Evaluation-interpolation, its elegance and utility notwithstanding, does suffer from one notable disadvantage: it is largely unable to exploit sparseness (= many zero-coefficient terms) in polynomial data. For example consider the multiplication of $3x^{100}+2$ by $4x^{97}+5x^{20}$. Only four coefficient multiplications are required using the obvious direct method, whereas evaluation-interpolation would require many more (in the thousands). It is precisely this phenomenon that makes "direct" methods—methods that operate on polynomial data using classical algorithms—an attractive alternative to evaluation-interpolation. The use of specially tailored Gaussian elimination schemes for integer and polynomial systems of linear equations is the topic of E. H. Bareiss [*Math. Comp.* **22** (1968), 565–578; *J. Inst. Math. Applic.* 10 (1972), 68–104] and J. D. Lipson [in R. G. Tobey (ed.), *Proc. 1968 Summer Institute on Symbolic Mathematical Computation*, IBM Programming Laboratory Report FSC69-0312, June 1969, 235–303]. The issue of sparseness in the development and analysis of polynomial matrix algorithms (determinant calculation, linear equations solution) is the central topic of E. Horowitz and S. Sahni, *J. ACM* **22** (1975), 38–50; W. M. Gentleman and S. C. Johnson, *ACM Trans. Math. Software* **2** (1976), 232–241; M. L. Griss, *ACM Trans. Math. Software* **2** (1976), 31–49.

2. For one efficient such algorithm $C(A,F)$, see *Danilevsky's Method* in V. N. Fadeeva, *Computational Methods of Linear Algebra* (New York: Dover, 1959), Section 24. This method computes the characteristic polynomial of an $n \times n$ matrix A in $O(n^3)$ field operations.

CHAPTER

IX

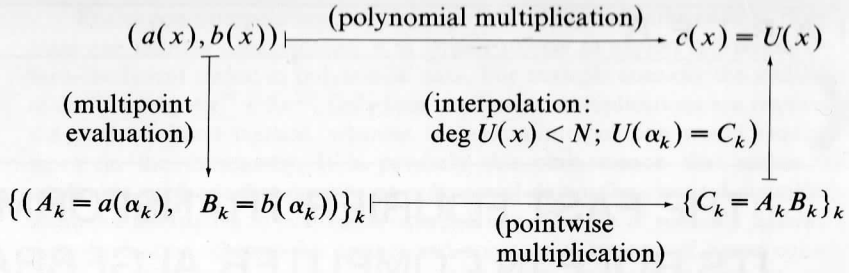
THE FAST FOURIER TRANSFORM: ITS ROLE IN COMPUTER ALGEBRA

An algorithm may be appreciated on a number of grounds: on technological grounds because it efficiently solves an important practical problem, on aesthetic grounds because it is elegant, or even on dramatic grounds because it opens up new and unexpected areas of application. The *fast Fourier transform* (popularly referred to as the "FFT"), perhaps because it is strong in *all* of these departments, has emerged as one of the "super" algorithms of Computer Science since its discovery in the mid sixties. This concluding chapter is devoted to this remarkable algorithm and some of its major applications to algebraic computing.

1. WHAT IS THE *FAST* FOURIER TRANSFORM?

Recall our application of evaluation-interpolation to polynomial multiplication: to compute the product $c(x)$ over $F[x]$ of $a(x)=\sum_{i=0}^{n}a_i x^i$ and $b(x)=\sum_{i=0}^{n}b_i x^i$, evaluation-interpolation requires that we choose (at least) $N=2n+1$ distinct points $\alpha_k \in F$ and then proceeds according to the mapping diagram below.

John D. Lipson, Elements of Algebra and Algebraic Computing. ISBN 0-201-04115-4.



As we discovered, this application was far from successful. Although evaluation-interpolation yields an $O(n^2)$ algorithm, the constant is considerably larger than that of the school method. But we did lay open the tantalizing prospect of performing the evaluation and interpolation steps proper in time $< O(n^2)$, which would then yield a superior algorithm (at least asymptotically—for large n). This is the departure point for this chapter.

How can such a speedup be achieved? The key idea, the one that lies at the heart of the FFT, is simply this: the evaluation-interpolation points (the α_k's), though they must be distinct, are otherwise completely arbitrary. *So let us choose them wisely.*

1.1 The Forward Transform: Fast Multipoint Evaluation

The forward transform of evaluation-interpolation is multipoint polynomial evaluation over a field F. We shall focus our attention on the following

> *Problem P_N of "size" N:* Evaluate a polynomial $a(x) = \sum_{i=0}^{N-1} a_i x^i$ of "length" N (length = degree + 1) at each of a set $E_N = \{\alpha_k\}_{k=0}^{N-1}$ of N distinct points $\alpha_k \in F$ (the "evaluation points").

The solution to P_N is the collection of polynomial values $A_k = a(\alpha_k)$ $(k = 0, \ldots, N-1)$.

To analyze and compare algorithms for solving P_N, we shall count $M(N)$, the required number of multiplications over F. ($M(N)$ is a valid figure of merit, since multiplications dominate the arithmetic work of the algorithms to be discussed.)

To show off our more inspired solutions to P_N, we record the pedestrian

Proposition 1. For arbitrary evaluation points, P_N can be solved in $M(N) = N^2 + O(N)$.

Proof. Compute each $a(\alpha_k)$ $(k = 0, \ldots, N-1)$ by Horner's rule. □

The idea now is to impose some structure on the evaluation points that can be exploited to speedup the solution to P_N.

Definition. Let $N = 2n$ be even. A collection of N distinct points $E_N = \{\alpha_k\}_{k=0}^{N-1}$ is said to have *Property S* if E_N can be written as

$$E_N = \{\pm\alpha_k\}_{k=0}^{n-1}.$$

('S' thus stands for symmetry of sign; if β is in E_N, then so is $-\beta$.)

Let E_N have Property S. Then, since $(-\beta)^2 = (+\beta)^2$, we see that only $N/2$ of the *squares* of points in E_N are distinct. This little observation is the key to speeding up multipoint evaluation.

Proposition 2. Let N be even, $N = 2n$, and let $E_N = \{\alpha_k\}_k$ have property S. Then P_N can be solved in $M(N) = N^2/2 + O(N)$.

Proof. We can decompose $a(x) = \sum_{i=0}^{N-1} a_i x^i$ according to

$$(*) \qquad a(x) = b(y) + xc(y)$$

where $$y = x^2, \quad b(y) = \sum_{i=0}^{n-1} a_{2i} y^i, \quad c(y) = \sum_{i=0}^{n-1} a_{2i+1} y^i.$$

[Example ($N = 4$):

$$a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$= \underbrace{(a_0 + a_2 y)}_{b(y)} + x\underbrace{(a_1 + a_3 y)}_{c(y)} \quad \text{where } y = x^2.]$$

Thus $A_k = a(\alpha_k)$, $A_{-k} = a(-\alpha_k)$ can be evaluated according to

Algorithm 1 (*Binary splitting scheme*).

Step 1. Compute $\beta_k = \alpha_k^2$ $(k = 0, \ldots, n-1)$.

Step 2. With $b(y), c(y)$ given by $(*)$, use Horner's rule to compute

$$B_k = b(\beta_k),$$
$$C_k = c(\beta_k) \qquad (k = 0, \ldots, n-1).$$

Step 3. Return $A_k = B_k + \alpha_k C_k$,

$$A_{-k} = B_k - \alpha_k C_k \qquad (k = 0, \ldots, n-1).$$

Steps 1 and 3 require $O(n)$ multiplications while step 2 requires the evaluation of two polynomials at n points, which using Horner's rule requires $2n^2 + O(n)$ multiplications (Proposition 1). Thus the overall number of multiplications is $M(2n) = 2n^2 + O(n)$, or $M(N) = N^2/2 + O(N)$. □

Thus, by exploiting the small amount of structure of points E_N enjoying Property S, we have achieved a speedup of a factor of two over the solution of Proposition 1—a solid improvement even though the asymptotic character of the solution to our size N multipoint evaluation problem still remains $O(N^2)$.

We note that the above algorithm has essentially solved the original problem P_N of size N in terms of two problems $P_{N/2}$ of half the size. Since the evaluation points for these two subproblems are not special (in particular they will not in general have Property S), Algorithm 1 must be content to use the Horner's rule solution of Proposition 1 to solve these two subproblems. Now wouldn't it be nice (we muse) if we could speed up the solution to these two subproblems as well? To this end we make the leap to a very special class of evaluation points.

In field theory, we refer to a field element ω that has multiplicative order N as a *primitive Nth root of unity:* an Nth root of unity in that ω satisfies $x^N - 1 = 0$; a *primitive* Nth root of unity in that ω satisfies $x^k - 1 = 0$ for no positive $k < N$. (Thus a primitive element in a field of order r (Section VI.2) is a primitive $(r-1)$st root of unity.)

Example 1. In **C**, $e^{2\pi i/N}$ is a primitive Nth root of unity (Example 3 of Section III.3.4). In $\mathbf{Z}_{13}$, 8 is a primitive 4-th root of unity. (Check: $8^2 = 12$, $8^3 = 5$, $8^4 = 1$). □

We refer to the N distinct integral powers of a primitive Nth root of unity ω,

$$[\omega] = \{1, \omega, \ldots, \omega^{N-1}\},$$

as a set of N *Fourier points*. These are the points that we shall choose for our multipoint evaluation problem. (We shall shortly elucidate their connection with Fourier transforms.) Let us call a multipoint evaluation problem P_N at N Fourier points a *Fourier evaluation problem* F_N (*of size* N).

Lemma 1. Let $N = 2n$, ω a primitive Nth root of unity. Then the N Fourier points $[\omega]$ have Property S, with $\omega^{k+n} = -\omega^k$ $(k = 0, \ldots, n-1)$.

Proof. First we note that

$$(\omega^{k+n})^2 = (\omega^k)^2 \omega^N = (\omega^k)^2,$$

using only that ω is an Nth root of unity. But $x^2 = y^2$ in a field implies that $x = \pm y$. Since $\omega^{k+n} \neq \omega^k$ (otherwise we would have $\omega^n = 1$ contradicting the fact that ω is a *primitive* Nth root of unity), it must be the case that $\omega^{k+n} = -\omega^k$, as required. □

Consequence. We can apply the binary splitting scheme of Algorithm 1 to solve a Fourier evaluation problem F_N.

Now for the crucial additional property of Fourier points that will let us speed up the solution to the subproblems arising in step 2 of Algorithm 1.

Lemma 2. Let $N = 2n$, ω a primitive Nth root of unity. Then ω^2 is a primitive $(N/2)$th root of unity.

Proof. Since $(\omega^2)^n = \omega^N = 1$, ω^2 is an nth root of unity. But $(\omega^2)^j \neq 1$ for $0 < j < n$, otherwise we would have $\omega^k = 1$ with $0 < k < 2n = N$ in contradiction to ω being a *primitive* Nth root of unity. Thus ω^2 is a primitive nth root of unity, as required. □

Consequence. Provided $n = N/2$ is even, we can apply the binary splitting scheme of Algorithm 1 not only to solve a given Fourier evaluation problem F_N at N Fourier points $[\omega]$ *but also to solve the two subproblems* P_n *arising in step 2 of Algorithm 1.* For these two subproblems each involve the multipoint evaluation of a length n polynomial at the n points $\{1, \omega^2, \ldots, (\omega^2)^{n-1}\}$. By Lemma 2, these points are a set $[\omega^2]$ of n Fourier points. Hence the two subproblems P_n are in fact *Fourier* evaluation problems F_n, so that Lemma 1 and its consequence applies to these subproblems.

Moreover, the same argument applies to the sub-subproblems that arise, and so on, inductively, for as long as the number of evaluation points remains even. Thus, if we choose $N = 2^m$, then the binary splitting scheme can be carried out (through m levels of recursion) until a trivial problem is reached: the evaluation of a polynomial of length one at one point.

These considerations lead to the abstract recursive FFT procedure of Algorithm 2 for evaluating a polynomial of length N at N Fourier points. By "abstract" we mean that the procedure is operational over any field having the requisite $N = 2^m$th root of unity.

Algorithm 2 (FFT—*fast Fourier transform*).

Input arguments.
integer $N = 2^m$,
polynomial $a(x) = \sum_{i=0}^{N-1} a_i x^i$,
primitive Nth root of unity ω.

Output argument.
array $\mathbf{A} = (A_0, \ldots, A_{N-1})$ where $A_k = a(\omega^k)$.

Auxiliary data.
integer $n = N/2$,
polynomials $b(x) = \sum_{i=0}^{n-1} b_i x^i$, $c(x) = \sum_{i=0}^{n-1} c_i x^i$,
arrays $\mathbf{B} = (B_0, \ldots, B_{n-1})$, $\mathbf{C} = (C_0, \ldots, C_{n-1})$.

Procedure FFT is displayed in Fig. 1. (Of course the procedure assumes a data type corresponding to the field over which it is to operate.)

The following correctness proof for procedure FFT makes explicit the intuitive inductive argument that we used in its derivation.

```
procedure FFT(N, a(x), ω, A);
  if N = 1
    then
      {Basis.}  A_0 := a_0
    else
    begin
      {Binary split.}
        n := N/2;
        b(x) := Σ_{i=0}^{n-1} a_{2i} x^i;
        c(x) := Σ_{i=0}^{n-1} a_{2i+1} x^i;
      {Recursive calls.}
        FFT(n, b(x), ω^2, B);
        FFT(n, c(x), ω^2, C);
      {Combine.}
        for k := 0 until n - 1 do
        begin
          A_k     := B_k + ω^k × C_k;
          A_{k+n} := B_k - ω^k × C_k
        end
    end
```

Fig. 1 FFT procedure.

Theorem 3 (*Procedure FFT works*). If

$N = 2^m$,
$a(x) = \sum_{i=0}^{n-1} a_i x^i \in F[x]$ is a polynomial of length N,
ω is a primitive Nth root of unity over F,

then FFT$(N, a(x), \omega, \mathbf{A})$ returns $A_k = a(\omega^k)$ for $k = 0, \ldots, N-1$.

Proof. Let $p(m)$ be the assertion of Theorem 3. Then the correctness proof of FFT is tantamount to the proof of $p(m)$ for all $m \in \mathbf{N}$.

Basis ($m = 0$). For this case, $N = 1$, FFT returns $A_0 = a_0 = a(\omega^0)$, the latter equality holding because $a(x)$ is the constant polynomial a_0. Thus $p(0)$ holds.

Induction. Assume $p(m)$ where m is an arbitrary natural number. We now establish $p(m+1)$. If $N = 2^{m+1}$, then $N > 1$, and the "binary split" step of FFT gives $n, b(x), c(x)$ such that $n = N/2$ and

$$(*) \qquad a(x) = b(x^2) + xc(x^2).$$

By Lemma 2, ω^2 is a primitive nth root of unity. Moreover $b(x)$ and $c(x)$ are polynomials of length $n = 2^m$. Hence by the induction hypothesis, the "recursive calls" step of FFT returns

$$(**) \qquad B_k = b(\omega^{2k}), \quad C_k = c(\omega^{2k}) \qquad (k = 0, \ldots, n-1).$$

The "combine" step of FFT then yields, for $k = 0, \ldots, n-1$,

$$\begin{aligned} A_k &= B_k + \omega^k C_k \\ &= b((\omega^k)^2) + \omega^k c((\omega^k)^2) && \text{by } (**) \\ &= a(\omega^k) && \text{by } (*), \\ A_{k+n} &= B_k - \omega^k C_k \\ &= b(\omega^{2k}) - \omega^k c(\omega^{2k}) && \text{by } (**) \\ &= b((\omega^{k+n})^2) + \omega^{k+n} c((\omega^{k+n})^2) && \text{by Lemma 1} \\ &= a(\omega^{k+n}) && \text{by } (*). \end{aligned}$$

Thus FFT$(N = 2^{m+1}, a(x), \omega, \mathbf{A})$ returns $A_k = a(\omega^k)$ $(k = 0, \ldots, N-1)$, which establishes $p(m+1)$ and completes the proof by induction of the correctness of procedure FFT. □

We now show that our FFT is indeed a *fast* Fourier transform.

Theorem 4. Procedure FFT requires

$$M(N) = (N/2) \log_2 N$$

field multiplications to solve a Fourier evaluation problem F_N.

Proof. According to the "else" clause of procedure FFT, $M(N)$ satisfies

$$M(N) = 2M(N/2) + N/2$$

(the "$2M(N/2)$" term due to the two recursive calls, the "$N/2$" term due to the "combine" step), which for $N = 2^m$ becomes

$$M(2^m) = 2M(2^{m-1}) + 2^{m-1}.$$

Iterating this relationship m times gives

$$M(2^m) = m2^{m-1} + M(1)2^m.$$

But $M(1) = 0$, in accordance with the "then" clause (basis) of FFT. Thus we have $M(2^m) = m2^{m-1}$, or $M(N) = (N/2)\log_2 N$ as required. □

Take, for example, $N = 1000$. Then classical multipoint evaluation requires 10^6 multiplications, whereas the FFT requires only about 10^4 multiplications. In more global terms, the FFT has the pleasing quasilinear property that doubling the problem size roughly doubles the computation time (more precisely, $M(2N)/M(N) \to 2$ for large N). This is in contrast with the classical $O(N^2)$ algorithm, which has the distressing property that doubling the problem size quadruples the required computation time.

So our FFT procedure is indeed fast. The factor of two speedup of the binary splitting scheme (Proposition 2 vs. Proposition 1) has been enjoyed at every level of the FFT's recursion, resulting in an asymptotic speedup from

$O(N^2)$ to $O(N \log N)$. This, then, is what the *fast* Fourier transform is all about (Note 1).

We now turn to the companion problem of *interpolation* with respect to Fourier points.

1.2 The Inverse Transform: Fast Interpolation

Let $\alpha_0, \ldots, \alpha_{N-1}$ be N points in a field F, to be used for evaluation and interpolation. Our size N multipoint evaluation problem (with respect to these points) is the problem of computing, for a given polynomial $a(x) = \sum_{i=0}^{N-1} a_i x^i \in F[x]$, the values $b_k = a(\alpha_k)$ $(k = 0, \ldots, N-1)$. Our size N interpolation problem, on the other hand, is the inverse problem of computing, for given $b_k \in F$ $(k = 0, \ldots, N-1)$, the coefficients of the (unique) interpolating polynomial $a(x) = \sum_{i=0}^{N-1} a_i x^i$ which satisfies $b_k = a(\alpha_k)$. In brief: with respect to the relationship $b_k = a(\alpha_k)$ $(k = 0, \ldots, N-1)$, evaluation determines the b_k's from the a_i's, interpolation determines the a_i's from the b_k's.

This inverse relationship between (multipoint) evaluation and interpolation becomes especially transparent when examined in matrix terms. To this end we introduce the $N \times N$ *Vandermonde matrix* $V(\alpha_0, \ldots, \alpha_{N-1})$ *associated with* $\alpha_0, \ldots, \alpha_{N-1}$:

$$V(\alpha_0, \ldots, \alpha_{N-1}) = \begin{bmatrix} 1 & \alpha_0 & \alpha_0^2 & \cdots & \alpha_0^{N-1} \\ 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{N-1} \\ \vdots & & & & \\ 1 & \alpha_{N-1} & \alpha_{N-1}^2 & \cdots & \alpha_{N-1}^{N-1} \end{bmatrix}.$$

Let $\mathbf{a} = (a_0, \ldots, a_{N-1})$, $\mathbf{b} = (b_0, \ldots, b_{N-1})$. The definition of matrix-vector multiplication immediately gives

Proposition 1. For $a(x) = \sum_{i=0}^{N-1} a_i x^i$ and $V = V(\alpha_0, \ldots, \alpha_{N-1})$,

$$V\mathbf{a} = \mathbf{b} \quad \Leftrightarrow \quad b_k = a(\alpha_k).$$

In Proposition 1, interpolation theory guarantees that if the α_k's are distinct then the coefficients a_i of $a(x)$ can be uniquely determined from the b_k's, which by Proposition 1 is to say that $V\mathbf{a} = \mathbf{b}$ can be solved uniquely for $\mathbf{a}$. But by elementary matrix theory this means that V is nonsingular (invertible). Thus we have shown that the Vandermonde matrix $V(\alpha_0, \ldots, \alpha_{N-1})$ for distinct α_k's is nonsingular, which allows us to embellish Proposition 1 to

Proposition 1′. For $a(x) = \sum_{i=0}^{N-1} a_i x^i$ and $V = V(\alpha_0, \ldots, \alpha_{N-1})$ (α_k's distinct),

$$V\mathbf{a} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{a} = V^{-1}\mathbf{b} \quad \Leftrightarrow \quad b_k = a(\alpha_k).$$

Example 1. In $\mathbf{Z}_7$, let $\alpha_0 = 5$, $\alpha_1 = 2$, $\alpha_2 = 3$, and let $a(x) = 2 + 6x + x^2$. Then

with $V = V(5, 2, 3)$ we have

$$V\mathbf{a} = \begin{bmatrix} 1 & 5 & 5^2 = 4 \\ 1 & 2 & 2^2 = 4 \\ 1 & 3 & 3^2 = 2 \end{bmatrix} \begin{bmatrix} 2 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} a(5) \\ a(2) \\ a(3) \end{bmatrix}.$$

And if $b_0 = 1$, $b_1 = 4$, $b_2 = 1$, then the solution to $V\mathbf{a} = \mathbf{b}$ is $a_0 = 2$, $a_1 = 6$, $a_2 = 1$ —the coefficients of $a(x)$ satisfying $a(\alpha_k) = b_k$ $(k = 0, 1, 2)$. □

So, in the light of Proposition 1′ we conclude:

(1) Multipoint evaluation (the forward transform) corresponds to a matrix product of the form $V\mathbf{a}$ where V is a Vandermonde matrix. Thus the FFT can be regarded as a fast algorithm for computing $V\mathbf{a}$ when V is a Vandermonde matrix $V(1, \omega, \ldots, \omega^{N-1})$ associated with a set of N Fourier points—$O(N \log N)$ for $V(1, \omega, \ldots, \omega^{N-1})$ vs. $O(N^2)$ for an arbitrary Vandermonde matrix.

(2) Interpolation (the inverse transform) corresponds to a matrix product of the form $V^{-1}\mathbf{b}$ where V is a Vandermonde matrix. Thus Newton's Interpolation Algorithm can be regarded as a fast algorithm for solving a system of linear equations $V\mathbf{a} = \mathbf{b}$ for $\mathbf{a} = V^{-1}\mathbf{b}$ when V is a Vandermonde matrix (associated with any set of distinct points, not necessarily Fourier points)—$O(N^2)$ in the Vandermonde case versus $O(N^3)$ for an arbitrary linear system.

But (2) is not what we are after (noteworthy though it might be). We want a *faster* than $O(N^2)$ interpolation algorithm, which exploits the case where V is a Vandermonde matrix associated with Fourier points.

Notation. If ω is a primitive Nth root of unity, then we write $V([\omega])$ for $V(1, \omega, \ldots, \omega^{N-1})$.

Theorem 2. Let ω be a primitive Nth root of unity in a field F in which N^{-1} $[= (N \cdot 1)^{-1}]$ exists. Then

$$V([\omega])^{-1} = N^{-1} V([\omega^{-1}]).$$

Proof. First, it is trivially shown (do it) that ω^{-1}, like ω, is a primitive Nth root of unity. Thus $V([\omega^{-1}])$ denotes the $(N \times N)$ Vandermonde matrix $V(1, \omega^{-1}, \ldots, (\omega^{-1})^{N-1})$. If we show that

$$V([\omega])V([\omega^{-1}]) = NI = \begin{bmatrix} N & & 0 \\ & \ddots & \\ 0 & & N \end{bmatrix},$$

then we are done.

So let $W = V([\omega])V([\omega^{-1}])$. Then

$$w_{ij} = \sum_{k=0}^{N-1} \omega^{ik}\omega^{-kj} \qquad (0 \le i, j < N).$$

CASE 1. $i=j$. Then $w_{ii}=\sum_{k=0}^{N-1}1=N$.

CASE 2. $i\neq j$. Then $w_{ij}=\sum_{k=0}^{N-1}(\omega^{i-j})^k$. Since $0<|i-j|<N$, it follows that $\omega^{i-j}\neq 1$, otherwise we would have a contradiction to $o(\omega)=N$. Hence we can apply the identity $\sum_{k=0}^{N-1}x^k=(x^N-1)/(x-1)$ $(x\neq 1)$ to obtain

$$w_{ij}=\frac{(\omega^{i-j})^N-1}{\omega^{i-j}-1}=\frac{(\omega^N)^{i-j}-1}{\omega^{i-j}-1}=0. \ \square$$

Example 2. In $\mathbf{Z}_{13}$, 8 is a primitive 4th root of unity, with $8^{-1}=5$. Here we have

$$V([8])=\begin{bmatrix}1&1&1&1\\1&8&12&5\\1&12&1&12\\1&5&12&8\end{bmatrix},\qquad V([5])=\begin{bmatrix}1&1&1&1\\1&5&12&8\\1&12&1&12\\1&8&12&5\end{bmatrix},$$

and

$$V([8])V([5])=\begin{bmatrix}4&0&0&0\\0&4&0&0\\0&0&4&0\\0&0&0&4\end{bmatrix},$$

in accordance with Theorem 2. $\square$

Finally, let us consider the "Fourier interpolation" problem of size N, that of computing the (unique) interpolating polynomial $a(x)=\sum_{i=0}^{N-1}a_ix^i$ that satisfies $a(\omega^k)=b_k$ for arbitrarily specified b_k $(k=0,\ldots,N-1)$, where $[\omega]$ is a set of N Fourier points.

Denote $V([\omega])$ by V and $V([\omega^{-1}])$ by V'. The coefficients a_i of $a(x)$ are then given by

$$\begin{aligned}\mathbf{a}&=V^{-1}\mathbf{b} && \text{Proposition 1'}\\ &=N^{-1}V'\mathbf{b} && \text{Theorem 2.}\end{aligned}$$

Let $\mathbf{c}=V'\mathbf{b}$. Since $V'=V([\omega^{-1}])$ is a Vandermonde matrix, Proposition 1 tells us that c_k is the value of $b(x)=\sum_{i=0}^{N-1}b_ix^i$ at $x=(\omega^{-1})^k$, which is to say that the c_k's can be computed by *Fourier evaluation* of $b(x)$ at the N Fourier points $[\omega^{-1}]$. For this latter problem we have an efficient algorithm—the FFT. Thus the problem of Fourier interpolation can be essentially solved by Fourier evaluation in accordance with

Algorithm 1 (*Fast Fourier interpolation—FFI*).

Input.
integer $N=2^m$,
primitive Nth root of unity ω,
sample values $\mathbf{b}=(b_0,\ldots,b_{N-1})$.

Output. $a(x)=\sum_{i=0}^{N-1}a_ix^i$ where $a(\omega^k)=b_k$ $(k=0,\ldots,N-1)$.

The following procedure uses the FFT procedure of Algorithm 2, Section 1.1.

procedure FFI(N, **b**, ω, $a(x)$);
begin
1. $b(x):=\sum_{i=0}^{N-1}b_ix^i$;
2. FFT(N, $b(x)$, ω^{-1}, **c**);
3. $a(x):=\sum_{i=0}^{N-1}(N^{-1}c_i)x^i$
end $\square$

Clearly the above Fourier interpolation algorithm operates in FFT time $O(N\log N)$—i.e., it really is a *fast* Fourier interpolation algorithm as advertised. This we record as the companion to Theorem 4 of Section 1.1.

Theorem 3. The solution to a Fourier interpolation problem of size N can be computed in time $O(N\log N)$ (assuming $O(1)$ time for operations over the underlying field).

Example 3. Let us use Algorithm 1 to compute the solution $a(x)=\sum_{i=0}^{N-1}a_ix^i$ to the following $N=4$ point Fourier interpolation problem over $\mathbf{Z}_{13}$ with respect to the Fourier points $[\omega=8]$.

k	$\alpha_k=8^k$	$b_k=a(\alpha_k)$
0	1	7
1	8	5
2	12	10
3	5	12

Line 1 of Algorithm 1 forms the polynomial $b(x)=7+5x+10x^2+12x^3$. Line 2 evaluates $b(x)$ via the FFT at the Fourier points $[\omega^{-1}=5]$, returning $c_0=b(1)=8$, $c_1=b(5)=1$, $c_2=b(12)=0$, $c_3=b(8)=6$. Line 3 then multiplies these c_i's by $4^{-1}=10$, returning $a(x)=2+10x+8x^3$ as the required interpolating polynomial. $\square$

1.3 Feasibility of mod *p* FFTs

To compute the solution of a Fourier evaluation-interpolation problem of size $N=2^m$ over a field F, the FFT requires that F have a primitive Nth root of unity ω. Over the complex field (the field of traditional "analytic" applications —see Note 1), this requirement can always be satisfied: for any N, $e^{2\pi i/N}$ is a primitive Nth root of unity in $\mathbf{C}$. Over finite (modular) fields $\mathbf{Z}_p$, on the other hand, the situation is not nearly so cut and dry, but as we intend to show, it is still quite favorable: mod p FFTs are indeed computationally feasible. In applications to algebraic computing we are interested in primes p near computer wordsize, with a view to using the Chinese remainder and evaluation-interpolation techniques of Chapter VIII. Thus we are interested in very large primes, say on the order of 10^9.

This section, then, is devoted to answering two questions in the affirmative:

1. Do fields $\mathbf{Z}_p$ exist (p very large) having primitive $N = 2^m$th roots of unity (m in the 10–20 range, say)? Indeed, do such fields exist in abundance?

2. Given a field $\mathbf{Z}_p$ having a primitive Nth root of unity, can we find it efficiently (keeping in mind that $\mathbf{Z}_p$ may have billions of elements)?

A key result towards answering both of these questions is

Theorem 1. $\mathbf{Z}_p$ has a primitive Nth root of unity if and only if $N | (p-1)$.

Proof. By Lagrange's Theorem, the order of a group element divides the order of the group (as proved explicitly in Corollary 2 of Theorem 3, Section III.3.4). Since $\mathbf{Z}_p^*$ has order $p-1$, we obtain the "only if" direction.

As for the "if" direction, let $N | (p-1)$. Now $\mathbf{Z}_p$ contains a primitive element (Theorem 2 of Section VI.2.1), call it α. It is trivially verified (do it) that

$$\beta = \alpha^{(p-1)/N}$$

has order N in $\mathbf{Z}_p^*$, making β a primitive Nth root of unity. □

Thus to compute mod p FFTs of size $N = 2^m$, we require primes p such that $2^m | (p-1)$, i.e., primes of the form $p = 2^e k + 1$ ($e \geq m$). We call a prime p of the form

$$p = 2^e k + 1 \qquad (k \text{ odd})$$

a *Fourier prime having (binary) exponent* e. Evidently any such prime can be used to compute FFTs of size $N = 2^m$ for $m \leq e$.

The assurance that Fourier primes exist in abundance rests on a result from analytic number theory (Note 2):

Generalized Prime Number Theorem (Cf. the "ordinary" Prime Number Theorem of Appendix 1 to Section VIII.3). Let integers a and b be relatively prime. The number of primes $\leq x$ in the arithmetic progression $ak + b$ ($k = 1, 2, \ldots$) is approximately (and somewhat greater than)

$$(x/\log x)/\phi(a) \qquad (\phi = \text{Euler's phi function}).$$

Consequence. The number of Fourier primes $p = 2^f k + 1 \leq x$ is approximately

$$(*) \qquad (x/\log x)/2^{f-1}.$$

This, then, is our assurance that Fourier primes exist in reasonable abundance.

Example 1. Let us take $x = 2^{31}$ (corresponding to the wordsize of a certain popular class of machines) and $f = 20$ in $(*)$. We conclude that there are approximately 180 Fourier primes $p = 2^e k + 1$ (k odd) with exponent $e \geq f = 20$. Any such Fourier prime could be used to compute FFTs of size 2^{20} (very large transforms indeed). □

Now for the second question that we posed: Can we efficiently determine primitive Nth roots of unity in a very large finite field $\mathbf{Z}_p$?

In the proof of Theorem 1, we noted that if α is primitive in $\mathbf{Z}_p$, then $\beta = \alpha^{(p-1)/N}$ is a primitive Nth root of unity. Even if the exponent $(p-1)/N$ is very large (as it may well be), the powering of α can still be efficiently carried out (in log exponent time) using the fast algorithm developed in Example 2 of the Appendix to Chapter VII. So our problem of finding primitive Nth roots of unity boils down to that of *finding a primitive element in large finite fields* $\mathbf{Z}_p$.

What proportion of elements in $\mathbf{Z}_p$ are primitive? (Are we looking for a needle in a haystack?) From finite field theory (Theorem 3 of Section VI.2.1) the number of primitive elements in $\mathbf{Z}_p$ is given by $\phi(p-1)$ (ϕ = Euler's phi function). Analytic number theory tells us that the average value of $\phi(n)$ over all integers n is $6n/\pi^2$; it is readily argued that its average value over even integers is greater than $3n/\pi^2$ (Note 3). We conclude, therefore, that primitive elements abound in finite fields; on the average (over p) we would expect better than three of every π^2 elements to be primitive, or, in probabilistic terms, we would expect an element drawn at random from $\mathbf{Z}_p$ to be primitive with probability greater than $3/\pi^2 \approx 0.3$.

Thus we propose to find a primitive element of $\mathbf{Z}_p$ by testing the elements $2, 3, \ldots$ in turn until a primitive one is found, armed with the above probabilistic assurance that even for very large values of p we should not have to test many elements.

But how should we test an element α of $\mathbf{Z}_p$ for primitivity? We momentarily consider the obvious method of raising α to successive powers, checking whether or not $\alpha^n \neq 1$ for $1 < n < p-1$. Thankfully enough, the need for carrying out this terribly inefficient test is obviated by the following elegant and completely algebraic result:

Theorem 2. $\alpha \in \mathbf{Z}_p$ is primitive $\Leftrightarrow$ [$\alpha^{(p-1)/q} \neq 1$ in $\mathbf{Z}_p$ for any prime factor q of $p-1$].

Proof. ($\Rightarrow$) Trivial.

($\Leftarrow$) Assume α has order $n < p-1$. Then n divides $p-1$ by Lagrange, so that $p-1 = kn$. Let $k = qr$, where q is a prime in k's prime factorization. But then $p-1 = qrn$, so that q is also a prime in $(p-1)$'s (unique) prime factorization. We then have

$$\alpha^{(p-1)/q} = \alpha^{rn} = (\alpha^n)^r = 1. \quad \square$$

Example 2. Let us use Theorem 2 to find a primitive element in $\mathbf{Z}_{41}$. Here we have $p - 1 = 40 = 2^3 5$, so that $\alpha \in \mathbf{Z}_{41}$ is primitive if and only if $\alpha^{20}, \alpha^8 \neq 1$.

Computing over $\mathbf{Z}_{41}$, we see that $2^{20} = 1$, $3^8 = 1$, $4^{20} = (2^{20})^2 = 1$, $5^{20} = 1$, so none of $2, 3, 4, 5$ are primitive. But $6^8 = 10$ and $6^{20} = 40$, so that 6 is the smallest primitive element of $\mathbf{Z}_{41}$ ("smallest" in the usual integer ordering sense, of course). □

We propose the following algorithm, based on Theorem 2, for computing a list of Fourier primes.

Algorithm 1 (*Computing a list of Fourier primes*).

Input.
List L of primes $\leq W$ (where W is typically our computer's wordsize);
Auxiliary list L' of primes $\leq \sqrt{W}$ (if the sieve method of Appendix 1 to Section VIII.3 is used to compute L, then this list L' is already available);
Positive integer f.

Output.
List of Fourier primes of the form $p = 2^e k + 1$ (k odd) with $e \geq f$, together with a primitive element of $\mathbf{Z}_p$.

For each prime p in L:

Step 1 {Is p a Fourier prime of required exponent?}
Determine the exponent e of 2 in the prime factorization of $p-1$. If $e < f$, then discard p.

Step 2 {Prime power factorization of $p-1$}.
Determine the distinct primes a_i in the prime power factorization of $p-1$,

$$p - 1 = a_1^{e_1} a_2^{e_2} \cdots a_r^{e_r},$$

by cancelling out powers of primes in the auxiliary table L' (Note 4).

Step 3 {Find a primitive element}.
Test $\alpha = 2, 3, \ldots$ for primitivity, using Theorem 2: α is primitive $\Leftrightarrow$ $\alpha^{(p-1)/a_i} \neq 1$ for all a_i. (Use fast powering for the computation $\alpha^{(p-1)/a_i}$.) □

In Fig. 2 are presented the results of executing Algorithm 1 to find the 10 largest Fourier primes $\leq 2^{31} - 1$ having exponent $e \geq f = 20$. (Such Fourier primes could be used to compute FFTs of size $2^{20} > 10^6$ using a 32 bit word (31 bits + sign) machine. Perhaps we should call these primes "IBM Fourier primes.")

p	e	α = least primitive element of $\mathbf{Z}_p$
2130706433	24	3
2114977793	20	3
2113929217	25	5
2099249153	21	3
2095054849	21	11
2088763393	23	5
2077229057	20	3
2070937601	20	6
2047868929	20	13
2035286017	20	10

Fig. 2. Fourier primes $p = 2^e k + 1 \leq 2^{31} - 1$ ($e \geq 20$).

From the results of this section, we conclude that mod p FFTs *are* computationally viable: Fourier primes—primes p for which FFTs over $\mathbf{Z}_p$ are possible—exist in plenty. Moreover, we have a reasonably efficient algorithm (Algorithm 1) for determining such primes along with primitive elements in the associated fields. From these primitive elements, the required primitive roots of unity can be efficiently computed.

2. FAST ALGORITHMS FOR MULTIPLYING POLYNOMIALS AND INTEGERS

With the FFT in hand, it is an easy matter to give a faster than $O(n^2)$ algorithm for multiplying degree n polynomials (faster = faster in an asymptotic sense, i.e., for sufficiently large problems). With a little more effort, we can derive a faster than $O(n^2)$ algorithm for multiplying n digit integers. In view of the venerable character of the classical $O(n^2)$ integer and polynomial multiplication algorithms ("school algorithms"), perhaps the reader will agree that the new FFT-based algorithms deserve to be called "surprisingly fast." In any case, if the results of this section make the reader question the tried and true (at least in algorithm design), then fine.

2.1 Fast Polynomial Multiplication

We return to our polynomial multiplication algorithm by evaluation-interpolation (Example 1 of Section VIII.3.2). There we failed to do better than $O(n^2)$ time due to the time required by classical multipoint evaluation and interpolation. But now we have the FFT.

Algorithm 1 (*FFT multiplication over* $F[x]$).

Input. Polynomials $a(x), b(x) \in F[x]$ having degrees $\leq n$.
Output. $c(x) = a(x)b(x)$.

Choose $N = 2^m$ to be $> 2n$. Let ω be a primitive Nth root of unity in F. (The following algorithm requires the existence of such an element ω.)

Step 1 {Evaluation via FFT}.
(a) Invoke FFT($N, a(x), \omega, \mathbf{A}$),
FFT($N, b(x), \omega, \mathbf{B}$);
(b) Compute $C_k = A_k B_k$ ($k = 0, \ldots, N-1$).

Step 2 {Fourier interpolation—Algorithm 1 of Section 1.2}.
Invoke FFI($N, \mathbf{C}, \omega^{-1}, U(x)$).

Step 3.
Return $c(x) = U(x)$. □

Since ω has been chosen to be a primitive $N = 2^m$th root of unity, the use of procedures FFT and FFI for solving the N-point evaluation and interpolation problem of steps 1(a) and 2 is valid. The overall validity of Algorithm 1—that $c(x)$ returned in step 3 is in fact $a(x)b(x)$— is then a consequence of our choice of N to be $> 2n$, appealing only to the validity of the general evaluation-interpolation scheme for polynomial multiplications over $F[x]$ (Example 1 of Section VIII.3.2).

The computing time required by Algorithm 1 is clearly dominated by the FFT steps 1(a) and 2, hence is $O(N \log N)$, assuming as always that operations over F require $O(1)$ time. If N is chosen to be the least power of 2 that is $> 2n$, then N is $\leq 4n$. Thus in terms of the original size parameter n, the computing time required by Algorithm 1 is $O(4n \log 4n) = O(n \log n)$. This we record as

Theorem 1. Multiplication of two polynomials of degree n over $F[x]$ can be carried out in time $O(n \log n)$ (provided F has the requisite primitive root of unity).

Thus over the complex field Theorem 1 holds unconditionally, since **C** contains primitive Nth roots of unity for any N. Over subfields of **C**, notably **R** and **Q**, Theorem 1 does not hold, strictly speaking, unless we are willing to perform Algorithm 1 over **C**. For example, FFTs involving real polynomials generate complex values, because the Fourier points are necessarily complex. Over finite fields $\mathbf{Z}_p$, Theorem 1 holds provided p is a Fourier prime $p = 2^e k + 1$ with $2^e > 2n$. (In applications of finite fields, we would of course restrict our attention to Fourier primes of high exponents as discussed in Section 1.3.)

Some concluding remarks about the relative speeds of FFT-based and classical multiplication algorithms as a function of the possibly different degrees of the operands. If $a(x)$ and $b(x)$ have degrees m and n, with $m \leq n$ say, then our FFT-based multiplication algorithm requires $O(n \log n)$ time while classical multiplication requires $O(mn)$ time. For $m \approx n$, the FFT-based algorithm is clearly at its best relative to the classical algorithm: $O(n \log n)$ versus $O(n^2)$. For $m \ll n$, on the other hand, the performance of the FFT-based algorithm becomes relatively poor due to the fact that it cannot effectively exploit the smallness of m relative to n. As an extreme case in point, let $m = 1$. Then the FFT-based algorithm is still $O(n \log n)$, whereas the classical algorithm becomes $O(n)$.

Out of this simple analysis emerges the following *balancing principle* which transcends the details of successful applications of fast (FFT-based) polynomial multiplication: polynomial multiplication, wherever it occurs, should involve polynomials of roughly the same size (degree). We shall encounter this principle at work in Section 3.

2.2 Fast Integer Multiplication

We have already exploited the observation that the conventional positional notation for integers is essentially a polynomial-based representation (see Example 4 of Section VIII.1.1): If $a = (a_{n-1} \cdots a_0)_B$ is a base B integer, then a represents the value of its *associated polynomial* $a(x) = \sum_{i=0}^{n-1} a_i x^i$ at $x = B$; i.e., $a = a(B)$. In this section we exploit that same observation, this time to achieve an integer multiplication algorithm that works in faster than the $O(n^2)$ time required by the venerable school algorithm.

Let us consider a sample product $c = ab$ of decimal ($B = 10$) integers:

$$\begin{aligned} a &= 329 \\ b &= 617 \\ \hline c &= 202993. \end{aligned}$$

This result can also be achieved by the polynomial multiplication $c(x) = a(x)b(x)$ followed by the evaluation $c(10)$. For $c(10) = a(10)b(10)$, because polynomial evaluation (in this case at $x = 10$) is a morphism for multiplication. Thus $c = c(10)$ yields the product $a(10)b(10) = ab$. To illustrate with $a = 329$, $b = 617$:

$$\begin{aligned} a(x) &= 3x^2 + 2x + 9 \\ b(x) &= 6x^2 + x + 7 \\ \hline c(x) &= 18x^4 + 15x^3 + 77x^2 + 23x + 63. \end{aligned}$$

Then $c(10) = 202993 = 329 \times 617$.

Now for a computer implementation of this polynomial multiplication-evaluation algorithm for multiplying two n-digit base B integers $a = (a_{n-1} \ldots a_0)_B$ and $b = (b_{n-1} \ldots b_0)_B$, where B is chosen to be $< W =$ the wordsize of our computer. An especially convenient choice for B, assuming that the multiplicands are presented to the computer as decimal integers, is the largest power of ten $< W$. Then the base B digits of the multiplicands are obtained by simply grouping consecutive decimal digits (e.g., the base 10^3 digits of 46709834 are 046, 709, 834); moreover these base B digits, which constitute the coefficients of the associated polynomials $[a = (a_{n-1} \ldots a_0) \leftrightarrow \sum_{i=0}^{n-1} a_i x^i]$, are all single-precision.

We now propose to compute the polynomial product $c(x) = a(x)b(x)$ by our polynomial MHI scheme of Section VIII.3.3; i.e., we propose to compute $c^{(k)}(x) = a(x)b(x) \bmod p_k$ for a sufficient number K of large, but less than wordsize, primes p_k ($B \leq p_k \leq W$) in order to obtain $c(x)$ by the CRA. Choosing these p_k's to be Fourier primes of the form $p = 2^e l + 1$ for sufficiently large exponent e, we can use FFT-based polynomial multiplication to compute the $c^{(k)}(x)$'s. This, then, is the crux of our proposed algorithm: the replacement of the $O(n^2)$ base B digit calculations of classical long multiplication by a number (K) of *fast* polynomial multiplications.

For reasons that will become clear in the analysis phase of our development, we call the resulting algorithm the "three primes" algorithm.

Algorithm 1 (*"Three primes" algorithm for integer multiplication*).

Input. $a = (a_{n-1} \dots a_0)_B$, $b = (b_{n-1} \dots b_0)_B$.
Output. $c = ab$.

The algorithm requires K Fourier primes $p = 2^e l + 1 \leq W$ with sufficiently large exponent e (K to be determined).

Step 1 {Multiplication of associated polynomials $a(x) = \sum_{i=0}^{n-1} a_i x^i$, $b(x) = \sum_{i=0}^{n-1} b_i x^i$ by the MHI scheme for $\mathbf{Z}[x]$ (Algorithm 1 of Section VIII.3.3)}.

1.1 For K Fourier primes p_k ($B \leq p_k \leq W$): compute $c^{(k)}(x) = a(x)b(x)$ over $\mathbf{Z}_{p_k}[x]$ using FFT-based polynomial multiplication.

1.2 Solve the polynomial CRP

$$u(x) \equiv c^{(k)}(x) \pmod{p_k} \qquad (k = 0, \dots, K-1)$$

for the least-positive coefficient solution $U(x)$.

1.3 Return $c(x) = U(x)$.

Step 2 {Evaluation at radix}. Return $c = c(B)$. □

The algorithm and its analysis hinge on the determination of K (how big must K be?). Each coefficient $c_k = \sum_{i+j=k} a_i b_j$ of $c(x)$ is seen to be $< nB^2$. Hence step 1 correctly computes $c(x)$ provided

$$(*) \qquad p_0 p_1 \cdots p_{K-1} \geq nB^2.$$

Since each p_k is $\geq B$, $(*)$ is satisfied provided B^K is $\geq nB^2$, i.e., provided K is $\geq (\log_B n) + 2$. *Therefore over the range $n \leq B$, three primes are sufficient* (thus the name "three primes" algorithm).

Choosing $K = 3$, we now have a tacit restriction on the size of problem our algorithm can handle: n must be $\leq B$. But keeping in mind that B is a huge integer, on the order of 10^9 say, we can regard this restriction as being totally innocuous.

We have one other restriction on the size of problem our algorithm can handle. Step 1.1 requires three Fourier primes $p = 2^e l + 1$ ($B \leq p \leq W$) with $2^e \geq$ length of $c^{(k)}(x) = 2n - 1$ (recall: length = degree + 1). Thus if E is the largest integer for which we can find three Fourier primes each having exponent $e \geq E$, then n must satisfy $2n - 1 \leq 2^E$, which will be the case if n is $\leq 2^{E-1}$.

In summary, our algorithm imposes two constraints on n: $n \leq B$ and $n \leq 2^{E-1}$.

As for the computing time analysis of our algorithm, step 1.1 requires $O(n \log n)$ time, while step 1.2 requires $O(n)$ time ($2n - 1$ CRPs each involving three integer congruences). Thus step 1 requires $O(n \log n)$ time overall. We leave it as a not too difficult exercise to show that the polynomial evaluation of step 2 requires $O(n \log n)$ time (Exercise 4). This gives the following

Theorem 1. Let our computer have (fixed) wordsize W, so that mod p operations for $p \leq W$ can be carried out in $O(1)$ time. Then this computer can multiply two n digit base B integers, $B < W$, in $O(n \log n)$ time provided:

1. $n \leq B$;
2. $n \leq 2^{E-1}$ where three Fourier primes $p = 2^e l + 1$ ($B \leq p \leq W$) can be found with $e \geq E$, E the largest such integer.

Algorithm 1 thus has a curious *subasymptotic* property: Although it multiplies n-digit numbers in $O(n \log n)$ time (versus the $O(n^2)$ time required by the classical method), it is operational only over a finite range of n. Of course this range can in principle be extended by increasing W, the computer wordsize. But W cannot be extended indefinitely with increasing n, for then the assumption that mod p operations can be carried out in $O(1)$ time—time bounded by a constant independent of n—would become untenable. This, then, is why Algorithm 1 is *sub*asymptotic in character.

With such an algorithm it is obligatory to ask what kind of range of application it has. After all, if the algorithm turns out to be applicable only for small n (say $n \leq 100$), then it would have to be dismissed as uninteresting; table lookup would provide a much better subasymptotic algorithm. But as we now show, Algorithm 1 is "practically asymptotic"; for all intents and purposes its range of application for n is effectively infinite.

To establish this result we confine our attention to a specific "typical" wordsize, namely $W = 2^{31} - 1$ (corresponding to the same 31 bit + sign wordsize that we used for illustrative purposes in Section 1.3).

With $W = 2^{31} - 1$, we choose $B = 10^9$, the largest power of ten $< 2^{31} - 1$. The "$n \leq B$" constraint means that in Algorithm 1, n must be $\leq 10^9$.

From Fig. 3 we see that there are three Fourier primes $B \leq p \leq W$ having exponents ≥ 24. Thus E in the "$n \leq 2^{E-1}$" constraint can be taken as 24 (the largest possible value, as it turns out, for this particular word size), which means that n must be $\leq 2^{23} \approx 8.38 \times 10^6$. The latter, therefore, is the determining constraint.

$p = 2^e k + 1$ (k odd)	e	α = least primitive element of $\mathbf{Z}_p$
2013265921	27	31
2113929217	25	5
2130706433	24	3

Fig. 3 Three Fourier primes having exponent ≥ 24 (for a 32 bit word computer).

Thus Algorithm 1, *employed on a 32 bit word machine, is capable of multiplying integers having in excess of eight million decimal digits.* (Perhaps the reader concurs that Algorithm 1 is indeed "practically asymptotic"!)

In Note 1 we have gathered together some mainly historical remarks about fast integer multiplication algorithms.

3. FAST ALGORITHMS FOR MANIPULATING FORMAL POWER SERIES

Although the FFT will be vital to our achieving fast power series algorithms, it is Newton's method—that most venerable of numerical algorithms—that is the highlight of this concluding section.

3.1 Truncated Power Series Revisited

A (formal) power series $a(t) = \sum_{i=0}^{\infty} a_i t^i$ is in general an infinite mathematical object (because there are infinitely many coefficients a_i), unlike a polynomial or an integer, but very much like the infinite decimal expansion of a real number. For computational purposes it is usually both desirable and necessary to represent a power series $a(t) = \sum_{i=0}^{\infty} a_i t^i$ by its first (say) n terms $\sum_{i=0}^{n-1} a_i t^i$. This is the *truncated* power series $T_n[a(t)] = a(t) \bmod t^n$ introduced in Section V.2.2. (For convenience we are now writing $T_n[a(t)]$ rather than $T_n(a(t))$.) We regard $T_n[a(t)]$ as a $\bmod t^n$ *approximation* to $a(t)$ in that $T_n[a(t)]$ agrees with $a(t)$ in its first n terms, much as we regard 3.14159 as a six figure approximation to π.

Let us now review the truncated power series ring $F[[t]]_n$ (Example 3 of Section V.2.2) from the viewpoint of the complexity of its operations. We make the usual assumption that operations over the coefficient field F require $O(1)$ time.

Let $a(t), b(t) \in F[[t]]_n$. Then

$$(1) \qquad a(t) \oplus b(t) = T_n[a(t) + b(t)] = a(t) + b(t),$$

$$(2) \qquad a(t) \odot b(t) = T_n[a(t)b(t)],$$

where the right-hand side operations $+$, $\cdot$ are polynomial operations. As for inversion over $F[[t]]_n$, $a(t)^{\ominus 1}$, we have the general ring morphism result $\phi(a^{-1}) = \phi(a)^{\ominus 1}$ (Proposition 7 of Section IV.2.1). Here the morphism is $T_n : F[[t]] \to F[[t]]_n$, so we have

$$(3) \qquad a(t)^{\ominus 1} = T_n[a(t)^{-1}],$$

which states: to compute $a(t)^{\ominus 1}$, compute the first n terms of $a(t)^{-1}$ over $F[[t]]$ (for example, using the algorithm of Theorem 4, Section IV.3.1).

From (1), (2), and (3) we immediately conclude

Proposition 1. Over $F[[t]]_n$:

(1) $a(t) \oplus b(t)$ can be computed in $O(n)$ time;

(2) $a(t) \odot b(t)$ can be computed
  (i) in time $O(n^2)$ using classical polynomial multiplication,
  (ii) in time $O(n \log n)$ using FFT polynomial multiplication (provided F supports the FFT);

(3) $a(t)^{\ominus 1}$ can be computed in $O(n^2)$ time.

Now suppose it is desired to compute $T_n[a(t) + b(t)]$, $T_n[a(t)b(t)]$, $T_n[a(t)^{-1}]$ for specified power series $a(t)$, $b(t) \in F[[t]]$—call these operations T_n-addition, T_n-multiplication, and T_n-inversion. Since T_n is a morphism $F[[t]] \to F[[t]]_n$, we have

$$T_n[a(t) + b(t)] = T_n[a(t)] \oplus T_n[b(t)],$$

$$T_n[a(t)b(t)] = T_n[a(t)] \odot T_n[b(t)],$$

$$T_n[a(t)^{-1}] = T_n[a(t)]^{\ominus 1}.$$

These equations tell us that to compute T_n-sums, products, or inverses, the power series operands need only be specified $\bmod t^n$, and these operations can then be interpreted over $F[[t]]_n$; i.e., $\bmod t^n$. Henceforth we do not distinguish between $\bmod t^n$ operations and T_n-operations.

* * *

The gap between multiplication time and inversion time begs the question, can we find a faster inversion algorithm? For an affirmative answer to this question we look to numerical computing.

3.2 Fast Power Series Inversion; Newton's Method

Our objective is to derive a fast algorithm for power series inversion. *Newton's method*, of numerical computing fame, provides just the tool we need. (This section, together with the next, might well have been entitled "Newton's method: a great *algebraic* algorithm"—Note 1.)

Let us briefly review Newton's method in its familiar numerical setting, as an algorithm for solving $f(x) = 0$ for an approximation to a numerical (say real) root $\bar{x}$. The method consists of computing a sequence of so-called *iterates* $x_1, x_2, \ldots$ (approximants to $\bar{x}$) according to *Newton's iteration*,

$$(1) \qquad x_{k+1} = x_k - f(x_k)/f'(x_k),$$

starting from some specified initial approximation x_0 to the desired root $\bar{x}$.

The geometry of Newton's method is illustrated in Fig. 4. The tangent to the curve at $(x_k, f(x_k))$, where x_k is the current iterate, is seen to intercept the x-axis at a point that provides a closer approximation to the root $\bar{x}$. This point, then, is taken to be the next iterate x_{k+1}. The forementioned tangent has the equation

$$\frac{y - f(x_k)}{x - x_k} = f'(x_k)$$