CS 681: Computational Number Theory and Algebra **Lecture 6: Efficient integer multiplication using FFT**

Lecturer: Manindra Agrawal **Notes by: Shashi Mittal**

August 12, 2005.

1 The integer multiplication problem

Suppose we are given two n bit integers, a and b, where

$$a = \sum_{i=0}^{n-1} a_i 2^i,\ b = \sum_{i=0}^{n-1} b_i 2^i$$

We wish to find the product of the two integers as efficiently as possible. Using the straightforward multiplication algorithm, the two numbers can be multiplied in $O(n^2)$ time. Using fast fourier transform, however, the product can be evaluated in sub-quadratic time. In particular, in this lecture, we will discuss how to perform integer multiplication in $O(n\ \log^2 n\ \log\log\ n)$ time.

2 Using FFT for integer multiplication

We break the n-bit integers into t blocks of l bits each. That is, $t = n/l$. We assume that n is a perfect power of 2, and so are t and l. The exact values of t and l will be computed later. Thus, we can represent a and b as

$$a = \sum_{i=0}^{t-1} \hat{a}_i 2^{il},\ b = \sum_{i=0}^{t-1} \hat{b}_i 2^{il}$$

Consider the polynomials $a(x)$ and $b(x)$, given by

$$a(x) = \sum_{i=0}^{t-1} \hat{a}_i x^i,\ b(x) = \sum_{i=0}^{t-1} \hat{b}_i x^i$$

If we can efficiently calculate the product $a(x) \times b(x)$ efficiently, then we can calculate the integer product by evaluating the polynomial at 2^l. The latter operation can be done efficiently, since evaluation at 2^l can be done using only bit-shifts and addition, which can be done in linear time.

Intuitively, one can compute $a(x) \times b(x)$ using fast fourier transform, and then compute the integer product. However, there is a catch at this point : When we are calculating the FFT

of the resulting polynomial, we have to multiply the respective coefficients of the transforms of the two input polynomials, which itself involves integer multiplication. We can overcome this problem using the chinese remaindering theorem.

First of all, we note that the jth coefficient in the product $a(x) \times b(x)$ is $\sum_{i=0}^{j} \hat{a}_i \hat{b}_{j-i}$. Therefore, the maximum absolute value of the jth coefficient is bounded above by $(j+1)2^{2l}$, and hence the value of any coefficient in the resultant polynomial cannot exceed $t(2^{2l}+1)$. Therefore, if we can compute $a(x) \times b(x)$ $(mod\ t(2^{2l}+1))$ then the product of the original two polynomials can be recovered.

We separately compute $a(x) \times b(x)$ modulo t and modulo $2^{2l}+1$. Since t and $2^{2l}+1$ are co-primes (as t is a perfect power of 2, and the other number is an odd number), therefore we can use the chinese remaindering theorem to compute the modulo we desire.

We compute $a(x) \times b(x)$ $(mod\ t)$ using the normal multiplication algorithm. $a(x) \times b(x)$ $(mod\ 2^{2l}+1)$ is evaluated using FFT, whereby the $2t$ integer products are recursively computed using the given algorithm. Note that 2 is a $4l$ root of unity modulo $2^{2l}+1$. Therefore we need to have $2t \leq 4l$, which implies that $t \leq 2l$. The smallest value of l satisfying this constraint is $2^{\lfloor \frac{k}{2} \rfloor}$, where $n = 2^k$. Therefore, we choose $l = 2^{\lfloor \frac{k}{2} \rfloor}$ and $t = 2^{\lceil \frac{k}{2} \rceil}$.

The complete algorithm is presented in the next section, and its time complexity is analyzed in the section after that.

3 Efficient integer multiplication algorithm

Input : Two n bit integers, a and b where $n = 2^k$
Output : The $(n+1)$ bit product of a and b mod 2^{n+1}.

Algorithm

1. If n is small enough, then compute $a \times b(mod\ 2^{n+1})$ using the normal multiplication algorithm, and return the result.
2. Otherwise, let $l \leftarrow 2^{\lfloor \frac{k}{2} \rfloor}$, $t \leftarrow 2^{\lceil \frac{k}{2} \rceil}$.
3. Let $a = \sum_{i=0}^{t-1} \hat{a}_i 2^{il}$, $b = \sum_{i=0}^{t-1} \hat{b}_i 2^{il}$. Let the polynomials $a(x)$ and $b(x)$ be given by $a(x) = \sum_{i=0}^{t-1} \hat{a}_i x^i$, $b(x) = \sum_{i=0}^{t-1} \hat{b}_i x^i$.
4. Compute $a(x) \times b(x)(mod\ t)$ using FFT and straightforward integer multiplication algorithm.
5. Compute $a(x) \times b(x)(mod\ 2^{2l}+1)$ using FFT of $a(x)$ and $b(x)$ and then computing the product of the $2t$ coefficients using this algorithm recursively.

6. Combine the results obtained in steps 4 and 5 using chinese remaindering theorem to obtain $f(x) = a(x) \times b(x)(mod\ t(2^{2l}+1))$.

7. Compute $f(2^l)$, and return this value as the answer.

Note that this algorithm computes the product of a and b modulo 2^{n+1} only. If the product exceeds 2^{n+1}, then before multiplying, we will have to pad a and b with extra zeros to make them numbers with $2n$ bits. This does not have any effect on the complexity of the algorithm, as it increases the running time only by a constant factor.

4 Complexity of the above algorithm

Let $T_M(n)$ be the complexity of multiplying two n bit numbers.

- Step 4 takes $O(t\ \log\ t\ \log^2 t)$, i.e. $O(t\ \log^3 t)$ time. Since t is nearly $n^{1/2}$, therefore this step takes $O(n^{1/2}\log^3 n)$ time. This is sub-linear time, and can be neglected, because we expect at least $O(n)$ time complexity.
- Step 5 : The FFT takes $O(t\ \log\ t\ l)$ time, which is in $O(n\ \log\ n)$. Multiplying $2t$ coefficients of the Fourier transforms takes $2tT_M(2l)$ time. Hence this step takes $O(n\ \log\ t)\ + 2tT_M(2l)$ time.
- Step 6 takes $O(n)$ time, which again can be neglected.

Thus, we get the following the following recurrence relation

$$T_M(n) = O(n\ \log\ t) + 2tT_M(2l) \tag{1}$$

Theorem 4.1 $T_M(n) = O(n\ \log^2 n\ \log\log\ n)$

Proof: Let $\hat{T} = \frac{1}{n}T_M(n)$. Dividing both sides of 1 by n, we get

$$\hat{T}(n) = O(\log\ t) + 4\hat{T}(2l) \tag{2}$$

Claim 4.1 $\hat{T}(n) = O(\log^2 n\ \log\log\ n)$.

Substituting the above expression in 2, we get

$$\begin{aligned} \hat{T}(n) &\leq c_1 \log\ t\ +\ 4c_2\ \log^2 2l\ \log\log\ 2l \\ i.e.\ \hat{T}(n) &\leq c_1 \log\ n\ +\ 4c_2\ \log^2 2\sqrt{n}\ \log\log\ 2\sqrt{n} \\ i.e.\ \hat{T}(n) &\leq c_1 \log\ n\ +\ c_2 \log^2 n\ \log\log\ n \end{aligned}$$

For sufficiently large n, the second term is dominated by the second term. This proves our claim that $\hat{T}(n) = O(\log^2 n\ \log\log\ n)$, and hence we get $T_M(n) = O(n\ \log^2 n\ \log\log\ n)$. ■

5 Improving the upper bound

The bound obtained in 4.1 is not a tight upper bound. We can reduce the complexity by a factor of log n. Note that we need a factor of $\log^2 n$ in the complexity of $\hat{T}(n)$, because of the factor of 4 in the second term of right hand side in equation 2. This factor can be reduced to 2, by using negative wrapped convolution. In that case, the upper bound for integer multiplication will come out to be $O(n \log n \log\log n)$. Details of this algorithm can be found in [1].

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman. The Design and Analysis of Computer Algorithms. *Addison-Wesley*, 1974, pp. 270-274.