Based on K. H. Rosen: Discrete Mathematics and its Applications.

Lecture 17: Integer Representation. Algorithms for integer operations.
Section 4.2

1 Integer Representation. Algorithms for Integer Operations

1.1 Integer Representation

There different ways to represent integers based on choosing different basis b to write the numbers. Computers usually use binary notation (with 2 as the base) when carrying out arithmetic, and octal (base 8) or hexadecimal (base 16) notation when expressing characters, such as letters or digits.

Definition 1. Let b be an integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots a_1 b + a_0,$$

where k is a nonnegative integer, $a_0, a_1, \ldots, a_k$ are nonnegative integers less than b, and $a_k \neq 0$.

Example 2. (Binary Expansion) Choosing 2 as the base gives **binary expansions** of integers. In binary notation each digit is either a 0 or a 1 For example, suppose that we want to compute the decimal expression for $(101011111)_2$:

$$(101011111)_2 = 1(2^8)+0(2^7)+1(2^6)+0(2^5)+1(2^4)+1(2^3)+1(2^2)+1(2^1)+1(2^0) = 351$$

(BASE CONVERSION) We will now describe an algorithm for constructing the base b expansion of an integer n.

1. First, divide n by b to obtain a quotient and remainder, that is,

   $$n = bq_0 + a_0, \quad \text{where} \quad 0 \leq a_0 < b.$$

   The remainder, a_0, is the rightmost digit in the base b expansion of n.

2. Next, divide q_0 by b to obtain

   $$q_0 = bq_1 + a_1, \quad \text{where} \quad 0 \leq a_1 < b.$$

   We see that aa_1 is the second digit from the right in the base b expansion of n.

3. Continue this process, successively dividing the quotients by b, obtaining additional base b digits as the remainders. This process terminates when we obtain a quotient equal to zero. It produces the base b digits of n from the right to the left.

1.2 Algorithms for Integer Operations

Consider the problem of **adding two integers in binary notation**. To add a and b, first add their rightmost bits. This gives

$$a_0 + b_0 = c_0 \cdot 2 + s_0,$$

where s_0 is the rightmost bit in the binary expansion of $a + b$ and c_0 is the carry, which is the carry. Then add the next pair of bits and the carry,

$$a_1 + b_1 + c_0 = c_1 \cdot 2 + s_1,$$

where s_1 is the next bit (from the right) in the binary expansion of $a + b$, and c_1 is the carry. Continue this process.

> **procedure** add(a, b: positive integers)
> (The binary expressions of a, b are $(a_{n-1}, a_{n-2}, \ldots, a_0)$ and $(b_{n-1}, b_{n-2}, \ldots, b_0)$)
> $c = 0$
> **for** $j = 0$ **to** $n - 1$:
> $d = \lfloor (a_j + b_j + c)/2 \rfloor$ (quotient c_j)
> $s_j = a_j + b_j + c - 2d$ (remainder s_j)
> $c = d$
> **return** $(s_0, s_1, \ldots, s_n)$ (the binary expansion of the sum is $(s_n, s_{n-1}, \ldots, s_1, s_0)$)

Consider the **multiplication of two n-bit integers** a, b. Using the distributive law, we see that

$$ab = a(b_0 2^0 + ab_1 2^1 + \cdots + ab_n 2^n$$

Observe that $ab_j = a$ if $b_i = 1$ and 0 if $b_j = 0$. Each time we multiply a term by 2, we shift its binary expansion one place to the left and add a zero at the tail end of the expansion. Consequently, we can obtain $ab_j 2^j$ by shifting the binary expansion of ab_j j places to the left, adding j zero bits at the tail end of this binary expansion. To finish we need to add all the $ab_j 2^j$ including initial zero bits if necessary.

> **procedure** multiply(a, b: positive integers)
> (The binary expressions of a, b are $(a_{n-1}, a_{n-2}, \ldots, a_0)$ and $(b_{n-1}, b_{n-2}, \ldots, b_0)$)
> $c = 0$
> **for** $j = 0$ **to** $n - 1$:
>   **if** $b_j = 1$ **then** $c_j = a$ shifted j places to the left
>   **else** $c_j = 0$
>   ($(c_0, c_1, \ldots, c_{n-1})$ are the partial products)
> $p = 0$
> **for** $j = 0$ **to** $n - 1$: (adding all the partial products c_j)
> $p = p + c_j$
> **return** p (the value of ab)

Consider the situation now of finding **div** ad **mod** for integers a, d with $d > 0$. We can find q and r by using a brute-force algorithm, when a is posiitive we subtract d from a as many times as necessary until what is left is less than d. The number of times we perform this subtraction is the quotient and what is left over after all these subtractions is the remainder.

procedure division algorithm(a: an integer, d: positive integer)
$q = 0$
$r = |a|$
while $r \geq d$
    $r = r - d$
    $q = q + 1$
if $a < 0$ **and** $r > 0$ **then**
    $r = d - r$
    $q = -(q + 1)$
return (q, r) (the quotient and the remainder of the division of a by d)

In cryptography it is important to be able to find b^n mod m efficiently, where b, n and m are large integers. It is impractical to first compute b^n and then find its remainder when divided by m. We present an algorithm using the binary expansion of $n = (a_{k-1}, \ldots, a_1, a_0)$. We have that

$$b^n = b^{a_k 2^k} b^{a_{k-1} 2^{k-1}} \ldots b^{2a_1} b^{a_0}.$$

Therefore, we could simply compute $b, b^2, b^4, \ldots, b^{2^k}$ and multiply the elements in the list with $a_j = 1$. The algorithm will successively finds $b \mod m, b^2 \mod m$, ... $b^k \mod m$ and multiplies together those terms where $a_j = 1$.

procedure modular exponentiation(b: integer, $n = (a_{k-1}, \ldots, a_1, a_0)$)
        m: positive integer)
$x = 1$
power $= b \mod m$
for $i = 0$ **to** $k - 1$:
    if $a_i = 1$ **then** $x = (x \cdot \text{power}) \mod m$
return x ($x = b^n \mod m$)