm{DFT}_4^{(i)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}.$$

Remark 17. The DFT is a Vandermonde matrix. In particular, if $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$,

$$\mathrm{DFT}_n^{(\omega)} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}.$$

Definition 18. Let $f, g \in k^n$. The *product* $f \odot g$ is the vector whose i'th coordinate is given by $f_i g_i$. The *convolution* $f * g$ is the vector whose i'th coordinate is given by

$$\sum_{k=0}^{n-1} f_k g_{(i-k) \bmod n}.$$

Lemma 19. *Let ω be a primitive n'th root of unity. Then*

1. *there is a factorization*
$$X^n - 1 = (X - \omega)(X - \omega^2) \cdots (X - \omega^n);$$
2. *for any $j \in \{1, \dots, n-1\}$,*
$$\sum_{i=0}^{n-1} \omega^{ij} = 0.$$
3. *there is a group isomorphism*
$$\left(\{\omega^i : i \in \mathbb{Z}\}, \cdot\right) \simeq (\mathbb{Z}/n\mathbb{Z}, +)$$
4. *the DFT matrix is easy to invert:*
$$\left(\mathrm{DFT}_n^{(\omega)}\right)^{-1} = \frac{1}{n} \mathrm{DFT}_n^{(1/\omega)}$$
5. *if $m \mid n$, then ω^m is a primitive (n/m)'th root of unity*
6. *the DFT is compatible with convolution*
$$\mathrm{DFT}_n(f * g) = \mathrm{DFT}_n(f) \odot \mathrm{DFT}_n(g)$$

Proof. 1. All ω^i are distinct: if $\omega^i = \omega^j$ with $1 \leq i < j \leq n$, then $\omega^{j-i} = 1$ with $0 < j - i < n$, which is a contradiction because ω is a primitive root of unity. All ω^i are roots of $X^n - 1$, since $(\omega^i)^n = (\omega^n)^i = 1$, so the $X - \omega^i$ are n distinct factors of $X^n - 1$. By comparing the degree and leading coefficient, we get the wanted factorization.

2. Use the formula

$$\left(\sum_{i=0}^{n-1} X^i\right)(X-1) = X^n - 1$$

Evaluated at $X = \omega^j$ for $0 < j < n$, the right hand side is 0, the factor $(\omega^j - 1)$ is non-zero, so the sum has to be zero.

3. Clear.

4. Evaluate the product:

$$\begin{aligned}
\mathrm{DFT}_n^{(\omega)}\mathrm{DFT}_n^{(1/\omega)} &= \left(\omega^{ij}\right)_{i,j=0}^{n-1} \cdot \left(\omega^{-ij}\right)_{i,j=0}^{n-1} \\
&= \left(\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj}\right)_{i,j=0}^{n-1} \\
&= \left(\sum_{k=0}^{n-1} \omega^{k(i-j)}\right)_{i,j=0}^{n-1} \\
&= \left(n\delta_{ij}\right)_{i,j=0}^{n-1}.
\end{aligned}$$

5. Clear.

6. If we associate the vector $f = (f_0, \ldots, f_{n-1})$ with the polynomial $f(X) = f_0 + \cdots + f_{n-1}X^{n-1}$, convolution is equivalent to multiplication in $k[X]/\langle X^n - 1\rangle$, that is

$$(f * g)(X) = f(X)g(X) + q(X) \cdot (X^n - 1)$$

for some $q \in k[X]$. Indeed, write

$$\begin{aligned}
f(X)g(X) &= \sum_{i,j=0}^{n-1} f_i g_j X^{i+j} \\
&= \sum_{i+j<n} f_i g_j X^{i+j} + \sum_{n \leq i+j<2n} f_i g_j X^{i+j} \\
&= \underbrace{\sum_{i+j<n} f_i g_j X^{i+j} + \sum_{n \leq i+j<2n} f_i g_j X^{i+j-n}}_{(f*g)(X)} \underbrace{- \sum_{n \leq i+j<2n} f_i g_j X^{i+j-n} + \sum_{n \leq i+j<2n} f_i g_j X^{i+j}}_{(\sum_{n \leq i+j<2n} f_i g_j X^{i+j-n})(X^n-1)}
\end{aligned}$$

The claim follows by evaluation at ω^i. □

The remark, together with property 4, makes powers of ω a good choice for evaluation and interpolation: if we can just find a fast way to evaluate $\mathrm{DFT}_n \cdot f$, we can perform both steps in a fast way.

3.2 Fast Fourier transform

Given $f = \begin{pmatrix} f_0 \\ \vdots \\ f_{2n-1} \end{pmatrix}$, we want to compute $\bar{f} = \mathrm{DFT}_{2n} \cdot f$.

Let's expand the j'th coefficient:

$$
\begin{aligned}
\left(\mathrm{DFT}_{2n}^{\omega} f\right)_j &= \sum_{i=0}^{2n-1} \omega^{ij} f_i \\
&= \sum_{i=0}^{n-1} \omega^{2ij} f_{2i} + \sum_{i=0}^{n-1} \omega^{(2i+1)j} f_{2i+1} \\
&= \sum_{i=0}^{n-1} (\omega^2)^{ij} f_{2i} + \omega^j \sum_{i=0}^{n-1} (\omega^2)^{ij} f_{2i+1} \\
&= \begin{cases} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{even}}\right)_j + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{odd}}\right)_j & \text{for } 0 \le j < n \\ \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{even}}\right)_{j-n} + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{odd}}\right)_{j-n} & \text{for } n \le j < 2n \end{cases} \\
&= \begin{cases} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{even}}\right)_j + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{odd}}\right)_j & \text{for } 0 \le j < n \\ \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{even}}\right)_{j-n} - \omega^{j-n} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\mathrm{odd}}\right)_{j-n} & \text{for } n \le j < 2n \end{cases}
\end{aligned}
$$

We can use this property to perform the evaluation and interpolation steps.

Algorithm 4 Fast Fourier Transform

Input: $f \in k^n$, ω a primitive n'th root of unity, $n = 2^k$

Output: $\bar{f} = \mathrm{DFT}_n^{(\omega)} f$

1. If $n = 1$ then return (f_0)
2. $u \leftarrow \mathsf{FFT}([f_0, f_2, \ldots,], \omega^2, n/2)$, $v \leftarrow \mathsf{FFT}([f_1, f_3, \ldots,], \omega^2, n/2)$
3. Return $[u_0 + v_0, u_1 + \omega v_1, u_2 + \omega^2 v_2, \ldots, u_{n/2-1} + \omega^{n/2-1} v_{n/2-1},$
   $u_0 - v_0, u_1 - \omega v_1, u_2 - \omega^2 v_2, \ldots, u_{n/2-1} - \omega^{n/2-1} v_{n/2-1}]$

Theorem 20. *Algo. 4 requires $O(n \log(n))$ operations in k.*

Proof. Similar to before, with the recurrence

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

□

This allows us to rewrite Algo. 3 with the FFT.

Algorithm 5 Evaluation/interpolation multiplication using FFT

Input: $f = f_0 + \cdots + f_{k-1}X^k$, $g = g_0 + \cdots + g_{l-1}X^l$ with $k + l < n$

Output: $h = h_0 + \cdots + h_{n-1}X^{n-1}$ such that $h = fg$

1. $\omega \leftarrow$ primitive n'th root of unity
2. $\bar{f} \leftarrow \mathsf{FFT}(f, \omega)$, $\bar{g} \leftarrow \mathsf{FFT}(g, \omega)$
3. $\bar{h} \leftarrow \bar{f} \odot \bar{g}$
4. Return $\frac{1}{n}\mathsf{FFT}(\bar{h}, \omega^{-1})$

Theorem 21. *Multiplication in $k[X]$ can be done with $O(n \log n)$ operations in k if k is algebraically closed.*

Remark 22. This complexity is currently the best known complexity for polynomial multiplication.

Remark 23. Let P be the permutation matrix such that

$$P \cdot f = \begin{pmatrix} f_{\text{even}} \\ f_{\text{odd}} \end{pmatrix}$$

and Δ be the diagonal matrix

$$\Delta = \begin{pmatrix} 1 & & & \\ & \omega & & \\ & & \omega^2 & \\ & & & \ddots \end{pmatrix}.$$

Then the computations above yield that

$$\begin{aligned} \mathrm{DFT}_{2n} &= \begin{pmatrix} \mathrm{DFT}_n & \Delta\mathrm{DFT}_n \\ \mathrm{DFT}_n & -\Delta\mathrm{DFT}_n \end{pmatrix} \cdot P \\ &= \begin{pmatrix} I & \Delta \\ I & -\Delta \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & \\ & \mathrm{DFT}_n \end{pmatrix} \cdot P \\ &= \begin{pmatrix} I & I \\ I & -I \end{pmatrix} \cdot \begin{pmatrix} I & \\ & \Delta \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & \\ & \mathrm{DFT}_n \end{pmatrix} \cdot P \end{aligned}$$

This can be generalized to divisions by m instead of 2. Skipping over the details, this gives

$$\mathrm{DFT}_{mn} = \begin{pmatrix} I & I & I & \cdots \\ I & \omega^n I & \omega^{2n} I & \cdots \\ I & \omega^{2n} I & \omega^{4n} I & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \cdot \begin{pmatrix} I & & & \\ & \Delta & & \\ & & \Delta^2 & \\ & & & \ddots \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & & & \\ & \mathrm{DFT}_n & & \\ & & \mathrm{DFT}_n & \\ & & & \ddots \end{pmatrix} \cdot P.$$

This is a result due to Cooley and Tuckey, which can be used to refine Algo. 4 so that it reduces an FFT of *any* size quickly to FFT's of prime size.