Arithmetic – Multiplication, DFT, FFT

In this lecture, we study the bit-complexity of various fundamental arithmetic operations for integers and polynomials. This lecture is based upon [2, Chap. 1] .

¶1. Representation. The computational model that we will use throught is the Random Access Machine (RAM) with one difference. A RAM consists of unbounded memory cells where each cell can store an arbitrary integer. Since we are interested in bit-complexity, we will restrict the second condition, namely the cells can contain integers of absolute value smaller than some constant β. The standard instruction set remains unchanged. An **integer** a **will be represented in base** β as

$$a = s(a_{n-1}\beta^{n-1} + a_{n-2}\beta^{n-2} + \cdots + a_0)$$

where $s \in \{+1, -1\}$ and $a_i \in \{0, \dots, \beta - 1\}$. The number of bits n needed to represent a in base β is bounded by $\log_\beta |a|$. If $\beta = 1$ then we get the unary representation of a. However, for our purposes we will always assume $\beta > 1$; in particular, a power of 2. Then n is the bit-length of a, denoted by $\mathrm{len}(a) = O(\log |a|)$; we do not emphasize upon β, since $\mathrm{len}(a)$ differs by a constant for all $\beta > 1$. We will often write the bit-representation of a as the n-tuple $\mathbf{a} := (a_{n-1}, \dots, a_0)$. A degree n polynomial over a ring R, where R can be $\mathbb{C}, \mathbb{R}, \mathbb{Z}$ in our setting, is given as $f(x) = \sum_{i=0}^{n} a_i x^i$, where $a_i \in R$, and $a_n \neq 0$. Define $\|f\|_\infty := \max\{|a_0|, |a_1|, \dots, |a_n|\}$, as the infinity-norm of f. The set of all univariate polynomials over R is denoted as $R[x]$. The coefficient representation will be succinctly written as C-rep.

We study computational complexity of the four basic operations on integers and polynomials: addition, subtraction, multiplication and division with remainder. The classical algorithms for adding and subtracting two integers a, b have optimal complexity, namely $\max\{\mathrm{len}(a), \mathrm{len}(b)\}$. Similarly, the bit-complexity of adding or subtracting two integer polynomials $a(x)$, $b(x)$ with degree m and n resp is $O(\min\{m, n\} \max\{\|a\|_\infty, \|b\|_\infty\})$. These algorithms are optimal and therefore we only focus on multiplication and division with remainder. In this lecture we deal with the first of these two operations.

1 A question of Representation

Our input is two n-bit numbers $a = (a_{n-1}, \dots, a_0)$, $b = (b_{n-1}, \dots, b_0)$. We want to compute their product $c := a \times b$. Let $M(n)$ denote the bit-complexity of multiplying two n-bit numbers. We know that for the classical algorithm for multiplying two n-bit numbers takes $M(n) = \Theta(n^2)$. Here we will describe the subsequent improvements.

An integer $(a_{n-1}, \dots, a_0)$ represented in base β has a natural correspondence with the polynomial $a(x) := \sum_{i=0}^{n-1} a_i x^i$, since $a = a(\beta)$. Thus to multiply a and b we can compute $c(x) := a(x) \times b(x)$, and then evaluate $c(x)$ at β to get the desired product c. Thus all the algorithms for computing integer products turn out to be algorithms for polynomial multiplication, which is what we focus on. Moreover, we focus on algebraic complexity for the moment.

The classical algorithm for multiplying two degree n integer polynomials takes $\Theta(n^2)$ algebraic operations. But is this complexity intrinsic to polynomial multiplication or is it an artefact of our representation? Can we think of a representation where multiplication can be done more efficiently? Yes! In fact, we can do it constant time. From the FTA we know that any polynomial $a(x) \in \mathbb{C}[x]$ can be expressed as $a(x) = a_n \prod_{i=1}^{n}(x - \alpha_i)$, where α_i's are the roots of $a(x)$. Given two polynomials with their list of roots, the roots of the product is just the concatenation of the two lists. But what about evaluation and addition? Well, evaluation can be done in linear time, but for addition we don't see a straightforward algorithm. However, computing the roots of a polynomial is a challenging task in the first place. We need a representation where multiplication can be done efficiently, i.e. better than quadratic time. From FTA we know that if two degree n polynomials are equal at $n + 1$ points then they are in fact identical. Thus an alternative representation for $a(x)$ is the set of $n + 1$ pairs $\{(x_0, a(x_0)), \dots, (x_n, a(x_n))\}$, where $x_i \in \mathbb{C}$ are $n + 1$ distinct

complex numbers. Call this representation as E-rep. Addition of two polynomials is clearly linear. Multiplication is also linear with the caveat that we need $2n+1$ pairs in our tuple. The drawback of E-rep, ironically, is evaluation. To evaluate a polynomial in E-rep we need to convert it to C-rep. This conversion step is called the **interpolation** step, and one can either use Lagrange's or Newton's interpolation to get the polynomial. For instance, using Lagrangian interpolation we know that given (x_i, y_i), $i = 0, \ldots, n$, x_i's distinct the polynomial $a(x)$ such that $a(x_i) = y_i$ is given as

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

The interpolation step takes $O(n^2)$ operations. To summarize: C-rep takes $O(n)$ time for addition and evaluation, but $O(n^2)$ for multiplication; E-rep takes $O(n)$ time for addition and multiplication, but $O(n^2)$ for evaluation. If only we could convert between the two reps in better than quadratic time we could utilize their advantages fully. What we have not yet utilized is our freedome to choose the evaluation points.

¶2. Converting between E-rep and C-rep. The evaluation of a degree n polynomial $f = \sum_{i=0}^{n} a_i x^i$ at the points $x_0, \ldots, x_n$ can be expressed using matrices as

$$[f(x_0), f(x_1), \ldots, f(x_n)]^t = V(x_0, \ldots, x_n)[a_0, \ldots, a_n]^t$$

where V is the Vandermonde matrix, i.e., the ith row is x_i^j, $j = 0, \ldots, n-1$. Assuming we have precomputed V, this conversion takes quadratic time as we have to compute a matrix-vector product. Can we do better? We can do the following divide-and-conquer approach: express

$$f(x) := f_e(x^2) + x f_o(x^2)$$

where f_e is the collection of even terms of f and f_o the odd terms; e.g., if $f(x) = x^5 + x^4 - x^3 + x^2 + x + 1 = x(x^4 - x^2 + 1) + x^4 + x^2 + 1$ then $f_o = x^2 - x + 1$ and $f_e = x^2 + x + 1$. The degrees of f_e and f_o are at most $n/2$. Does it help to reduce our evaluation complexity? To compute f at x_i, we have to compute f_e and f_o at x_i^2; but there are still $n+1$ *distinct values* x_i^2 and computing the two polynomials takes $O(n)$ operations for each value. So we haven't gained any thing. If, however, there were only $n/2$ distinct values x_i^2 then we could recurse the computation on degree $n/2$ polynomials f_e and f_o at these $n/2$ points, which would yield us the recursion:

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

where $T(n)$ is the time to evaluate f at $n+1$ distinct values $x_0, \ldots, x_n$. Where can we find $n+1$ numbers, $x_0, \ldots, x_n$, such that squaring them gives us only $n/2$ distinct values?

¶3. Roots of Unity. The roots of the equation $x^n = 1$ are called the nth roots of unity. Let $\omega := \exp(2\pi i/n)$. Then the n roots are given as ω^k, $k = 0, \ldots, n-1$; ω is called a primitive nth root of unity since $\omega^k \neq 1$ for $0 < k < n$; in fact, if $\omega^k = \omega^j$, for $0 \leq k < j < n$ the $\omega^{j-k} = 1$, which implies $n | j - k$, a contradiction. Do the roots of unity have the desired property? Yes, since ω^{2k}, $0 \leq k < n/2$, are the same as ω^k, $n/2 \leq k < n$. Not only that the squares of ω^k, $k = 0, \ldots, n-1$, are also $n/2$th roots of unity.

Let's rewrite the map $V \cdot \mathbf{a}$, as $\mathbf{A} := \mathrm{DFT}_n(\mathbf{a})$, where $\mathrm{DFT}_n := V(\omega)$, the Vandermonde matrix corresponding to the $(n-1)$th primitive root of unity ω. We will call the vector $\mathbf{A}$ as the **discrete fourier transform** of $\mathbf{a}$. The **inverse discrete fourier transform** of $\mathbf{A}$ is the vector $\mathbf{a} := \mathrm{DFT}_n^{-1}(\mathbf{A})$. We now have the following claim:

THEOREM 1. *The DFT and inverse DFT can be computed in $O(n \log n)$ arithmetic operations in $\mathbb{C}$.*

Proof. The following algorithm computes the DFT of a vector in $\mathbb{C}^n$.

`Fast Fourier Transform Algorithm`
INPUT: An vector $\mathbf{a} \in \mathbb{C}^n$, and ω an nth root of unity.
OUTPUT: $\mathrm{DFT}_n \cdot \mathbf{a}$.
1. Let $a_e(x), a_o(x)$ be the polynomials corresponding to the polynomial $\sum_{i=0}^{n-1} a_i x^i$.
2. Evaluate $a_e(x^2)$ and $a_o(x^2)$ at $x = \omega^k$, $k = 0, \ldots, n-1$.
3. Compute $\omega^k a_o(\omega^{2k})$, $k = 0, \ldots, n-1$.
4. Return the vector with the kth entry as $a_e(\omega^{2k}) + \omega^k a_o(\omega^{2k})$, $k = 0, \ldots, n-1$.

Let $T(n)$ be the time to compute $\mathrm{DFT}_n(\mathbf{a})$. Since ω^{2k}, $k = 0, \dots, n-1$, are $n/2$th roots computing $a_e(\omega^{2k})$ and $a_o(\omega^{2k})$ is the same as $\mathrm{DFT}_n(\mathbf{a}_e)$ and $\mathrm{DFT}_n(\mathbf{a}_o)$ resp. Then, by our earlier observations, we have the recursion $T(n) = 2T(n/2) + O(n)$, where the $O(n)$ operations are required in setp 3 and 4 of the algorithm.

For the inverse DFT we observe that $\mathrm{DFT}_n\mathrm{DFT}_n{-1} = \mathrm{DFT}_{n-1}\mathrm{DFT}_n = nI$. This follows from the cancellation property of the roots of unity: for any $s \in \mathbb{Z}$

$$\sum_{k=0}^{n-1} \omega^{ks} = \begin{cases} 0 & \text{if } s \mod n \neq 0 \\ n & \text{if } s \mod n = 0. \end{cases} \tag{1}$$

Q.E.D.

The overall algorithm for multiplying two degree n polynomials is the following:

INPUT: Two degree n polynomials $a(x), b(x) \in \mathbb{C}[x]$.
OUTPUT: The degree $2n$ polynomial $a(x) \cdot b(x)$.
1. Compute $\mathbf{A} := \mathrm{DFT}_{2n}(\mathbf{a})$ and $\mathbf{B} := \mathrm{DFT}_{2n}(\mathbf{b})$.
2. Let $\mathbf{C}$ be the pointwise multiplication of $\mathbf{A}$ with $\mathbf{B}$.
3. Return $\mathrm{DFT}_{2n}^{-1}(\mathbf{C})$.

Thus polynomial multiplication is reduced to two DFTs and one inverse DFT, and hence can be done in $O(n \log n)$ arithmetic operations in $\mathbb{C}$.

2 Modular DFT and FFT

What about multiplying integer polynomials? One approach, which Strassen took, was to use complex roots of unity, as above, but with enough accuracy; this yielded a time bound that satisfied the recursion $T(n) = O(nT(\log n))$; in particula, this implies $T(n) = O(n \log n (\log \log n)^{1+\epsilon})$. Schönhage and Strassen later improve the bound to $O(n \log n \log \log n)$. The improvement is mostly theoretical, however, it is their introduction of modular DFT to avoid approximate arithmetic was a breakthrough.

Recall that we wanted the following two properties from the nth roots of unity:

(1) Squaring the nth roots of unity give us the $n/2$th roots of unity and

(2) Cancellation property as in (1).

If an arbitrary ring R has such nth roots of unity, then we can define the DFT operation and carry out the FFT algorithm. Since $\mathbb{Z}$ does not have roots of unity, except for 1, we need to work with a discrete ring that has roots of unity. The key idea of Schönhage-Strassen was that the ring $\mathbb{Z}_M$, where $M := 2^L + 1$, has the desired roots of unity, and FFT can be implemented for this ring; L will be an integer multiple of some power of two. From the choice of M it immediately follows that $2^{2L} \equiv 1 \mod M$. Let $n := 2^{k+1}$, be such that n divides $2L$. Further define $\omega := 2^{2L/n}$. Then we have the folowing result.

LEMMA 2. *The number ω is a primitive nth root of unity in $\mathbb{Z}_M$, i.e., ω^j, $j = 0, \dots, n-1$, are all solutions of the equation $x^n \equiv 1 \mod M$.*

Proof. We have to show that $\omega^j \not\equiv 1 \mod M$, for $j < 2K$. There are two cases to consider:

1. If $j \leq K$ then $\omega^j = 2^{jL/K} \leq 2^L < M$. Therefore, $\omega^j \not\equiv 1 \mod M$.

2. If $j > K$ then ...

Q.E.D.

We will now demonstrate the two properties needed for teh FFT algorithm. The "squaring-halving" property is easily seen, since the square of the first-half roots ω^j, $0 \leq j < n/2$, is equal to the square of the second-half roots, ω^j, $n/2 \leq j < 2K$; also, ω^2 is a $n/2$th root of unity. Thus we only demonstrate the cancellation property.

$$\sum_{j=0}^{n-1} \omega^{js} = \begin{cases} 0 & \text{if } s \mod n \neq 0 \\ n & \text{if } s \mod n = 0. \end{cases} \tag{2}$$

The second case is straightforward, since if $s = nt$, then $\omega^{js} = \omega^{2Ljt} \equiv 1 \mod M$. Otherwise, let $s \equiv 2^p q \mod n$, where q is odd. Define $r := n/2^p$; since $2^p q < n$, this is well defined. We now break the sum into 2^p parts each containing r terms.

$$\sum_{j=0}^{n-1} \omega^{js} = \sum_{j=0}^{r-1} \omega^{js} + \sum_{j=r}^{2r-1} \omega^{js} + \cdots + \sum_{j=n-r}^{n-1} \omega^{js}.$$

Pulling out the smallest term from each of the sums on the RHS we get

$$\sum_{j=0}^{n-1} \omega^{js} = \sum_{j=0}^{r-1} \omega^{js} + \omega^{rs} \sum_{j=0}^{r-1} \omega^{js} + \cdots + \omega^{rs(2^p-1)} \sum_{j=0}^{r-1} \omega^{js}.$$

Since $\omega^{rs} \equiv \omega^{2^p qr} = \omega^{nr} \equiv 1$ we further obtain

$$\sum_{j=0}^{n-1} \omega^{js} \equiv 2^p \sum_{j=0}^{r-1} \omega^{js}.$$

Also note that $\omega^{rs/2} \equiv \omega^{Kq} \equiv (-1)^q \equiv -1$, as q is odd. Thus

$$\sum_{j=0}^{n-1} \omega^{js} = \sum_{j=0}^{r/2-1} (\omega^{js} + \omega^{s(j+r/2)}) \equiv \sum_{j=0}^{r/2-1} (\omega^{js} - \omega^{sj}) = 0.$$

The cancellation property implies that $\text{DFT}^{-1} = V(\omega)^{-1} = V(\omega^{-1})/n$; note that n is invertible in $\mathbb{Z}_M$ since M is odd and so n is relatively prime to M.

We now have all the ingredients to implement FFT in $\mathbb{Z}_M$. The algorithm is as before. The input is a vector $\mathbf{a} \in \mathbb{Z}_M^n$, or a degree $n-1$ polynomial in $\mathbb{Z}_M[x]$, and the output is $\text{DFT}_n(\mathbf{a})$; similarly for the inverse. We have the following result equivalent to Theorem 1.

THEOREM 3. *The modular DFT and modular inverse DFT can be computed in $O(nL \log n)$ bit-operations.*

The recursion is $T(n) = 2T(n/2) + O(nL)$. The $O(nL)$ cost comes from doing $O(n)$ additions modulo M; each addition involves working with numbers of bit-length L.

Now that we have our modular DFT, let's see how to implement Fast Integer Multiplication. Here we present a simplification of Schönhage-Strassen algorithm.

3 Fast Integer Multiplication

Goal to compute product of two N-bit binary numbers a, b. We will reduce the problem recursively into a problem of multiplying numbers of bit-length $\sqrt{N}$. Suppose $N = 2^k$ (if not pad it the numbers with zeros) and define

$$n := 2^{\lfloor k/2 \rfloor} \text{ and } \ell := N/b.$$

Thus $n\ell = N$, both n and ℓ are $\Theta(\sqrt{N})$, and n divides ℓ (since $n \leq \ell$).

We express a as a binary number containing n blocks of ℓ bits each, i.e.,

$$a = \sum_{i=0}^{n-1} A_i 2^{\ell i},$$

where $0 \leq A_i < 2^\ell$. Similarly, express $b := \sum_{i=0}^{n-1} B_i 2^{\ell i}$. The product ab can be expressed similarly in terms of a $2n-1$ dimensional vector $\mathbf{c} = (C_0, \ldots, C_{2n-2})$, where

$$ab = \sum_{i=0}^{2n-2} C_i 2^{\ell i} \tag{3}$$

such that

$$C_i := \sum_{j=0}^{i} A_j B_{i-j}. \tag{4}$$

How large can C_i's be? The summation index is bounded by n and the $A_j, B_j < 2^\ell$. Thus $C_i < n2^{2\ell} < 2^{3\ell}$. Thus we choose our modulus $M := 2^{3\ell} + 1$, and define $L := 3\ell$. Our primitive root of unity $\omega = 2^{6\ell/n}$, which is an nth primitive root in M.

To compute the product ab we proceed just as in the multiplication algorithm given earlier, except after doing the modular inverse DFT we evaluate the resulting polynomial at 2^ℓ, i.e. compute the sum $\sum_{i=0}^{2n-1} C_i 2^{\ell i}$. The bit-complexity of the modular DFT and inverse DFT is $O(nL \log n)$. The componentwise product of the vectors obtained by $\mathrm{DFT}_{2n}(\mathbf{a})$ and $\mathrm{DFT}_{2n}(\mathbf{b})$ recursively has bit-complexity $2nM(L)$. What is the cost of computing the sum $\sum_{i=0}^{2n-1} C_i 2^{\ell i}$? Notice that since $C_i < 2^{3\ell}$, it contains at most three blocks C_{i0}, C_{i1}, C_{i2} each of bit-length ℓ. Consider the sum $C_0 + 2^\ell C_1 + 2^{2\ell} C_2$. In this sum at most three blocks can overlap, namely C_{02}, C_{11}, C_{22}, and thus the cost of performing this addition depends only on the bit-length of $C_0 + 2^\ell C_1 + 2^{2\ell} C_2$, which is $O(L)$. Thus computing $\sum_{i=0}^{2n-1} C_i 2^{\ell i}$ has $O(nL)$ bit-complexity, which is dominated by the cost of doing a DFT. We thus have the following recursion:

$$M(N) \le 2nM(L) + O(nL \log n) = 2nM(3\ell) + O(n\ell \log n).$$

Recall that $n \le \sqrt{N}$, $\ell \le \epsilon\sqrt{N}$, and $n\ell = N = 2^k$, thus

$$M(N) \le 2\sqrt{N} M(3\sqrt{N}) + O(N \log N).$$

Let $t(k) := M(2^k)/2^k$. Then dividing the equation above by N we obtain

$$t(k) = 6t(k/2) + O(k).$$

This can be solved to obtain $t(k) = k^{\lg 6}$. Substituting back, we obtain $M(N) = O(N(\log N)^{\lg 6})$.

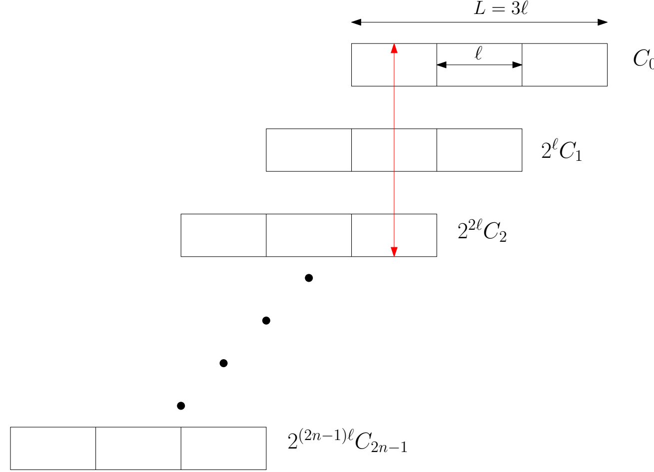

Figure 1: Computing $\sum_{i=0}^{2n-1} C_i 2^{\ell i}$. Atmost three blocks overlap.

References

[1] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.