Advanced Algorithms	**Lecture 9 : April 25, 1995**
Lecturer: Johan Håstad	*Scribe: Staffan Ulfberg*

9.1 An Application of Lovàsz's Algorithm

We are given a number α with n significant bits, which is known to be an approximation of an algebraic number:

$$\exists a_0, a_1, \ldots, a_k \in \mathbb{Z} : |a_i| \leq D \wedge \sum_{i=0}^{k} a_i \alpha^i \leq Dk2^{-n}.$$

Our task is to find $a_0, a_1, \ldots, a_k$ that satisfy the equation above, i.e., we want to find the equation corresponding to an algebraic number.

We set up a lattice in which a short vector corresponds to a good equation:

$$\begin{array}{rcl} b_0 & = & (\epsilon \quad 0 \quad \ldots \quad 0 \quad 1), \\ b_1 & = & (0 \quad \epsilon \quad 0 \quad \ldots \quad 0 \quad \alpha), \\ b_2 & = & (0 \quad 0 \quad \epsilon \quad 0 \quad \ldots \quad 0 \quad \alpha^2), \\ & \vdots & \\ b_{k-1} & = & (0 \quad \ldots \quad 0 \quad \epsilon \quad \alpha^{k-1}), \\ b_k & = & (0 \quad \ldots \quad 0 \quad \alpha^k). \end{array}$$

We know that there is a vector $c = (c_0, c_1, \ldots, c_k) = \sum_{i=0}^{k} a_i b_i$ in the lattice for which $c_i \leq \epsilon D, i = 0, 1, \ldots, k-1$ and $c_k = \sum_{i=0}^{k} a_i \alpha^i \leq Dk2^{-n}$. This vector's length is bounded by $\sqrt{\epsilon^2 D^2 k + D^2 k^2 2^{-2n}}$; we know that Lovàsz's algorithm will find a vector that is not more than $2^{k/2}$ times as long. If we choose $\epsilon = 2^{-n}$ the vector we find has a length bounded by $2^{(k+1)/2} Dk2^{-n}$. This bound must hold for all components in the vector, specifically the last one:

$$\left| \sum_{i=0}^{k} a_i \alpha^i \right| \leq 2^{(k+1)/2} Dk2^{-n}.$$

Finding a vector corresponding to an algebraic number could be used to factor polynomials: given a polynomial $p(x)$ of degree n, numerically solve for a root of the equation $p(x) = 0$. Now if $p(x) = a(x)b(x)$, and we have that $p(\alpha) = 0$, we know that either $a(\alpha)$ or $b(\alpha)$ is zero. Thus, finding the polynomial of degree at most $n/2$ that corresponds to the root α will give us one factor of $p(x)$.

There are more details involved if we are going to use this result for factorization as proposed above. Among other things we have to consider what accuracy we need, and adjust D and n so that we obtain the correct equation.

9.2 The Discrete Fourier Transform

9.2.1 Definition

The discrete fourier transform $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}$ of the sequence $a_0, a_1, \ldots, a_{n-1}$ is defined by

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i,$$

where $\omega^n = 1$ and $\omega^k \neq 1, k < n$ ($\omega = e^{2\pi i/n}$). The inverse of the transform is

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{-jk} \hat{a}_j.$$

One of the reasons to use the fourier transform is that convolution of two sequences is transformed to pairwise multiplication in the transform domain:

$$c = a * b \sim \hat{c}_k = \hat{a}_k \hat{b}_k.$$

The convolution is defined by

$$a * b = \sum_{i+j \equiv k \pmod{n}} a_i b_j.$$

9.2.2 The Fast Fourier Transform

To calculate the fourier transform in an efficient way, we use a method known as the fast fourier transform (FFT). To see how the FFT works, suppose that n is a power of 2 and rewrite the definition of the transform:

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i = \sum_{k=0}^{n/2-1} \omega^{2kj} a_{2k} + \sum_{k=0}^{n/2-1} \omega^{(2k+1)j} a_{2k+1}.$$

Noticing that the two sums can be calculated easily once we have the DFT of the even and odd terms respectively:

$$\mathrm{DFT}(a_0, a_2, \ldots, a_{n-2}, \sigma = \omega^2) = \hat{a}_0^e, \hat{a}_1^e, \ldots, \hat{a}_{n/2-1}^e,$$

and

$$\mathrm{DFT}(a_1, a_3, \ldots, a_{n-1}, \sigma = \omega^2) = \hat{a}_0^o, \hat{a}_1^o, \ldots, \hat{a}_{n/2-1}^o,$$

we can calculate

$$\hat{a}_j = \hat{a}_j^e + \omega^j \hat{a}_j^o, \ j = 0, 1, \ldots, n/2 - 1,$$

and

$$\hat{a}_j = \hat{a}_{j-n/2}^e + \omega^{j+n/2} \hat{a}_{j-n/2}^o, \ j = n/2, n/2+1, \ldots, n-1.$$

Thus we can calculate the fourier transform with the help of a divide and conquer strategy. The number of required multiplications will be $O(n \log n)$.

9.3 Multiplication of Large Integers

How many bit operations are needed to multiply two n-bit integers? The algorithm used when carrying out such a calculation by hand requires $O(n^2)$ operations. The basic idea to make a faster algorithm is to use the fourier transform: the main work in multiplying seems to resemble convolution, so by using the fourier transform, we can multiply the numbers bitwise in the transform domain.

9.3.1 A First Attempt

Our first attempt to make an algorithm for multiplication will be the following:

> **Input:** Two numbers a and b represented by the binary sequences $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$.
> **Output:** The binary sequence $c_0, c_1, \ldots, c_{n-1}$ representing the product $c = ab$.
> (1) Calculate the transforms $\hat{a}, \hat{b}$.
> (2) $\hat{c}_k \leftarrow \hat{a}_k \hat{b}_k$
> (3) Compute c by doing the inverse transformation.

Step 1 will use $O(n \log n)$ exact complex multiplications. Step 2 can be calculated in $O(n)$ operations and step 3 will take the same time as step 1.

A minor problem is that

$$c_0 = a_0 b_0 + a_1 b_{n-1} + a_2 b_{n-2} + \cdots$$

is not really the least significant bit of the product. To get the correct answer we have to use $2n$ bits, and then to fill out the vectors of double length by zeros. The corresponding problem exists for all of the result bits; the solution is the same.

What we really want is to derive a method for multiplication that is fast measured in the number of bit operations needed. So, obviously, using exact complex arithmetics is not desirable. There seem to be two possible solutions:

1. Use the method above, but try to limit the need of accuracy in the complex multiplications.
2. Try to use the fourier transform over some other field than the complex numbers, e.g., over $\mathbb{Z}_N$ for some N.

9.3.2 Use complex numbers

If we want to use the complex fourier transform as proposed above, the number of bits of accuracy we need is in $O(\log n)$: there will be $\log n$ iterations, and we will lose at most one bit of accuracy in each iteration. So, if $M(n)$ is the number of bit operations needed to multiply two n bit numbers, we get

$$\begin{aligned}
M(n) &= cn \log n M(\log n) \\
&= cn \log n \, c \log n \log\log n M(\log\log n) \\
&= c^2 n (\log n)^2 \log\log n M(\log\log n) \\
&= \cdots \in O(n(\log n)^{2+\epsilon}).
\end{aligned}$$

One improvement would be to have a_i and b_i represent $\log n$ bits each. We will have to multiply vectors of length $n/\log n$, containing integers $0, 1, \ldots, n-1$; $O(\log n)$ bits of accuracy will still suffice.

$$\begin{aligned}
M(n) &= c(n/\log n)\log(n/\log n)M(logn) \\
&\approx cnM(\log n) \\
&= cn \log n M(\log\log n) \\
&\in O(n \log n (\log\log n)^{1+\epsilon}).
\end{aligned}$$

9.3.3 Make calculations in $\mathbb{Z}_N$

We can multiply a bit faster by making our calculations in $\mathbb{Z}_N$ instead. We require that $n = 2^k$. First let $B = 2^{\lfloor k/2 \rfloor}$ and $l = n/B$. We let the input vectors a and b be of length B containing integers $0, 1, \ldots, 2^l - 1$. Calculations are to be performed modulo $2^{2l} + 1$.

The number $2^{4l/B}$ is a Bth unit root since $(2^{4l/B})^B = 2^{4l} = (2^{2l})^2 = (-1)^2 = 1 \pmod{2^{2l}+1}$. An examination of the DFT and the FFT reveals that setting $\omega = 2^{4l/B}$ will do.

We represent a number modulo $2^{2l} + 1$ by $2l$ bits. We use that $x2^{2l} + y \equiv y - x \pmod{2^{2l}+1}$ so that multiplication by powers of the form $2^{4l/B}$ and addition can be done in $O(l)$ bit operations.

If we analyze our proposed multiplication algorithm, we find that step 1 requires $O(B \log B)$ multiplications, each of which can be performed in $O(l)$ bit operations. The multiplication of the transforms in step 2 requires $BM(2l)$ operations where $M(n)$ denotes the number of operations needed to multiply numbers of length n. We get

$$\begin{aligned} M(n) &= O(lB \log B) + BM(2l) \\ &= O(n \log n) + BM(2l) \\ &= cn \log n + BM(2l), \end{aligned}$$

where we set $M(n) = M'(n)n$:

$$\begin{aligned} nM'(n) &= cn \log n + B2lM'(2l) \\ M'(n) &= c \log n + 2M'(2l) \\ &\leq c \log n + 2M(4\sqrt{n}) \\ &\leq c \log n + 2c \log 4\sqrt{n} + 4M'(4\sqrt{4\sqrt{n}}) \\ &\leq c \log n + c \log n + 4c \log(4\sqrt{4\sqrt{n}}) + 8M'(dn^{1/8}). \end{aligned}$$

We will reach $M'(1)$ after $\log \log n$ steps; the total cost will be in $O(n \log n \log \log n)$.

Unfortunately, the method described above will not work correctly. A few things to worry about are:

1. If the components of the result are larger than 2^{2l} the algorithm will give an erroneous result; c_k is $\sim B2^{2l}$ and we only know c_k modulo $2^{2l} + 1$.

2. What happens if some number becomes exactly 2^{2l}? $2l$ bits are not enough to represent such a number.

3. How do we convert the output vector to a number? Is knowledge of c_k enough to calculate the product?

4. We want to determine a product. In the recursion we calculate the product modulo $2^{2l}+1$.

Problem 2 is a simple special case; we do not draw any further attention to it. Problem 4 is simply solved by making computation of product modulo 2^n+1 our basic problem and doubling n on the first level of recursion.

When we multiply a and b represented by the vectors $a = (a_0, a_1, \ldots, a_{B-1})$ and $b = (b_0, b_1, \ldots, b_{B-1})$:

$$a = \sum_{i=0}^{B-1} 2^{li} a_i$$

and

$$b = \sum_{i=0}^{B-1} 2^{li} b_i,$$

we want to get the product

$$ab = \sum_{k=0}^{2B-2} \left(2^{kl} \sum_{i+j\equiv k} a_i b_j \right) \equiv \sum_{k=0}^{B-1} 2^{kl} c'_k \pmod{2^n+1}$$

where

$$c'_k = \sum_{i+j=k} a_i b_j - \sum_{i+j=k+B} a_i b_j.$$

To calculate c'_k, use the vectors

$$\begin{aligned} a'' &= (a_0 \quad \Psi a_1 \quad \Psi^2 a_2 \quad \ldots \quad \Psi^{B-1} a_{B-1}), \\ b'' &= (b_0 \quad \Psi b_1 \quad \Psi^2 b_2 \quad \ldots \quad \Psi^{B-1} b_{B-1}), \end{aligned}$$

where $\Psi^B = -1$ ($\Psi = 2^{2l/B}$, 4 or 16). If we use the fourier transform to compute the convolution of a'' and b'' ($\hat{c}''_k = \hat{b}''_k \hat{a}''_k$), we will get

$$c'' = (c'_0 \quad \Psi c'_1 \quad \Psi^2 c'_2 \quad \ldots \quad \Psi^{B-1} c'_{B-1}).$$

It is now easy to calculate the sum

$$\sum_{k=0}^{B-1} 2^{kl} c'_k$$

in $O(n)$ operations. This solves problem 3.

For problem 1 we know $c'_k \bmod 2^{2l}+1$. If we calculate $c'_k \bmod B$ we also know $c'_k \bmod B(2^{2l}+1)$:

$$c'_k \equiv \begin{cases} x & \pmod{2^{2l}+1} \\ y & \pmod{B} \end{cases}$$

$$\Rightarrow c'_k = (y-x)(2^{2l}+1)+x.$$

We can calculate $c'_k \pmod{B}$ as

$$c'_k = \sum_{i+j=k} a_i b_j - \sum_{i+j=k+B} a_i b_j \pmod{B}.$$

We will have to deal with $\log B$ bit numbers, to compare with B bit numbers that we have used until now, so this should be fairly simple. Let $A_i = a_i \bmod B$ and $B_i = b_i \bmod B$, and also let

$$\alpha = \sum_{i=0}^{B-1} B^{3i} A_i,$$

$$\beta = \sum_{i=0}^{B-1} B^{3i} B_i.$$

Now calculate the product $\alpha\beta$, which is the product of two $O(B \log B)$ bit numbers, for example, by using the algorithm described in the next section which will use $O((B \log B)^{1.58}) \subset O(n)$ operations:

$$\alpha\beta = \sum_{k=0}^{2B-2} B^{3k} \underbrace{\sum_{i+j=k} A_i B_j}_{\leq B^3} = \sum_{k=0}^{2B-2} B^{3k} \gamma_k.$$

We can now calculate $c'_k \pmod{B}$ as $\gamma_k - \gamma_{k+B} \bmod B$.

9.4 Another Algorithm for Multiplication

In this section we will have a look at a completely different algorithm for multiplication. We want to multiply the n bit numbers x and y. Suppose that n is a power of 2 and rewrite x and y:

$$x = a + 2^{n/2}b,\ y = c + 2^{n/2}d,$$

where $a, b, c, d < 2^{n/2}$. What we want to calculate is the product

$$xy = (a + 2^{n/2}b)(c + 2^{n/2}d) = ac + 2^{n/2}(ad + bc) + 2^n bd.$$

Noticing that $ad + bc = (a + b)(c + d) - ac - bd$, it suffices to compute three $n/2$ bit products: ac, bd, and $(a+b)(c+d)$. We also need a few additions and shift operations in each recursion step. Denoting the number of bit operations needed to multiply two n bit numbers $M(n)$, we get:

$$\begin{aligned} M(n) &= 3M(n/2) + O(n) \Rightarrow \\ M(n) &\in O(n^{\log 3}) \approx O(n^{1.58}). \end{aligned}$$