

## Divisibility

Definition: When  $a$  and  $b$  are integers and  $a \neq 0$  we say  $a$  **divides**  $b$ , and write  $a|b$ , if  $b/a$  is a whole number.

**THEOREM:** Let  $a$ ,  $b$  and  $c$  be integers. If  $a|b$  and  $b|c$ , then  $a|c$ .

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  $a|c$ .

**THEOREM:** Let  $a$ ,  $b$ ,  $c$ ,  $x$  and  $y$  be integers. If  $a|b$  and  $a|c$ , then  $a|bx + cy$ .

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  $a|(bx + cy)$ .

**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  $\geq x$ .

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

## 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  $k$  is a positive 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.

Cryptographic algorithms require 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 hardware 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. On the other hand, it would be difficult to perform arithmetic on large numbers if each word were filled completely with bits of the large integer. A standard compromise often uses all but two bits of each word to store bits of large numbers. For example, many libraries of procedures for arithmetic with large integers pack 30 bits into each 32-bit word.

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 with their digits 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  $k$ ) {

$c_i = a_i + b_i + \text{carry}$

    if ( $c_i < b$ ) { carry = 0 }

    else { carry = 1;  $c_i = c_i - b$  }

}

$c_{k+1} = \text{carry}$

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) { ci = 0 }
for (i = 0 to k - 1) {
    carry = 0
    for (j = 0 to m - 1) {
        t = ai × bj + ci+j + carry
        ci+j = t mod b
        carry = ⌊t/b⌋
    }
    cm+i+1 = carry }
}
```

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. 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.

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

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

## 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.

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

## 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: Compute the greatest common divisor of 165 and 285.

$$285 = 1 \times 165 + 120$$

$$165 = 1 \times 120 + 45$$

$$120 = 2 \times 45 + 30$$

$$45 = 1 \times 30 + 15$$

$$30 = 2 \times 15 + 0,$$

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

$$15 = 45 - 30 = 45 - (120 - 2 \times 45)$$

$$15 = 3 \times 45 - 120$$

$$15 = 3(165 - 120) - 120$$

$$15 = 3 \times 165 - 4 \times 120$$

$$15 = 3 \times 165 - 4(285 - 165)$$

$$15 = 7 \times 165 - 4 \times 285.$$

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

[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** ( $g, x, y$ )

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.

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.