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

Lecture 10: Logspace Division and Its Consequences

David Mix Barrington and Alexis Maciel
July 28, 2000

1. Overview

Building on the last two lectures, we complete the proof that DIVISION is in FOMP (first-order with majority quantifiers, BIT, and powering modulo short numbers) and thus in both L-uniform TC^0 and L itself. We then examine the prospects for improving this result by putting DIVISION in FOM or truly uniform TC^0, and look at some consequences of the L-uniform result for complexity classes with sublogarithmic space.

- We showed last time that we can divide a long number by a *nice* long number, that is, a product of short powers of distinct primes. We also showed that given any Y, we can find a nice number D such that $Y/2 \leq D \leq Y$. We finish the proof by finding an approximation N/A for D/Y, where A is nice, that is good enough so that $\lfloor XN/AD \rfloor$ is within one of X/Y.
- We review exactly how DIVISION, ITERATED MULTIPLICATION, and POWERING of long integers (poly-many bits) are placed in FOMP by this argument.
- We review the complexity of powering modulo a short integer, and consider how this affects the prospects of placing DIVISION and the related problems in FOM.
- Finally, we take a look at *sublogarithmic space* classes, in particular problems solvable in space $O(\log\log n)$. These classes are sensitive to the definition of the model. We argue that the more interesting classes are obtained by marking the space bound in the memory before the machine starts. It is known that a machine with $O(\log\log n)$ space cannot recognize $\{0^n : n \text{ is prime}\}$ if it starts

with empty memory. If this lower bound can be extended to the case where the machine has a marked space bound, this would show that L $\neq$ NP.

2. Completing the Chiu-Davida-Litow Proof

To recapitulate, we have long numbers X and Y and we want to find $\lfloor X/Y \rfloor$. We choose a large enough M, a product of distinct odd primes, and convert both X and Y to CRR_M form. Using results from Advanced Lecture 9, we find a number D such that $Y/2 \leq D \leq Y$ and D is *nice*, meaning that all its maximal prime-power divisors are short ($O(\log n)$ bits). We showed in that lecture that in FOMP, we can divide an arbitrary long number by a nice long number, as long as the inputs and output are all in CRR_M.

We define the rational number u to be $1 - Y/D$, so that $u \leq 1/2$ and D/Y is $1/(1-u) = \sum_{j=0}^{\infty} u^j$. By Lemma 1 below, we will find numbers N and A such that A is nice and the rational number N/A is within ϵ of D/Y, where $\epsilon < 2^{-2n}$. Then we may be assured that the integer $Z = \lfloor XN/DA \rfloor = \lfloor (X/D)(N/A) \rfloor$ is within one of $\lfloor X/Y \rfloor$, so that by evaluating $X - YZ$ we can determine the exact answer. This will complete the proof that DIVISION, with output in CRR_M, is in FOMP.

Lemma 1 *Let $u = (D-Y)/D \leq 1/2$ be given as a quotient of two numbers in* CRR_M *and let $\epsilon = 2^{-O(n)}$. In* FOMP *we can compute two numbers N and A such that A is nice and N/A is within ϵ of $\sum_{j=0}^{\infty} u^j$.*

Proof The first thing to notice is that if we truncate the infinite series to the finite series $\sum_{j=0}^{cn} u^j$ for a sufficiently large constant c, we will cause an error much smaller than ϵ because $u \leq 1/2$. We will pick a product of primes A_j for each value of 1 from 0 to cn, using distinct primes so that the product A of all the A_j's will be nice.

For each such j, we will approximate the rational number u^j by a fraction with denominator $A_1 \cdots A_j$, making an error of less than $\epsilon/2cn$ each time. When we add these fractions with common denominator A (getting the desired numerator N) the total error will be less than ϵ (see the Exercises). Note that finding the common denominator involves an iterated product, but this is not problematic because all the numbers involved are in CRR_M and we want the output in CRR_M as well.

It suffices to approximate u separately as a fraction Z_j/A_j with denominator A_j for each j, and then to take the product $Z_1 \cdots Z_j / A_i \cdots A_j$ as our approximation of u^j. Recall that u is defined to be $(D-Y)/D$ where D is nice. As long as A_j is long enough (say, $(c+1)n$ bits) we can use our nice-denominator DIVISION algorithm to get $N_j = \lfloor (D-Y)A_i/D \rfloor$ and be assured that N_j/A_j is sufficiently close to u. $\square$

3. DIVISION and Related Problems in FOMP

We have completed our proof that DIVISION is in FOMP with the input and output in CRR_M. But recall that this is sufficient to solve the original DIVISION problem, with input and output in binary, in FOMP. Given an arbitrary number Z in CRR_M, we can find the k'th bit of Z in binary by computing the number $\lfloor Z/2^k \rfloor - 2\lfloor Z/2^{k+1} \rfloor$ in CRR_M and seeing whether it is equal to 0 or 1. This is a first-order definition given the CRR_M version of DIVISION.

The same trick allows us to solve the binary versions of ITMULT and POWERING in FOMP. We have solved ITMULT with input and output in CRR_M, and POWERING is simply a special case of ITMULT. (Recall that our definition of POWERING requires the exponent to be a short number (polynomial in n) — otherwise the answer might not be expressible even as a long number.) In each case, computing the product or power in CRR_M and converting the answer to binary solves the problem.

By definition, the predicate for powering modulo a short number is complete for FOMP via FOM reductions. (Actually these are "Turing" rather than the reductions defined in the Basic Lectures. These are the natural reductions to use in the first-order context. A is FOM-Turing reducible to B if A can be defined by a first-order formula with majority quantifiers, BIT, and access to decision predicate for B. In the circuit world, you can think of a Turing reduction from A to B as a circuit that contains gates that input a string and decide whether it is in B.)

What about our other problems? Can we show that they are complete for FOMP under FOM reductions? We have seen several reductions among these problems already. POWERING has an easy $\mathrm{FO} + \mathrm{BIT}$ reduction to ITMULT, taking the pair $\langle A, b \rangle$ to a vector consisting of b copies of A. POWERING also has an $\mathrm{FO} + \mathrm{BIT}$ reduction to DIVISION, because the powers of A can be read from the bits of the binary expansion of $2^{p(n)}/(1 - 2^{q(n)})$ for large enough polynomials p and q. DIVISION has an FOM reduction to POWERING that uses the same principle, but needs ITERATED ADDITION to sum up the appropriate powers.

The completeness of ITERATED MULTIPLICATION and DIVISION for FOMP under FOM reductions will follow from that of POWERING. But we can easily get the predicate for powering modulo a short number from POWERING and DIVISION together: simply compute a^i as an integer and divide it by m to get a quotient Q, then let $b = a^i - Q$ and we have a^i congruent to b modulo m.

Thus if any of these four problems is in FOM, then $\mathrm{FOM} = \mathrm{FOMP}$ and all four problems are in FOM, which is equal to DLOGTIME-uniform TC^0. In the next section we look more closely at the difficulty of powering modulo a short number to get some idea whether this might be true.

4. Prospects for DIVISION in FOM

When we first looked at powering modulo a short number in Advanced Lecture 8, we noted two algorithms to solve the problem (to calculate a^i modulo m):

- Computing $a^1, a^2, \ldots, a^i$ in turn, remembering only one at a time and thus using logarithmic space, and

- Calculating all the predicates "$b^k = c$ modulo m" in parallel for $k = 1, 2, 4, 16, \ldots$, reaching $k = i$ in $O(\log\log i)$ phases.

The second algorithm puts this problem in the class FOLL, languages decided by first-order formulas with $O(\log\log n)$ quantifiers on inputs of size n. These formulas are equivalent to DLOGTIME-uniform circuits of depth $O(\log\log n)$, unbounded fan-in, and polynomial size. In Basic Lecture 10, we will prove a lower bound on the size needed for circuits of a given depth to decide the PARITY language, with the consequence that PARITY is *not* in the class FOLL.

As we showed in an earlier exercise, the standard reductions applied to FOLL put it within depth $\log n(\log\log n)$ with bounded fan-in, and thus within DSPACE$(\log n(\log\log n))$, not within L or NC^1. The smallest standard class known to contain FOLL is AC^1, well above the rest of the classes involved in the analysis of DIVISION. We can say even more about FOLL given the observation above that PARITY is not in it. Since FOLL is closed under FO + BIT (Turing) reductions, a language in FOLL *cannot be hard* under such reductions for any class that contains PARITY, such as TC^0, NC^1, L, or NL.

This *suggests*, but does not prove, that the predicate for powering modulo a short number does not use the full power of L and thus that FOMP might be strictly contained in L-uniform TC^0. The latter class can be defined as FOM augmented with any numerical predicates computable in L. But of course, since it is consistent with our knowledge that FOM = L, all of these varying uniform versions of TC^0 might be equal.

What would it take to prove that DIVISION is in FOM, the truly uniform version of TC^0? The obvious thing to do would be to solve powering modulo a short number in FOM, if this is possible. Even the weaker result of solving powering modulo a short number in truly uniform NC^1 would put DIVISION in truly uniform NC^1, something currently not known. (There is no particular reason to believe that NC^1 is more powerful than TC^0 for problems dealing with integers, but you never know.)

If, on the other hand, powering modulo a short number is *not in* truly uniform TC^0, then it seems that an entirely new approach would be needed to put DIVISION there. This is because any use of CRR when the original input and output are in binary would seem to require putting binary numbers *into* CRR, which by definition includes computing powers of two modulo a short number. (Might 2 be easier than other bases? The number 2 is a generator modulo some primes and not modulo others. If it is a generator for a constant fraction of all primes, for instance, one could develop CRR using entirely such primes. Maybe number theorists know about this...)

5. Sublogarithmic Space Classes

We have so far not thought much about space complexity classes between DSPACE(1) (the regular languages) and DSPACE($\log n$) or L. Savitch's Theorem for space classes, for example, required that the space bound be at least $\log n$. One reason for ignoring sublogarithmic space classes is that they are less *robust* than the more familiar classes. For example, for any function $s(n) = o(\log n)$, we can define the following two complexity classes:

- dspace(s) is the set of languages that can be decided by a machine with $O(s(n))$ bits of memory on inputs of size n, provided that the memory is empty at the start of the computation.
- DSPACE(s) is the set of languages that can be decided by a machine with $O(s(n))$ bits of memory on inputs of size n, provided that the memory begins with the string $0^{s(n)}\$$. That is, there is an endmarker that tells the machine the limits of its allowed memory.

The first definition is arguably the more natural one, because all of our other machines were forced to begin with blank input. At least for sensible space bounds of $O(\log n)$ and above, the two classes are the same, as you will show in the exercises. But we will argue here that the second definition may make more sense, because the DSPACE classes are more closely related to other, more interesting complexity classes. To prove this relationship we will make use of the Chiu-Davida-Litow result that binary numbers can be converted to CRR in L.

In the exercises we ask you to prove some lower bounds on the power of dspace(s) and DSPACE(s) where $s = o(\log n)$. Some of these work equally well for both classes because they exploit the fundamental limitations of devices with too little memory to remember where they are in the input. But others, such as the proof that the unary strings of prime length are not in dspace($\log\log n$), for example, seem to rely on another limitation of dspace machines, the fact that they are required to treat many different inputs in the same way. (In fact these bounds rely on the very Pumping Lemma disparaged in Advanced Lecture 3.) You may find it instructive to try to generalize this lower bound to show that the unary strings of prime length are not in DSPACE($\log\log n$). (But don't work too hard on it until you have read below!)

Given any language $A \subseteq \Sigma^*$ with $\Sigma = \{0, 1\}$, we can consider the *unary translation* of A, called $un(A)$. We get this by treating every string w in A as a binary integer $1w$ (we must put the one in front so that leading zeros will not cause different strings to be mapped to the same number), and letting $un(A)$ be the set of all strings 0^{1w} for $w \in A$.

The two languages A and $un(A)$ contain the same information, but their complexity may be different because each complexity measure is computed with respect to the input size, and the sizes of w and 0^{1w} differ exponentially. For conventional space bounds, this is all that is happening, and a language $un(A)$ is in DSPACE(s) iff A is in DSPACE($\log n + s(\log n)$) (if $s(n) = \Omega(n)$). But when some of the bounds are sublogarithmic, this can break down. In the exercises you are asked to find a language A that is regular (in dspace(1) = DSPACE(1)) but such that $un(A)$ is not in dspace(s) for any $s = o(\log n)$.

With the DSPACE measure, however, the behavior of conversion between unary and binary representation does not change as radically for small space classes:

Lemma 2 *(Allender-Barrington) Let $s(n) = \Omega(\log n)$ be fully space constructible. Then for any language A, $A \in$ dspace($s(n)$) if and only if $un(A) \in$ DSPACE($\log\log n + s(\log n)$). In particular, $A \in$ L iff $un(A) \in$ DSPACE($\log\log n$).*

Proof For the forward direction, we assume that A is in dspace(n) and provide a small-space algorithm to decide $un(A)$. On input 0^n, we do not have enough space to store a binary representation of n. But we can choose an appropriate modulus M and obtain the CRR_M form of n one prime at a time, since we require only primes of $O(\log\log n)$ bits. Whenever we need n modulo m_i, we can sweep the input keeping a modulo m_i counter in our memory.

From the Chiu-Davida-Litow Theorem, we know that we can obtain the bits of the binary representation of a b-bit number in $O(\log b)$ space. Since here $b = \log n$, the space needed is $O(\log\log n)$, which is available to us. We can then use space $s(\log n)$ to simulate the machine needed to determine whether the binary string represented by n is in A.

For the converse, we have input x (a string interpreted as a binary number) and we want to determine whether $x \in A$ in dspace(s) with the aid of a machine M that can decide the language $un(A)$ in DSPACE($\log\log x$) + $s(\log x)$. (We know $s(\log x)$ is at least $\Omega(\log\log x)$.)

We cannot carry out a step-by-step simulation of M on input 0^x, because we do not have enough memory to record the head position of M. However, we can at least start such a simulation because we have enough memory to simulate the memory of M. We can carry out the simulation watching for two possible events:

- M's input head returns to an endmarker before M repeats a configuration (same state, same memory contents, same memory head positions), or
- M does repeat a configuration.

Each time the former case occurs we continue the simulation with the space we have available. In the latter case, we can determine the net movement of M's input head during the loop from the first occurrence of the configuration to the second. We can then determine, by some simple arithmetic, what will happen after repeated performance of this loop. Eventually (since M halts on all inputs and thus cannot loop forever) M's input head must hit an endmarker, and we can figure out which one and which state it will be in. We can thus continue the computation as long as necessary to determine its final outcome, using only the allowed space. $\square$

Suppose then that we can show some *unary* language to be outside of DSPACE($\log\log n$), such as the set of unary strings of prime length. This would show that the corresponding *binary* language is outside of L. But the set of binary representations of prime numbers is in NP, so proving this lower bound would separate L from NP, making a major breakthrough in complexity theory. This suggests that problems involving even these very weak complexity classes can be very hard.

6. Exercises

1. Exactly what happens in our division algorithm if D should happen to be equal to Y or to $Y/2$?

2. Suppose that $\alpha_1, \ldots, \alpha_n$ are real numbers such that for some $0 \leq u \leq 1/2$, $|\alpha_i - u| < \epsilon$. Prove that the product of the α_i's is within $n\epsilon$ of u^n. What assumptions do you need on ϵ?

3. Show that given any two integers b and c each greater than one, we can convert integers from base-b to base-c in FOMP. (We assume that the individual "digits" of each representation are given in binary.)

4. Show that the base-conversion problem above is complete for FOMP under FOM reductions.

5. Prove that $\mathrm{dspace}(\log n) = \mathrm{DSPACE}(\log n)$. What hypothesis on $s(n) = \Omega(\log n)$ do you need to prove that $\mathrm{dspace}(s(n)) = \mathrm{DSPACE}(s(n))$?

6. Prove that the set of palindromes over $\Sigma = \{0, 1\}$ is not in $\mathrm{DSPACE}(s)$ if $s = o(\log n)$. (Hint: Think of the machine as a two-way DFA with $2^{O(s)}$ states and apply the argument of Advanced Lecture 3, paying attention to the numbers. If you are careful you can strengthen the hypothesis to $s \leq (1 - \epsilon) \log n$ for any $\epsilon > 0$.)

7. Prove that the set of unary strings of prime length is not in $\mathrm{dspace}(s)$ for $s = o(\log n)$. (Hint: Consider a terminating computation that uses space exactly k on the input 0^n. Show that the machine also uses space exactly k on infinitely many other strings in 0^*, and that these must contain strings of both prime and composite length.)

8. Show that $\mathrm{dspace}(\log \log n)$ is strictly contained in $\mathrm{DSPACE}(\log \log n)$. (Hint: one separating language is $\{a^i b^{2^i} : i \geq 0\}$.)

9. Find a set of numbers such that the set of its binary encodings is in $\mathrm{DSPACE}(1)$ but the set of its unary encodings is not in $\mathrm{dspace}(s)$ with $s = o(\log n)$.

10. Show that all unary languages in $\mathrm{DSPACE}(\log \log n)$ are in $\mathrm{FO} + \mathrm{BIT}$.