IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Advanced Course on Computational Complexity

Lecture 7: More Arithmetic and Fun With Primes

David Mix Barrington and Alexis Maciel
July 25, 2000

1. Overview

We continue our exploration of arithmetic operations on short ($O(\log n)$-bit) and long ($n^{O(1)}$-bit) numbers by putting multiplication and iterated addition of long numbers in FOM or uniform TC^0. To continue to more complex operations such as division, we will need to take a digression to deal with *prime numbers*. In more detail:

- Using the iterated addition of short numbers from last lecture, we will carry out iterated addition of long numbers, and hence multiplication of long numbers, in FOM. We'll also solve the easier problem of sorting polynomially many long numbers in FOM.
- We'll begin our study of prime numbers by reviewing the fundamental theorem of arithmetic and exploring the *density* of primes (the number of primes with a given number of bits). Close bounds on this density are given by the Prime Number Theorem, but we will derive bounds adequate for our purposes with a simpler argument using *Kolmogorov complexity.*
- We'll give an application of the *small primes trick* in complexity theory, a theorem of Lipton and Zalcstein that strings in the *free group on two generators* can be tested for equality in deterministic log space.
- In another application, we'll show that poly-log ($(\log n)^{O(1)}$) *threshold functions* can be computed in FO + BIT.

2. Multiplication and Other Operations in FOM

In the last lecture we saw how to add together a polynomial number of short numbers (numbers of $O(\log n)$ bits) in FOM. We can now use this to solve the general ITERATED ADDITION problem, where the numbers have polynomial length, in FOM as well. For definiteness we will assume that we have n inputs $X_1, \ldots, X_n$ of n bits each, so that each X_i is equal to $\sum_j x_{ij} 2^j$. The sum Z of these n numbers will have at most $n + \log n$ bits.

We lay the numbers out in an n by n array of bits and once again divide this array into vertical stripes of $\log n$ columns each. That is, we write each column number j as $k \log n + \ell$ with $0 \leq \ell < \log n$ and define a stripe to be the columns that share a given value of k. We then separate the bits in the stripes for odd values of k from those for even values of k. We define Z_{odd} to be the sum of the odd-stripe bits, and Z_{even} to be the sum of the even-stripe bits. Our answer Z is the sum of Z_{odd} and Z_{even}, and we will show that each of these two numbers is computable in FOM.

If we look at a single stripe, we see that we are adding n short numbers and then multiplying the result by $2^{k \log n}$. We can do this addition in FOM as we've seen, getting a number of at most $2 \log n$ bits times $2^{k \log n}$. But then note that in the computation of Z_{odd} or Z_{even}, results from one stripe cannot cause carries with results from other stripes. We can thus specify each bit of Z_{odd} or Z_{even} as the appropriate bit of one of the stripe sums. In this way we compute Z_{odd} and Z_{even} in FOM, and a single addition gives us Z.

Multiplication of two n-bit numbers is clearly easy once we have iterated addition, as if $X = \sum_i x_i 2^i$, $XY = \sum_i (x_i 2^i Y)$, a sum of n numbers each of at most $2n$ bits. (If we were keeping track carefully, we could note that this is an $\mathrm{FO} + \mathrm{BIT}$ reduction from the problem MULTIPLICATION to the problem ITERATED ADDITION. In the exercises we ask you to construct such a reduction from ITERATED ADDITION to MULTIPLICATION. That is, given $X_1, \ldots, X_n$ can you construct Y and Z such that the sum of the X_i's can be read off somewhere from the binary expansion of YZ?)

The FOM algorithm for ITERATED ADDITION allows us to say more about the relationship between two problems that will concern us greatly later, DIVISION and POWERING. Neither of these problems is known to be in FOM. It is not hard to reduce POWERING to DIVISION in $\mathrm{FO} + \mathrm{BIT}$, using the identity $1/(1 - U2^{-n}) = \sum_i (U2^{-n})^i$. But reducing DIVISION to POWERING using the same idea requires iterated addition. We can write $1/Y$ as $1/(1 - U) = \sum_i U^i$, where $U = 1 - Y$. We can use POWERING to calculate the terms of this sum to polynomially many bits of accuracy, and then add them together to get an approximation of $1/Y$ good enough to get an approximation of X/Y that is within one of $\lfloor X/U \rfloor$. Then testing the two integers nearest this approximation, using MULTIPLICATION, allows us to find the correct value of $\lfloor X/U \rfloor$.

Finally we note that another interesting problem, the sorting of n n-bit integers, is easily seen to be in FOM. We can compare integers in FO + BIT, and thus determine, for each input integer, exactly how many other input integers are less than it. (We compute a bit for each other candidate and then add these $n-1$ bits in FOM.) Then to find the i'th integer in the sorted list, for example, we find the input integer that is larger than exactly $i-1$ others and use that. (There are some minor details to cover the case of ties.)

3. How Many Prime Numbers are There?

We now take a slight digression into *number theory*, because our eventual algorithms for DIVISION and other problems will require us to know some facts about the distribution of prime numbers. There are infinitely many prime numbers, of course, but we will want to know how the *number of primes less than x* varies as a function of x.

The right answer to this problem is known to mathematics by the *Prime Number Theorem*, which says that this function is asymptotically $(1+o(1))x/(\log_e x)$. That is, roughly one in $\log_e x$ numbers in the range of x is prime. Of course, the best answers to these questions involve the use of almost arbitrarily different mathematics. Can we get a reasonable lower bound on the number of primes by more accessible methods? Papadimitriou's book gives an easy argument that there must be at least $\sqrt{x}$ primes less than x and discusses other approaches to getting better bounds. Here we prove that there are $\Omega(x/(\log^2 x))$ primes less than x, giving us an excuse to introduce the concept of *Kolmogorov complexity*. (For much more on this subject see the book by Li and Vitanyi.)

The usual method of representing numbers (non-negative integers) by strings of bits requires $\log n$ bits (actually $\lceil \log(n+1) \rceil$, except for 0) to represent n. It should be clear that it is impossible to design a system that would represent all numbers using substantially fewer bits, as there would simply not be enough bit strings to give each number a unique string. Kolmogorov complexity uses this idea to argue that there *exist* strings that cannot be compressed by any algorithm. Here we will show that if there were somehow fewer primes than our bound requires, we could design a system that would represent each number n in $\log n - \omega(1)$ bits, a contradiction.

First note that every number n has a prime-power factor that is $\Omega(\log n/\log\log n)$.

This is because n is the product of its maximal prime-power factors, which are distinct numbers each bounded by the largest such factor d, and thus n is no greater than $d!$. (We calculated the inverse of the factorial function, up to asymptotics, in an exercise in Basic Lecture 1.) If this number should not be a prime itself, we can use this property to represent n with a saving of bits. We write n as $p^e y$, and encode n by giving e, p, and y.

Of course to give these three numbers we must be careful of the details of the encoding, as our output must be an unadorned bit sequence with no punctuation and no risk of ambiguous interpretation. If x and y are two bit strings, we can encode the pair $\langle x, y\rangle$ using $2\log|x| + |x| + |y| + O(1)$ bits as follows. (Here $|x|$ denotes the length of the string x, as usual.) We first give the number $|x|$ in binary but with each bit doubled (so 101 would appear as 110011), followed by an 01. Then exactly the next $|x|$ bits are to be read as x, and the remaining bits are to be read as y.

Encoding a triple $\langle e, p, y\rangle$ this way uses $|e|+|p|+|y|+2(\log|e|+\log|p|)+O(1)$ bits. If $e > 1$, one can check that the savings over $|n| = e|p| + |y| + O(1)$ are substantial, as going from $e|p|$ to $|p|$ more than makes up for the other costs. We may thus concern ourselves only with numbers that have a *prime* factor that is $\Omega(\log n/\log\log n)$.

We write these numbers as pairs $\langle i, y\rangle$ where p_i is the i'th prime and $n = p_i y$. The length of n is just $|p_i|+|y|+O(1)$, and we can write the pair in $2\log|i|+|i|+|y|+O(1)$ bits. We have a contradiction unless $|i|$ is at least $|p_i| - 2\log i - O(1)$ bits. Since $\log i$ is less than $\log p_i$, we can conclude that $|i|$ is at least $|p_i| - 2\log p_i - O(1)$, and thus that i itself is at least $\Omega(p_i/\log^2 p_i)$.

We will also have occasion to use the *primorial* of a number x, which is defined to be the product of all primes less than x. It is easy to see that this number is at least 2 to the number of such primes, since each prime is at least 2. Thus our estimate tells us that the primorial of x is $2^{\Omega(x/\log^2 x)}$.

4. The Word Problem for the Free Group

Now let's apply our newfound knowledge of primes to solve a problem in complexity theory. The *free group* on two generators is defined to be the set of all strings over the alphabet $\{a, b, a^{-1}, b^{-1}\}$, with the understanding that cancellations of (or insertions of) the strings aa^{-1}, $a^{-1}a$, bb^{-1}, and $b^{-1}b$ anywhere in the string yield the same element of the group. (Formally, the group can be thought of as a set of equivalence classes of strings under this equivalence relation.) The *word problem* for the free group is to input a string and determine whether it represents the identity element of the group. (This is equivalent in difficulty to taking two words and determining whether they are equivalent — we ask you to show this in the exercises.)

We can solve the word problem in P easily by cancelling pairs greedily until we cancel all the letters or get stuck. But this algorithm uses space linear in the input size. We can do better:

Theorem 1 *(Lipton-Zalcstein) The word problem for the free group on two generators is in* L.

Proof The first key observation is that the free group can be represented as a group of two-by-two integer matrices with determinant one (a subgroup of the group called $SL_2(Z)$, which consists of all such matrices). We map the free group element a to the matrix $\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ and map b to $\begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$, with the inverses being mapped to $\begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$ respectively. Clearly this is a homomorphism and the correct cancellations will result in identity matrices. We have to check that the homomorphism is one-to-one, that is, that no two non-equivalent strings of letters map to the same matrix. In the exercises you will prove this, showing a unique factorization property for matrices that are products of these generators.

The obvious idea to solve the word problem, then, is to generate the sequence of matrices corresponding to the input string, multiply them together, and see whether the result is the identity matrix. This algorithm is correct, but unfortunately cannot be implemented in logarithmic space as it is. The matrix corresponding to the string $(ab)^n$, for example, has entries whose length in bits is linear in n. We can't remember such a matrix in our memory.

What we can do, however, is remember the four entries of a matrix each *modulo a short prime*. That is, we carry out the matrix multiplication not in $SL_2(Z)$ but in $SL_2(Z_p)$ where p is a prime of $O(\log n)$ bits. If the answer in $SL_2(Z)$ is really the identity matrix, we will also get the identity matrix in $SL_2(Z_p)$ for every p. But what about the converse? Suppose we try the calculation for all primes of up to $c \log n$ bits, and get the identity matrix every time. We then know that the actual integer matrix entries of the product are congruent to the desired numbers modulo each of these primes. By the Chinese Remainder Theorem, the product is also congruent to the identity matrix modulo the *product* of all these primes, the primorial of $2^{c \log n} = n^c$. We showed above that the primorial of x grows as $2^{\Omega(x/\log x)}$, so by choosing c correctly we can cause the primorial to exceed any given exponential function of n (any function that is $2^{n^{O(1)}}$). It is easy to show that the coefficients of the product of n matrices with $O(1)$ coefficients must be $2^{O(n)}$, so we are done. Our algorithm is to carry out the matrix multiplication modulo all such primes and accept iff each gives the identity matrix. $\square$

Note that this algorithm does *not* permit us to evaluate a matrix product in L, but only to check whether it is equal to some particular matrix. It is also not clear how to take a string in the free group and produce the unique string that is equivalent to it but has no possible cancellations, *unless* this string happens to be the empty string.

5. Poly-Log Threshold Functions in $\mathrm{FO} + \mathrm{BIT}$

Another use of prime numbers is a surprising fact due to Denenberg, Gurevich, and Shelah, and independently to Fagin, Klawe, Pippenger, and Stockmeyer and to Ajtai and Ben-Or. It is easy to design FO formulas that count letters in a string up to some constant. The natural conjecture would be that if a function is $\omega(1)$, first-order formulas would be unable to count up to it. But in fact:

Theorem 2 *Let $t(n) = (\log n)^{O(1)}$ be a function definable in $\mathrm{FO} + \mathrm{BIT}$. In $\mathrm{FO} + \mathrm{BIT}$ it is possible to determine whether an input string has at least $t(n)$ ones.*

(Note: This is possible *if and only if* $t(n) = (\log n)^{O(1)}$ — the other direction will follow from the exponential lower bound for parity circuits to be proved in Basic Lecture 10.)

Proof It is sufficient to test whether there are *exactly* $t(n)$ ones, where $t(n)$ is polylog. This will be a bit simpler.

We know from Advanced Lecture 4 how to add up a poly-log length sequence of bits (in the Exercises you are invited to provide some more details of this). Our problem is that the poly-log many bits we want to add are spread out along an input of n bits. We'll deal with this by using *hashing.*

Any number $m < n$ gives us a hash function that takes strings of n bits to strings of m bits. We set bit i of the output to be one iff there is *any* position in the input that both has a one in it and is congruent to i modulo m. A *collision* occurs for a particular input if there are two or more positions that each have a one and are congruent modulo m. *If* there are no collisions, however, the number of ones in the output is exactly equal to the number in the input.

We need to show that for every possible input string w with at most $t(n)$ ones, there is an m that is poly-log in n and that hashes w to a string $h_m(w)$ with no collisions. If this is true, our formula can define the hash function for the least such m and count the ones in $h(w)$ as before. Note that the hash function itself, and the test for whether it has collisions, are each definable in FO + BIT. (We are using arithmetic on various short numbers, which is clearly definable from PLUS and TIMES.)

If we consider an arbitrary input w with $t(n)$ ones, for which values of m could it have a collision? There are $\binom{t}{2} \leq t^2$ pairs of ones in w, and each one will cause a collision if and only if m divides the distance between the two ones in the pair. There can be a collision only if m divides the *product* P of these distances, a number that is at most $n^{t^2} = 2^{t^2 \log n}$. But look at just the primes less than some number q. If *every one* of these primes divided P, then so would the product of all those primes, the primorial of q. We can choose a poly-log q so that the primorial of q exceeds P. Thus at least one prime less than q fails to divide P, meaning that it fails to divide any of the distances between positions with ones. Hashing by this number has no collisions, so there is a least number with no collisions and our formula will find and use it.

□

6. Exercises

1. Construct a reduction from ITERATED ADDITION to MULTIPLICATION. That is, given n n-bit numbers $X_1, \ldots, X_n$, construct numbers Y and Z such that the sum of the X_i's can be easily determined from the binary representation of YZ.

2. We showed above that the primorial of x is $2^{\Omega(x/\log^2 x)}$, using our $\Omega(x/\log^2 x)$ bound for the number of primes less than x. Calculate this number more carefully to show that the primorial is $2^{\Omega(x/\log x)}$, still using our bound on the number of primes. (Hint: The log of the primorial is the sum of the logs of all these primes.) How do these results change if we use the bound on the number of primes given by the Prime Number Theorem?

3. Recall that S_n is the group of all permutations of an n-element set. The order of an element a in a group is the least number $q > 0$ such that $a^q = e$, where e is the identity. What is the maximum order of any element of S_n?

4. The equality problem for the free group on two generators is to input two strings and determine whether they are equivalent (represent the same element of the

free group). Show that the equality problem is easily solved given a solution to the word problem, and vice versa.

5. Prove the claim that any product of our four given generator matrices has a unique factorization into such matrices that allows no cancellations. (Hint: Consider any two-by-two integer matrix C with determinant 1. Show that there is only one way to write C as DG where G is one of our generator matrices and D is "smaller" than C. (One definition of the size of a matrix is the sum of the squares of its entries.) Continue this process until you get the identity or a matrix that can't be reduced in size by factoring out a generator.)

6. (Only if you liked the one before.) Prove that any two-by-two matrix of *non-negative* integers with determinant one has a unique factorization as a product of the matrices $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. (Dave proved this fact in his undergraduate honors thesis in 1981.)

7. Explain carefully why adding up *any poly-log* sequence of bits is in FO + BIT. (We proved this for a particular poly-log function earlier.)

8. Suppose that $f(n)$ is greater than poly-log ($f(n) = \log^{\omega(1)} n$), and that we have a circuit family of depth $d(n)$ and size $s(n)$ that tests whether the input has exactly $f(n)$ ones. What is the depth and size of a circuit family for EXACTLY-HALF built from these circuits? What about a circuit family for PARITY? (If $d(n) = O(1)$ and $s(n)$ is polynomial, this hypothesis would violate known lower bounds for parity that we will prove later.)