Large Numbers, the Chinese Remainder Theorem, and the Circle of Fifths

S. A. FULLING

Texas A&M University

College Station, TX 77843-3368

fulling@math.tamu.edu

1. **The problem.** Consider a recurrence relation such as

$$a_{n+1} = a_n{}^2 + (n+3)na_n\,, \qquad a_0 = 1,$$

whose solutions are integers that grow rapidly with n. (This is a cooked-up example. For a realistic problem of this nature, look at [2], especially the formula at the end, and contemplate calculating Y_{40}; or look at equation (3.65) of [9], whre the last coefficient is a fraction whose denominator has 80 decimal digits.) Suppose that: (1) We do not know how to solve the recurrence in closed form, so we want to use a computer to grind out the values of a_n for, say, $n \le 20$:

$$a_1 = 1, \quad a_2 = 5, \quad a_3 = 75, \quad a_4 = 6975, \quad a_5 = 48845925, \quad \dots.$$

(2) We insist on knowing the answers exactly; floating-point numbers of a fixed precision are not adequate.

The problem is that eventually the numbers will overflow the natural "word size" of the computer. If integers are represented by 16 bits, the largest signed integer is $2^{15} - 1$, the largest unsigned integer is $2^{16} - 1$. If "long" (32-bit) integers are used, we can get up to $2^{32} - 1$ unsigned.

What happens when an integer variable overflows depends on the programming language used. Some systems will give an error message and abort the program. In standard C, the storage of the number "rolls over" like a car's odometer: the most significant digits are lost, without warning. Programs like *Maple* and *Mathematica* do arithmetic with integers of (in principle) arbitrary length, but let us assume that we are not using one of those (else we would have no story to tell).

The most obvious response to this situation would be to represent large integers as arrays of regular (say 16-bit) integers, and to program the arithmetical operations directly on them. Each integer piece is treated exactly like a single digit in hand arithmetic. For example, if $n = n_0 + 2^{16}n_1$ and $m = m_0 + 2^{16}m_1$, then $nm = n_0m_0 + 2^{16}(n_1m_0 + n_0m_1) + 2^{32}n_1m_1$; if we have n_0, etc., stored as 32-bit integers (that are actually smaller than 2^{16}), then n_0m_0 is a 32-bit integer, and we can break it up as $p_0 + 2^{16}p_1$, so that $nm = p_0 + 2^{16}(p_0 + n_1m_0 + n_0m_1) + 2^{32}n_1m_1$. Next we can break up the coefficient of 2^{16}

as $p_1 + 2^{16}p_2$ and throw p_2 into the coefficient of 2^{32}. And so on. Addition and subtraction are easier, but similar; one needs to handle carries and borrows between adjacent 16-bit parts.

This method can be programmed straightforwardly enough, but it has some disadvantages. Multiplying two arrays of length N requires N^2 multiplications of 16-bit numbers. Addition and subtraction require only N operations, but these must be done sequentially, to handle the carries and borrows. This is a disadvantage on a modern parallel computer.

In contrast, the modular method to be described here requires only N operations for input of size N, even for multiplication, and the operations are independent, so they can be performed in parallel. (And, most importantly for our present purposes, the mathematical theory behind this method is much more interesting!)

2. **Review of modular (residue) arithmetic.** An integer $m > 1$ defines a partition of $\mathbf{Z}$ into equivalence classes

$$[0], \ [1], \ \ldots, \ [m-1].$$

(For more leisurely introductions to this subject, see the background reading recommended at the end of this article.) Two numbers are "equivalent (or congruent) modulo m" if their difference is divisible by m. Thus $n \in [r]$ means that $n = qm + r$ with $0 \le r < m - 1$. The modern computer notation is "$r = n \% m$"; r is called the "residue" of n modulo m; m is called a "modulus". (If n_1 and n_2 are in the same equivalence class, the traditional mathematical notation is "$n_1 \equiv n_2$ modulo m", without any implication that either n_j lies in the range of the residues.)

These classes can be added, subtracted, and multiplied according to definitions

$$[a] + [b] = [a + b], \quad \text{etc.}$$

We need to show that these operations are well-defined; that is, the result of applying the definitions does not depend on which elements a and b of the respective classes are chosen:

Suppose that $a = q_a m + r_a$ and $b = q_b m + r_b$. Then $a + b = (q_a + q_b)m + (r_a + r_b)$. If $r_a + r_b \ge m$, we subtract m from it to bring it back into the proper range, and compensate by adding 1 to $q_a + q_b$. If $r_a + r_b < m$, we do nothing. In either case, we get the residue of the sum as

$$r_{a+b} = (r_a + r_b) \% m$$

regardless of what q_a and q_b were.

Similarly,

$$ab = (q_a q_b m + r_a q_b + r_b q_a)m + (r_a r_b),$$

so $r_{ab} = (r_a r_b) \% m$.

This arithmetical consistency makes it possible to blur the distinctions among a, $[a]$, and r_a, representing equivalence classes by residues and understanding all arithmetic to be modulo m. The equivalence classes are called "the integers modulo m"; the set of integers modulo m is denoted by $\mathbf{Z}_m$.

The range 0 to $m - 1$ for the residues is the most obvious one to choose, but actually any string of m consecutive integers can be used to represent the equivalence classes. The

range from $\lceil -m/2 \rceil$ to $\lfloor m/2 - \epsilon \rfloor$, with 0 in the center, is often useful. One should really think of $\mathbf{Z}_m$ as forming a circle ($[m]$ being the same thing as $[0]$), and the labeling of its elements by a particular choice of residues as analogous to a choice of range $a \le \theta < a+2\pi$ for the angular coordinate on a circle.

Observe that odometer-style integer overflow is actually an implementation of arithmetic modulo 2^{16} (or 2^{32}, as the case may be). For unsigned integers, the range of the residues is 0 to $2^{16} - 1$; for signed integers in "two's-complement representation", the range is -2^{15} to $2^{15} - 1$. ("One's-complement representation" is a complication which we shall ignore.)

3. **The Chinese remainder theorem and its musical realization.** This theorem is called "Chinese" because a numerical example of it is stated in a Chinese manuscript of circa 300 A.D. (with author Sun Tsu), and the general case was stated and proved by Ch'in Chiu-Shao in 1247 A.D.

Two integers are called "relatively prime" if their greatest common divisor is 1 (in other words, their prime factorizations have no number in common).

Chinese Remainder Theorem: *Let m_1, ..., m_R be positive integers that are pairwise relatively prime:*

$$\gcd(m_j, m_k) = 1 \quad \text{if } j \ne k.$$

Let $M = m_1 m_2 \cdots m_R$. For any R-tuple of integers, $(u_1, \ldots, u_R)$, there is exactly one integer u such that

$$0 \le u < M \quad \text{and} \quad u \equiv u_j \text{ modulo } m_j \quad \text{for each } j.$$

(In particular, if u_j is restricted to the range 0 to $m_j - 1$, then $u_j = u \mathbin{\%} m_j$. The same statement applies if the ranges of the residues are shifted, or if the range of u is shifted in a similar manner.)

Let's look at an extended example. Suppose that $m_1 = 12$ and $m_2 = 7$. Thus $M = 84$. For any number u between 0 and 83, say $u = 49$, we can calculate the residues:

$$u_1 = 1 \quad \text{because } 49 = 4 \times 12 + 1,$$
$$u_2 = 0 \quad \text{because } 49 = 7 \times 7.$$

In fact, it is easy to see that all the numbers fit into a table, labeled by u_1 horizontally and u_2 vertically, starting with 0 in the upper left corner and moving diagonally downward, then "wrapping" to the top when the bottom is reached, and wrapping to the left whenever the right side is reached:

		C	$C^\sharp$	D	$D^\sharp$	E	F	$F^\sharp$	G	$G^\sharp$	A	$A^\sharp$	B
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
tonic	**0**	0	49	14	63	28	77	42	7	56	21	70	35
minor 2nd	**1**	36	1	50	15	64	29	78	43	8	57	22	71
major 2nd	**2**	72	37	2	51	16	65	30	79	44	9	58	23
minor 3rd	**3**	24	73	38	3	52	17	66	31	80	45	10	59
major 3rd	**4**	60	25	74	39	4	53	18	67	32	81	46	11
4th	**5**	12	61	26	75	40	5	54	19	68	33	82	47
minor 5th	**6**	48	13	62	27	76	41	6	55	20	69	34	83

In other words, although we can continue to think of $\mathbf{Z}_{84}$ as a circle, it also has the two-dimensional structure of a torus, the "product" of two circles of circumferences 12 and 7:

$$\mathbf{Z}_{84} = \mathbf{Z}_{12} \times \mathbf{Z}_7 \,.$$

This example has a musical interpretation. Think of the integers modulo 84 as labeling musical notes, or the keys on a piano. (A standard piano has 88 keys, so there is an overlap of 4 keys at the ends.)
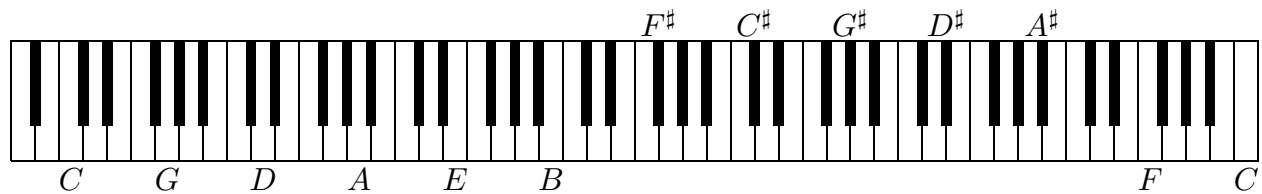
The ratio of the frequencies of the sounds produced by adjacent keys is $\sqrt[12]{2}$, so when we move up by 12 keys we reach the note of frequency twice the one we started from — an interval of one "octave". If we started from a C, this note is also a C. On the other hand, if we move up by 7 keys, we reach a note of frequency almost exactly $\frac{3}{2}$ of the original one. This is called an interval of a "fifth". If we started from C (as "tonic"), this (so-called "dominant") note is a G. If we move up another 7 keys from G, we reach D; and continuing in this way we reach every one of the 12 notes of the scale exactly once before arriving back at a C. This structure is called "the circle of fifths".

(One may wonder, since the basic numbers involved here are 12 and 7, why the intervals are called "octaves" and "fifths". For complicated historical reasons, musicians have two strange rules for counting the notes between C and G or C: (1) don't count the black keys; (2) count both endpoints (in mathematical terms, both $[0]$ and $[m]$). The result is to convert the 12 to an 8 and the 7 to a 5.)

It is important to understand that the existence of the circle of fifths depends crucially on the fact that 12 and 7 are relatively prime. If we replaced the 7 by a 6, we would go from C to $F^\sharp$ to C again, without ever visiting the other notes. If we replaced 7 by 8 or 9 (not a divisor of 12 but not relatively prime to it either), the cycle would be longer but would still end prematurely.

For an example of modular arithmetic in $\mathbf{Z}_{84}$, note that the residues of $v = 2$ are $(v_1, v_2) = (2, 2)$. As previously observed, the residues of 49 are $(u_1, u_2) = (1, 0)$. Then

$$(u_1, u_2) + (v_1, v_2) \equiv ((u_1 + v_1) \,\%\, 12, (u_2 + v_2) \,\%\, 7) = (3, 2),$$

and we see from the table that these are the residues of 51, which is indeed $49+2$. Similarly,

$$(u_1, u_2) \cdot (v_1, v_2) \equiv (u_1 v_1 \,\%\, 12, u_2 v_2 \,\%\, 7) = (2, 0),$$

which is the residue representation of 14; well, $49 \cdot 2 = 98 = 84 + 14$, so $49 \cdot 2$ does equal 14 modulo 84.

For another example of the Chinese remainder theorem, see Example 14.18 in [4], especially the table on p. 656, where the integers modulo $M = 30$ are represented by their residues modulo 2, 3, and 5.

A quick proof of the theorem is possible: We can restate the conclusion this way: For every list of residues, $(u_1, \ldots, u_R)$, there is a unique $u \in \mathbf{Z}_M$ such that $u \% m_j = u_j$. If u and u' are integers such that $u \equiv u'$ modulo m_j for $1 \le j \le R$, then $u - u'$ is a multiple of each m_j. Therefore, $u - u'$ is a multiple of M. (This is the step where relative primality is used.) This shows that the residues uniquely determine u as a member of $\mathbf{Z}_M$. In other words, we have shown that the function mapping $u \in \mathbf{Z}_M$ into its list of residues, $(u_1, \ldots, u_R) \in \mathbf{Z}_{m_1} \times \cdots \times \mathbf{Z}_{m_R}$, is injective. It remains to show that the function is surjective: every R-tuple of residues corresponds to some $u \in \mathbf{Z}_M$. But the number of R-tuples is $m_1 \cdots m_R = M$, the same as the number of integers modulo M, so there is exactly one pigeon in every pigeonhole: the two sets are in one-to-one correspondence.

As we saw earlier in the example, addition, subtraction, and multiplication modulo M can be carried out by performing the corresponding operations on the residues modulo the respective m_j; for example,

$$(u_1, \ldots, u_R) + (v_1, \ldots, v_R) = ((u_1 + v_1) \% m_1, \ldots, (u_R + v_R) \% m_R)$$

yields the residue representation of $u + v$ modulo M. We can think of the representation of an integer (less than M) by a string of residues as analogous to the representation of an integer by a string of decimal digits, with the important difference that under addition, subtraction, and multiplication the residues in each "place" combine only among themselves; there is no "carry" problem!

4. **Implementation.** The practical point of this discussion should now be clear: We choose R relatively prime integers, m_j, all fairly big but smaller than 2^{16}, and we do computer arithmetic with arrays of R integers, $(u_1, \ldots, u_R)$, modulo $(m_1, \ldots, m_R)$. These arrays represent all the integers from 0 up to one less than $M = m_1 \cdots m_R$, which may be much larger than 2^{32}. (In practice one may want to accommodate negative numbers by using the symmetric range, $\lceil -M/2 \rceil \le u \le \lfloor M/2 - \epsilon \rfloor$, but for simplicity let's not discuss the details of that.) This representation of large integers has the advantage previously advertised, that arithmetic can be performed in parallel on the residue components, without carries and borrows and without cross terms in multiplication. (It has some disadvantages, too, which will be mentioned later.)

As a quick toy example, let's calculate 7! with respect to the moduli 13, 11, 9, and 7 — without ever encountering a number larger than $256 = 2^8$. (This is possible because the moduli are all smaller than 2^4.) The residues of 7 with respect to these moduli are $(7, 7, 7, 0)$. For any smaller integer n the moduli are (n, n, n, n). However, it is more efficient to combine some of them:

$$2 \times 5 \equiv (10, 10, 1, 3), \qquad 3 \times 4 \equiv (12, 1, 3, 5).$$

Then

$$2 \times 3 \times 4 \times 5 \equiv (120, 10, 3, 15) \equiv (3, 10, 3, 1);$$

$$6! \equiv (18, 60, 18, 6) \equiv (5, 5, 0, 6);$$

$$7! \equiv (35, 35, 0, 0) \equiv (9, 2, 0, 0).$$

To check this, note that

$$7! = 5040$$
$$= 387 \times 13 + 9$$
$$= 458 \times 11 + 2$$
$$= 560 \times 9 + 0$$
$$= 6! \times 7 + 0.$$

(Later we'll see how to find the 5040 from the $(9, 2, 0, 0)$ without knowing the answer beforehand.)

Several years ago an undergraduate student, Davin Potts, wrote under my direction some C++ functions to implement residue arithmetic for large integers [10]. We recognized immediately that the problem was a natural one for object-oriented programming — specifically for "polymorphism", or "operator overloading". The point is that in ordinary C code, when large integers are represented by arrays, one can no longer write operations like

$$\mathtt{c = a + b;}.$$

One needs to write something like

$$\mathtt{c = sum(a, b);},$$

where $\mathtt{sum}$ is the function (subroutine) one has written to carry out the addition. Even worse, if the integer is represented by a C structure that must be passed by reference, one might have

$$\mathtt{add(\&a, \&b, pc);}$$
$$\mathtt{c = *pc;}.$$

In C++, instead, we can define the R-tuples of residues to be a "class" named $\mathtt{residue\_int}$ and can define "+", etc., to act on that class in the usual way:

$$\mathtt{residue\_int\ a,\ b,\ c;}$$
$$\mathtt{c = a + b;}.$$

We also need to be able to add or multiply a $\mathtt{residue\_int}$ by an ordinary integer; this is done by defining "friend" functions in C++.

The default moduli in the program [10] are the 5 largest prime numbers less than 2^{16}:

$$65449, \quad 65479, \quad 65497, \quad 65519, \quad 65521.$$

Their product is $1,204,964,463,846,332,731,259,513 \approx 10^{24}$, so we can represent all unsigned integers less than that value. The program is flexible enough to accommodate various numbers of moduli. The residues themselves are represented as long integers (32 bits), so that they can be multiplied without overflow.

5. **Reconstructing an integer from its residues.** Unfortunately, the list of residues of a large integer is not intuitively meaningful to a human reader. After a calculation one will probably want to convert from residues back into standard decimal notation. In the example of $\mathbf{Z}_{12} \times \mathbf{Z}_7$ we could do that by reading the rectangular table, but that method is not practical if the moduli are large and there are more than 2 of them.

A good reconstruction algorithm requires some more number theory. Here is how it works in the context of our example, $m_1 = 12$ and $m_2 = 7$.

First we need to find a number c that is the reciprocal (inverse) of one modulus modulo the other modulus. *Claim:* 3 is the reciprocal of 12 modulo 7 (in other words, $[12]^{-1} = [3]$ in the number system $\mathbf{Z}_7$). *Proof:*

$$12 \times 3 = 36 = 5 \times 7 + 1 \equiv 1 \text{ modulo } 7.$$

Exercise: Show that 7 is its own reciprocal modulo 12. The calculation is hidden in the previous text. Both reciprocals can also be read off from the table. A systematic way of finding reciprocals is mentioned below.

Knowing that $c = [m_1]^{-1}$ in $\mathbf{Z}_{m_2}$, and given a residue list (u_1, u_2), we can now define

$$v_2 = (u_2 - u_1)c \% m_2$$

and get the corresponding integer modulo M as

$$u = v_2 m_1 + u_1 \, .$$

Proof: Obviously $u \% m_1 = u_1$.

$$\begin{aligned} u \% m_2 &= [(u_2 - u_1)cm_1 \% m_2] + [u_1 \% m_2] \\ &= [(u_2 - u_1) \% m_2] + [u_1 \% m_2] \\ &= u_2 \% m_2 \, . \end{aligned}$$

Finally, one can check that $0 \le u < M = m_1 m_2$.

Example: In $\mathbf{Z}_{84}$ consider $(6, 4)$. Since $c = 3$ in this case, $v_2 = -6 \% 7 = 1$. Thus $u = 1 \times 12 + 6 = 18$. This agrees with our table ($F^\sharp$, major 3rd).

This method can be iterated for more than two moduli. One needs first to find the reciprocal of each modulus with respect to each later modulus in the list:

$$c_{jk} m_j \equiv 1 \text{ modulo } m_k \, .$$

Start the algorithm by setting $(v_1, \ldots, v_R) = (u_1, \ldots, u_R)$. Then at step j for $1 \le j < R$,

$$\text{replace } v_k \text{ by } (v_k - v_j)c_{jk} \% m_k$$

for $j < k \le R$. Finally,

$$u = v_1 + v_2 m_1 + v_3 m_2 m_1 + \cdots + v_R m_{R-1} \cdots m_1 \, . \tag{$*$}$$

Exercise: Show in this way that $7! = 5040$.

Of course, when this process is carried out in a real problem, u and some of the terms in $(*)$ must be expected to exceed the word size of the machine. It may be that at this point only the rough magnitude of u is needed, and a conversion of the residue representation into a floating-point number, rather than an exact integer, will suffice. A C++ function to do that is included in [10].

Modular arithmetic does have some drawbacks. Although a modular representation is very convenient for addition, subtraction, and multiplication, it is not good at all for division, or for telling which of two integers is the larger, or for testing whether the integer is about to grow outside the allowed range set by the product of the moduli, M. (These last two points are especially severe when the range 0 to $M - 1$ is used, since it may be impossible to predict whether the result of a subtraction will be a negative number, which is not allowed.) To perform these operations one almost has to convert back to the form $(*)$. (See, however, [7] and [8].)

Another problem is that for a given underlying word size (say 2^{16}), there is an upper limit on the integers that can be represented, since there is a largest M that can be obtained as a product of prime (or relatively prime) numbers less than the word size. This limitation is seldom important in practice.

6. **The extended Euclidean algorithm.** One hole left in our description of the method is an algorithm for finding the reciprocals c_{jk}. An example of how this can be done appears in [4] on p. 649 (Example 14.13). The point is that the Euclidean algorithm for finding the greatest common divisor of m_j and m_k yields as byproducts two integers a and b such that

$$am_1 + bm_2 = \gcd(m_1, m_2) \quad [\, = 1, \text{ since the } m_j \text{ are relatively prime}].$$

It follows that $[a] = [m_1]^{-1}$ in $\mathbf{Z}_{m_2}$ and $[b] = [m_2]^{-1}$ in $\mathbf{Z}_{m_1}$.

The "extended Euclidean algorithm" (which keeps track of the numbers a and b as well as the gcd) is extensively discussed by Knuth at the beginning of his first volume [5] as an example of the proof of validity of an algorithm by mathematical induction. It is implemented in [10].

7. **Bibliographical remarks.** The primary research citation on the practical use of modular arithmetic for large numbers is [1]. The method is extensively discussed by Knuth in his second volume [6]; that book is my primary source for the Chinese remainder theorem and its proof. Szabo and Tanaka [11] also studied computer applications of modular arithmetic, but they were more interested in working with small moduli and building the modular algorithms into the hardware. Davis and Hersh [3] give a readable introduction to the Chinese remainder theorem consisting primarily of interesting historical commentary. Grimaldi [4] provides a standard textbook account of residue arithmetic.

References

1. I. Borosh and A. S. Fraenkel, Exact solutions of linear equations with rational coefficients by congruence techniques, *Math. Comput.* **20**, 107–112 (1966).

2. J. A. Campbell, Computation of a class of functions useful in the phase-integral approximation. I, *J. Comput. Phys.* **10**, 308-315 (1972).

3. P. J. Davis and R. Hersh, *The Mathematical Experience* (Birkhäuser, Boston, 1981), pp. 187–195.

4. R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, 3rd ed. (Addison–Wesley, Reading, 1994), Chap. 14.

5. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, 2nd ed. (Addison–Wesley, Reading, 1973), pp. 14–17.

6. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, 2nd ed. (Addison–Wesley, Reading, 1981), Sec. 4.3.2.

7. M.-L. Lin, E. Leiss, and B. McInnis, Division and sign detection algorithms for residue number systems, *Comp. & Maths. with Appls.* **10**, 331-342 (1984).

8. M. Lu and J.-S. Chiang, A novel division algorithm for the residue number system, *IEEE Transac. Comput.* **41**, 1026–1032 (1992).

9. V. B. Mandelzweig, Quasilinearization method and its verification on exactly solvable models in quantum mechanics, *J. Math. Phys.* **40**, 6266–6291 (1999).

10. D. M. Potts, The RESIDUE_INT software package, `http://rainbow.uchicago.edu/~dmpotts/residue_int/` .

11. S. Szabo and R. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology* (McGraw–Hill, New York, 1967).