MULTIDIGIT MODULAR MULTIPLICATION WITH THE EXPLICIT CHINESE REMAINDER THEOREM

Daniel J. Bernstein

950518 (draft T)

Abstract. Fix coprime moduli $m_1, \dots, m_s$, of a few digits each. Let n be an integer of a few hundred digits. We show how arithmetic modulo n may be performed upon integers u represented as vectors $(u \bmod m_1, \dots, u \bmod m_s)$. This method involves no multiprecision arithmetic, except in an easy precomputation; it is practical in software and extremely well suited for hardware. Our main tool is the Explicit Chinese Remainder Theorem, which says exactly how u differs from a particular linear combination of the remainders $u \bmod m_i$.

1. Introduction

Let $m_1, \dots, m_s$ be positive integers. One may reconstruct a sufficiently small integer u from the sequence $(u \bmod m_1, u \bmod m_2, \dots, u \bmod m_s)$, which we call the **remainder representation** of u; here $u \bmod m_i = u - m_i \lfloor u/m_i \rfloor$ is the remainder when u is divided by m_i.

We may calculate $uv \bmod m_i$ as $(u \bmod m_i)(v \bmod m_i) \bmod m_i$. So we may find the remainder representation of uv from the remainder representations of u and v. This is an example of **remainder arithmetic**. Remainder arithmetic is particularly convenient in hardware since each m_i is handled independently.

In this paper we consider the problem of performing arithmetic modulo an integer n. To fix ideas we consider the problem of computing $x^k \bmod n$, where n is a few hundred digits long; the sufficiently imaginative reader will perhaps be able to find some use for such a computation.

Usual solution: Start from $x \bmod n$. Multiply it by itself to obtain $(x \bmod n)^2$. Reduce that modulo n to obtain $x^2 \bmod n$. Continue with an appropriate sequence of multiplications and reductions so as to end up with $x^k \bmod n$. (See [3, section 4.6.3] and [1, section 1.2].)

All we have to do, then, is multiply and reduce integers modulo n. Conventional wisdom states that it is pointless to use the remainder representation here, since one cannot compute $xy \bmod n$ without first, in effect, converting xy back to its usual representation as a binary expansion.

The conventional wisdom is wrong. In this paper we present a new method of reducing a remainder representation modulo n. We do not generally obtain

---

1991 *Mathematics Subject Classification.* Primary 11Y16.

This paper was included in the author's thesis at the University of California at Berkeley. The author was supported in part by a National Science Foundation Graduate Fellowship.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TeX

the smallest possible result, but we do obtain a small enough result that we may perform further operations safely. This method is highly parallel; we draw it to the attention of hardware designers. It is also practical in software.

The crux of our method is a fact that we dub the **Explicit Chinese Remainder Theorem**. Write $P = m_1 m_2 \cdots m_s$. The usual Chinese Remainder Theorem states that if m_i and m_j are coprime for every $i \neq j$ then the remainder representation of u determines $u \bmod P$. In fact, $u \bmod P$ is a particular linear combination of the remainders $u \bmod m_i$. The *Explicit* Chinese Remainder Theorem says, in a computationally useful form, exactly what multiple of P must be subtracted from that linear combination to obtain u.

The Explicit Chinese Remainder Theorem has been exploited by Montgomery, Silverman, and Couveignes; see [6], [5], and [2]. In fact, our method may be viewed as an optimization of a slight variant of [5, equation 3.4.4].

In section 2 we state and prove the Explicit Chinese Remainder Theorem. In section 3 we show how to compute one of the quantities in this theorem. In section 4 we explain how to convert an integer from the remainder representation to the usual representation. In section 5 we present our method to reduce a remainder representation modulo n. When n is fixed we may precompute various constants; we discuss the precomputation in section 6. In section 7 we discuss possible software and hardware implementation approaches. Everything here has a polynomial analogue, as outlined in section 8.

Remainder arithmetic is commonly called **modular arithmetic**. See [3, section 4.3.2] for an introduction. See also [3, exercise 4.5.2–15] for a method of computing the inverse of v modulo m_i, or more generally the ratio of u and v modulo m_i, when v and m_i are coprime.

2. The Explicit Chinese Remainder Theorem

Before stating this theorem we establish two pieces of notation. First, we write $u \equiv v \pmod{P}$ if $u \bmod P = v \bmod P$; in other words, if $u - v$ is divisible by P. Second, we write $\operatorname{round} z$ for the unique integer i with $|i - z| < 1/2$, when such an integer exists.

The Explicit Chinese Remainder Theorem. *Fix coprime positive integers $m_1, m_2, \ldots, m_s$ with product $P = m_1 m_2 \cdots m_s$. Let $k_1, k_2, \ldots, k_s$ be integers with $k_i P/m_i \equiv 1 \pmod{m_i}$. Let u be an integer with $|u| < P/2$. If $x_i \equiv k_i u \pmod{m_i}$ and $z = \sum_i x_i/m_i$ then $u = Pz - P \operatorname{round} z$.*

Observe that u is a linear combination of $\operatorname{round} z$ and the modified remainders x_i, with coefficients that do not depend on u. We will take advantage of this linearity in section 5.

Proof. We work modulo m_j:

$$Pz = \sum_i x_i \frac{P}{m_i} \equiv x_j \frac{P}{m_j} \equiv k_j u \frac{P}{m_j} \equiv u \pmod{m_j}.$$

The m_j's are pairwise coprime so $Pz \equiv u \pmod{P}$. Write $r = z - u/P$; then $Pr = Pz - u$ is a multiple of P, so r is an integer. But $|z - r| = |u/P| < 1/2$. Thus $r = \operatorname{round} z$, and $u = Pz - Pr = Pz - P \operatorname{round} z$. □

3. Computing the nearest integer to a sum of rationals

We retain the notation of the Explicit Chinese Remainder Theorem. Write $r = \operatorname{round} z$. Assume that $|u| < P/4$, so that $|z - r| = |u/P| < 1/4$.

Montgomery and Silverman [6] observed that to find r, the nearest integer to z, it suffices to have a low-precision approximation to z. They suggested computing z in floating-point arithmetic. We suggest computing z in fixed-point arithmetic:

Lemma 3.1. *Let $t_1, \ldots, t_s$ be real numbers. Let r be an integer with $|\sum_i t_i - r| < 1/4$. If $2^a \geq 2s$ and $q_i = \lfloor 2^a t_i \rfloor$ then $r = \lfloor 3/4 + 2^{-a} \sum_i q_i \rfloor$.*

For us $t_i = x_i/m_i$ and $z = \sum_i t_i$. We may compute q_i by repeatedly doubling x_i modulo m_i and checking for overflows.

Proof. By construction $0 \leq 2^a t_i - q_i < 1$. Add: $0 \leq 2^a \sum_i t_i - \sum_i q_i < s \leq 2^{a-1}$, so $|\sum_i t_i - 1/4 - 2^{-a} \sum_i q_i| \leq 1/4$, so $|r - 1/4 - 2^{-a} \sum_i q_i| < 1/2$. Thus $3/4 + 2^{-a} \sum_i q_i$ differs from $r + 1/2$ by less than $1/2$. □

4. Converting from the remainder representation

To illustrate the practical use of the Explicit Chinese Remainder Theorem we consider the following problem. Again fix coprime positive integers $m_1, m_2, \ldots, m_s$ with product $P = m_1 m_2 \cdots m_s$. Let u be an integer with $|u| < P/4$; say we know u's remainder representation $(u \bmod m_1, u \bmod m_2, \ldots, u \bmod m_s)$. How do we reconstruct the usual representation of u as a binary expansion?

Precomputation: Find $P/m_i \bmod m_i = \prod_{j \neq i} m_j \bmod m_i$ by successive multiplications. The result is coprime to m_i; invert it modulo m_i to obtain k_i. Do this for each i. (See section 6 for another method.)

Next: For each i find an integer $x_i \equiv k_i u \pmod{m_i}$. There are at least three plausible ways to do this: (1) set $x_i = k_i u$; (2) set $x_i = k_i u \bmod m_i$; (3) let x_i be the **least remainder**—either $k_i u \bmod m_i$ or $(k_i u \bmod m_i) - m_i$, whichever has smaller absolute value. We will use method (2).

Next: Compute $Pz = \sum_i x_i (P/m_i)$. We could compute P on the fly, divide by m_i and multiply by x_i, and sum the results. We could precompute P, or a table of P/m_i, in a variety of ways. Several further ideas are discussed in [3, section 4.3.2]. A reasonable approach in practice is to add up $z = x_1/m_1 + \cdots + x_s/m_s$ as a multiprecision rational number, either with binary splitting or term by term, and then multiply by P. The term-by-term version amounts to keeping track of the integers $m_1 \cdots m_i$ and $(x_1/m_1 + \cdots + x_i/m_i) m_1 \cdots m_i$ as i increases. The binary splitting version is useful for large s in conjunction with fast multiplication. It is also useful in hardware.

Finally: Compute $\operatorname{round} z$ by the method of section 3. Multiply $\operatorname{round} z$ by P. Subtract $P \operatorname{round} z$ from Pz to obtain u.

5. Reducing remainder representations modulo n

Again let u be an integer with $|u| < P/4$. Given the remainder representation of u we will, without multiprecision arithmetic, construct the remainder representation of an integer v congruent to u modulo n. Although $|v|$ is not in general smaller than n, it is not too much bigger—it is less than $n \sum_i m_i$. Our method is expressed in the following corollary of the Explicit Chinese Remainder Theorem:

Lemma 5.1. *Fix coprime positive integers $m_1, m_2, \ldots, m_s$ with product P. Let $k_1, k_2, \ldots, k_s$ be integers such that $k_i P/m_i \equiv 1 \pmod{m_i}$. Let u be an integer with $|u| < P/2$. Write $x_i = k_i u \bmod m_i$ and $r = \operatorname{round} \sum_i x_i/m_i$. Then $u \equiv v \pmod{n}$, where*

$$v = \sum_i x_i \left(\frac{P}{m_i} \bmod n \right) - (P \bmod n) r.$$

Furthermore

$$v \equiv \sum_i x_i \left(\frac{P}{m_i} \bmod n \bmod m_j \right) - (P \bmod n \bmod m_j) r \pmod{m_j}$$

and $|v| < n \sum_i m_i$.

Here we see the advantage of focusing on u and v, which are linear combinations of $x_1, \ldots, x_s, r$, rather than on $u \bmod n = v \bmod n$ as in [6], [5], and [2].

Proof. $u = \sum_i x_i P/m_i - Pr \equiv v \pmod{n}$ by the Explicit Chinese Remainder Theorem.

The formula for v modulo m_j follows from the definition of v.

Finally, v is the difference of two nonnegative terms. The first is less than $\sum_i m_i n$, since $x_i < m_i$. The second is less than ns since $r = \operatorname{round} \sum_i x_i/m_i \leq \operatorname{round} \sum_i 1 = \operatorname{round} s = s$. $\square$

It is easy to use these formulas in practice. Given n and the various m_i we precompute tables of k_i, $P/m_i \bmod n \bmod m_j$, and $P \bmod n \bmod m_j$, as explained in the next section. Then, given the remainder representation of u, we compute x_i directly as in section 4, and we compute r as in section 3. Finally, for each j, we compute $v \bmod m_j$ as an appropriate dot product modulo m_j.

In [6] and [2], x_i was chosen as $k_i u$. Like [5] we set $x_i = k_i u \bmod m_i$; this greatly improves our v bound for very little effort. We could reduce $|v|$ down to at most $(n/4) \sum_i m_i$ by using least remainders for x_i and for various numbers modulo n. If it is worth that much hassle to save two bits, then perhaps we should go even further, manipulating all the x_i to make v small.

6. Precomputing everything

To use the method of section 5 we must precompute the $s^2 + s$ quantities $P/m_i \bmod n \bmod m_j$ and $P \bmod n \bmod m_j$, as well as the inverse of P/m_j modulo m_j. We can do all this without too much trouble, if $P \geq 4nm_j$ for each j:

Lemma 6.1. *Fix coprime positive integers $m_1, m_2, \ldots, m_s$ with product P. Let n be a positive integer. Assume that $P/m_j \geq 4n$. For $i \neq j$ let c_{ij} be the inverse of m_i modulo m_j. Set $k_j = \prod_{i \neq j} c_{ij} \bmod m_j$, $n_j = n \bmod m_j$, $p_j = P \bmod n \bmod m_j$, $q_j = \lfloor P/n \rfloor \bmod m_j$, and $e_{ij} = q_i n_j + p_j \bmod m_j$. For $i \neq j$ set $d_{ij} = c_{ij} e_{ij} \bmod m_j$ and $x_{ij} = k_j e_{ij} \bmod m_j$. Set $r_j = \operatorname{round} \sum_{i \neq j} x_{ji}/m_i$ and let d_{jj} be the ratio modulo m_j of $\sum_{i \neq j} c_{ij} x_{ji} - r_j$ and k_j. Then $k_j P/m_j \equiv 1 \pmod{m_j}$ and $q_j n_j + p_j \equiv 0 \pmod{m_j}$ for every j; and $d_{ij} = P/m_i \bmod n \bmod m_j$ for every i and j. Also $|\sum_{i \neq j} x_{ji}/m_i - r_j| < 1/4$.*

Proof. First $k_j P/m_j \equiv \prod_{i \neq j} c_{ij} m_i \equiv 1 \pmod{m_j}$.

Abbreviate $q = \lfloor P/n \rfloor$, so that $P = qn + (P \bmod n)$. Then $q \equiv q_j \pmod{m_j}$ so $q_j n_j + p_j \equiv qn + (P \bmod n) = P \equiv 0 \pmod{m_j}$.

Next $qn \equiv q_i n \pmod{m_i n}$ so $P \equiv q_i n + (P \bmod n) \pmod{m_i n}$. But $0 \le q_i n + (P \bmod n) \le (m_i - 1)n + (n-1) < m_i n$. Hence $P \bmod m_i n = q_i n + (P \bmod n)$. Reduce modulo m_j:

$$m_i \left(\frac{P}{m_i} \bmod n \right) = P \bmod nm_i = q_i n + (P \bmod n) \equiv q_i n_j + p_j \equiv e_{ij} \pmod{m_j}.$$

To prove that $d_{ij} = P/m_i \bmod n \bmod m_j$ we consider separately $i \ne j$ and $i = j$. For $i \ne j$ we have $m_i d_{ij} \equiv m_i c_{ij} e_{ij} \equiv e_{ij} \equiv m_i (P/m_i \bmod n) \pmod{m_j}$. Since m_i is coprime to m_j we have $d_{ij} \equiv P/m_i \bmod n \pmod{m_j}$.

To handle $P/m_j \bmod n \bmod m_j$ we will apply the Explicit Chinese Remainder Theorem to all m_i *except* m_j. The product of this restricted set is P/m_j, and we have $(k_i m_j)(P/m_j)/m_i \equiv 1 \pmod{m_i}$ for $i \ne j$. Define $u = P/m_j \bmod n$; observe that $|u| < n \le P/4m_j$. Furthermore

$$x_{ji} \equiv k_i e_{ji} \equiv k_i m_j (P/m_j \bmod n) = (k_i m_j) u \pmod{m_i}.$$

Hence $u = (P/m_j)z - (P/m_j) \operatorname{round} z$ with $z = \sum_{i \ne j} x_{ji}/m_i$. By construction $r_j = \operatorname{round} z$, so $|z - r_j| = |u m_j / P| < 1/4$. Finally

$$\begin{aligned} k_j u &= k_j \frac{P}{m_j} \sum_{i \ne j} \frac{x_{ji}}{m_i} - k_j \frac{P}{m_j} r_j = k_j \sum_{i \ne j} x_{ji} \frac{P}{m_i m_j} - k_j \frac{P}{m_j} r_j \\ &\equiv k_j \sum_{i \ne j} x_{ji} \frac{P}{m_i m_j} m_i c_{ij} - k_j \frac{P}{m_j} r_j = k_j \sum_{i \ne j} c_{ij} x_{ji} \frac{P}{m_j} - k_j \frac{P}{m_j} r_j \\ &\equiv \sum_{i \ne j} c_{ij} x_{ji} - r_j \equiv k_j d_{jj} \pmod{m_j}. \end{aligned}$$

Hence $d_{jj} \equiv u = P/m_j \bmod n \pmod{m_j}$ as desired. □

We may compute all the quantities in Lemma 6.1 exactly as they are defined. It is convenient to store x_{ij} on top of e_{ij}. To find r_j we use the method of section 3.

We use multiprecision arithmetic in computing $\lfloor P/n \rfloor$ and $P \bmod n$, and in reducing $n, \lfloor P/n \rfloor, P \bmod n$ modulo each m_j. We can save some time and effort by computing p_j as $-q_j n_j \bmod m_j$. Alternatively, when n_j is coprime to m_j, we can compute q_j as the ratio of $-p_j$ and n_j modulo m_j, as suggested in [2]. (The situation of [2] was that P had far more digits than n, so it was impractical to compute $\lfloor P/n \rfloor$. Fortunately, n was coprime to P, so n_j was coprime to m_j.)

7. Implementation ideas

We return to the practical problem of computing $x^k \bmod n$.

Select moduli $m_1, m_2, \ldots, m_s$ with product $P \ge 4(n \sum_i m_i)^2$. Precompute the quantities described in section 6. Now, given the remainder representations of integers t, t' smaller than $n \sum_i m_i$ in absolute value, we may (1) form the remainder representation of tt', which is an integer of magnitude less than $P/4$; and (2) reduce tt' modulo n by the method of section 5. We end up with the remainder representation of a new integer $v \equiv tt' \pmod{n}$. Since v is smaller than $n \sum_i m_i$ in absolute value, it can participate in further multiplications.

So we start from the remainder representation of $x \bmod n$, which is smaller than $n\sum_i m_i$. We repeatedly multiply and reduce in the remainder representation, keeping all intermediate results below $n\sum_i m_i$, until we have reached $v \equiv x^k \pmod{n}$. We convert back to the usual representation of v by the method of section 4. Finally we compute $v \bmod n = x^k \bmod n$.

The same strategy works for arbitrary sequences of ring operations modulo n. For example, we can compute multiples of points on an elliptic curve modulo n, compute convolutions modulo n as in [6] or [5], and so on. We simply have to make sure that P is large enough for the intermediate results.

Software. In practice we spend almost our entire time computing

$$v \bmod m_j = \left(\sum_i x_i \left(\frac{P}{m_i} \bmod n \bmod m_j\right) - (P \bmod n \bmod m_j)r\right) \bmod m_j$$

as per Lemma 5.1. This is a matrix-vector product: we multiply our precomputed matrix, with entries of the form $P/m_i \bmod n \bmod m_j$ and $-(P \bmod n) \bmod m_j$, by the vector $x_1, x_2, \ldots, x_s, r$. We could reduce the jth component of the output modulo m_j in each step, to keep the numbers small, or in one sweep at the end.

Existing computers perform many operations more efficiently than multiplication and addition modulo m_j. In light of the restricted range of inputs we could easily replace all arithmetic with memory lookups.

Extreme example: We could precompute the outputs $x_i(P/m_i \bmod n) \bmod m_j$ for each possible x_i. Then, given $x_1, x_2, \ldots, x_s$, we can simply look up and add up the corresponding outputs. If n is fixed for a long time this might be worthwhile. Less extreme would be, e.g., to break each x_i into a sum of terms of the form $(2^b)(1, 3, 5, \text{or } 7)$ and to precompute corresponding tables. This may be worthwhile on computers with slow multiplication.

Another helpful technique is to insert the matrix entries into carefully written machine code. On some computers it will be possible to keep most or all of the x_i in registers at once. On computers with very fast arithmetic it would be worthwhile to eliminate housekeeping by expanding all loops out of the machine code.

Optimal modulus sizes will depend heavily on such choices and on further details of the computer's arithmetic facilities.

Note that we could store $x_i = k_i u \bmod m_i$ rather than $u \bmod m_i$, and we could modify our precomputed matrix to produce $k_j v \bmod m_j$ rather than $v \bmod m_j$.

Our moduli are not engraved in stone. Given n we could perhaps look for moduli permitting an unusually fast matrix-vector product: for example, where our matrix has many zeros or many repeated entries. Does a "fast matrix" exist for any n? If so, how could we find it? In some situations n itself is not fixed. How quickly can we find a good $(n, m_1, m_2, \ldots, m_s)$?

Hardware. Our reduction method is so dramatically parallel that it cries out for a hardware implementation. The author imagines several levels of chips, of increasing complexity and increasing effectiveness:

1. A straightforward integer matrix-vector multiplication chip.
2. A chip that multiplies a matrix by a vector and reduces the jth component of the output modulo m_j. It would be reasonable for the selection of m_j to be hardwired into the chip.

3. A chip that understands the entire procedure of section 5: given the remainder representation of u, it computes x_i, r, and the remainder representation of v.
4. Like level 3, but the chip stores several remainder representations in named "registers," and can add, subtract, multiply, or reduce registers upon command.
5. Level 4 plus high-level operations such as modular exponentiation.
6. Level 5 plus conversion to and from the remainder representation.
7. Level 6 plus the precomputation procedure of section 6.

8. The polynomial analogue

There is an analogue of the Explicit Chinese Remainder Theorem for polynomials over a field:

The Explicit Chinese Remainder Theorem for Polynomials. *Fix coprime polynomials* $m_1, m_2, \ldots, m_s$ *with product* $P = m_1 m_2 \cdots m_s$. *Let* $k_1, k_2, \ldots, k_s$ *be polynomials such that* $k_i P/m_i \equiv 1 \pmod{m_i}$. *Let* u *be a polynomial with* $\deg u < \deg P$. *Set* $x_i = k_i u \bmod m_i$. *Then* $u = \sum_i x_i P/m_i$.

Note that "round z" for integers has been replaced by 0 for polynomials. Our integer algorithms, *mutatis mutandis*, work for polynomials.

Proof. Work modulo m_j: $\sum_i x_i P/m_i \equiv x_j P/m_j \equiv k_j u P/m_j \equiv u \pmod{m_j}$. The m_j's are pairwise coprime so $\sum_i x_i P/m_i \equiv u \pmod{P}$. By construction $\deg x_i < \deg m_i$. Then both $\sum_i x_i P/m_i$ and u have degree smaller than $\deg P$, so they must be equal. □

When all m_i are monic and linear, the Explicit Chinese Remainder Theorem for Polynomials reduces to the familiar fact that a polynomial is a linear combination of its values at a sufficiently large set of points. See [3, pages 484–486].

References

1. Henri Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, Berlin, 1993.
2. Jean-Marc Couveignes, *Computing a square root for the number field sieve*, in [4], 95–102.
3. Donald E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1981.
4. Arjen K. Lenstra, Hendrik W. Lenstra, Jr. (editors), *The development of the number field sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.
5. Peter L. Montgomery, *An FFT Extension of the Elliptic Curve Method of Factorization*, Dissertation, University of California at Los Angeles, 1992.
6. Peter L. Montgomery, Robert D. Silverman, *An FFT Extension to the $P-1$ Factoring Algorithm*, Mathematics of Computation **54** (1990), 839–854.

5 Brewster Lane, Bellport, NY 11713