Fast Fourier Transform (FFT) on Rings

1. Generalities

1.1 Definition. A *semigroup* is a system $(S, \circ)$, where S is any set and $\circ$ is a *binary operation* on S satisfying the *associative* law:
$\forall\, a, b, c \in S$ we have $(a \circ b) \circ c = a \circ (b \circ c)$. □

1.2 Definition. $(S, \circ, 1)$, where $1 \in S$ is a "distinguished" element, is a *monoid* iff $(S, \circ)$ is a semigroup and for all $a \in S$

$$a \circ 1 = 1 \circ a = a$$

□

NOTE. "1" is *not* to be confused with the "1" of ordinary arithmetic; it is here an abstract symbol. It is called the *identity for* $\circ$.

1.3 Lemma. *If $(S, \circ, 1)$ is a monoid, then 1 is the only identity, i.e., the only member of S satisfying $a \circ 1 = 1 \circ a = a$ for all $a \in S$.*

Proof. Let $u \in S$ be another identity. Then $a \circ u = u \circ a = a$ for all $a \in S$. In particular,
$u = u \circ 1$, since 1 is an identity. Also,
$1 = u \circ 1$, since u is an identity. Thus, $u = 1$. □

1.4 Definition. If $(S, \circ, 1)$ is a monoid and $a \in S$, then $b \in S$ is *an* inverse of a iff

$$a \circ b = b \circ a = 1$$

□

1.5 Lemma. *An element of monoid $(S, \circ, 1)$ has at most one inverse.*

Proof. Let b and c be inverses of a. We will show that $b = c$.
Indeed, $b = b \circ 1 = b \circ (a \circ c) = (b \circ a) \circ c = 1 \circ c = c$. □

NOTE. a^{-1} will denote the unique inverse of a, whenever it exists. Moreover, the definition of inverse yields immediately that if b is the inverse of a, then a is the inverse of b, in other words, $(a^{-1})^{-1} = a$. This follows from writing the defining equation as $b \circ a = a \circ b = 1$, using the symmetry of equality.

1.6 Definition. $(S, \circ, 1)$ is a *group* iff it is a monoid such that *every* element in S has an inverse. □

1.7 Example. (1) The set of all 2×2 matrices over the integers $\mathbb{I}$ is a semigroup under matrix multiplication (also a monoid).

(2) The set of nonempty strings over a set Σ, i.e., Σ^+, is a semigroup under concatenation, but it is not a monoid.

(3) $\mathbb{I}$ is a monoid under $+$. 0 is the identity element (i.e., 0 is "1" here). It is also a group. On the other hand, $\mathbb{I}$ under $\times$ is a monoid with 1 as the identity. It is not a group though, as nothing besides 1 and -1 has an inverse.

(4) $\mathbb{R}_+$, the set of positive reals, is a group under $\times$ (1 is the identity, and, of course, $a^{-1} = 1/a$).

(5) $\mathbb{I}_m$, the set of integers *modulo* m is a monoid with respect to $\times$. We will see that it is a group if m is prime. □

1.8 Definition. An *abelian* group is one with a *commutative* operation $\circ$. It is also called a *commutative group.*

Traditionally, "+" denotes the operation of an abelian group. In that case we write "0" instead "1" for the identity. □

1.9 Definition. A *ring* $(S, +, \circ, 0, 1)$ is a structure where $(S, +, 0)$ is an *abelian* group, $(S, \circ, 1)$ is a monoid and the *distributive laws* hold, i.e., for all a, b, c in S,

$$a \circ (b + c) = a \circ b + a \circ c$$
$$(b + c) \circ a = b \circ a + c \circ a$$

where we use the convention that $\circ$ has priority over $+$ in the above notation (so we minimize use of brackets).

A *field* is a ring of which *all* members other than 0 have an inverse with respect to $\circ$. □

If a field $(S, +, \circ, 0, 1)$ is such that $\circ$ is commutative, then it is a *commutative field*.

1.10 Exercise. (1) Show that in the definition of ring, the requirement that $(S, +, 0)$ is commutative is automatically satisfied because of the other requirements in the definition, hence is redundant. (*Hint.* Consider, for arbitrary a, b in S, $(a + b) \circ (1 + 1)$ and evaluate in two different ways ...)

(2) Show that in a field $(S, +, \circ, 0, 1)$, $(S - \{0\}, \circ, 1)$ is a group.

1.11 Example. (1) $(\mathbb{I}, +, \times, 0, 1)$ is a ring. Indeed it is a *commutative* ring in the sense that the $\times$ is commutative.

(2) The ring $(M, +, \times, 0, 1)$ of 2×2 matrices over $\mathbb{I}$ (i.e., with entries in $\mathbb{I}$) is not commutative. Here, $+, \times$ are matrix addition and multiplication, while $\binom{00}{00}$ and $\binom{10}{01}$ are the "0" and "1" respectively.

(3) $(\mathbb{I}_m, +, \times, 0, 1)$ is a ring. □

1.12 Lemma. *For all a in a ring,* $0 \circ a = 0$.

Proof.
$$0 + 0 = 0 \tag{1}$$
hence we get the following equations
$$\begin{aligned}(0+0) \circ a &= 0 \circ a \text{ by (1)}\\ 0 \circ a + 0 \circ a &= 0 \circ a \text{ by the distributive law}\\ -(0 \circ a) + (0 \circ a + 0 \circ a) &= -(0 \circ a) + 0 \circ a\\ (-(0 \circ a) + 0 \circ a) + 0 \circ a &= 0\\ 0 + 0 \circ a &= 0\\ 0 \circ a &= 0\end{aligned}$$

□

1.13 Lemma. *For all a, b in a ring,* $-(a \circ b) = (-a) \circ b = a \circ (-b)$.

Proof. Show that $-(a \circ b) = (-a) \circ b$, the proof of the other equality being similar.

Now,
$$\begin{aligned}a \circ b + (-a) \circ b &= (a + (-a)) \circ b \text{ distributive law}\\ &= 0 \circ b = 0\end{aligned}$$
Thus $-(a \circ b) = (-a) \circ b$ by uniqueness of inverses (i.e., $(-a) \circ b$ behaves like the inverse of $a \circ b$, so it *must be it.* □

1.14 Lemma. *For all a, b in a ring,* $a \circ b = (-a) \circ (-b)$.

Proof. Exercise. □

NOTE. In what follows, $+$ and $\cdot$ will denote the group and monoid operations of rings respectively, and, by abuse of language, we shall talk of a ring S [T, W, etc.] instead of the more proper $(S, +, \cdot, 0, 1)$ [$(T, +, \cdot, 0, 1)$, $(W, +, \cdot, 0, 1)$, etc.]. In other words, we identify a ring with its *set* and allow the operations to be understood from the context. We shall also often use the "implied multiplication notation", writing ab instead of $a \cdot b$.

1.15 Definition. Let Z be a ring. Then $Z[x]$ denotes the algebraic system of *polynomials in x over Z*, i.e.,

(1) $a \in Z[x]$ iff $a = (a_0, a_1, \ldots)$, an infinite sequence such that for all i, $a_i \in Z$ and *only finitely many* a_i's are non zero.

(2) "+" in $Z[x]$ is defined *componentwise*, i.e., if $a = (a_0, a_1, \ldots)$ and $b = (b_0, b_1, \ldots)$, then $c = a + b$ iff $c_i = a_i + b_i$ for all i.

(3) "$\cdot$" on $Z[x]$ is defined by $c = a \cdot b$ iff

$$c_m = \sum_{i+j=m} a_i \cdot b_j \text{ for all } m$$

Intuitively, $(a_0, a_1, \ldots)$ stands for $a_0 + a_1 x + a_2 x^2 + \cdots$, which motivates (2) and (3) above, the latter being our familiar "convolution". $\square$

1.16 Definition. For every $a \in Z[x]$ there exists a function $f_a : Z \to Z$ such that for all $m \in Z$

$$f_a(m) = \sum_{i=0}^{\infty} a_i m^i.$$

Note that $\sum_{i=0}^{\infty}$ is a *finite* sum since from a certain index i and onwards $a_i = 0$. Usually, we denote f_a by $a(x)$ and $f_a(m)$ by $a(m)$. We say that $a(m)$ is the *value* of the polynomial $a(x)$ at $x = m$. $\square$

In what follows, we shall also (by abuse of language and notation) say things like "let $a(x)$ be the polynomial $\sum_{i=0}^{n-1} a_i x^i$" instead of saying "let $a = (a_0, \ldots)$ be a polynomial with $a_i = 0$ for all $i > n - 1$". If, moreover, $a_{n-1} \neq 0$, then $n - 1$ is the *degree* of the polynomial.

1.17 Exercise. Show that $Z[x]$ is a ring.

2. Polynomial Evaluation

Let $a(x) = \sum_{i=0}^{n-1} a_i x^i \in Z[x]$, Z a ring. To evaluate $a(x)$ at $x = b$ we use *Horner's Rule* which is the fastest (indeed optimal) way to do this *in general*:

$eval \leftarrow a_{n-1}$
for $i = n-2$ **to** 0 **by** -1 **do begin**
$\quad eval \leftarrow eval \cdot b + a_i$
end

It involves $n-1$ multiplications.

NOTE. In assessing *arithmetic* complexity we usually count the number of multiplications as a *measure* of complexity, with the understanding that, at least at the present state of knowledge, multiplication is more "expensive" (i.e., slower) than addition.

To evaluate $a(x)$ at n different points $b_0, \ldots, b_{n-1}$ it takes *in general* $n(n-1)$ multiplications, applying Horner's rule n times.

In what follows we restrict our study to *commutative rings*, i.e., those whose "multiplication" monoid is commutative.

3. Polynomial Representations in a Machine

(1) The formal definition of a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$ suggests the vector representation $(a_0, a_1, \ldots, a_{n-1})$, i.e., the n-vector of coefficients represents the polynomial.

(2) However, an $(n-1)$-degree polynomial can also be represented by an n-vector of *values* at n *distinct* points x_i for $i = 0, \ldots, n-1$, that is, $(a(x_0), a(x_1), \ldots, a(x_{n-1}))$. $\blacksquare$

This representation is not unique as it depends on the chosen points x_i for $i = 0, \ldots, n-1$, which are *entirely* up to us.

Let us now justify this type of representation. It suffices to show that we can go from one representation to the other.

(i) Clearly, given form (1) we can go to form (2), for *any* choice of x_i for $i = 0, \ldots, n-1$, in $O(n^2)$ steps (1 step = 1 multiplication).

(ii) Given the values $a(x_i)$ for $i = 0, \ldots, n-1$ we can *compute* the a_i's since we can verify that

$$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{n-1} a(x_i) \prod_{\substack{j=0 \\ i \neq j}}^{n-1} \frac{x - x_j}{x_i - x_j}$$

The right hand side is the *Lagrange interpolation formula*. It should be noted that the Lagrange interpolation computation can be carried out in a ring Z (for polynomials in $Z[x]$) if *all* of $\prod_{\substack{j=0\\ i\neq j}}^{n-1} x_i - x_j$ for $i = 0, \ldots, n-1$ have multiplicative inverses in the monoid $(Z, \cdot, 1)$. This is *not* the case in $(\mathbb{I}, \cdot, 1)$ for example.

Indeed, start with $a(x) = x^2 - 2x + 2$. Take $x_0 = 0, x_1 = 1, x_2 = -1$. Hence,

$$
\begin{aligned}
a(x_0) &= 2\\
a(x_1) &= 1\\
a(x_2) &= 5
\end{aligned}
$$

and

$$
\begin{aligned}
a(x) &= \sum_{i=0}^{2} a(x_i) \prod_{\substack{j=0\\ i\neq j}}^{2} \frac{x - x_j}{x_i - x_j}\\
&= 2\frac{(x-1)(x+1)}{(0-1)(0+1)} + 1\frac{(x-0)(x+1)}{(1-0)(1+1)} + \frac{(x-0)(x-1)}{(-1-0)(-1-1)}\\
&= -2(x^2-1) + \frac{1}{2}x(x+1) + \frac{5}{2}x(x-1)\\
&= x^2 - 2x + 2
\end{aligned}
$$

Clearly, $1/2$ and $5/2$ cannot be computed in $\mathbb{I}$ even though the final result *is* in $\mathbb{I}[x]$.

4. The importance of representation (2)

Observe that to *multiply* two polynomials given in the form (1), say $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{i=0}^{n-1} b_i x^i$ we must find all the convolutions $\sum_{i+j=p} a_i b_j$ for $p = 0, \ldots, 2(n-1)$, that is, a total of $O(n^2)$ multiplications.

On the other hand, if $a(x)$ and $b(x)$ were given in the form (2), i.e., via their values at the $2(n-1)+1$ distinct points x_i for $i = 0, 1, \ldots, 2(n-1)$, then the product of the two polynomials *expressed in the form* (2) is the vector $\big(a(x_0)b(x_0), \ldots, a(x_{2(n-1)})b(x_{2(n-1)})\big)$ and this can be obtained in $O(n)$ multiplications!

Of course, this would be "cheating" if we did not have *fast* methods to convert between forms (1) and (2), because for most applications form (1) is required.

So what we are looking for is an efficient polynomial multiplication scheme along the following lines (assume without loss of generality that both polynomials have length n in their form (1) representation):

(A) Evaluate $a(x)$ and $b(x)$ at *appropriately chosen points* $(x_i)_{i=0,\ldots,2(n-1)}$ so that the evaluation can be done in $o(n^2)$ ("small-oh") steps.

(B) Multiply the obtained forms (of type (2)) componentwise in $O(n)$ steps.

(C) Transform the result from (B) into form (1). Note that the form (2) product has length $2(n-1)+1$ which is appropriate for finding the $2(n-1)+1$ coefficients of the form (1) product by interpolation. Of course, we hope here to be able to do both interpolation and evaluation in time $o(n^2)$ by choosing the evaluation points judiciously.

5. *Evaluation at n points*

We shall restrict attention to the problem of n evaluations of $a(x) = \sum_{i=0}^{n-1} a_i x^i$, and to the related interpolation problem (i.e., the number of points equals the vector length of the polynomial). Observe that to evaluate $a(x)$ at m points, where $m > n$, is the same as evaluating $p(x) = \sum_{i=0}^{m-1} p_i x^i$ at m points where $p_i = a_i$ for $i = 0, \ldots, n-1$ and $p_i = 0$ for $i = n, n+1, \ldots, m$.

As a *first attempt* let us assume, without loss of generality, that n is even and that the evaluation points are $(\pm x_i)_{i=0,1,\ldots,n/2-1}$.

We can write

$$a(x) = A_e(x^2) + xA_o(x^2) \tag{1}$$

where

$$A_e(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i \qquad \text{(the even coefficients part)}$$

and

$$A_o(y) = \sum_{i=0}^{n/2-1} a_{2i+1} y^i \qquad \text{(the odd coefficients part)}$$

Therefore,

$$A_e(x^2) = \sum_{i=0}^{n/2-1} a_{2i} x^{2i}$$

and

$$xA_o(x^2) = \sum_{i=0}^{n/2-1} a_{2i+1} x^{2i+1}$$

which verifies equation (1).

Hence, to evaluate $a(\pm x_i), i = 0, \ldots, n/2-1$ we need do (since $(-x_i)^2 = x_i^2$):

(I) Evaluate $A_e(x^2)$ and $A_o(x^2)$ for $i = 0, \ldots, n/2-1$.

(II) Compute $A_e(x_i^2) + x_i A_o(x_i^2)$ and $A_e(x_i^2) - x_i A_o(x_i^2)$ for $i = 0, \ldots, n/2-1$.

Timing: For (I) we do $n/2$ multiplications to find x_i^2, for $i = 0, \ldots, n/2-1$. We then do $2(n/2)$ applications of Horner's rule on polynomials of length $n/2-1$, that is, $n(n/2-1)$ multiplications.

For (II) we do $n/2$ multiplications. Thus, the total is $n + n(n/2-1) = n^2/2$ multiplications, about a twofold improvement over "straight" Horner (i.e., with arbitrarily chosen evaluation points).

But still we have not improved the asymptotic behaviour; we only improved the constant (and have not even discussed interpolation). However, we can do much better by a still cleverer choice of evaluation points. We would like to apply the above scheme recursively, but as it stands we cannot do so as the x_i^2's are all "positive".

6. *Primitive roots of unity*

6.1 Definition. Let $a \in S$, where $(S, \cdot, 1)$ is a monoid. We define $a^0 = 1$ and $a^{m+1} = a^m \cdot a$ for every natural number m. $\square$

According to this definition, $a^1 = a^{0+1} = a^0 \cdot a = 1 \cdot a = a$, as it should be intuitively.

6.2 Exercise. Show that the "laws of exponents" hold in any monoid. I.e., $a^{m+n} = a^m \cdot a^n$ and $(a^m)^n = a^{mn}$ for all natural numbers m, n.

6.3 Remark. It follows that the powers of an element commute with respect to the operation of *any* monoid, for $a^m \cdot a^n = a^{m+n} = a^n \cdot a^m$.

6.4 Definition. $w \in S$, where $(S, \cdot, 1)$ is a monoid, is a *primitive n-th root of unity* ($n > 1$) iff all the following hold:
$w^n = 1$
$w^i \neq 1$ for $0 < i < n$. $\square$

If w is a primitive n-th root of unity, then $w \neq 1$, for otherwise, $1 = w = w^1$ and $0 < 1 < n$.

6.5 Definition. If for $a^m \in S$, where $(S, \cdot, 1)$ is a monoid and m a natural number, the inverse $(a^m)^{-1}$ exists, then we write $a^{-m} = (a^m)^{-1}$. $\square$

6.6 Remark. It is always the case that $a^{-m} = (a^{-1})^m$ as well. Indeed, for $m = 0$ this is trivial. Assume the claim for m and go to $m+1$:

We need to verify that $(a^{-1})^{m+1} \cdot a^{m+1} = 1$. Now, (see also 6.3)

$$(a^{-1})^{m+1} \cdot a^{m+1} = (a^{-1})^m \cdot (a^{-1} \cdot a) \cdot a^m = (a^{-1})^m \cdot 1 \cdot a^m = (a^{-1})^m \cdot a^m \overset{\text{I.H.}}{=} 1$$

6.7 Example. If $m > r > 0$, then $a^m \cdot a^{-r} = a^{m-r}$. Indeed,

$$a^m \cdot a^{-r} \cdot a^r = a^m \tag{1}$$

and

$$a^{m-r} \cdot a^r = a^m \tag{2}$$

By (1) and (2) and cancellation of a^r (Why is this cancellation allowed?) we get $a^m \cdot a^{-r} = a^{m-r}$. $\square$

6.8 Exercise. Show that for $r > 0$, $a^{-(-r)} = (a^{-r})^{-1}$ if a^{-r} exists. Conclude that definition 6.5 makes sense even for negative m. Proceed then to redo exercise 6.2 for *any integers* m and n.

6.9 Lemma. *If w is a primitive n-th root of unity, then w^{-m} exists for all $m \in \mathbb{N}$.*

Proof. If $m < n$, then $w^m \cdot w^{n-m} = w^n = 1$. So, w^{-m} exists, it is w^{n-m}.

If $m > n$, then let $m = qn + r$, where $0 \le r < n$. Then $w^m = w^{qn+r} = w^{qn} \cdot w^r = (w^n)^q \cdot w^r = 1^q \cdot w^r = w^r$. But the previous case established that w^r has an inverse. $\square$

6.10 Lemma. *If w is a primitive n-th root of unity and $n > 2$ is* even*, then w^2 is a primitive $n/2$-th root of unity.*

Proof. $(w^2)^{n/2} = w^n = 1$. Also, $(w^2)^k = w^{2k}$, hence, for $0 < k < n/2$, $\quad (w^2)^k \neq 1$. $\square$

6.11 Lemma. *The points $1, w, w^2, \ldots, w^{n-1}$ are all distinct if w is a primitive n-th root of unity.*

Proof. Let $w^i = w^j, 0 \le i < j \le n-1$. Then $1 = w^{-i} \cdot w^i = w^{-i} \cdot w^j = w^{j-i}$.

Since $0 < j - i < n$ we have a contradiction as n is the *smallest* non zero natural number with the property $w^n = 1$. $\square$

6.12 Lemma. *If w is a primitive n-th root of unity, where n is* even*, then $(w^j)^2 = (w^{n/2+j})^2$ for $j = 0, 1, \ldots, n/2 - 1$.*

Proof. Trivial. $\square$

NOTE. The above Lemma provides a generalization of the property $(-x_i)^2 = x_i^2$. Here $w^{n/2+j}$ *plays the role* of $-w^j$; and "plays the role" is *all* we can say.

Indeed, $(w^j)^2 - (w^{n/2+j})^2 = (w^j + w^{n/2+j}) \cdot (w^j - w^{n/2+j}) = 0$. Now, $w^j - w^{n/2+j} \neq 0$ (Why?), *but this does not imply that* $w^j + w^{n/2+j} = 0$, since the monoid may well have 0-divisors (for example, $\mathbb{I}_6$ does). Thus we *cannot* conclude in general that $-w^j = w^{n/2+j}$.

6.13 Lemma. *If w is an n-th primitive root of unity, then w^{-1} is too.*

Proof. $(w^{-1})^n = (w^n)^{-1} = 1$. If $0 < k < n$, then $(w^{-1})^k = w^{n-k}$ (Why?), but $w^{n-k} \neq 1$ (Why?). $\square$

7. Evaluation at the Fourier points

7.1 Definition. Let w be a primitive n-th root of unity in a ring Z.
The n distinct points $1, w, w^2, \ldots, w^{n-1}$ are called the *Fourier points*. $\square$

We will evaluate a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$ over Z at the n Fourier points. We shall assume without loss of generality that the length n of the polynomial is a power of 2. If not, we add leading 0-coefficients until n becomes a power of two. Since the run time of the algorithm we are after will be $O(n \log n)$, this increase in the length of the polynomial is inconsequential, for in the worst case the original length is almost doubled. But then n applications of Horner take time $n(n-1)$ (with the *original* polynomial) while our proposed scheme will take about $2n \log 2n$ with the *lengthened* polynomial; this is still $O(n \log n)$ hence still faster than Horner.

Now the scheme:
As before,

$$a(x) = A_e(x^2) + xA_o(x^2) \tag{1}$$

We wish to find all the values $a(w^j), j = 0, \ldots, n-1$. To this end we do:

(I) Compute $A_e((w^j)^2)$ and $A_o((w^j)^2)$ for $j = 0, \ldots, n/2-1$ (recall that $(w^j)^2 = (w^{n/2+j})^2$).

(II) Compute $A_e((w^j)^2) + w^j A_o((w^j)^2)$ and $A_e((w^j)^2) + w^{n/2+j} A_o((w^j)^2)$ for $j = 0, \ldots, n/2-1$.

Step (I) can be performed recursively because $(w^j)^2 = (w^2)^j$ and w^2 is a $n/2$-th root of unity, while $n/2$ is still a power of 2. Moreover, this means that $(w^2)^j$ for $j = 0, \ldots, n/2-1$ are distinct Fourier points. So we are asked to solve the *original* problem (it was for $a(x)$), only now it is at *half* the original size (the problem is now for $A_e(x)$ and $A_o(x)$).
Step (II) takes n multiplications.

Thus, if $T(n)$ is the timing function for the problem and n is the original polynomial length, we get the recurrence:

$$\begin{aligned} T(1) &= c \quad \text{(a constant)} \\ T(n) &= 2T(n/2) + n \end{aligned}$$

We know that the above recurrence has solution $T(n) = O(n \log n)$.

NOTE. Tradition has it that the vector $\big(a(1), a(w), \ldots, a(w^{n-1})\big)$ is called the *Fourier Transform* of the vector of the polynomial coefficients $(a_0, a_1, \ldots, a_{n-1})$. This is motivated from the fact that $a(w^j) = \sum_{i=0}^{n-1} a_i w^{ij}$ for $j = 0, 1, \ldots, n-1$, and if our ring is the ring of the complex numbers, $\mathbb{C}$, then we can take $w = e^{2\pi\sqrt{-1}/n}$ which *is* an n-th primitive root.

Thus, $a(w^j) = \sum_{i=0}^{n-1} a_i e^{2\pi ij\sqrt{-1}/n}$ which is the "discrete" version of the Fourier Transform of Calculus. Naturally then, the fast algorithm developed above for the computation of the Fourier Transform is called the *Fast* Fourier Transform, in short FFT.

8. Congruences

8.1 Definition (Congruences on $\mathbb{I}$). The alternative notations $a \equiv b \quad (m)$ or $a \equiv_m b$ or $a \equiv b \pmod m$, read a *is congruent to* b *modulo* m, mean that m divides $a - b$ (in symbols, $m \mid a - b$). $\square$

NOTE. If a and b are not congruent modulo m, then they are *incongruent*. We write $a \not\equiv b \quad (m)$ or $a \not\equiv_m b$ or $a \not\equiv b \pmod m$.

8.2 Proposition. *$\equiv_m$ is an equivalence relation on $\mathbb{I}$.*

Proof. $a \equiv_m b \to b \equiv_m a$ (i.e., $m \mid a - b \to m \mid b - a$). Also, for all $a \in \mathbb{I}$, $a \equiv_m a$ (that is, $m \mid a - a$) is true.

Finally, transitivity:

Let $a \equiv_m b$ and $b \equiv_m c$. That is, $m \mid a - b$ and $m \mid b - c$. Therefore, for some integers k_1, k_2, $a - b = mk_1$ and $b - c = mk_2$, from which follows that $a - c = m(k_1 + k_2)$ or $m \mid a - c$. $\square$

8.3 Definition. We denote $\mathbb{I}/\equiv_m$ (i.e., the set of the equivalence classes of $\mathbb{I}$ under $\equiv_m$) by $\mathbb{I}_m$. $\square$

8.4 Proposition. *$a \equiv b \quad (m)$ iff a and b leave the same remainder when divided by m.*

Proof. *if*-part. Let $a = mk + r$ and $b = mk_1 + r$, where $0 \le r < m$. Then $a - b = m(k - k_1)$, hence $a \equiv b \quad (m)$.

only if-part. Let $a - b = mq$, for some q, and also $a = mk + r$ with $0 \le r < m$, and $b = mk_1 + r_1$ with $0 \le r_1 < m$ (we argue the non trivial case where $a \ne b$).

Then $a - b = m(k - k_1) + r - r_1$, hence $m(q + k_1 - k) = r - r_1$, from which

$$m|q + k_1 - k| = |r - r_1| \tag{1}$$

If $r = r_1$, then we are done, so let $r \neq r_1$. It follows that $q + k_1 - k \neq 0$ hence $|q + k_1 - k| \geq 1$, thus, by (1),

$$|r - r_1| \geq m \tag{2}$$

But

$$\begin{aligned} 0 \leq r < m \ \& \ 0 \leq r_1 < m &\rightarrow \\ 0 \leq r < m \ \& \ -m < -r_1 \leq 0 &\rightarrow \\ -m < r - r_1 < m &\rightarrow \\ |r - r_1| < m & \end{aligned} \tag{3}$$

(3) contradicts (2) and we are done. □

8.5 Corollary. *$\mathbb{I}_m$ is finite, in fact its cardinality is m.*

Proof. There are as many *different* equivalence classes as there are different remainders mod m. □

NOTE. By definition, $\mathbb{I}_m$ is the set of all the equivalence classes of $\mathbb{I}$ mod m. These are $[0], [1], \ldots, [m-1]$. Since, however, $0 \equiv_m m$, $[m], [1], \ldots, [m-1]$ is another *notation* for the equivalence classes. In fact, there are infinitely many ways to represent each class $[i]$: namely, $[km + i]$ for *any* $k \in \mathbb{I}$ will do since $i \equiv km + i \pmod m$.

If we decide to abuse notation and drop the brackets $[\ldots]$, then the elements of $\mathbb{I}_m$ are represented by just m *pairwise incongruent* numbers from $\mathbb{I}$.

Thus, $0, 1, \ldots, m-1$ is the *least non-negative* representation of $\mathbb{I}_m$, while if m is odd, $-(m-1)/2, \ldots, 0, \ldots, (m-1)/2$ is the *least absolute value* representation (check that m consecutive numbers are necessarily pairwise incongruent mod m).

In this abused notation (without the brackets) we must always remember that while we may write $a = b$ *on* $\mathbb{I}_m$, we really mean $[a] = [b]$, i.e., $a \equiv b \pmod m$.

We now define arithmetic operations on $\mathbb{I}_m$.

8.6 Definition. Let x, y be arbitrary in $\mathbb{I}_m$.

(1) $x + y$ is the unique $r \in \mathbb{I}_m$ such that $x + y = r$ (remember what "=" means!).

(2) $x \cdot y$ is the unique $r \in \mathbb{I}_m$ such that $x \cdot y = r$. □

8.7 Remark. (a) Why is r unique in either case? Well, for example, if there were r, r' in $\mathbb{I}_m$ such that $r \equiv x \cdot y \equiv r' \pmod m$, then $r \equiv r' \pmod m$ by transitivity of $\equiv_m$, impossible since any two elements of $\mathbb{I}_m$ are incongruent mod m. O.K., but what about existence? Would there always be an $r \in \mathbb{I}_m$ such that $x + y \equiv_m r$? This asks whether $x + y \in [r]$ for some $r \in \mathbb{I}_m$, and the answer is "yes" since there are m equivalence classes and m pairwise incongruent r's in $\mathbb{I}_m$.

(b) In the case of $+$ and the least non-negative representation

$$x + y = \begin{cases} x + y & \textbf{if } x + y < m \\ m - (x + y) & \textbf{otherwise} \end{cases}$$

(c) In what follows—unless otherwise specified—we shall assume the least non-negative representation, $0, 1, \ldots, m - 1$, as the default *representation* for $\mathbb{I}_m$.

8.8 Proposition. *$a \equiv_m b$ and $c \equiv_m d$ imply:*
(1) $a + c \equiv_m b + d$
(2) $a \cdot c \equiv_m b \cdot d$.

Proof. Let $a - b = mk_1$ and $c - d = mk_2$. Then

(1) $a - b + c - d = (a + c) - (b + d) = m(k_1 + k_2)$, hence $a + c \equiv_m b + d$.

(2) $(a - b)c = mk_1c$, hence $ac \equiv_m bc$. Similarly, $bc \equiv_m bd$. By transitivity, $a \cdot c \equiv_m b \cdot d$. $\square$

8.9 Remark. We can phrase the above Proposition as follows: *If $a = b$ and $c = d$ in $\mathbb{I}_m$, then also $a + c = b + d$ and $ac = bd$ in $\mathbb{I}_m$. That is, as in $\mathbb{I}$, in $\mathbb{I}_m$ too we can add or multiply "equalities" term-wise and obtain equalities.*

8.10 Proposition. *$(\mathbb{I}_m, +, \cdot)$ is a* commutative *ring, where* 0 *is the additive identity and* 1 *is the multiplicative identity.*

Proof. Easy exercise. $\square$

We shall now consider *linear equations in* $\mathbb{I}_m$, $ax + b = 0$ in the *unknown* x.†

The problem is to find $x \in \mathbb{I}_m$ such that $ax + b = 0$ (in $\mathbb{I}_m$; this very annoying qualification, whenever is being safely understood, will be omitted). We shall restrict attention to the case that

(1) the *greatest common divisor* of a and m equals 1, in symbols, $(a, m) = 1$—i.e., when a and m are *relatively prime*—and

(2) $b = -1$.

I.e., we consider the equations $ax = 1$ in $\mathbb{I}_m$ (congruences $ax \equiv_m 1$ in $\mathbb{I}$).

† Or, as we can also call them, *linear congruences in* $\mathbb{I}$ (*not* $\mathbb{I}_m$!), in symbols $ax + b \equiv 0 \pmod{m}$.

8.11 Proposition. *$ax = 1$ has exactly* one *solution in $\mathbb{I}_m$ if $(a, m) = 1$. It has* no *solution if $(a, m) \neq 1$.*

Proof. Let us start with a Lemma: *For any m, n in $\mathbb{I}$, such that $|m| + |n| \neq 0$, there are k, l in $\mathbb{I}$ such that $(m, n) = m \cdot k + n \cdot l$.* **Proof** of **Lemma.** Let $d = (m, n)$. Consider the set $P = \{mx + by : x \in \mathbb{I} \ \ \& \ \ y \in \mathbb{I}\}$. P contains positive elements (one of $x = \pm 1, y = 0$ or $x = 0, y = \pm 1$ will confirm this). Let then δ be the *smallest positive* member of P. We observe (by definition of P) that

$$\delta = m \cdot k + n \cdot l \text{ for some } k, l \text{ in } \mathbb{I} \tag{1}$$

Moreover, it must be that $\delta \mid m$ and $\delta \mid n$. For example, if $\delta \not| m$, then $m = q \cdot \delta + r$, where $0 < r < \delta$, hence $m = q \cdot (m \cdot k + n \cdot l) + r$ which after some arithmetic gives

$$r = m(1 - q \cdot k) + n \cdot (-l) \tag{2}$$

By (2), $r \in P$, but since $0 < r < \delta$ we have just contradicted the choice of δ. Thus, δ is one of the (positive) common divisors of m, n and therefore $\delta \leq d$. On the other hand, by (1), $d \mid \delta$, hence $d \leq \delta$. Thus, $d = \delta$. The Lemma is now settled.

We return to the proof of the Proposition. Here $(a, m) = 1$, thus, for some k, l in $\mathbb{I}$, $1 = a \cdot k + m \cdot l$, or

$$a \cdot k = 1 \text{ in } \mathbb{I}_m$$

Thus, $x = k$ solves the equation $a \cdot x = 1$ in $\mathbb{I}_m$. The solution is unique, since another solution k' yields $a \cdot (k - k') = 0$ in $\mathbb{I}_m$, or $m \mid a(k - k')$, from which $m \mid k - k'$ due to $(a, m) = 1$.† That is, $k = k'$ in $\mathbb{I}_m$.

Finally, if $(a, m) = d > 1$ and $a \cdot k = 1$ in $\mathbb{I}_m$ we have $a \cdot k - 1 = m \cdot q$ for some $q \in \mathbb{I}$ therefore $d \mid a \cdot k - m \cdot q$, i.e., $d \mid 1$—impossible. □

8.12 Corollary. *If $(a, m) = 1$, then an x such that $a \cdot x = 1$ in $\mathbb{I}_m$ can be algorithmically found.*

Proof. Try $0, 1, \ldots, m - 1$ as candidates. Exactly one *will* work. □

NOTE. There are more efficient methods than the above.

8.13 Corollary. *The invertible elements a of $\mathbb{I}_m$ are precisely those for which $(a, m) = 1$.*

Proof. $a \in \mathbb{I}_m$ is invertible iff for some $x \in \mathbb{I}_m$ $a \cdot x = 1$ in $\mathbb{I}_m$. □

8.14 Corollary. *If m is* prime*, then $\mathbb{I}_m$ is a field.*

Proof. Now for all $a \in \mathbb{I}_m$, $a \neq 0$ in $\mathbb{I}_m \rightarrow (a, m) = 1$. □

† If $(a, m) = 1$ and $m \mid a \cdot b$, then $m \mid b$, for using the same k, l as above we see that $b = (a \cdot b) \cdot k + m \cdot b \cdot l$.

9. The Chinese Remainder Theorem (CRT)

9.1 Theorem. *Consider*

$$x \equiv a \pmod{m_1} \tag{1}$$
$$x \equiv b \pmod{m_2} \tag{2}$$

where

$$(m_1, m_2) = 1 \tag{3}$$

Then there is a u that makes the congruences (1) and (2) in the unknown x simultaneously true. Moreover, there is exactly one such u in $\mathbb{I}_{m_1 m_2}$.

Proof. To see this, try at first a "guess" $\hat{u} = am_2 + bm_1$ at a solution. Clearly,

$$\hat{u} \equiv am_2 \pmod{m_1} \text{ (since } bm_1 \equiv 0)$$
$$\hat{u} \equiv bm_1 \pmod{m_2} \text{ (since } am_2 \equiv 0)$$

The problem is that we still have m_1, m_2 around. A bit of thought suggests that we should have multiplied am_2 with the inverse of m_2 *in* $\mathbb{I}_{m_1}$ and bm_1 with the inverse of m_1 *in* $\mathbb{I}_{m_2}$.

So try next a better "guess", namely $u = am_2c_2 + bm_1c_1$ where

$$m_2c_2 \equiv 1 \pmod{m_1}$$
$$m_1c_1 \equiv 1 \pmod{m_2}$$

Such c_1, c_2 exist by (3). One can verify immediately that

$$u \equiv a \pmod{m_1} \tag{4}$$
$$u \equiv b \pmod{m_2} \tag{5}$$

Moreover, if u' also satisfies (4) and (5), then $u \equiv u' \pmod{m_1}$ and $u \equiv u' \pmod{m_2}$, hence $m_1 \mid u - u'$ and $m_2 \mid u - u'$. By (3), $m_1m_2 \mid u - u'$,† i.e., $u = u'$ in $\mathbb{I}_{m_1m_2}$. □

NOTE. The above theorem is the "Chinese Remainder Theorem".

9.2 Corollary. *The CRT is true for any number of pairwise prime moduli $m_1, \ldots, m_r$.*

---

† Say, $m_1 \mid n$ and $m_2 \mid n$, $n = m_1q_1$ and $n = m_2q_2$ for some q_1, q_2. From $1 = m_1k + m_2l$ for some k, l follows $n = m_1m_2q_2k + m_2m_1q_1l$.

10. An application of the CRT

Suppose that we have an algorithm on $\mathbb{I}$ involving $+, -, \times, /$ with inputs x_i for $i = 0, \ldots, n-1$ which generates outputs y_i for $i = 0, \ldots, n-1$.

It may be that this algorithm is not as efficient as we like, but that instead, if we *do all the arithmetic in* $\mathbb{I}_{m_1}, \mathbb{I}_{m_2}$ and $\mathbb{I}_{m_3}$, where $m_1 < m_2 < m_3$ and the m_i are pairwise relatively prime, we are satisfied with the efficiency of the algorithm.

Now "doing all the arithmetic in $\mathbb{I}_{m_j}$, for $j = 1, 2, 3$" means that with inputs x_i for $i = 0, \ldots, n-1$ we get outputs $y_i \pmod{m_j}$ for $i = 0, \ldots, n-1$ and $j = 1, 2, 3$. In other words, instead of the "correct" (or "exact") y_i for $i = 0, \ldots, n-1$ we have computed (albeit fast) *aliases* or *representations* $\tilde{y}_i, \tilde{\tilde{y}}_i, \tilde{\tilde{\tilde{y}}}_i$ for $i = 0, \ldots, n-1$, such that

$$y_i \equiv \tilde{y}_i \pmod{m_1} \tag{1}$$
$$y_i \equiv \tilde{\tilde{y}}_i \pmod{m_2} \tag{2}$$
$$y_i \equiv \tilde{\tilde{\tilde{y}}}_i \pmod{m_3} \tag{3}$$

By the CRT, we can find integers u_i that satisfy all the above congruences in y_i, and these solutions are unique $\pmod{m_1 m_2 m_3}$.

If moreover m_1, m_2, m_3 were chosen to be large enough to satisfy $0 \le y_i < m_1 m_2 m_3$, where as before y_i are the "exact" solutions, then $y_i = u_i$ *in* $\mathbb{I}$ (since this equality holds in $\mathbb{I}_{m_1 m_2 m_3}$).

In conclusion, if the modular method that finds the $\tilde{y}_i$ etc., is much faster than the "straight" method in $\mathbb{I}$ (that finds the exact y_i directly), *and* if we can solve (1), (2) and (3) above *fast* (to find u_i—that turn out to be y_i!), then we are better off to use modular arithmetic.

11. Back to FFT; Fast Interpolation

The FFT given in section 7 was in terms of an abstract (uninterpreted) ring Z and its operations $+, -$. We reproduce it below in some detail:

proc $FFT(a, w, n, s)$

{The procedure "FFT" finds the FFT of $a = (a_s, a_{s+1}, \ldots, a_{s+n-1})$ and stores it in a as $(\hat{a}_s, \hat{a}_{s+1}, \ldots, \hat{a}_{s+n-1})$, i.e., as $\big(a(w^0), a(w^1), \ldots, a(w^{n-1})\big)$, where n is a power of 2, w is a primitive n-th root of unity in the ring Z of the coefficients.

A is a *global* array defined outside FFT.}

if $n > 1$ **then begin**

{Split a in A_e and A_o halves. A_o is to occupy the left half and A_o the right half of a.}

for $k = 0$ **to** $n - 1$ **by** 2 **do begin**
$A(k/2) \leftarrow a(s + k)$
$A(n/2 + k/2) \leftarrow a(s + 1 + k)$
$a \leftarrow A$
end

call $FFT(a, w^2, n/2, s)$ $\{FFT$ of A_e at $w^2\}$
call $FFT(a, w^2, n/2, s + n/2)$ $\{FFT$ of A_0 at $w^2\}$

{Now we have the $A_o(w^{2j})$'s and $A_e(w^{2j})$'s. Next, form the $a(w^j)$'s.}

for $k = 0$ **to** $n/2 - 1$ **do begin**
$T \leftarrow a(k + s) + w^k a(k + s + n/2)$
$S \leftarrow a(k + s) + w^{n/2+k} a(k + s + n/2)$
$a(k + s) \leftarrow T$
$a(k + s + n/2) \leftarrow S$
end

end

end FFT

In any concrete application, we must adapt the above FFT-algorithm and its "arithmetic" in terms of the concrete ring operations we have in mind. Observe that $\mathbb{I}$ is not an appropriate ring to do FFT in, for it has no primitive roots of unity. On the other hand this is not too disastrous, for the rings $\mathbb{I}_m$, where $m = 2^{2^n} + 1$, do have roots of unity, and as the previous section already told us we can, essentially, do $\mathbb{I}$-arithmetic in $\mathbb{I}_m$, for appropriately large m.

The numbers $2^{2^n} + 1$, for $n \geq 0$ are denoted by F_n and are called "Fermat Numbers". Fermat had conjectured that they are all prime, but this turned out to be not true. Rings $\mathbb{I}_m$, where $m = F_n$, we will henceforth denote by $\mathbb{I}_{F_n}$.

Let us turn to the converse problem, that of *interpolation*, i.e., going from $\big(a(w^0), a(w^1), \ldots, a(w^{n-1})\big)$ back to $(a_0, \ldots, a_{n-1})$.

We just saw that the FFT produces $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{n-1})$, where $\hat{a}_i = a(w^i)$, from the vector (polynomial) $a = (a_0, \ldots, a_{n-1})$ in $O(n \log n)$ multiplications. Let us see how to recover a from $\hat{a}$ also in $O(n \log n)$ multiplications. To this end start with the polynomial

$$p(x) = \sum_{i=0}^{n-1} \hat{a}_i x^i \tag{1}$$

and evaluate at the Fourier points $1, w^{-1}, (w^{-1})^2, \ldots, (w^{-1})^{n-1}$. This gives for each $m = 0, 1, \ldots, n-1$,

$$\begin{aligned} p(w^{-m}) &= \sum_{i=0}^{n-1} \hat{a}_i w^{-mi} \\ &= \sum_{i=0}^{n-1} a(w^i) w^{-mi} \end{aligned} \tag{2}$$

Now, $a(w^i) = \sum_{j=0}^{n-1} a_j (w^i)^j = \sum_{j=0}^{n-1} a_j w^{ij}$, so that (2) becomes

$$\begin{aligned} p(w^{-m}) &= \sum_{i=0}^{n-1} \Big(\sum_{j=0}^{n-1} a_j w^{ij} \Big) w^{-mi} \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_j w^{i(j-m)} \\ &= \sum_{j=0}^{n-1} a_j \sum_{i=0}^{n-1} w^{i(j-m)} \end{aligned} \tag{3}$$

For convenience, let us put $b_{jm} = \sum_{i=0}^{n-1} w^{i(j-m)}$, so (3) becomes

$$p(w^{-m}) = \sum_{j=0}^{n-1} a_j b_{jm} \tag{4}$$

Observe that

(a) $0 \le |j - m| \le n - 1$

(b) w^{-1} is a primitive n-th root of unity.

Hence, b_{jm} has the form

$$\sum_{i=0}^{n-1} u^{ik}, \quad 0 \le k \le n-1 \tag{5}$$

where u is a primitive n-th root of unity (if $j > m$, then $u = w$ and $k = j - m$; if $j \leq m$, then $u = w^{-1}$ and $k = m - j$).

Clearly, if $k = 0$ (i.e., $j = m$), then

$$b_{jm} = \overbrace{1 + \cdots + 1}^{n \text{ times}}$$

NOTE. $\overbrace{1 + \cdots + 1}^{n \text{ times}}$ is usually denoted by $n \cdot 1$. Be careful to distinguish that while $n \in \mathbb{N}$ "1" is *the multiplicative identity in a ring, and is* not *necessarily the "1" of* $\mathbb{N}$.

Here we are witnessing a *mixed* operation, i.e., one whose operands do *not* necessarily come from the *same set*. Also note how this situation is entirely similar to a^n of Definition 6.1. Similarly, one can define the mixed operation "$\cdot$" (not to be confused with the "multiplication" $\cdot$ of the ring or the multiplication $\times$ of $\mathbb{N}$) by the equations

$$\begin{aligned} 0 \cdot a &\stackrel{\text{def}}{=} 0 \text{ (the } \textit{left } 0 \text{ is in } \mathbb{N}\text{; the } \textit{right } 0 \text{ is in the ring)} \\ (n+1) \cdot a &\stackrel{\text{def}}{=} n \cdot a + a \text{ (for any } a \text{ in the ring)} \end{aligned}$$

The above captures $n \cdot a = \overbrace{a + \cdots + a}^{n \text{ times}}$.

Now, we will compute b_{jm} for $j \neq m$, i.e., we will compute (5) in the case $0 < k \leq n - 1$ (u is still an n-th root!):

There are rings where the sum (5) equals 0 (for $0 < k$). For example, if the ring *does not have divisors of zero*, then this will be the case, for

$$(u^k - 1) \sum_{i=0}^{n-1} u^{ik} = (u^k)^n - 1 = (u^n)^k - 1 = 0$$

but $u^k - 1 \neq 0$ (Why?), and the absence of 0-divisors forces (5) to equal 0.

It turns out that a much wider class of rings, namely, $\mathbb{I}_{F_r}$, for any $r \in \mathbb{N}$, have the property that (5) becomes 0 whenever $0 < k \leq n - 1$ (and n is the order of the primitive root w). This is due to a more "complicated" reason (explained later) and is true *even if* divisors of 0 are present.

So what happens if $b_{jm} = 0$ (for $j \neq m$)? We get (from (4)) $p(w^{-m}) = (n \cdot 1) a_m$,† and therefore, *if* $(n \cdot 1)^{-1}$ *exists* in the ring in question (we shall arrange that this is so in our applications) then

$$a_m = (n \cdot 1)^{-1} p(w^{-m})$$

† We are guilty of employing "implied multiplication" all over the place, without warning. Here, a more pedantic way of writing things down would be $p(w^{-m}) = (n \cdot 1) \circ a_m$, where "$\circ$" is the ring multiplication. Note that while $n \in \mathbb{N}$, $1, (n \cdot 1)$ and a_m are ring elements.

Therefore, to go from

$$\big(a(w^0), \ldots, a(w^{n-1})\big)$$

that is

$$\hat{a} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})$$

to the polynomial (vector of coefficients)

$$(a_0, a_1, \ldots, a_{n-1})$$

i.e., to do *interpolation*, we simply call $FFT(\hat{a}, w^{-1}, n, 0)$ and multiply each obtained entry by $(n \cdot 1)^{-1}$.

This can be done in $O(n \log n)$ multiplications. We just did *fast interpolation* by doing *fast evaluation* on the appropriate polynomial.

Let us justify now our claims above:

11.1 Lemma. *Let S be a ring, $a \in S$, $n \in \mathbb{N}$, n even. Then*

$$\sum_{i=0}^{n-1} a^{ik} = (1 + a^{(n/2)k}) \sum_{i=0}^{n/2-1} a^{ik}$$

Proof.

$$\begin{aligned}
\sum_{i=0}^{n-1} a^{ik} &= \sum_{i=0}^{n/2-1} a^{ik} + \sum_{i=n/2}^{n-1} a^{ik} \\
&= \sum_{i=0}^{n/2-1} a^{ik} + a^{(n/2)k} \sum_{i=0}^{n/2-1} a^{ik} \\
&= \big(1 + a^{(n/2)k}\big) \sum_{i=0}^{n/2-1} a^{ik}
\end{aligned}$$

□

11.2 Lemma. *Let S be a ring, $a \in S$, $n \in \mathbb{N}$, $n = 2^m$, for some $m \in \mathbb{N}$. Then*

$$\sum_{i=0}^{n-1} a^{ik} = \prod_{i=1}^{m} (1 + a^{(n/2^i)k}) = \prod_{i=0}^{m-1} (1 + a^{2^i k})$$

Proof. Repeated application of the previous Lemma. □

11.3 Lemma. *Let S be a ring, $a \in S$, $n \in \mathbb{N}$, $n = 2^m$, for some $m \in \mathbb{N}$ and $a^{n/2} = -1$. Then $\sum_{i=0}^{n-1} a^{ik} = 0$ if $0 < k < n$.*

Proof. $k = 2^l(2p+1)$ for some l, p in $\mathbb{N}$ (Why?). Thus,

$$\sum_{i=0}^{n-1} a^{ik} = \prod_{i=1}^{m} (1 + a^{(n/2^i)2^l(2p+1)}) \tag{1}$$

From $0 < k < n$ follows that $0 \le l < m$, thus

$$1 + a^{(n/2^{l+1})2^l(2p+1)} \tag{2}$$

is a factor of the right hand side of (1). But (2) equals to

$$\begin{aligned} 1 + a^{(n/2)(2p+1)} &= 1 + (a^{n/2})^{2p+1} \\ &= 1 + (-1)^{2p+1} \\ &= 1 + (-1) \\ &= 0 \end{aligned}$$

Hence (1) is 0. □

11.4 Proposition. *The ring $\mathbb{I}_{F_n}$, has 2 as a 2^{n+1}-th primitive root of unity. Moreover, $2^{2^n} = -1$ in $\mathbb{I}_{F_n}$, thus, by the previous Lemma,*

$$\sum_{i=0}^{2^{n+1}-1} 2^{ik} = 0$$

if $0 < k < 2^{n+1}$.

Proof. Trivially,

$$2^{2^n} \equiv -1 \pmod{2^{2^n} + 1}$$

hence

$$\left(2^{2^n}\right)^2 \equiv (-1)^2 \equiv 1 \pmod{2^{2^n} + 1}$$

i.e.,

$$2^{2^{n+1}} \equiv 1 \pmod{2^{2^n} + 1}$$

Let t be the *smallest* for which

$$2^t \equiv 1 \pmod{2^{2^n} + 1} \tag{1}$$

Observe that:

(a) $t \leq 2^{n+1}$

(b) Suppose $0 < t < 2^{n+1}$ and let $2^{n+1} = qt + r$ where $0 \leq r < t$. Then

$$1 \equiv 2^{2^{n+1}} = 2^{qt+r} = (2^t)^q 2^r \equiv 2^r \pmod{2^{2^n} + 1} \tag{2}$$

Thus, $r = 0$ (otherwise, r satisfies $0 < r < t$ and (1), contrary to the choice of t). It follows that $t = 2^m$ for some $0 < m < n + 1$ (from $0 < t = 2^m < 2^{n+1}$).

(c) Now $2^{2^m} \equiv 1 \pmod{2^{2^n} + 1}$ iff $2^{2^n} + 1 \mid 2^{2^m} - 1$; impossible since $2^{2^n} + 1 > 2^{2^m} - 1$ (by $m < n + 1$). □

From the previous Proposition we gather the following facts about $\mathbb{I}_{F_n}$ for any $n > 1$:

(a) 2 is a 2^{n+1}-th primitive root of unity; so is 2^{-1} ($= 2^{2^{n+1}-1}$).

(b) *Evaluation* of

$$a(x) = \sum_{i=0}^{2^{n+1}-1} a_i x^i$$

at the Fourier points

$$1, 2, 2^2, \ldots, 2^{2^{n+1}-1}$$

is done with $O(n2^n)$ multiplications.

(c) *Interpolation*, to get

$$a_0, \ldots, a_{2^{n+1}-1}$$

from

$$\hat{a} = \big(a(1), a(2), a(2^2), \ldots, a(2^{2^{n+1}-1})\big)$$

is done by FFT on input $\hat{a}$, using 2^{-1} (i.e., $2^{2^{n+1}-1}$) as a 2^{n+1}-th primitive root of unity. The resulting intermediate vector is multiplied by $(2^{n+1} \cdot 1)^{-1} = 2^{2^{n+1}-n-1}$ in order to finally get the vector $(a_0, \ldots, a_{2^{n+1}-1})$. All this also takes $O(n2^n)$ multiplications.

(c) Multiplications by powers of 2 are very easy to do, if the numbers are in base-2 notation.

In practice then, $\mathbb{I}_{F_n}$ are very attractive to do FFT in.

Not every $\mathbb{I}_m$ (for m *not* of the form $2^{2^r} + 1$ that is) has a primitive root of order n such that $\sum_{i=0}^{n-1} w^{ik} = 0$ for $0 < k < n$. For example, in $\mathbb{I}_{15}$ $w = 4$ is a 2-nd primitive root (i.e., $n = 2$), since $4^2 = 16 \equiv 1 \pmod{15}$.

But for $k = 1$,

$$\begin{aligned}\sum_{i=0}^{n-1} w^{ik} &= \sum_{i=0}^{n-1} w^i \\ &= 1 + 4 \\ &= 5 \\ &\not\equiv 0 \pmod{15}\end{aligned}$$

12. Fast Polynomial Multiplication

We want to multiply

$$a(x) = \sum_{i=0}^{2^n-1} a_i x^i \tag{A}$$

and

$$b(x) = \sum_{i=0}^{2^n-1} b_i x^i \tag{B}$$

both over $\mathbb{I}$. We assume that the polynomials have just enough leading zero-coefficients† so that

$$c(x) = a(x)b(x) = \sum_{i=0}^{2^n-1} c_i x^i$$

where

$$c_r = \sum_{r=i+j} a_i b_j$$

We are told that there is a bound B on the a_i's and b_i's, i.e.,

$$|a_i| \le B, |b_i| \le B, i = 0, 1, \ldots, 2^n - 1$$

hence

$$|c_i| \le 2^n B^2, i = 0, 1, \ldots, 2^n - 1 \tag{1}$$

The length of the polynomials suggests that we work in $\mathbb{I}_{F_{n-1}}$, so that 2 is a 2^n-th root of unity. Since some coefficients may be negative, we should use the least absolute value representation, in which we can faithfully portray results of absolute size $\le 2^{2^{n-1}-1}$.

Now if B satisfies $2^n B^2 \le 2^{2^{n-1}-1}$, i.e.,

$$B^2 \le 2^{2^{n-1}-n-1} \tag{2}$$

then we can carry out the FFT and all its arithmetic in $\mathbb{I}_{F_{n-1}}$ (represented by the least absolute value scheme). Indeed, let the so *computed* coefficients be $\tilde{c}_i$, while the "exact" ones (i.e., the ones obtained if we did *not* do arithmetic (mod F_{n-1})) are $c_i, i = 0, 1, \ldots, 2^n - 1$.

Then

$$c_i \equiv \tilde{c}_i \pmod{F_{n-1}}, i = 0, 1, \ldots, 2^n - 1 \tag{3}$$

† Two $r-1$ degree—therefore, length r, counting 0-coefficient terms—polynomials have a product of degree $2r-2$, hence of length $2r-1$. We can therefore "pad" the factor polynomials by leading zero coefficients, so that all three have length $2r-1$.

Now, $|c_i| \le 2^{2^{n-1}-1}$ by (2), and $|\tilde{c}_i| \le 2^{2^{n-1}-1}$ since we use the least absolute value representation while doing $\mathbb{I}_{F_{n-1}}$-arithmetic.

By (3), since any two integers in the range $-(F_{n-1}-1)/2, \ldots, (F_{n-1}-1)/2$ are incongruent $\pmod{F_{n-1}}$,

$$c_i = \tilde{c}_i, i = 0, 1, \ldots, 2^n - 1 \textit{ in "real life", i.e., in } \mathbb{I} \tag{4}$$

12.1 Example. As a "practical" example, let us assume a computer with word length 31 bits (plus sign), thus of maximum word value $W = 2^{31} - 1$ $(= \underbrace{11\ldots1}_{31\ 1\text{'s}})$.

In the interest of letting the hardware do for us all the integer arithmetic let us restrict n so that $F_{n-1} \le W$, that is, $2^{2^{n-1}} + 1 \le 2^{31} - 1$, which yields $n = 5$ (maximum), hence $F_{n-1} = 65537$.

By (2), the maximum value of B we can handle with this F_{n-1} is 32; not very exciting. To summarize:

For polynomials ((A) and (B) above) of length $2^n = 32$ (allowing leading 0's so that the product has the same length) and coefficients of absolute size ≤ 32 we can obtain the product $c(x)$ in $O(n2^n)$ operations doing arithmetic modulo 65537, by doing the following:

(i) $FFT(a, 2, 2^n, 0)$ [This yields the vector $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{2^n-1})$ in $O(n2^n)$ steps],

(ii) $FFT(b, 2, 2^n, 0)$ [This yields the vector $\hat{b} = (\hat{b}_0, \ldots, \hat{b}_{2^n-1})$ in $O(n2^n)$ steps],

(iii) Multiply out the vectors $\hat{a}, \hat{b}$ componentwise to obtain $\hat{c} = (\hat{c}_0, \ldots, \hat{c}_{2^n-1})$ in $O(2^n)$ steps,

(iv) $FFT(\hat{c}, 2^{-1}, 2^n, 0)$ [This yields the vector c, i.e., $c(x)$, in $O(n2^n)$ steps; $2^{-1} = 2^{2^n-1} \pmod{F_{n-1}}$, of course].

Throughout, a "step" is an integer multiplication (or division). □

Let us make the above example more "practical".

12.2 Example. We still want to multiply the polynomials $a(x)$ and $b(x)$ above, only this time we assume a more realistic scenario that

$$B = 2^{2^{n-1}}$$

Here we will use the CRT and apply (i)–(iv) twice; the first time $\pmod{F_{n-1}}$ and the second time $\pmod{F_n}$.

First of all, by (1), $|c_i| \le 2^n 2^{2^n} = 2^{2^n+n}, i = 0, 1, \ldots, 2^n - 1$. These numbers are too big to "fit" in $\mathbb{I}_{F_{n-1}}$. Therefore:

Working in $\mathbb{I}_{F_{n-1}}$.

(i) $FFT(a, 2, 2^n, 0)$ [This yields the vector, *in* $\mathbb{I}_{F_{n-1}}$, $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{2^n-1})$ in $O(n2^n)$ steps],

(ii) $FFT(b, 2, 2^n, 0)$ [This yields the vector $\hat{b} = (\hat{b}_0, \ldots, \hat{b}_{2^n-1})$ in $O(n2^n)$ steps],

(iii) Multiply out the vectors $\hat{a}, \hat{b}$ componentwise to obtain $\hat{c} = (\hat{c}_0, \ldots, \hat{c}_{2^n-1})$ in $O(2^n)$ steps,

(iv) $FFT(\hat{c}, 2^{-1}, 2^n, 0)$ [This yields the vector

$$\tilde{c} = (\tilde{c}_0, \ldots, \tilde{c}_{2^n-1})$$

in $\mathbb{I}_{F_{n-1}}$—i.e., $\tilde{c}(x)$—in $O(n2^n)$ steps; $2^{-1} = 2^{2^n-1} \pmod{F_{n-1}}$, of course].

Working in $\mathbb{I}_{F_n}$. Here we apply additional padding to our polynomials, using leading 0's, so that (using still a and b to denote the so padded polynomials with—hopefully—no danger of confusion)

$$a = (a_0, \ldots, a_{2^n-1}, \underbrace{0, \ldots, 0}_{2^n \text{ 0's}})$$

and

$$a = (b_0, \ldots, b_{2^n-1}, \underbrace{0, \ldots, 0}_{2^n \text{ 0's}})$$

This, of course, we do to avail of the services of 2 as a primitive 2^{n+1}-th root in $\mathbb{I}_{2^{2^n}+1}$.

(i) $FFT(a, 2, 2^{n+1}, 0)$ [This yields the vector, *in* $\mathbb{I}_{2^{2^n}+1}$, $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{2^{n+1}-1})$ in $O(n2^n)$ steps],

(ii) $FFT(b, 2, 2^{n+1}, 0)$ [This yields the vector $\hat{b} = (\hat{b}_0, \ldots, \hat{b}_{2^{n+1}-1})$ in $O(n2^n)$ steps],

(iii) Multiply out the vectors $\hat{a}, \hat{b}$ componentwise, *in* $\mathbb{I}_{2^{2^n}+1}$, to obtain $\hat{c} = (\hat{c}_0, \ldots, \hat{c}_{2^{n+1}-1})$ in $O(2^n)$ steps,

(iv) $FFT(\hat{c}, 2^{-1}, 2^{n+1}, 0)$ [This yields the vector

$$\tilde{\tilde{c}} = (\tilde{\tilde{c}}_0, \ldots, \tilde{\tilde{c}}_{2^{n+1}-1})$$

in $\mathbb{I}_{F_n}$—i.e., $\tilde{\tilde{c}}(x)$—in $O(n2^n)$ steps; $2^{-1} = 2^{2^{n+1}-1} \pmod{F_n}$, of course].

Again, a "step", throughout, is an integer multiplication (or division).

Now to get what we really are after: $c = (c_0, \ldots, c_{2^n-1})$.

We have that

$$c_i \equiv \tilde{c}_i \pmod{F_{n-1}}, i = 0, \ldots, 2^n - 1 \tag{5}$$

and

$$c_i \equiv \tilde{\tilde{c}}_i \pmod{F_n}, i = 0, \ldots, 2^n - 1 \tag{6}$$

(**NB.** The congruences (6) go all the way to $i = 2^{n+1} - 1$, but these additional 2^n congruences are redundant, since we already know that $c_i = 0$ for $i = 2^n, \ldots, 2^{n+1} - 1$.) Moreover,

$$|c_i| \leq 2^{2^n+n} \leq 2^{2^n+2^{n-1}-1} < \frac{F_{n-1}F_n - 1}{2} \text{ for } n \geq 3$$

and $(F_{n-1}, F_n) = 1$ [indeed, $F_n = F_{n-1}(2^{2^{n-1}} - 1) + 2$, thus $(F_{n-1}, F_n) \mid 2$. But F_{n-1} and F_n are odd].

Thus, by CRT, we can recover the "real-life" c_i's from (5) and (6) in another $O(2^n)$ steps.

The overall task then is completed in $O(n2^n)$ steps.

Now, how "expensive" are all these individual "steps" (integer multiplications)? In an unrestricted setting (unbounded n) their cost grows with n, so that the task—theoretically—is more expensive than $O(n2^n)$ *assuming that we follow the described algorithm.*† However, in a "practical" setting we can do all our multiplications/divisions by the hardware mechanism in single precision arithmetic, and hence charge $O(1)$ for each step (= multiplication or division).

For example, let $n = 6$. Then, $F_{n-1} = 2^{32}+1$ and $F_n = 2^{64}+1$ and any of these numbers can be represented by no more than 3 contiguous words $W = 2^{31} - 1$, therefore each single multiplication/division by numbers up to size F_n can still be done in $O(1)$ hardware (single precision) multiplications/divisions (elaborate to convince yourself).

Thus, our "practical" polynomial multiplication scheme takes $O(N \log N)$ integer multiplications (where $N = 2^n$, the length of the *padded* polynomials) and handles polynomials of lengths up to $2^{n-1} = 2^5 = 32$‡ and with coefficient sizes (in absolute value) up to 4.29×10^9, which is a pretty satisfactory practical scenario. □

† We cannot make any absolute claims about what is the "real" expense for doing an integer multiplication, for what is the theoretically best possible is not known.

‡ Recall that the polynomials $a(x)$ and $b(x)$ of *actual* length 2^{n-1} will be padded in the first instance to length 2^n so that the product will have the same length as the inputs.

13. A “practical” Fast Integer Multiplication

We can apply the result of Example 12.2, with minor modifications and some ramifications, to obtain a “practical”, and fast, integer multiplication scheme.

We are to multiply two numbers, of *binary* length 2^n each (without loss of generality, we achieve a power of 2 length by leading 0 padding; moreover, the padding is sufficient to guarantee an output of length 2^n as well).

Before proceeding further, let us elaborate on some important aspects of doing (mod F_n) arithmetic (in the least non-negative representation setting) for any $n \geq 0$:

Arithmetic (mod F_n)**.** Any number $x \in \mathbb{I}_{F_n}$ can be represented as a bit-string

$$x_{2^{n+1}-1} x_{2^{n+1}-2} \ldots x_1 x_0 \text{ where } x_i \in \{0,1\}$$

with the understanding

$$x = \sum_{j=0}^{2^{n+1}-1} x_j 2^j \tag{1}$$

The above *looks* like the expansion of x base-2 in $\mathbb{N}$, except that here we have added leading 0 bits (digits) so that—as bit strings—*all* the numbers $x \in \mathbb{I}_{F_n}$ have (binary) length 2^{n+1}.

There is a small price to pay for doing arithmetic in $\mathbb{I}_{F_n}$: Such representations are *not* unique (contrary to the $\mathbb{N}$ or $\mathbb{I}$ case) since, among other things, $2^{2^{n+1}} = 1$ *in* $\mathbb{I}_{F_n}$. Thus, for example, the number 0 is represented as both

$$\underbrace{000\ldots0}_{2^{n+1}}$$

and

$$\underbrace{111\ldots1}_{2^{n+1}}$$

the latter because

$$0 = 2^{2^{n+1}} - 1 = \underbrace{111\ldots1}_{2^{n+1}} \text{ in } \mathbb{I}_{F_n}$$

This will not trouble us though, for there are overriding benefits too. Such a representation has the important property that multiplication by a power of 2 (say, 2^k) is *extremely* easy:

$$\begin{aligned}
x2^k &= \sum_{i=0}^{2^{n+1}-1} x_j 2^{k+j} \\
&= \sum_{i=0}^{2^{n+1}-k-1} x_j 2^{k+j} + \sum_{i=2^{n+1}-k}^{2^{n+1}-1} x_j 2^{k+j}
\end{aligned}$$

Consult now the following Figures (1–3), where Fig. 1 is the “picture” before multiplication, Fig. 2 is the situation after multiplication but before rearranging the “bits” in their proper sequence, and Fig. 3 is the final picture.

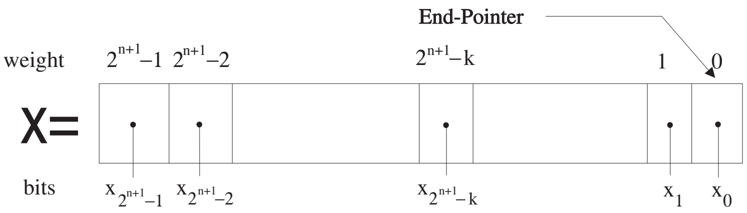


Fig. 1 (Original)

Recall that $2^{2^{n+1}} = 1 = 2^0$ in $\mathbb{I}_{F_n}$, thus *after* multiplication by 2^k we have

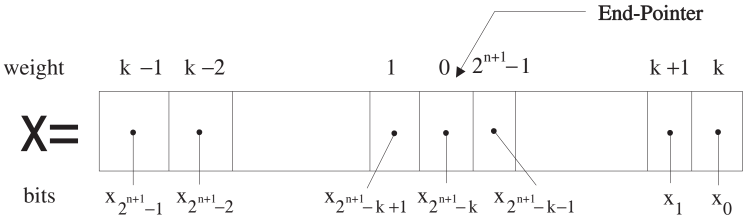


Fig. 2 (Intermediate)

where “weight”, of course, is the power of 2 in the representation. Finally,

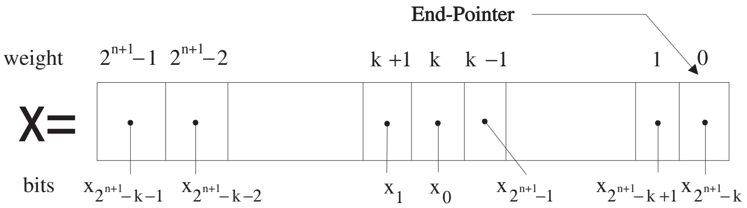


Fig. 3 (Final)

That is, to multiply by 2^k we *cyclically shifted* the representation of x by k places to the *right*, and this can be done in $O(2^{n+1})$ *bit operations* (i.e., as many as the *bit-length* of x) and this is $O(2^n)$ $(2^{n+1} = 2^n 2)$.

Actually, if we move the *end-pointer* by k positions to the *left*, rather than shifting the entire x to the right, we accomplice the operation in $O(k)$ *bit operations*, which does not depend on n.

Addition in $\mathbb{I}_{F_n}$ is trivially doable in $O(2^n)$ bit operations (observe that overflow of a carry is really addition of $2^{2^{n+1}}$, thus the carry will appear as a $+1$ in the least significant position). Finally, subtraction is multiplication by $-1 = 2^{2^n}$ followed by addition, thus also takes $O(2^n)$ bit operations.

13.1 FUNDAMENTAL Conclusion. *FFT in $\mathbb{I}_{F_n}$, where $w = 2$, a 2^{n+1}-th root of unity, can be done in $O(n2^n)$* integer multiplications.

Observing that each such multiplication has one operand (factor) of the form 2^k, *we conclude that FFT in $\mathbb{I}_{F_n}$ can be done in $O(n2^{2n})$* **bit operations***.*

Next, let us partition the numbers into contiguous blocks of “bits”, so that we have as many blocks in each partition as each block has bits.

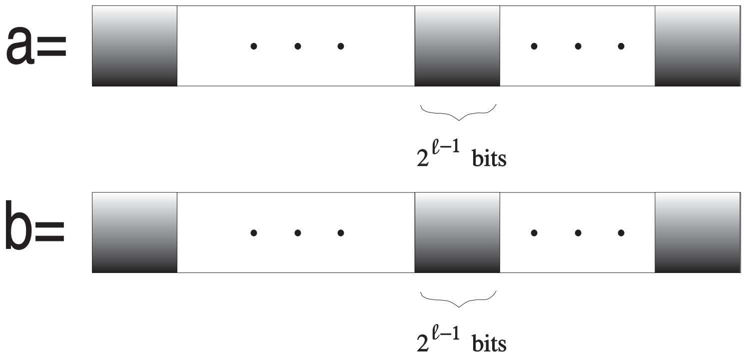


Fig. 4

There are two cases, $n = 2l - 1$ or $n = 2l$ for some $l \in \mathbb{N}$. Let us deal with the case $n = 2l - 1$, leaving the other out here as being similar (see however next section). Thus,

$$a = \sum_{i=0}^{2^l-1} a_i (2^{2^{l-1}})^i \quad \text{where } 0 \le a_i < 2^{2^{l-1}}, i = 0, \ldots, 2^l - 1 \tag{1}$$

and

$$b = \sum_{i=0}^{2^l-1} b_i (2^{2^{l-1}})^i \quad \text{where } 0 \le b_i < 2^{2^{l-1}}, i = 0, \ldots, 2^l - 1 \tag{2}$$

Consider also the polynomials associated with (1) and (2),

$$A(x) = \sum_{i=0}^{2^l-1} a_i x^i \tag{3}$$

and

$$B(x) = \sum_{i=0}^{2^l-1} b_i x^i \tag{4}$$

To find $c = a \cdot b$ amounts to

(i) Find $C(x) = A(x)B(x)$

(ii) Evaluate $C(x) = \sum_{i=0}^{2^l-1} c_i x^i$ at $x = 2^{2^{l-1}}$

From (1) and (2)

$$0 \le c_i < 2^{2^l+l}, i = 0, 1, \ldots, 2^n - 1$$

We know, from Example 12.2, how to do stage (i) in $O(l2^l)$ *integer multiplications* involving numbers that have binary *length* about 2^l. Here we represent $\mathbb{I}_{F_{l-1}}$ and $\mathbb{I}_{F_l}$ by the least non-negative representation. These moduli are sufficient to recover the c_i's by the CRT since $2^{2^l+l} < 2^{2^{l-1}+2^l} < F_{l-1}F_l$ for $l > 1$.

It is also instructive to note that a solution to

$$\begin{aligned} z &\equiv p \pmod{F_{l-1}} \\ z &\equiv q \pmod{F_l} \end{aligned} \tag{5}$$

is

$$z = F_l 2^{2^l-1}\big((p-q) \mod F_{l-1}\big) + q \text{ (Verify!)} \tag{6}$$

from which a z satisfying the system (5) and also $0 \le z < F_{l-1}F_l$ can be easily found, indeed in $O(1)$ multiplications, if the hardware can multiply numbers of the order $\approx 2^{2^l}$.

When we write $a = b \mod m$ we mean that a is the *remainder of the division*

$$\left\lfloor \frac{b}{m} \right\rfloor$$

i.e., for some q, $b = q \cdot m + a$ and $0 \le a < m$. As such, the above notation is *stronger* than the notation $a \equiv b \pmod{m}$.

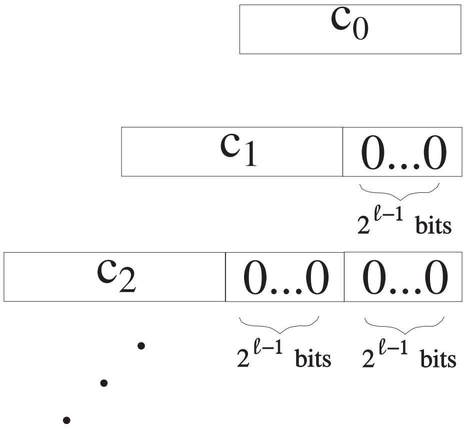
Item (ii) can be done the "brute force" way, i.e., as follows"

eval $\leftarrow c_0$
for $i = 1$ **to** $2^l - 1$ **by** 1 **do begin**
  eval $\leftarrow$ *eval* $+ c_i \cdot 2^{i2^{l-1}}$
end

Now multiplying a number r by $2^{2^{l-1}}$ (we are in $\mathbb{I}_{F_n}$ now—since the numbers "fit"; *not* in $\mathbb{I}_{F_l}$!) is the same as shifting the right-end pointer (i.e., the one that points to the least significant bit of r) 2^{l-1} positions to the left (we think of the representation of a number r in $\mathbb{I}_{F_n}$ as being on a "circular list"—review Figures 1–3 and the accompanying discussion).

This is a $O(2^{l-1})$ task in terms of *bit operations* (*not* integer multiplications).

The above algorithm is pictured as



Given that the binary length $\|c_i\|$ of the c_i's could be no larger than $2^l + l = O(2^l)$, and addition of two length l numbers takes no more than $l + 1$ bit operations, the above program—implemented according to the above figure—takes no more than

$$\begin{aligned} O(2^l(2^l + l + 1)) &= O(2^{2l}) \\ &= O(2^n) \quad \textit{bit operations}. \end{aligned} \tag{7}$$

Using as yardstick the venerable "school-method" integer multiplication which takes $O(d^2)$ *bit operations* (for numbers of binary length d) we recognize that the timing for stage (ii) is *better* (faster) than $O(2^n)$ *integer multiplications*. (What if we used Horner's method instead of the "brute force" algorithm? Would this be better? As good? Why?)

The whole task then is doable in $O(l2^l)$ integer multiplications *performed by the hardware on single-precision numbers* of length $< 2^l$.

A "practical" value of l is—as in 12.2—6, which makes $n = 11$. This scheme therefore is applicable to binary numbers of length up to 2^{10} (recall that there is padding), that is to numbers of *value*—in decimal notation—up to

17976931348623159077293051907890247336179769789423065 72
7343008115773267580550096313270847732240753602112011 3
8798713933576587897688144166224928474306394741243777 6
7893424865485276302219601246094119453082952085005768 8
3815068234246288147391311054082723716335051068458629 8
2399472459384797163048353563296242241372 15

or

$$1.7976931348623159 \times 10^{308}$$

Can we disengage the above scheme from hardware considerations? Well, let us revisit stage (i):

Say we work in $\mathbb{I}_{F_{l-1}}$ (the case for $\mathbb{I}_{F_l}$ is similar) We will do all our timing in terms of bit operations, since we do not care any more how the hardware does multiplications, nor do we want any more to constrain our timing analysis by restricting the size of l (and hence n) so that there are $O(1)$ "words" in F_{l-1}'s binary representation.

Stage (i) does:

(A) $FFT(a, 2, 2^l, 0)$ [This yields the vector, *in* $\mathbb{I}_{F_{l-1}}$, $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{2^l-1})$ in $O(l2^{2l})$ *bit operations*],

(B) $FFT(b, 2, 2^l, 0)$ [This yields the vector $\hat{b} = (\hat{b}_0, \ldots, \hat{b}_{2^l-1})$ in $O(l2^{2l})$ *bit operations*],

(C) Multiply out the vectors $\hat{a}, \hat{b}$ componentwise to obtain $\hat{c} = (\hat{c}_0, \ldots, \hat{c}_{2^l-1})$ in $2^l O((2^l)^2)$ *bit operations*, since there are 2^l number pairs to multiply, all in $\mathbb{I}_{F_{l-1}}$ hence of length $O(2^l)$.

It is understood that we apply the "school-method" $O(d^2)$*-multiplication for each pair—call it "method* M_0*—hence the charge of* $O((2^l)^2)$ *for each.*

So stage (C) costs $O(2^{3l})$ *bit operations*,

(D) $FFT(\hat{c}, 2^{-1}, 2^l, 0)$ [This yields the vector

$$\tilde{c} = (\tilde{c}_0, \ldots, \tilde{c}_{2^l-1})$$

in $\mathbb{I}_{F_{l-1}}$—i.e., $\tilde{c}(x)$—in $O(l2^{2l})$ *bit operations*].

The above timings are dominated by $O(2^{3l})$ *bit operations.*

Another $O(2^{3l})$ *bit operations* later (this time working in $\mathbb{I}_{F_l}$) and we obtain the vector

$$\tilde{\tilde{c}} = (\tilde{\tilde{c}}_0, \ldots, \tilde{\tilde{c}}_{2^{l+1}-1})$$

We recover the c_i's by the CRT using (6). That is from

$$\left.\begin{array}{l} c_i \equiv \tilde{c}_i \pmod{F_{l-1}} \\ c_i \equiv \tilde{\tilde{c}}_i \pmod{F_l} \end{array}\right. \text{for } i = 0, \ldots, 2^l - 1 \tag{8}$$

we get

$$c_i = F_l 2^{2^l-1}\big((\tilde{c}_i - \tilde{\tilde{c}}_i) \mod F_{l-1}\big) + \tilde{\tilde{c}}_i \tag{9}$$

It is known that if multiplication of two r-bit numbers takes $T(r)$ *bit operations*, then so does—within a constant factor—the division of a length $2r$ number by a length r number (see, for example, Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974).

Moreover, if $r_i = (\tilde{c}_i - \tilde{\tilde{c}}_i) \mod F_{l-1}$, then

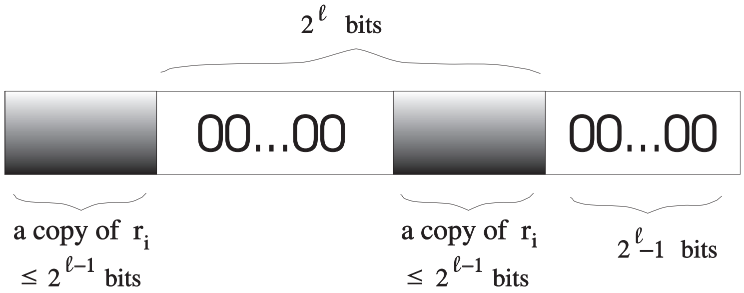
$$r_i = \begin{cases} \tilde{c}_i - \tilde{\tilde{c}}_i & \textbf{if } \tilde{c}_i - \tilde{\tilde{c}}_i \geq 0 \\ 2^{2^{l-1}} & \textbf{if } \tilde{c}_i = 0 \text{ and } \tilde{\tilde{c}}_i = 2^{2^l} \\ \big((2^{2^l} - 1 - \tilde{\tilde{c}}_i) + \tilde{c}_i\big) \mod F_{l-1} & \textbf{otherwise} \end{cases}$$

so that, by the previous remark, r_i is obtained in $O(2^{2l})$ *bit operations using procedure* M_0, and, of course, $\|r_i\| \leq 2^{l-1}$.

By (9),

$$c_i = (F_l 2^{2^l - 1} r_i + \tilde{\tilde{c}}_i) \mod F_{l-1} F_l$$

The $F_l 2^{2^l-1} r_i$'s look like the picture below, so that the c_i's are obtained (each) in $O(2^{2l})$ *bit operations* using M_0-multiplication (note that $\|F_l 2^{2^l-1} r_i + \tilde{\tilde{c}}_i\| \approx 2^{l-1} + 2^{l+1}$ while $\|F_{l-1} F_l\| \approx 2^{l-1} + 2^l$).



The entire task of multiplying the numbers a and b of length $N = 2^n$ therefore is complete in $O(2^{3l})$, i.e.,

$$O(N^{3/2}) \textit{ bit operations} \qquad (M_1)$$

Let us call this multiplication procedure M_1. We see that M_1 is slightly better than Karatsuba's $O(N^{\log_2 3})$ multiplication.

Suppose now that we upgrade the above integer multiplication procedure M_1, so that where it was (internally) using M_0 to do integer multiplications and divisions of binary numbers of length $O(2^l)$, it now uses M_1.

We call this upgraded version M_2. And upgraded it is, for the dominant timing term now is $2^l O((2^l)^{3/2}) = O(2^{5l/2})$ for an overall timing of

$$O(N^{5/4}) \textit{ bit operations} \qquad (M_2)$$

Similarly we can define a sequence of successively faster multiplication algorithms, M_i for $i = 0, 1, 2, \ldots$ †

† Schönhage and Straßen, in their original paper of 1970 have proposed a similar sequence of multiplication schemes, however they did FFT in $\mathbb{C}$.

An easy induction on i shows that M_i runs in $O(N^{(1+2^i)/2^i})$ bit operations for any two integers that have binary length N (it uses M_{i-1} integer multiplication "internally", whenever integer multiplications are called for).

How "practical" *is* M_i?

Let us look at $N = 2^{31}$ (each multiplicand's binary *length*—in decimal $N = 2147483648$). Then we obtain the following table, *on the simplifying assumption that the constants hidden in the Big-Oh notation are all equal* (normalized to 1 for comparison purposes). It is expected (although no experimentation was undertaken to verify this statement) that the Schönhage-Straßen recursive method has much more overhead (i.e., a higher multiplication constant in the $O(\ldots)$-timing) than any of the other methods (except, possibly, Karatsuba's).

Multiplication Comparison TABLE 1 $N = 2^{31}$	
Multiplication Method	**Result** (*bit operations*)
Schönhage-Straßen $O(N \log_2 N \log_2 \log_2 N)$	3.29811×10^{11}
"School-Method" $O(N^2)$	4.61169×10^{18}
Karatsuba $O(N^{\log_2 3})$	6.17673×10^{14}
M_1 $O(N^{1.5})$	9.95164×10^{13}
M_2 $O(N^{1.25})$	4.62288×10^{11}
M_3 $O(N^{1.125})$	3.1508×10^{10}

It is not surprising that for any *given length* N we can find an M_i-multiplication which, for numbers of length N, outperforms Schönhage-Straßen multiplication. This due to $\lim_{i\to\infty}(1 + 2^i)/2^i = 1$.

What *is* pleasantly surprising is that for numbers as large as 2^N, M_3 will run 10 times faster than Schönhage-Straßen multiplication.

The picture for numbers of smaller binary length is also promising for "practical" multiplication schemes M_i and is captured in TABLE 2 below:

Multiplication Comparison TABLE 2 $N = 400$	
Multiplication Method	**Result** (*bit operations*)
Schönhage-Straßen $O(N \log_2 N \log_2 \log_2 N)$	10758
"School-Method" $O(N^2)$	160000
Karatsuba $O(N^{\log_2 3})$	13309
M_1 $O(N^{1.5})$	8000
M_2 $O(N^{1.25})$	1789
M_3 $O(N^{1.125})$	846

14. Schönhage-Straßen Fast Integer Multiplication

In this section we show how the asymptotically fastest known integer multiplication scheme, due to Schönhage and Straßen, ticks.

On first impulse, we may think that pushing the M_i schemes to the limit will do the trick. Namely, instead of having M_{i+1} use M_i internally, why not use M_{i+1} itself (recursion!), in other words, why not devise a scheme M_∞ which uses M_∞ internally, when multiplications/divisions of length $O(2^l)$ numbers are called for.

This is the idea alright, and such a scheme would have a run-time function given by the recurrence

$$T(1) = 1 \text{ (a "nominal" constant)} \tag{1}$$
$$T(2^{2l}) = k2^l T(2^l) + O(l2^{2l}) \tag{2}$$

which is the "limiting case" of the equations that relate the timings of M_{i+1} and M_i ($T_{i+1}$and T_i respectively), namely

$$T_{i+1}(1) = 1 \text{ (a "nominal" constant)}$$
$$T_{i+1}(2^{2l}) = k2^l T_i(2^l) + O(l2^{2l})$$

k is the number of times we had to resort (in M_{i+1}) to "internal" integer multiplications of numbers of length 2^l, at a bit-cost $2^l T_i(2^l)$ each time. This happened twice when multiplying the $\hat{a}$ and $\hat{b}$ vectors obtained in $\mathbb{I}_{F_{l-1}}$ and $\mathbb{I}_{F_l}$, and again when doing the CRT algorithm to obtain the vector c in $\mathbb{I}$ (equivalently $\mathbb{I}_{F_n}$).

The $O(l2^{2l})$ term is due to the four invocations of FFT. So

$$k > 2 \tag{3}$$

The recurrence (1)–(2) can be rewritten, after division by 2^{2l} and setting

$$\hat{T}(2^x) \stackrel{\text{def}}{=} 2^{-x} T(2^x)$$

as

$$\hat{T}(1) = 1 \text{ (a "nominal" constant)} \tag{4}$$
$$\hat{T}(2^{2l}) = k\hat{T}(2^l) + O(l) \tag{5}$$

which *if* $k = 2$, has the solution

$$\hat{T}(2^l) = O(l \log l)$$

from which

$$T(2^l) = O(2^l l \log l)$$

i.e., the famous

$$T(N) = O(N \log N \log\log N) \tag{6}$$

For our $k > 2$ (following the method of the previous section) we get rather uninspiring solutions. For example, if $k = 3$, then $\hat{T}(2^l) = O(l^{\log_2 3})$, thus

$$T(2^l) = O(2^l l^{\log_2 3})$$

hence

$$T(N) = O(N(\log N)^{\log_2 3})$$

In general,

$$T(N) = O(N(\log N)^{\log_2 k})$$

Let us see how Schönhage and Straßen managed to get $k = 2$, and hence timing (6).

Let once more a, b be two N-digit numbers, written in *binary notation*. Thus, $0 \le a \le 2^N - 1$ and $0 \le b \le 2^N - 1$. As before, we will work in $\mathbb{I}_{F_n}$ for the smallest n such that $2N \le 2^n$, i.e., $n = \lceil \log_2 2N \rceil$ (this n can be computed in $O((\log N)^2)$ bit operations).

Therefore, the multiplication of two N-digit binary numbers in $\mathbb{N}$ is reduced to the following subproblems:

(a) For each a, b to be multiplied choose the appropriate $\mathbb{I}_{F_n}$, i.e.,
  (i) Find N (finding N from a—or b—can be done in $O(N)$ bit operations)
  (ii) Find $n = \lceil \log_2 2N \rceil$ ($O((\log N)^2)$ bit operations)
  Hence, for (a) the timing is $O(N)$; not to worry.

(b) Multiply a, b *doing all the arithmetic* in $\mathbb{I}_{F_n}$.

As before, let us split each of a, b, c in blocks of 2^{l-1} bits each. Let us consider the case $n = 2l$ here for a change. Thus,

$$a = \sum_{i=0}^{2^{l+1}-1} a_i (2^{2^{l-1}})^i, \qquad 0 \le a_i < 2^{2^{l-1}}$$

$$b = \sum_{i=0}^{2^{l+1}-1} b_i (2^{2^{l-1}})^i, \qquad 0 \le b_i < 2^{2^{l-1}}$$

Observe that $c = a \cdot b = \sum_{i=0}^{2^{l+1}-1} c_i (2^{2^{l-1}})^i$, where

$$0 \le c_m = \sum_{m=i+j} a_i b_j < 2^{l+1} \left(2^{2^{l-1}}\right)^2 = 2^{l+1+2^l} \tag{A}$$

Remarks. (1) $a \cdot b$ has $2^{l+1} - 1$ as upper bound in the summation since l is chosen so that the *result* $a \cdot b$ can "fit" in $\mathbb{I}_{F_n}$, where $n = 2l$ (subitem (ii) above).

(2) As before, once we have the c_m's (in $\mathbb{I}_{F_n}$), then to find c we need just to evaluate

$$c(x) = \sum_{i=0}^{2^{l+1}-1} c_i x^i \in \mathbb{I}_{F_n}[x]$$

at $x = 2^{2^{l-1}}$. We already saw that this is doable in $O(2^n)$ *bit operations.* So let us see how the c_m's can be obtained.

We set

$$a(x) = \sum_{i=0}^{2^{l+1}-1} a_i x^i$$
$$b(x) = \sum_{i=0}^{2^{l+1}-1} b_i x^i$$

Then $c(x) = a(x) \cdot b(x)$.

We work in $\mathbb{I}_{F_l}$, doing FFT, where 2 is a 2^{l+1}-th primitive root and each of $a(x), b(x), c(x)$ have 2^{l+1} terms, thus FFT can be used to find the c_m's *in* $\mathbb{I}_{F_l}$, *that is, to find* $\tilde{c}_m$*'s such that for all* m, $c_m \equiv \tilde{c}_m \pmod{F_l}$.

As we encountered already, there is a problem: By (A), $0 \le c_m < 2^{l+1+2^l}$ while, of course, $0 \le \tilde{c}_m \le 2^{2^l}$. Thus, $c_m \equiv \tilde{c}_m \pmod{F_l}$ *does not in general imply that* $c_m = \tilde{c}_m$.

So, although we got the $\tilde{c}_m$'s in $\mathbb{I}_{F_l}$, we still have *not* got the c_m's in $\mathbb{I}_{F_n}$!

CRT to the rescue. Here we deviate from our earlier solution (where we repeated our modular solution in $\mathbb{I}_{F_{l+1}}$). Suppose that along with the $\tilde{c}_m$'s in $\mathbb{I}_{F_l}$ $(0 \le \tilde{c}_m < 2^{2^l} + 1)$ we have also found $\tilde{\tilde{c}}_m$'s, this time $\bmod 2^{l+1}$ (i.e., for all m, $c_m \equiv \tilde{\tilde{c}}_m \pmod{2^{l+1}}$).

Then we can also find *a unique* $\hat{c}_m$ (for each m) such that

$$0 \le \hat{c}_m < 2^{l+1}(2^{2^l} + 1) \tag{B}$$
$$\hat{c}_m \equiv \tilde{c}_m \pmod{F_l} \tag{C}$$
$$\hat{c}_m \equiv \tilde{\tilde{c}}_m \pmod{2^{l+1}} \tag{D}$$

(**NB.** Trivially, 2^{l+1} and $2^{2^l} + 1$ are relatively prime.)

By uniqueness, since $0 \le c_m < 2^{l+1+2^l} < 2^{l+1} F_l$, and by $(B), (C)$ and (D) above, it follows that $c_m = \hat{c}_m$ *in* $\mathbb{I}$.

To see how to obtain the above ingredients we need the following:

14.1 Lemma. *The system*

$$
\begin{aligned}
z &\equiv a \pmod{F_l}, \qquad 0 \le a < F_l \\
z &\equiv b \pmod{2^{l+1}}, \qquad 0 \le b < 2^{l+1} \\
0 \le z &< 2^{l+1} F_l
\end{aligned}
$$

has a unique z-solution, which can be computed, in $\mathbb{I}$, *in* $O(2^l)$ bit operations.

Proof. (1) Verify—observing that $F_l \equiv 1 \pmod{2^{l+1}}$ and working with least non-negative representation throughout—that

$$u = F_l\big((b-a) \mod 2^{l+1}\big) + a$$

will satisfy the two congruences (compare with the similar situation in the previous section). Moreover, u satisfies

$$0 \le u < F_l 2^{l+1} + F_l = 2^{2^l+l+1} + 2^{2^l} + 2^{l+1} + 1 < 2^{2^l+l+2} < 2F_l 2^{l+1}$$

Thus,

$$z = \begin{cases} u & \textbf{if } u < 2^{l+1}F_l \\ u - 2^{l+1}F_l & \textbf{otherwise} \end{cases}$$

satisfies the two congruences, *and* the inequality constraint.

(2) Also observe that we can get a least non-negative $v = \big((b-a) \mod 2^{l+1}\big)$ by

$$v = \begin{cases} b-a & \textbf{if } b \ge a \\ (2^{2^l} - a + b) \mod 2^{l+1} & \textbf{otherwise} \end{cases}$$

(3) Addition of numbers up to 2^{2^l} takes $O(2^l)$ bit operations, while taking remainders of divisions by 2^{l+1} takes $O(l)$ bit operations. Also, multiplying any r by F_l is to multiply by 2^{2^l} and then add r. So, for $\|r\| = O(2^l)$, it also takes $O(2^l)$ bit operations. □

We now resume our task of finding $\tilde{c}_m$ and $\tilde{\tilde{c}}_m$ such that

$$
\begin{aligned}
c_m &\equiv \tilde{c}_m \pmod{F_l} \qquad (i) \\
c_m &\equiv \tilde{\tilde{c}}_m \pmod{2^{l+1}} \qquad (ii)
\end{aligned}
$$

for $m = 0, \ldots, 2^{l+1} - 1$.

Now, (i) can be done as before, using the FFT, namely

- $FFT(a, 2, 2^{l+1}, 0) \to \hat{a}$: in $O(l2^{2l})$ bit operations.
- $FFT(b, 2, 2^{l+1}, 0) \to \hat{b}$: in $O(l2^{2l})$ bit operations.
- Multiply the $\hat{a}, \hat{b}$ *componentwise* to obtain $\hat{c}$ (in $\mathbb{I}_{F_l}$): in $2^{l+1}M(2^l)$ bit operations. Here there are 2^{l+1} pairs of numbers of length $\le 2^l$ to multiply.

We use the Schönhage-Straßen method *recursively* to do these multiplications ($M(\dots)$ is the timing function of the under description procedure).

- $FFT(\hat{c}, 2^{-1}, 2^{l+1}, 0) \to \tilde{c}$: in $O(l2^{2l})$ bit operations.

Let us turn to (ii):

(1) Compute $\tilde{\tilde{a}}_i$ and $\tilde{\tilde{b}}_i$, for $i = 0, \dots, 2^{l+1} - 1$, so that

$$\begin{aligned} \tilde{\tilde{a}}_i &= a_i \mod 2^{l+1} \\ \tilde{\tilde{b}}_i &= b_i \mod 2^{l+1} \end{aligned} \tag{iii}$$

For each i, (iii) takes $O(l)$ bit operations. Let also

$$\begin{aligned} u &= \sum_{i=0}^{2^{l+1}-1} \tilde{\tilde{a}}_i 2^{(3l+3)i} \\ v &= \sum_{i=0}^{2^{l+1}-1} \tilde{\tilde{b}}_i 2^{(3l+3)i} \end{aligned} \tag{iv}$$

Clearly,

$$u \cdot v = \sum_{i=0}^{2^{l+1}-1} \gamma_i 2^{(3l+3)i} \text{ and } \gamma_m = \sum_{m=i+j} \tilde{\tilde{a}}_i \tilde{\tilde{b}}_j, m = 0, \dots, 2^{l+1} - 1$$

Observe that

$$\begin{aligned} u \text{ (and } v) &< \sum_{i=0}^{2^{l+1}-1} (2^{l+1} - 1) 2^{(3l+3)i} \\ &= (2^{l+1} - 1) \frac{2^{(3l+3)2^{l+1}} - 1}{2^{3l+3}} \\ &< 2^{(3l+3)2^{l+1}} \end{aligned}$$

Same for v.

Also, $0 \le \gamma_m < 2^{l+1}(2^{l+1})^2 = 2^{3l+3}$ for $m = 0, \dots, 2^{l+1} - 1$.

We draw two conclusions:

(1) The γ_m's can be "read off" the bit representation of $u \cdot v$. Indeed, each block of $3l + 3$ bits (starting from the right end—i.e., $m = 0$, say) is a γ_m (it has, of course, $2l + 2$ leading 0's).

(2) Since, as we just saw, $\|u\| = O(l2^l)$ and $\|v\| = O(l2^l)$, any $O(n^{1+\epsilon})$ integer multiplication scheme ($0 < \epsilon < 1$) will yield $u \cdot v$ in $O((l2^l)^{1+\epsilon}) = O(2^{2l})$ *bit operations.*

Putting (i) and (ii) together we have a timing recurrence

$$\begin{aligned} M(1) &= 1 \text{ (a nominal constant)} \\ M(2^{2l}) &= 2 \cdot 2^l M(2^l) + O(l2^{2l}) \end{aligned}$$

which as we have already remarked leads to a "closed" form $M(2^l) = O(2^l l \log l)$, or $M(N) = O(N \log N \log\log N)$.