CSG399: Gems of Theoretical Computer Science. Lecture 9. Feb. 06, 2009.
Instructor: Emanuele Viola Scribe: Dimitrios Kanoulas

Powering, iterated product, and weak prime number theorem

1 Powering

Today we restate and look more thoroughly at each step of the powering algorithm we begun at the end of the last class; we also present the weak form of prime number theorem.

Firstly recall the Chinese Remainder Theorem that we will use in our proofs.

Theorem 1 (Chinese Remainder Theorem). *Let $p_1, \dots p_l$ be distinct primes and $\bar{p} := \Pi_i(p_i)$. Then $\mathbb{Z}_{\bar{p}}$ is isomorphic to $\mathbb{Z}_{p_1}, \dots \mathbb{Z}_{p_l}$, with the following mapping: $x \in \mathbb{Z}_{\bar{p}} \to (x \text{ mod } p_1, \dots, x \text{ mod } p_l) \in \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_l}$.*

For the converse direction of the isomorphism, $\exists \alpha_1, \dots, \alpha_l$ with $|\alpha_i| \leq \text{poly}(\bar{p})$ such that for any $(x \text{ mod } p_1, \dots, x \text{ mod } p_n) \in \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_l}$, $x = \sum_{i=1}^{l} \alpha_i (x \text{ mod } p_i)$.

We restate the powering theorem we have seen at the end of previous class.

Theorem 2 (Powering). *Given $x \in \{0,1\}^n$, we can compute x^n by a circuit of depth $O(\log n)$ with fan-in 2 (and hence $\text{poly}(n)$).*

The algorithm for powering is the following:

Algorithm 1 (Powering). *Input: $x \in \{0,1\}^n$. Let $l := n^3$.*

1. *Compute: $(x \text{ mod } p_1, \dots, x \text{ mod } p_l)$*
2. *Compute: $(x^n \text{ mod } p_1, \dots, x^n \text{ mod } p_l)$*
3. *Compute: x^n.*

Last time we proved the correctness of Algorithm 1. Today we will prove that each step is computable in depth $O(\log n)$. We recall the following fact which we saw earlier in the course.

Fact 1. *Any function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of depth $O(n)$ and size $2^{O(n)}$, with fan-in 2.*

Before presenting the proof, we note that in general we may want to take as input both $x \in \{0,1\}^n$ and $i \leq n$ and output x^i. This generalization is easily obtained, e.g. by developing n different circuits one for each possible value of i. This more general version is used e.g. in the division algorithm.

1.1 Algorithm 1 in depth $O(\log n)$

Step 1

Let $p_i, \ldots, p_l$ be the first l prime numbers. It is an immediate consequence of the Prime Number Theorem that they are all poly(n).

Then, if x_j is the j^{th} bit of the binary representation of x, we have that

$$\begin{aligned} x\ (mod\ p_i) &= \sum_{j=0}^{n-1} (2^j x_j)(mod\ p_i) \\ &= [\sum_{j=0}^{n-1} (2^j mod\ p_i) x_j](mod\ p_i). \end{aligned}$$

So, $\forall j, i$ we precompute $a_{i,j} = (2^j mod\ p_i)$, which are independent of x. We only have to compute

$$= [\sum_{j=0}^{n-1} a_{i,j} x_j](mod\ p_i).$$

Each $a_{i,j}x_j$ can easily be done in depth $O(\log n)$. Using iterated addition, we can also do the sum $\sum_{j=0}^{n-1} a_{i,j}x_j$ in depth $O(\log n)$. This final sum is at most poly(n), and thus the length of its binary representation is $O(\log n)$. The same holds for p_i as we noted before. So the final modular reduction is an operation on $O(\log n)$ bits and can be computed by brute-force circuits of depth $O(\log n)$ (Fact 1).

Step 2

In Step 2, we want to compute $(x^n\ mod\ p_i)$ given $(x\ mod\ p_i)$, $\forall i \leq n$. Note that this is a function from $O(\log(n))$-bits to $O(\log(n))$-bits. Again the result follows via brute-force (Fact 1).

Step 3

By the Chinese Remainder Theorem, $x^n = \sum_{i=1}^{l} \alpha_i (x^n\ mod\ p_i)$. We use multiplication to compute each $\alpha_i(x^n\ mod\ p_i)$ and iterated addition to sum them all.

2 Iterated Product

In iterated product our input is: $x_1, \ldots x_n \in \{0,1\}^n$ and the output is: $\Pi_i x_i$.

Theorem 3 (Iterated Product)**.** *Iterated product can be computed in depth $O(\log n)$.*

Iterated product is of particular interest because it can be used to compute in small depth "pseudorandom functions" based e.g. on the hardness of factoring. Such objects in turn shed light on our ability to prove lower bounds via the "Natural Proofs" connection which we expect to see later in this course.

The algorithm for iterated product is similar to Algorithm 1:

Algorithm 2. *Input:* $x_1, \ldots, x_n \in \{0,1\}^n$. *Let* $l := n^3$.

1. *Compute:* $(x_1 \textit{ mod } p_1, \ldots, x_1 \textit{ mod } p_l), \ldots, (x_n \textit{ mod } p_1, \ldots, x_n \textit{ mod } p_l)$

2. *Compute:* $(\Pi_{i=1}^n x_i \textit{ mod } p_1), \ldots, (\Pi_{i=1}^n x_i \textit{ mod } p_l)$

3. *Compute:* $\Pi_i x_i$.

The correctness proof of Algorithm 2 is the same as that of Algorithm 1 which we have seen in the previous lecture.

Proof that Algorithm 2 can be implemented by circuits of depth $O(\log n)$. Step 1 and Step 3 are implemented as the corresponding steps in Algorithm 1.

Step 2 amounts to a smaller version of the problem: for each $j \leq l$, we want to compute $(\Pi_{i=1}^n x_i \textit{ mod } p_j)$ from $(x_1 \textit{ mod } p_j, \ldots, x_n \textit{ mod } p_j)$, where each $(x_i \textit{ mod } p_j)$ is at most poly(n) and thus has a representation of $O(\log n)$ bits. Let us fix j.

First, if some $(x_i \textit{ mod } p_j) = 0$, then the whole product is 0.

So let us assume that for every i, $(x_i \textit{ mod } p_j) \neq 0$. We use the following fact:

Fact 2. *If* p *is a prime, then* $(\mathbb{Z}_p - \{0\})$ *is a cyclic group, meaning that there exists a generator* $g \in (\mathbb{Z}_p - \{0\}) : \forall x \in (\mathbb{Z}_p - \{0\}), x = g^i$, *for some* $i \in \mathbb{Z}$.

Therefore, we write:

$$(x_1 \textit{ mod } p_j, \ldots, x_n \textit{ mod } p_j) \quad = \quad (g^{\log_g x_1}, \ldots, g^{\log_g x_n}).$$

Then the product we want is:

$$(\Pi_{i=1}^n x_i \textit{ mod } p_j) \quad = \quad (g^{\sum_{i=1}^n \log_g x_i}).$$

We now show the above approach can be implemented in depth $O(\log n)$.

Taking logs, i.e. going from $(x \textit{ mod } p_j)$ to $\log_g x$ can be done via a brute-force circuit (Fact 1).

Summing the logs is iterated addition.

Exponentiating, i.e. going from $\sum_{i=1}^n \log_g x_i$ to $(g^{\sum_{i=1}^n \log_g x_i}) = (\Pi_{i=1}^n x_i \textit{ mod } p_j)$ can be done via brute-force (Fact 1), because note that the sum of the logs is at most poly(n). $\square$

All the circuits we have seen in this lecture and in the previous one can be constructed "very explicitly," cf. Eric Allender's survey: "The Division Breakthroughs."

3 Weak Prime Number Theorem

Theorem 4 (Weak Prime Number Theorem). *For arbitrarily large j, (# of primes $\leq j$) $\geq j/\log^c j$, for an absolute constant c.*

Remark 5. *The Prime Number Theorem gives $c = 1$. We will get "close" to this result. The proof can be easily extended to get $c = 1 + \epsilon$ for arbitrarily small ϵ.*

The Prime Number Theorem holds $\forall j$ sufficient large. We will get arbitrarily large j.

We will show an equivalent statement that for arbitrary large j, $p_j \leq j \cdot \log^c j$, where p_j is the j^{th} prime. This means that: (# primes $\leq j \log^c j) \geq j$, which gives the theorem by inverting the function $(j \log^c j)$.

We are going to show how we can "compress" every n-bit integer in s bits: We show that there exists a function $f : \{0,1\}^s \to \{0,1\}^n$, such that f is onto. Since f is onto we must have $s \geq n$. On the other hand, we will prove that one can take $s \approx p_j - j$ and thus get our Theorem.

Proof. Given any integer k, we will show that $\exists j \geq k : p_j \leq j \log^c j$ (this is the meaning of arbitrarily large j). Consider the function $f : \{0,1\}^s \to \{0,1\}^n$, for large n and a suitable s to be determined later, defined for input $x \in \{0,1\}^s$ by:

If the first bit of x is 1, then interpret the rest of x as $a_1, \ldots, a_k$ and $f(x) := p_1^{a_1}, \ldots, p_k^{a_k}$.

If the first bit of x is 0, then interpret the rest of x as $j \geq k, b$ and $f(x) := p_j \cdot b$. Note that we assume that $j \geq k$, we can e.g. think of summing k to the appropriate bits of x, or of the function being undefined if $j \leq k$

Claim 1. *f is onto $\{0,1\}^n$.*

Proof. For any $y \in \{0,1\}^n$, let p_j be the largest prime that divides y. If $p_j \leq p_k$ then $y = p_1^{a_1} \cdots p_k^{a_k}$ and we encode y as $(1, a_1, \ldots, a_k)$. Otherwise, we encode y as $(0, j, y/p_j)$. □

Claim 2. *Can implement f with $s = 1 + max\{$ bits from case 1, bits from case 2 $\}$ = $1 + max\{k(\log n + O(1)), max_{j \geq k}\{2 \log\log j + \log j + n - \log p_j + O(1)\}\}$.*

Proof. There are two cases:

Case 1: We need to specify k exponents $a_1, \ldots, a_k$. Since each prime is at least 2, the exponents are at most n: $\forall i : a_i \leq n$. Therefore we just need $k(\log(n) + 1)$-bits.

Warm-up for Case 2: We need to specify j, b. Observe that $b \leq 2^n/p_j$, and so the binary representation of b takes $n - \log p_j + O(1)$ bits. If we write down j, b using a self-terminating encoding for j: $(1\ j_0\ 1\ j_1\ 1\ j_2, \ldots,\ 1\ j_{\log j}\ 0\ b)$, we get length $2(\log j) + \log b + O(1) \leq 2(\log j) + n - \log p_j + O(1)$ bits. This doesn't quite give the claim and the theorem (the theorem would have $\sqrt{j}$ instead of $j/\text{poly}\log j$).

Case 2: We use a self-terminating encoding for $\log j$:

$$(1\ (\log j)_1\ 1\ (\log j)_2\ 1\ \ldots\ 1\ (\log j)_{\log\log j}\ 0\ j\ b).$$

This has length $2 \log\log j + \log j + \log b + O(1) \leq 2 \log\log j + \log j + n - \log p_j + O(1)$, and gives the claim. □

Since f is onto we have:

$$\begin{aligned}
s \geq n \quad &\Rightarrow \quad 1 + max\{k(\log n + O(1)), max_{j \geq k}\{2 \log\log j + \log j + n - \log p_j + O(1)\}\} \geq n \\
&\Rightarrow \quad For\ n \geq k^2 :\ max_{j \geq k}\{2 \log\log j + \log j + n - \log p_j + O(1)\} \geq n \\
&\Rightarrow \quad \exists j \geq k : 2 \log\log j + \log j + O(1) \geq \log p_j \\
&\Rightarrow \quad j \cdot \text{poly} \log j \geq p_j.
\end{aligned}$$

$\square$