Computational Number Theory and Algebra	May 2, 2012
Lecture 5	
Lecturers: Markus Bläser, Chandan Saha	*Scribe: Chandan Saha*

In the previous lecture, we saw how to use FFT to multiply two polynomials in $\mathcal{R}[x]$ with degree less than $n/2$ using $O(n \log n)$ operations in $\mathcal{R}$. This is a significant improvement over the naïve polynomial multiplication algorithm that runs in $O(n^2)$ time over $\mathcal{R}$. But, to achieve this improvement it is crucial that $\mathcal{R}$ has a principal n^{th} root of unity. In today's class, we will see how to attach a 'virtual' root of unity to $\mathcal{R}$, if $\mathcal{R}$ doesn't have such a root to begin with. We will see this idea at work in an integer multiplication algorithm which we discuss next. The topics for today's discussion are:

- Integer multiplication via polynomial multiplication,
- Reducing polynomial division to polynomial multiplication,

1 Integer multiplication via polynomial multiplication

We want to design an asymptotically efficient algorithm to multiply two N-bit integers a and b. Once again, a naïve integer multiplication algorithm takes $O(N^2)$ bit operations. We want to do significantly better than this complexity. Let $a = \sum_{i=0}^{N-1} a_i 2^i$ and $b = \sum_{i=0}^{N-1} b_i 2^i$, where $a_{N-1}a_{N-2}\ldots a_0$ and $b_{N-1}b_{N-2}\ldots b_0$ are the binary representations of a and b, respectively. Assume without loss of generality that N is a power of 2. Further, for the sake of simplicity of exposition, we will assume that N is a number of the form 2^{2^ℓ} - this is just to ensure that $N^{\frac{1}{2^j}}$ is an integer for every $j \leq \ell$. One can certainly avoid making this second assumption by using appropriate 'ceil' and 'floor' notations.

Split each of the two binary numbers into blocks of size $\sqrt{N}$ bits, and write them as, $a = \sum_{i=0}^{\sqrt{N}-1} A_i \cdot 2^{\sqrt{N}\cdot i}$ and $b = \sum_{i=0}^{\sqrt{N}-1} B_i \cdot 2^{\sqrt{N}\cdot i}$, respectively, where A_i and B_i are $\sqrt{N}$-bit numbers i.e. $0 \leq A_i, B_i \leq 2^{\sqrt{N}} - 1$. Consider the polynomials $A(x) = \sum_{i=0}^{\sqrt{N}-1} A_i \cdot x^i$ and $B(x) = \sum_{i=0}^{\sqrt{N}-1} B_i \cdot x^i$. Now notice that the product $a \cdot b$ is equal to the product of the polynomials $A(x)$ and $B(x)$ evaluated at $2^{\sqrt{N}}$, i.e. $a \cdot b = A(2^{\sqrt{N}}) \cdot B(2^{\sqrt{N}})$. This simple observation suggests an integer multiplication algorithm *via* polynomial multiplication: encode the integers as polynomials, multiply the polynomials using FFT, and finally evaluate the product polynomial at $2^{\sqrt{N}}$.

But, there is an issue here. The polynomials $A(x)$ and $B(x)$ have degree less than $\sqrt{N}$. So, by Lemma 4 of the previous lecture, we need a principal $2\sqrt{N}$-th root of unity in the underlying ring (in order to multiply these two polynomials using FFT). The coefficients of these polynomials are integers in the range $[0, 2^{\sqrt{N}} - 1]$. Although, $\mathbb{Z}$ does not contain a $2\sqrt{N}$-th root of unity, the ring $\mathbb{Z}/(2^{\sqrt{N}} + 1)$ does indeed contain such a root - because, the element 2 is a principal $2\sqrt{N}$-th root of unity in $\mathbb{Z}/(2^{\sqrt{N}} + 1)$ (why?). Why not pretend that the polynomials $A(x)$ and $B(x)$ are polynomials over the ring $\mathbb{Z}/(2^{\sqrt{N}} + 1)$ (as the coefficients A_i and B_i are anyway less than $2^{\sqrt{N}}$), and multiply them over this ring using FFT? The only problem is that the product polynomial $C(x) = A(x) \cdot B(x)$, might have coefficients as large as (about) $\sqrt{N} \cdot 2^{2\sqrt{N}}$ - which means, some of the coefficients of $C(x)$ might end up being different numbers in the ring $\mathbb{Z}/(2^{\sqrt{N}} + 1)$ i.e. when we go modulo $(2^{\sqrt{N}} + 1)$. There is a relatively easy fix for this - instead of working with the ring $\mathbb{Z}/(2^{\sqrt{N}} + 1)$, work with the ring $\mathcal{R} = \mathbb{Z}/(2^{3\sqrt{N}} + 1)$. In the ring $\mathcal{R}$, the element $2^3 = 8$ is a principal $2\sqrt{N}$-th root of unity. Moreover, the coefficients of $C(x)$ remain unchanged in $\mathcal{R}$, as $2^{3\sqrt{N}} + 1 > \sqrt{N} \cdot 2^{2\sqrt{N}}$ for any $N \geq 1$. This suggests the following integer multiplication algorithm.

Algorithm 1 Integer multiplication using FFT

1. `Encode integers` a `and` b `as polynomials` $A(x) = \sum_{i=0}^{\sqrt{N}-1} A_i x^i$ `and` $B(x) = \sum_{i=0}^{\sqrt{N}-1} B_i x^i$.
2. `Multiply` $A(x)$ `and` $B(x)$ `over the ring` $\frac{\mathbb{Z}}{(2^{3\sqrt{N}}+1)}$ `using 8 as the` $2\sqrt{N}$`-th root of unity.`
3. `Evaluate the product` $C(x) = A(x) \cdot B(x)$ `at` $2^{\sqrt{N}}$.

Time complexity - In the following analysis, log stands for $\log_2$. Encoding the integers as polynomials in Step 1 takes $O(N)$ bit operations. By Lemma 4 of the previous lecture, multiplication of $A(x)$ and $B(x)$ over $\mathcal{R} = \mathbb{Z}/(2^{3\sqrt{N}} + 1)$ in step 2, takes $O(\sqrt{N} \log \sqrt{N})$ additions in $\mathcal{R}$, $O(\sqrt{N} \log \sqrt{N})$ multiplications by powers of $\omega = 8$, $2\sqrt{N}$ multiplications by the inverse of $2\sqrt{N}$ in $\mathcal{R}$, and $2\sqrt{N}$ multiplications in $\mathcal{R}$.

An element in $\mathcal{R}$ is an integer in the range $[0, 2^{3\sqrt{N}}]$, hence it can be represented by $3\sqrt{N}$ bits (except for the element $2^{3\sqrt{N}}$ which takes $3\sqrt{N} + 1$ bits). We can add two elements r_1 and r_2 in $\mathcal{R}$ in the following way: First add r_1 and r_2 over integers - say, $r = r_1 + r_2$, and then find $r \mod (2^{3\sqrt{N}} + 1)$. Adding r_1 and r_2 over integers takes $O(\sqrt{N})$ bit operations, and moreover, the value of r is at most $2^{3\sqrt{N}+1}$, as $r_1, r_2 \leq 2^{3\sqrt{N}}$. Express r as $r = c_1 \cdot 2^{3\sqrt{N}} + c_0$, where $0 \leq c_1 \leq 2$ and $0 \leq c_0 < 2^{3\sqrt{N}}$. Then, $r \mod (2^{3\sqrt{N}} + 1) = c_0 - c_1 \mod (2^{3\sqrt{N}} + 1)$. If $c_0 - c_1 \geq 0$ then sum of r_1 and r_2 is $c_0 - c_1$ in $\mathcal{R}$. Else, if $c_0 - c_1 < 0$ then $c_0 - c_1 = -1$ or -2 (as $c_1 \leq 2$). The element -1 is the same as the element $2^{3\sqrt{N}}$ in $\mathcal{R}$, and similarly, -2 is equal to $2^{3\sqrt{N}} - 1$ in $\mathcal{R}$. Therefore, adding two elements in $\mathcal{R}$ takes $O(\sqrt{N})$ bit operations. Hence, $O(\sqrt{N} \log \sqrt{N})$ additions in $\mathcal{R}$ takes $O(N \log N)$ bit operations.

What is the cost of multiplying an element $r \in \mathcal{R}$ by a power of $\omega = 2^3$? The maximum power of ω that is multiplied with an element of $\mathcal{R}$ in the FFT algorithm is $\sqrt{N} - 1$ (see Algorithm 1 in the previous lecture note). So, let us find out the complexity of multiplying r by $\omega^{\sqrt{N}-1} = 2^{3(\sqrt{N}-1)}$. First, multiply r by $2^{3(\sqrt{N}-1)}$ over integers - this essentially amounts to *shifting* r by $3(\sqrt{N} - 1)$ bits, which takes $O(\sqrt{N})$ bit operations (why?). As $r \in [0, 2^{3\sqrt{N}}]$, $2^{3(\sqrt{N}-1)} \cdot r \leq 2^{6 \cdot \sqrt{N} - 3}$. Now find $2^{3(\sqrt{N}-1)} \cdot r \mod (2^{3\sqrt{N}} + 1)$. By the same argument as above, we can show that $2^{3(\sqrt{N}-1)} \cdot r \mod (2^{3\sqrt{N}+1})$ can be computed using $O(\sqrt{N})$ bit operations. (The details are left as an exercise.) Hence, $O(\sqrt{N} \log \sqrt{N})$ multiplications by powers of ω in $\mathcal{R}$ takes $O(N \log N)$ bit operations. The purpose of choosing ω, a power of 2, is to reduce multiplications by powers of ω to shift operations, which can be done efficiently.

Verify that $\eta = 2^{6\sqrt{N} - \log \sqrt{N} - 1}$ is the inverse of $2\sqrt{N}$ in $\mathcal{R}$. By a similar argument as above, multiplication by η amounts to shift operations, which takes $O(\sqrt{N})$ bit operations. Hence, $2\sqrt{N}$ multiplications by the inverse of $2\sqrt{N}$ in $\mathcal{R}$ takes $O(N)$ bit operations. (We leave the details as an exercise.)

We are left with finding the time complexity of $2\sqrt{N}$ multiplications in $\mathcal{R}$. Since, elements of $\mathcal{R}$ are numbers in the range $[0, 2^{3\sqrt{N}}]$, a multiplication in $\mathcal{R}$ can be viewed as multiplication of two $3\sqrt{N}$ bit integers (multiplication by $2^{3\sqrt{N}} \in \mathcal{R}$ is just a shift operation), followed by going modulo $2^{3\sqrt{N}} + 1$. Arguing along the same line as before, we can derive that the 'going modulo $(2^{3\sqrt{N}} + 1)$' step can be achieved using $O(\sqrt{N})$ bit operations. Therefore, if $T(N)$ is the bit complexity of multiplying two N-bit integers, then $2\sqrt{N} \cdot T(3\sqrt{N}) + O(N)$ is the bit complexity of $2\sqrt{N}$ multiplications in $\mathcal{R}$. (Once again we leave the details as an exercise.)

Finally, in step 3, the evaluation of the product polynomial $C(x)$ at $2^{\sqrt{N}}$ can be done using $O(N)$ bit operations (by shift operations) (why?). Putting everthing together, we get the bit complexity of multiplying two N-bit integers as,

$$T(N) = O(N \log N) + 2\sqrt{N} \cdot T(3\sqrt{N}).$$

Solving the recurrence relation, we get $T(N) = O(N \log^{2+\alpha} N)$, where $\alpha = \log_2 3 - 1$.

Till date, the asymptotically fastest integer multiplication algorithm is due to Fürer [Für07]. Fürer's algorithm multiplies two N-bit integers using $N \cdot \log N \cdot 2^{O(\log^* N)}$ bit operations. A slight (but, perhaps simpler) variant of Fürer's algorithm can be found here [DKSS08]. The previous best integer multiplication algorithm (by Schönhage and Strassen [SS71]) had running time $O(N \log N \log \log N)$ bit operations.

Superlinear property of multiplication complexity

Denote the time complexity of multiplying two polynomials of degree less than n by $\mathsf{M}(n)$, and the complexity of multiplying two N-bit integers by $\mathsf{M_I}(N)$. By Schönhage-Strassen's algorithm, $\mathsf{M}(n) = O(n \log n \log\log n)$, and by Fürer's algorithm $\mathsf{M_I}(N) = N \cdot \log N \cdot 2^{O(\log^* N)}$. Thus, these multiplication complexities satisfy the following superlinear property:

Observation 1 *For all $n, m \in \mathbb{Z}^+$, $\mathsf{M}(n+m) \geq \mathsf{M}(n) + \mathsf{M}(m)$. Similarly, $\mathsf{M_I}(n+m) \geq \mathsf{M_I}(n) + \mathsf{M_I}(m)$.*

This observation will be useful later, when we show that some other problems have essentially the same complexity as polynomial multiplication.

2 Reducing polynomial division to polynomial multiplication

Let f, g be polynomials in $\mathcal{R}[x]$, where $\mathcal{R}$ is a commutative ring with unity, $\deg(f) = n$, $\deg(g) = m$ $(m \leq n)$, and g is a monic polynomial (meaning, the coefficient of the highest degree term in g is 1). We want to design an algorithm to divide f by g and find the quotient and the remainder q and r, respectively. Since g is monic, division by g is well defined over $\mathcal{R}$. Once again, the naïve division algorithm might take $O(nm)$ operations over $\mathcal{R}$, which can be $O(n^2)$ when $m = O(n)$. We want to do significantly better than this complexity.

Let $f(x) = q(x)g(x) + r(x)$, where $q, r \in \mathcal{R}[x]$ and $\deg(r) < m$. Then,

$$x^n f(1/x) = (x^{n-m} q(1/x)) \cdot (x^m g(1/x)) + x^{n-m+1} \cdot (x^{m-1} r(1/x)).$$

For any polynomial $h(x) \in \mathcal{R}[x]$, denote $x^k h(1/x)$ by $h'_k(x)$, where $k \geq \deg(h)$. By the above equation,

$$\begin{aligned} f'_n(x) &= q'_{n-m}(x) \cdot g'_m(x) + x^{n-m+1} \cdot r'_{m-1}(x) \\ \Rightarrow f'_n(x) &= q'_{n-m}(x) \cdot g'_m(x) \mod x^{n-m+1} \\ \Rightarrow q'_{n-m}(x) &= f'_n(x) \cdot (g'_m(x))^{-1} \mod x^{n-m+1} \end{aligned}$$

Does the last equation make sense? What do we mean by $(g'_m(x))^{-1}$? It means, a polynomial $h(x)$ such that $h(x) \cdot g'_m(x) = 1 \mod x^{n-m+1}$. But, does such an inverse of g'_m exist modulo x^{n-m+1}? The following lemma shows that it does. Since g is monic, the constant term of g'_m is 1. Therefore, $g'_m \cdot h_0 = 1 \mod x$, where $h_0 = 1$.

Lemma 2 *If $g'_m \cdot h_i = 1 \mod x^{2^i}$ then $h_{i+1} \overset{\text{def}}{=} 2h_i - g'_m h_i^2 \mod x^{2^{i+1}}$ is such that $g'_m \cdot h_{i+1} = 1 \mod x^{2^{i+1}}$.*

Proof $g'_m \cdot h_{i+1} = 2g'_m h_i - {g'_m}^2 h_i^2 = 1 - (g'_m h_i - 1)^2 \mod x^{2^{i+1}}$. Note, $(g'_m h_i - 1)^2 = 0 \mod x^{2^{i+1}}$. ■

Let $\ell = \lceil \log(n-m+1) \rceil$. By the above lemma, there is a polynomial h_ℓ such that $g'_m(x) \cdot h_\ell = 1 \mod x^{2^\ell}$. Therefore, $g'_m(x) \cdot h_\ell = 1 \mod x^{n-m+1}$, as x^{n-m+1} divides x^{2^ℓ}. Once we compute h_ℓ, we can find q'_{n-m} as,

$$q'_{n-m} = f'_n \cdot h_\ell \mod x^{n-m+1} \quad \text{(note that } \deg(q'_{n-m}) \leq n-m).$$

Now observe that $q = x^{n-m} q'_{n-m}(1/x)$. So, we can find q in this way, and once we find q, we can compute the remainder r as, $r = f - qg$. This suggests the following algorithm.

Algorithm 2 Polynomial division

1. Compute $f'_n = x^n f(1/x)$ and $g'_m = x^m g(1/x)$.
2. Find $h = {g'_m}^{-1} \mod x^{n-m+1}$, and compute $q'_{n-m} = f'_n \cdot h \mod x^{n-m+1}$.
3. Compute the quotient $q = x^{n-m} q'_{n-m}(1/x)$, and the remainder $r = f - qg$.

Time complexity - Step 1 takes $O(n+m)$ operations over $\mathcal{R}$. In step 2, we compute the inverse of g'_m modulo x^{n-m+1}. This is done by computing $h_\ell = {g'_m}^{-1} \mod x^{2^\ell}$, where $\ell = \lceil \log(n-m+1) \rceil$. We compute h_ℓ by iteratively computing the inverses $h_0, h_1, \ldots, h_\ell$ modulo $x^{2^0}, x^{2^1}, \ldots, x^{2^\ell}$, respectively, using lemma 2. In the beginning $h_0 = 1$. At the i^{th} iteration, we already have the inverse h_{i-1} modulo $x^{2^{i-1}}$, and we want to compute h_i. Since, h_{i-1} is computed modulo $x^{2^{i-1}}$, we can assume that $\deg(h_{i-1}) < 2^{i-1}$. By lemma 2, $h_i = 2h_{i-1} - g'_m h_{i-1}^2 \mod x^{2^i}$. So, we need to multiply g'_m with h_{i-1}^2. Since, this computation is modulo x^{2^i}, we can drop those terms in g'_m whose degree is greater than $2^i - 1$. Therefore, computing the product $g'_m h_{i-1}^2$ is like multiplying three polynomials with degree of each bounded by 2^i. So, this takes $O(\mathsf{M}(2^i))$ operations over $\mathcal{R}$. Once, we compute $2h_{i-1} - g'_m h_{i-1}^2$ we can go modulo x^{2^i} by dropping all terms of degree higher than $2^i - 1$. Therefore, to compute h_i from h_{i-1}, it takes $O(\mathsf{M}(2^i))$ operations over $\mathcal{R}$. Which means, to compute h_ℓ we have to spend a total of $\sum_{i=0}^{\ell} O(\mathsf{M}(2^i))$ operations. By Observation 1, this sum is $O(\mathsf{M}(n))$. We can derive h from h_ℓ by dropping all terms with degree higher than $n-m$. Finally, computing $q'_{n-m} = f'_n \cdot h \mod x^{n-m+1}$ takes another $O(\mathsf{M}(n))$ time. Therefore, step 2 can be executed using $O(\mathsf{M}(n))$ operations over $\mathcal{R}$. In step 3, we compute r by doing one more polynomial multiplication. Hence, the total complexity of the above algorithm is $O(\mathsf{M}(n))$ operations over $\mathcal{R}$.

A remark on Lemma 2: Lemma 2 gives us a method to 'lift' an inverse of a polynomial modulo x^{2^i} to an inverse modulo $x^{2^{i+1}}$. This is part of a more general technique called *Hensel lifting*, which we will discuss in a later lecture.

Another remark: Just like polynomial division reduces to polynomial multiplication, integer division too has the same bit complexity as integer multiplication. In other words, division with remainder of N-bit integers can be done using $O(\mathsf{M}_\mathsf{I}(N))$ bit operations. If you're interested, you can look up the details of this reduction in Chapter 9 of [GG03], or Section 1.3 of this lecture note [AK09].

Exercises:

1. Fill in the missing details in the time complexity analysis of Algorithm 1.
2. Design an algorithm to multiply two polynomials (over a ring $\mathcal{R}$) of degree less than n using $O(n \log^2 n)$ operations in $\mathcal{R}$. The ring $\mathcal{R}$ may not have a principal $2n$-th root of unity.

References

[AK09] Manindra Agrawal and Purushottam Kar. Lecture 1 & 2: Integer and Modular Arithmetic, July 2009. Available from `http://www.cse.iitk.ac.in/users/manindra/CS681/lecture1and2.pdf`.

[DKSS08] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC*, pages 499–506, 2008.

[Für07] Martin Fürer. Faster integer multiplication. In *STOC*, pages 57–66, 2007.

[GG03] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2003.

[SS71] A Schönhage and V Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.