Remainders

We learned how to multiply and divide in elementary school.

As adults we perform division mostly by pressing the $\div$ key on a calculator. This key supplies the *quotient*. In numerical analysis and most other parts of math, the quotient is what is needed.

But when division is performed in number theory and cryptography, the quotient usually does not matter and we just need the *remainder*.

Example. Divide 23 by 4. The answer is 5.75 using floating-point arithmetic. Using integer arithmetic, the quotient is 5 and the remainder is 3. We have $23 \div 4 = 5$ with 3 left over as the remainder. We can also write $23 \div 4 = 5 + \frac{3}{4}$. We can multiply both sides by 4 to get $23 = 5 \times 4 + 3$.

In a computer, numbers are stored either as floats (doubles) or as integers (ints). The arithmetic operations performed on numbers differ according as what type they are.

If x and y are floats and $y \neq 0.0$, then x/y is a floating-point number which is a good approximation to the real value of x/y.

If x and y are ints and $y \neq 0$, then x/y is the quotient (an int) equal to the floor of the real number x/y.

Example. The code

```
int x, y, z; x = 23; y = 4; z = x/y;
```

produces $z = 5$. The remainder is ignored.

There is an operator to give remainders in computer languages. In C and Java the operator is "%".

Example. The code

```
int x, y, z; x = 23; y = 4; z = x%y;
```

produces $z = 3$. The quotient is ignored.

Many computers have a machine instruction that produces both the quotient and remainder when it divides an integer by another integer. When a compiler uses this instruction it usually keeps one result and discards the other.

Example. The code

```
int x, y, z; x = 23; y = 4; z = (x/y)*y;
```

produces $z = 20$. The compiler does not "simplify" this code to `z = x;` by canceling the `y`s, as it would if `x`, `y` and `z` were floats. In the integer case, the code `(x/y)*y` gives the largest multiple of y that is $\leq x$.

The integer divide machine instruction on different computers may give different answers when one or both operands are negative, but all machines agree on positive operands.

For example, when a computer divides -23 by 4 it may give a quotient of -6 and a remainder of $+1$, the "correct" answer for number theory, or it may truncate the quotient to -5 and give a remainder of -3.

You should experiment with integer division of positive and negative integers on your computers. What answer does it give for $(+23)/(+4)$, $(-23)/(+4)$, $(+23)/(-4)$, $(-23)/(-4)$, $(+23)\%(+4)$, $(-23)\%(+4)$, $(+23)\%(-4)$, and $(-23)\%(-4)$?

Divisibility

The most important remainder is 0. Note that $b\%a = 0$ iff the real number b/a is an integer.

Definition. When a and b are integers and $a \neq 0$ we say a *divides* b, and write $a \mid b$, if $b\%a = 0$, that is, if b/a is a whole number.

Theorem. Let a, b and c be integers. If $a \mid b$ and $b \mid c$, then $a \mid c$.

Proof: By hypothesis, the two quotients b/a and c/b are whole numbers. Therefore their product, $(b/a) \times (c/b) = c/a$, is a whole number, which means that $a \mid c$.

Theorem. Let a, b, c, x and y be integers. If $a \mid b$ and $a \mid c$, then $a \mid (bx + cy)$.

Proof: We are given that the two quotients b/a and c/a are whole numbers. Therefore the linear combination $(b/a) \times x + (c/a) \times y = (bx + cy)/a$ is a whole number, which means that $a \mid (bx + cy)$.

Theorem (The division algorithm): Suppose $a > 0$ and b are two integers. Then there exist two unique integers q and r such that $0 \leq r < a$ and $b = aq + r$.

Definition. The integers q and r in this theorem are called the *quotient* and *remainder* when b is divided by a.

We use the notation $\lfloor x \rfloor$, the *floor* of x, to mean the largest integer $\leq x$, and $\lceil x \rceil$, the *ceiling* of x, to mean the smallest integer $\geq x$.

Example. $\lfloor 3 \rfloor = 3 = \lceil 3 \rceil$. $\lfloor 3.14 \rfloor = 3$. $\lceil 3.14 \rceil = 4$. $\lfloor -3.14 \rfloor = -4$. $\lceil -3.14 \rceil = -3$.

With this notation, the quotient q may be written $q = \lfloor b/a \rfloor$. We use the notation $b \bmod a$ for the remainder r. Note that $b \bmod a = b - a \times \lfloor b/a \rfloor$.

Arithmetic with Large Integers

Theorem (positional number systems): Let b be an integer greater than 1. Let n be a positive integer. Then n has a unique representation in the form

$$n = \sum_{i=0}^{k} d_i b^i,$$

where $k \geq 0$ is an integer, the d_i are integers in $0 \leq d_i \leq b-1$ and $d_k \neq 0$.

The number b is called the *base* of the number system. The d_i are the *digits* of n in base b.

Base 10 is familiar:
$362_{10} = 3 \cdot 10^2 + 6 \cdot 10^1 + 2 \cdot 10^0$.

Computer scientists often use base 2:
$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13_{10}$.

Cryptographic algorithms need arithmetic with large integers. Computer hardware has a fixed maximum size, such as $2^{31} - 1$, for the integers it can handle directly. Cryptographic algorithms use much larger integers than this maximum value.

Integers greater than the natural word size are stored in arrays with a fixed number of bits per word. It would be wasteful memory usage to store only one bit per word. Usually, all or nearly all bits of a computer word are used to store a multi-precision integer in an array. Several libraries of procedures for arithmetic with large integers are available, such as GNU-MP and the Java Bigint class.

The basic operations of arithmetic are addition, subtraction, multiplication and division. In order to perform these operations on large integers we represent the numbers in a convenient base, like $b = 2^{30}$, with their digits in this base stored in arrays.

Suppose we use base b and we wish to add $A = \sum_{i=0}^{k} a_i b^i$ to $B = \sum_{i=0}^{m} b_i b^i$. If $k \neq m$, prepend enough leading 0 digits to the shorter number to give the two numbers the same length. After this has been done, assume the problem is to add $A = \sum_{i=0}^{k} a_i b^i$ to $B = \sum_{i=0}^{k} b_i b^i$. Call the sum $C = \sum_{i=0}^{k+1} c_i b^i$. Note that the sum might have one more digit than the summands. The addition algorithm is to add corresponding digits of A and B to form each digit of C, and carry a 1 if the digit sum is $\geq b$. Here is the algorithm.

[Addition: $C = A + B$ using base b arithmetic]
Input: The base b digits of A and B.
Output: The base b digits of $C = A + B$.

```
carry = 0
for (i = 0 to k) {
        c_i = a_i + b_i + carry
        if (c_i < b) { carry = 0 }
        else { carry = 1; c_i = c_i - b }
        }
c_{k+1} = carry
```

Trace through the algorithm to add 462 and 159 in base $b = 10$.

The product of a k-digit integer times an m-digit integer has either $k + m$ or $k + m - 1$ digits (or is zero). Suppose we wish to multiply $A = \sum_{i=0}^{k-1} a_i b^i$ times $B = \sum_{i=0}^{m-1} b_i b^i$. Call the product $C = \sum_{i=0}^{k+m-1} c_i b^i$. Note that the high-order digit might be 0. The elementary school method forms partial products $b_i \times A$, shifts their digits into appropriate columns and adds the shifted partial products. In a computer, it saves space to do the addition concurrently with the multiplication. Here is the algorithm in pseudocode.

[Multiplication: $C = A \times B$ using base b arithmetic]
Input: The base b digits of A and B.
Output: The base b digits of $C = A \times B$.

```
carry = 0
for (i = 0 to k + m − 1) { c_i = 0 }
for (i = 0 to k − 1) {
        carry = 0
        for (j = 0 to m − 1) {
                t = a_i × b_j + c_{i+j} + carry
                c_{i+j} = t mod b
                carry = ⌊t/b⌋
        c_{m+i+1} = carry }
        }
```

Trace through the algorithm to multiply 462 times 159 in base $b = 10$.

In order to analyze the complexity of algorithms that use arithmetic we will need to know the time taken by the four arithmetic operations. We do not concern ourselves with the actual time taken, since this time depends on the computer hardware. Rather we will count the number of basic steps. The basic steps we consider are adding, subtracting or multiplying two 1-bit numbers, or dividing a 2-bit number by a 1-bit number, giving a quotient and a remainder. These are called *bit operations*.

Furthermore, we will not worry about the exact count of bit operations. We will use the big-O notation to approximate the growth rate of the number of bit operations as the length of the operands grows.

Definition. If f and g are functions defined and positive for all sufficiently large x, then we say f is $O(g)$ if there is a constant $c > 0$ so that $f(x) < cg(x)$ for all sufficiently large x.

Theorem (Complexity of arithmetic): One can add or subtract two k-bit integers in $O(k)$ bit operations. One can multiply two k-bit integers in $O(k^2)$ bit operations. One can divide a $2k$-bit dividend by a k-bit divisor to produce a k-bit quotient and a k-bit remainder in $O(k^2)$ bit operations.

There is a similar theorem for complexity of arithmetic on operands that do not have the same length.

Definition. We say that an algorithm *runs in polynomial time* if there is a d and a constant $c > 0$ so that for every input I of length k bits, the algorithm on input I finishes in no more than ck^d bit operations.

Base changing, addition, subtraction, multiplication and division of integers can be done by algorithms that run in polynomial time.

The degree d of the polynomial is 1 for addition and subtraction. It is 2 for multiplication and division.

There are fancier algorithms for multiplication and division when k is very large that reduce the execution time to about $ck \log k$.

Greatest Common Divisors

Definition. When a and b are integers and not both zero we define the *greatest common divisor* of a and b, written $\gcd(a, b)$, as the largest integer which divides both a and b. We say that the integers a and b are *relatively prime* if their greatest common divisor is 1.

It is clear from the definition that $\gcd(a, b) = \gcd(b, a)$. One way to compute the greatest common divisor of two nonzero integers is to list all of their divisors and choose the largest number which appears in both lists. Since d divides a if and only if $-d$ divides a, it is enough to list the positive divisors.

Example. Find the greatest common divisor of 4 and 14.

The positive divisors of 4 are 1, 2, 4.

The positive divisors of 14 are 1, 2, 7, 14.

The largest number in the intersection of these two lists is $\gcd(4, 14) = 2$.

A much faster way to compute GCDs depends on this theorem.

Theorem (GCDs and division): If a is a positive integer and b, q and r are integers with $b = aq + r$, then $\gcd(b, a) = \gcd(a, r)$.

Proof: Let $g = \gcd(b, a)$ and $h = \gcd(a, r)$.

Since h divides both a and r, it must divide the linear combination $b = aq + r$. Therefore, h divides both b and a, so it divides $g = \gcd(b, a)$.

Since g divides both b and a, it must divide the linear combination $r = b - aq$. Therefore, g divides both a and r, so it divides $h = \gcd(a, r)$.

Both g and h are positive integers. We have shown that $g \mid h$ and $h \mid g$, that is, both h/g and g/h are positive integers. But $h/g = 1/(g/h)$ and the only positive integer that is the reciprocal of a positive integer is 1. Therefore, $g = h$.

Euclidean Algorithm

Theorem (Simple form of the Euclidean algorithm): Let $r_0 = a$ and $r_1 = b$ be integers with $a \geq b > 0$. Apply the division algorithm iteratively to obtain

$$r_i = r_{i+1}q_{i+1} + r_{i+2} \text{ with } 0 < r_{i+2} < r_{i+1}$$

for $0 \leq i < n-1$ and $r_{n+1} = 0$. Then $\gcd(a, b) = r_n$, the last nonzero remainder.

Example. To find $\gcd(4, 14)$ we let $r_0 = a = 14$ and $r_1 = b = 4$. Now $14 = 4 \cdot 3 + 2$, so $q_1 = 3$ and $r_2 = 2$. Next, $4 = 2 \cdot 2 + 0$, so $q_2 = 2$ and $r_3 = 0$. Since $r_3 = 0$, the answer is $\gcd(4, 14) = r_2 = 2$.

When 4 and 14 are changed to much larger numbers, this method is much faster than listing all the divisors of the two numbers.

Here is the algorithm in iterative form.

[Euclidean Algorithm]
Input: Integers $a \geq b > 0$.
Output: $g = \gcd(a, b)$.

$g = a$; $t = b$
`while` $(t > 0)$ $\{$
$\quad q = \lfloor g/t \rfloor$
$\quad w = g - qt$
$\quad g = t$
$\quad t = w$
$\quad \}$
`return` g

Here is a recursive version, one line in C.

```
gcd(a,b) int a,b; {return(b ? gcd(b, a%b) : a);}
```

Example. Compute the greatest common divisor of 165 and 285.

$$\begin{aligned} 285 &= 1 \times 165 + 120 \\ 165 &= 1 \times 120 + 45 \\ 120 &= 2 \times 45 + 30 \\ 45 &= 1 \times 30 + 15 \\ 30 &= 2 \times 15 + 0, \end{aligned}$$

so $\gcd(165, 285) = 15$.

Theorem. If the integers a and b are not both 0, then there are integers x and y so that $ax + by = \gcd(a, b)$.

Example. Above, we found that $\gcd(285, 165) = 15$. Now let us find x and y with $285x + 165y = \gcd(285, 165) = 15$.

Beginning with the next to last equation in that example and working backwards, we find

$$\begin{aligned} 15 &= 45 - 30 = 45 - (120 - 2 \times 45) \\ 15 &= 3 \times 45 - 120 \\ 15 &= 3(165 - 120) - 120 \\ 15 &= 3 \times 165 - 4 \times 120 \\ 15 &= 3 \times 165 - 4(285 - 165) \\ 15 &= 7 \times 165 - 4 \times 285. \end{aligned}$$

Thus $x = -4$ and $y = 7$.

The following algorithm finds the same answer without working backwards.

[Extended Euclidean Algorithm]
Input: Integers $a \geq b > 0$.
Output: $g = \gcd(a, b)$ and x and y with $ax + by = \gcd(a, b)$.

$x = 1$; $y = 0$; $g = a$; $r = 0$; $s = 1$; $t = b$
`while` $(t > 0)$ $\{$
$q = \lfloor g/t \rfloor$
$u = x - qr$; $v = y - qs$; $w = g - qt$
$x = r$; $y = s$; $g = t$
$r = u$; $s = v$; $t = w$
$\}$
`return` (g, x, y)

To prove that the simple form of the Euclidean Algorithm works, use induction and the theorem about GCDs and division to show that

$$\gcd(r_i, r_{i+1}) = \gcd(r_{i+1}, r_{i+2})$$

for each $i = 0, 1, 2, \ldots n - 1$. The final step is $\gcd(r_0, r_1) = \gcd(r_n, 0) = r_n$.

To prove that the Extended Euclidean Algorithm is correct, use induction to show that each time the `while` loop begins and ends we have

$$xa + yb = g \quad \text{and} \quad ra + sb = t.$$

Finally, note that if the variables x, y, r and s are deleted from the Extended Euclidean Algorithm, the result is the simple iterative form of the Euclidean Algorithm, so that it ends with $g = \gcd(a, b)$.

An example of Extended Euclid

Trace through the Extended Euclidean algorithm to find x and y with
$285x + 165y = \gcd(285, 165) = 15$.

The values are shown each time the `while` loop begins.

x	y	g	r	s	t	q
1	0	285	0	1	165	1
0	1	165	1	-1	120	1
1	-1	120	-1	2	45	2
-1	2	45	3	-5	30	1
3	-5	30	-4	7	15	2
-4	7	15	11	-19	0	$-$

The last line shows that

$$285(-4) + 165(+7) = 15 = \gcd(285, 165).$$

Theorem (Complexity of the Euclidean algorithm, Lamé, 1845). The number of steps (division operations) needed by the Euclidean algorithm to find the greatest common divisor of two positive integers is no more than five times the number of decimal digits in the smaller of the two numbers.

This was the first nontrivial algorithm to have its worst-case complexity determined. The proof shows that the worst case happens when one computes the GCD of two consecutive Fibonacci numbers. The average complexity is much harder to determine. (It is about 1.94 times the number of decimal digits in the smaller of the two input numbers.)

Corollary. The number of bit operations needed by the Euclidean algorithm to find the greatest common divisor of two positive integers is $O((\log_2 a)^3)$, where a is the larger of the two numbers.

This shows that we can compute GCDs in polynomial time, and the polynomial has low degree (3), so it is easy to compute GCDs.

Primes

Definition. A *prime number* is an integer greater than 1 which is divisible only by 1 and itself, and by no other positive integer. A *composite number* is an integer greater than 1 which is not prime.

A composite number n has a positive divisor other than 1 and itself. This factor must be less than n and greater than 1.

The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 and 37.

The first few composite numbers are 4, 6, 8, 9, 10, 12, 14, 15, 16, 18 and 20.

The integers $4 = 2 \cdot 2$, $12 = 2 \cdot 2 \cdot 3$ and $63 = 3 \cdot 3 \cdot 7$ are all composite because they each have divisors other than 1 and themselves.

Theorem. Let a, b and c be positive integers. If $a \mid bc$ and $\gcd(a, b) = 1$, then $a \mid c$.

Proof: Since a and b are relatively prime, there are integers x and y so that $ax+by = \gcd(a, b) = 1$. Multiply by c to get $axc + bcy = c$. Clearly $a \mid a$. Also, $a \mid bc$ by the hypothesis. Therefore, a divides $a(xc) + (bc)y = c$ by the theorem just before the division algorithm.

Theorem. If a prime p divides a product $a_1 a_2 \cdots a_n$ of positive integers, then it divides at least one of them.

Proof: We use mathematical induction on the number n of factors. If $n = 1$, there is nothing to prove. Assume the statement is true for n factors. Suppose the prime p divides a product of $n + 1$ positive integers $a_1 a_2 \cdots a_n a_{n+1}$. If $p \mid a_1$, we are done. Otherwise, p is relatively prime to a_1 because p has only the divisors 1 and p, and p doesn't divide a_1, so $\gcd(p, a_1) = 1$. By the previous theorem, p divides the product $a_2 a_3 \cdots a_n a_{n+1}$ of n factors, and so p must divide one of these n numbers by the induction hypothesis.

The Fundamental Theorem of Arithmetic

Theorem. Every integer greater than 1 can be written as a product of primes, perhaps with just one prime, and this product is unique if the primes are written in nondecreasing order.

Proof (by contradiction).

Suppose some number could not be written as the product of primes. Let n be the smallest such number. Then n cannot be prime, because each prime is the "product" of one prime. Thus n is composite, say, $n = ab$ with $1 < a < n$ and $1 < b < n$. Since a and b are smaller than n, they can be written as the product of primes, and so $n = ab$ can be so written, too.

Proof (by contradiction) continued.

Now suppose the factorization of n into primes is not unique. Then n has two different factorizations

$$n = p_1 p_2 \cdots p_j = q_1 q_2 \cdots q_k$$

into primes. Cancel any common primes on the two sides, so that no prime appears on both sides. There must be at least one prime on each side since we assume the two factorizations of n differ. Then the prime p_1 must divide one number q_i on the other side. But q_i is also prime, so $q_i = p_1$ and so not all common factors have been canceled. This contradiction shows that the factorization of n must be unique.

Suppose the positive integer n is factored into the product of primes, and the primes are in nondecreasing order. The fundamental theorem of arithmetic says that this representation is unique. If we collect repeated prime factors and write them as the power p^e of a prime, we have the following *standard representation*:

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k} = \prod_{i=1}^{k} p_i^{e_i},$$

where $p_1, p_2, \ldots, p_k$ are the distinct primes that actually divide n and $e_i \geq 1$ is the number of factors of p_i dividing n.