CS681: Computational Number Theory and Algebra **(Fall 2009)**

Lecture 1 & 2: Integer and Modular Arithmetic

July 30, 2009

Lecturer: Manindra Agrawal *Scribe: Purushottam Kar*

1 Integer Arithmetic

Efficient recipes for performing integer arithmetic are indispensable as they are widely used in several algorithms in diverse areas such as cryptology, computer graphics and other engineering areas. Hence our first object of study would be the most basic integer operations - namely addition, subtraction, multiplication and division. We will start off with algorithms that are typically referred to as "high-school" or "peasant" algorithms and move on to more efficient ones wherever scope for improvement is found.

1.1 Integer Addition and Subtraction

Given two n-bit numbers a and b, the high-school algorithms take $\mathcal{O}\left(n\right)$ time to perform addition and subtraction. Now this is optimal upto constant factors since the input size itself is $\Omega(n)$. Hence these do not pose a challenge to us. Of course for someone designing a processor, these constant factors may decide whether the chip becomes a market leader or not but we shall consider the issue of integer addition and subtraction closed from now on having obtained optimal algorithms for the same (many thanks to our respective high school Math teachers for this) .

1.2 Integer Multiplication

Unfortunately we have less to thank our high school teachers on this matter since the high school algorithm takes $\mathcal{O}\left(n^2\right)$ time to multiply two n-bit numbers. This is too slow for most scientific calculations. A slight improvement comes due to an observation of Gauss on multiplication of complex numbers.

Gauss observed that the product $(a+bi)(c+di)$ can be calculated using just three multiplications and five additions instead of four multiplications and four additions. Let $k_1 = ac, k_2 = bd, k_3 = (a+b)(c+d)$. Then $(a+bi)(c+di) = (k_1 - k_2) + (k_3 - k_1 - k_2)i$. This was the trick employed by Karatsuba and Ofman in [KO62] to arrive at an $\mathcal{O}\left(n^{\log_2 3}\right)$ algorithm for integer multiplication.

What is done in this algorithm is to express each n-bit number using two $\frac{n}{2}$-bit numbers. Thus $a = a_1 2^{\frac{n}{2}} + a_0$ and $b = b_1 2^{\frac{n}{2}} + b_0$. Hence we can use the above trick to multiply these two numbers using three $\frac{n}{2}$-bit multiplications and some constant number of $\mathcal{O}\left(n\right)$ operations which include adding the intermediate results, left shifts etc. So the time complexity of this algorithm is $M(n) = 3M\left(\frac{n}{2}\right) + \mathcal{O}\left(n\right)$ which gives $M(n) = \mathcal{O}\left(n^{\log_2 3}\right)$.

This method of breaking up numbers can be generalized to break numbers into more parts and save on some multiplications using similar tricks. This led to a family of multiplication algorithms known as Toom-Cook methods [Too63, Coo66] which give an asymptotic time complexity of $\mathcal{O}\left(n^{\frac{\log(2k-1)}{\log(k)}}\right)$ for any $k > 1$ and hence can give a complexity of $\mathcal{O}\left(n^{1+\epsilon}\right)$ for any $\epsilon > 0$.

This was improved by Schönage and Strassen using Spectral techniques (more specifically Fourier Analysis) to $\mathcal{O}\left(n \log n \log\log n\right)$ in [SS71]. After this came a long period of waiting before Fürer and De-Kurur-Saha-Saptharishi in a series of closely spaced papers [FÖ7, DKSS08] improved the time complexity to $\mathcal{O}\left(n \log n\ 2^{\log^* n}\right)$ where $\log^*$ denotes the iterated $\log$ function.

Of course one does not typically use the "fastest" method on all instances. As it turns out the algorithms by Fürer and De *et al* outperform the Schönage-Strassen algorithm in practice only on astronomically large numbers due to the large constants involved. Thus for really huge numbers, one uses the algorithms due to Fürer and De *et al* at the top level but as the recursive calls are made to smaller numbers, one switches to the Schönage-Strassen algorithm only to switch to the Karatsuba algorithm for still smaller numbers and finally to the high school algorithm for the bottom level recursive calls.

1.3 Integer Division

As it turns out, if asymptotic complexity is all that we are interested in then integer division has the same time complexity as integer multiplication. In fact the operations of multiplication, squaring, division, inversion and square-rooting all have the same time complexity. However in this lecture our aim is only to show that division is no more difficult than multiplication. For this we require the Newton-Raphson method for inversion.

This method is used in general to find roots of a real valued function f by starting with a well-informed guess x_0 for the root and improving it iteratively using the update rule $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. The method typically has a quadratic convergence but is also known to go astray at times. But in our case we will show that it approaches its goal in a steadfast manner. We shall of course use a slightly modified version of the algorithm keeping in mind not to keep very high precision numbers. Note that all logarithms are taken to base 2 unless otherwise mentioned.

Let us take the two n-bit integers to be divided as a and b where $b < a$. We want to find integers d and r such that $a = bd + r$ and $r < b$. We will always work only with positive integers for simplicity. For now let us assume that b has been scaled down to the range $\left(\frac{1}{2}, 1\right)$ by dividing by an appropriate power or 2. For this assume $b > 2$. The case $b = 2$ is clearly very simple to handle. From now on, till otherwise mentioned, we shall call this scaled down n-bit number as b. Our goal here is to find a good approximation of $\frac{1}{b}$. Note that $\frac{1}{b} \in (1, 2)$.

Let us choose $x_0 = 1$ as our first approximation. Clearly $x_0 < \frac{1}{b}$ and $1 - bx_0 < \frac{1}{2}$. Truncate $1 - bx_0$ to the first bit after the decimal point and call this quantity y_0. Clearly $y_0 < \frac{1}{2}$ as well. Define $y_i = y_{i-1}^2$ and $x_{i+1} = x_i(1 + y_i)$ for all $i > 0$. Consider the following results :

Lemma 1.1. *For any $i > 0$, we have the following to be true*

1. $y_i < \frac{1}{2^{2^i}}$

2. $1 - bx_i < \frac{1}{2^{2^i}}$

3. y_i *is at most* 2^i *bits long*

4. x_i *is at most* 2^i *bits long*

Proof. We present the respective proofs below. All proofs are by induction on i.

1. For $i = 0$ we have already shown the result. Suppose the result also holds upto some i, then we have $y_{i+1} = y_i^2 < \frac{1}{2^{2^{i+1}}}$.

2. For $i = 0$ we have already shown the result. Suppose the result also holds upto some i, then we have $1 - bx_{i+1} = 1 - bx_i(1 + y_i) = 1 - bx_i - bx_i y_i < \frac{1}{2^{2^i}} + \left(\frac{1}{2^{2^i}} - 1\right) y_i < \frac{1}{2^{2^i}} + \left(\frac{1}{2^{2^i}} - 1\right) \frac{1}{2^{2^i}} = \frac{1}{2^{2^{i+1}}}$.

3. For $i = 0$ this is true because of choice of y_0. Suppose the result also holds upto some i, then we have $y_{i+1} = y_i^2$ and hence the number of bits in y_{i+1} can be atmost twice that of y_i. Hence the result holds.

4. For $i = 0$ this is true by choice of x_0. Suppose the result also holds upto some i, then we have $x_{i+1} = x_i(1 + y_i)$. Since $y_i < 1$ and both x_i and y_i are at most 2^i bits long, the result holds.

□

At this point we fix the issue of scaling b to within the range $\left(\frac{1}{2}, 1\right)$. Let $m > 0$ such that $\widetilde{b} = \frac{b}{2^m} \in \left[\frac{1}{2}, 1\right)$. Then it is clear that if x is such that $1 - \widetilde{b}x < \epsilon$ then if $\widetilde{x} = \frac{x}{2^m}$ then $1 - b\widetilde{x} < \epsilon$ as well. Now we state a final lemma before moving on to prove our desired result.

Lemma 1.2. *Fix* $i = \log n + 1$. *Then if* $a = bd + r$ *then* $\|a\widetilde{x}_i - d| \leq 1$.

Proof. Clearly $\|a\widetilde{x}_i - d\| = \|d(b\widetilde{x}_i - 1) + r\widetilde{x}_i\| < \frac{d}{2^{2^i}} + \frac{r}{b} \leq \frac{d}{2^{2^i}} + 1 - \frac{1}{b} \leq 1$ since $r < b$, $\log b \leq n$ and $\log d \leq n$. □

The preceding lemmata give us an $\mathcal{O}(M(n))$ algorithm for integer division where $M(n)$ is the asymptotic complexity for integer multiplication. This can be shown as follows : at the i-th iteration, all we do is multiply and add 2^i-bit numbers (this includes the squaring step). This takes $M\left(\mathcal{O}\left(2^i\right)\right) + \mathcal{O}\left(2^i\right)$ time which is $\mathcal{O}\left(M\left(2^i\right)\right)$ since $M\left(\mathcal{O}\left(n\right)\right) = \mathcal{O}\left(M(n)\right)$ and $M(n) = \Omega(n)$. Another outcome of $M(n) = \Omega(n)$ is that $M(a) + M(b) \leq M(a + b)$ for all $a, b \geq 1$.

In the beginning we calculate $1 - bx_0$ which takes $\mathcal{O}\left(M(n)\right)$ time. The calculations at the end to find d from $x_{\log n+1}$ also takes $\mathcal{O}\left(M(n)\right)$ time. Due to some uncertainty in the value of d, we may have to try out a constant number of values in the vicinity of the d obtained. All this will take time $\mathcal{O}\left(M(n)\right)$. Hence integer division is no more difficult than integer multiplication.

Note: There are some minute details of the algorithm that need to be taken care of. It can be seen that the algorithm will fail if $y_0 = 0$. However this is actually a good thing for us because it tells us that our very first approximation is as good as something we hoped to get several iterations down the line. Thus if $\frac{1}{2^{l-1}} \geq 1 - bx_0 > \frac{1}{2^l}$, then we can retain $2^{\lceil \log l \rceil}$ bits of $1 - bx_0$ and call this the $\lceil \log l \rceil$-th iteration and proceed.

2 Modular Arithmetic

In this section we will deal with how to do the four basic integer operations on n-bit numbers a and b *modulo* another n-bit number m. It should be clear that modular addition, subtraction and multiplication can be done in $\mathcal{O}(M(n))$ time. As far as division is concerned we first have to define what does it mean for $\frac{a}{b}$ to be $z \bmod m$.

Definition 2.1. *For any three integers a, b and c, $\frac{a}{b} \bmod m$ is said to be defined if there exists an integer z such that $bz \equiv a \bmod m$. In such a situation we say $\frac{a}{b} \equiv z \bmod m$.*

Lemma 2.2. *$\frac{a}{b} \bmod m$ for integers a and b such that $(a, b) = 1$ is defined iff $(m, b) = 1$.*

Proof. $(\Rightarrow)$ Suppose $\frac{a}{b} \bmod m$ is defined and $\frac{a}{b} = rm + z$ where $z \in \mathbb{Z}$ and $r \in \mathbb{Q}$ such that $br \in \mathbb{Z}$. Hence $a = brm + bz$. Now suppose $(m, b) > 1$. This implies from the above equation that $(a, b) > 1$ as well which is contradictory to our assumption. Hence $(m, b) = 1$.
$(\Leftarrow)$ Suppose $(m, b) = 1$, hence there exist $p, q \in \mathbb{Z}$ such that $pm + qb = 1$. Hence $apm + aqb = a$ which implies $b(aq) \equiv a \bmod m$. Hence $\frac{a}{b} \bmod m$ is defined. □

Using the above lemma we can easily find $\frac{a}{b} \bmod m$ in $\mathcal{O}(n^2)$ time. First using the extended Euclid's algorithm determine if $(m, b) = 1$. If this is indeed the case then the algorithm will also output the pair (p, q) and clearly $q \equiv \frac{1}{b} \bmod m$. Hence $aq \equiv \frac{1}{b} \bmod m$. Since the Euclid's algorithm works in quadratic time and $M(n) = o(n^2)$ hence the algorithm takes only $\mathcal{O}(n^2)$ time.

References

[Coo66] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Department of Mathematics, Harvard University, 1966.

[DKSS08] Anindya De, Piyush Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast Integer Multiplication Using Modular Arithmetic. In *Fortieth Annual ACM Symposium on Theory of Computing*, 2008.

[F07] Martin Fürer. Faster Integer Multiplication. In *Thirty Ninth Annual ACM Symposium on Theory of Computing*, 2007.

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

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

[Too63] Andrei Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics - Doklady*, 3:714–716, 1963.