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

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

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

Remark

- Theorem 1 is employing formal mathematical terminology to express that our number system is a *positional* number system. For example, 954_{10}means 954 base 10 and can be expressed as

  $$9 \times 10^2 + 5 \times 10^1 + 4 \times 10^0$$

  The digits of the number 954 are in positions 2, 1, and 0 respectively. Furthermore, the positions are exponents of the base.

EXAMPLE 1.1 Find the Decimal equivalent of $(2AE0B)_{16}$

Solution:

position	**digit**	**decimal value**	**exponentiated base**	**subtotal**
4	2	2	65536	131072
3	A	10	4096	40960
2	E	14	256	3584
1	0	0	16	0
0	B	11	1	11
Total				**175627**

EXAMPLE 1.2 Find the decimal equivalent of 1101 1010 1011 0010

Solution:

1. First, convert the binary integer to hexadecimal by making groups of four bits starting from the binary point.
   1101 1010 1011 0010
2. Next convert each group of four – each nibble – to its hexadecimal equivalent.
   D A B 2
3. Use the method of example 1.1 to convert the hexadecimal value to decimal.

position	**digit**	**decimal value**	**exponentiated base**	**subtotal**
3	D	13	4096	53248
2	A	10	256	2560
1	B	11	16	176
0	2	2	1	2
Total				**55986**

The Radix Divide/Multiply Method

Objective:	Covert a decimal number to an equivalent number in another radix (base).	
Solution:	**Step**	**Discussion**
	1	Separate the integer and fractional portions of the decimal number
	Integer portion conversion algorithm:	
	Step	**Discussion**
	0	Divide the decimal number by the radix. The remainder is a_i, i=0,1,2,3 … *n*-1. Quotient q_i becomes the dividend in the next iteration. r : radix, divisor d_i : dividend in iteration *i*. d_0 is the initial dividend, the decimal number to be converted to the foreign base. q_i : quotient in iteration *i* a_i : remainder produced in iteration *i*, *i*th digit of the number in radix *r*. $a_i = d_i \bmod r, d_{i+i} = \lfloor d_i \div r \rfloor$
	1 … *n*-1	Perform step 0 until the dividend equal zero.

Example: Convert 29_{10} to binary.

Step	Radix-Divisor	Dividend	Quotient	Remainder	a_i
0	2	29	14	1	a_0
1	2	14	7	0	a_1
2	2	7	3	1	a_2
3	2	3	1	1	a_3
4	2	1	0	1	a_4
5	2	0 **stop!**			

$29_{10} = 11101_2$

11101_2	**=**		**1**	$\times$	**2^4**	**=**	**16**
		+	**1**	$\times$	**2^3**	**=**	**8**
		+	**1**	$\times$	**2^2**	**=**	**4**
		+	**0**	$\times$	**2^1**	**=**	**0**
		+	**1**	$\times$	**2^0**	**=**	**1**
							29

EXAMPLE 2.1 Find the binary equivalent of $(29)_{10}$

Solution:

1. First, convert the decimal number to hexadecimal using the radix-divide method shown above.

Radix-Divisor	Dividend	Quotient	Remainder	Hexadecimal
16	29	1	13	D
16	1	0	1	1
16	0 stop!			

2. Next, convert each hexadecimal digit to binary.
   1 D = 0001 1101
3. Last, remove leading zeroes.
   1 1101

Purpose: complements simplify binary subtraction

Binary subtraction requires:

1. complement
2. fixed field widths for binary numbers

One's complement:

invert all bits $1 \rightarrow 0, 0 \rightarrow 1$

Note: The *only* time a number is complemented is when the number is *negative*.

Example: consider

`10011110`, the one's complement is
`01100001`

Think of the one's complement as the difference between the initial operand and a number of equal length having a one in every position.

Example:

```
  11111111
 -10011110
 ---------
  01100001
```

Two's complement is one more than the one's complement

Example: find the two's complement of 10011110

1. Original value `10011110`
2. One's complement `01100001`
3. Add one `+ 1`

```
  10011110
  01100001
 +       1
 ---------
  01100010
```

Two's Complement Representation

1. Choose a field width. Common field widths are 4, 8, 16, 32, and 64 bits.
2. A binary number is positive if the most significant digit is zero (0), otherwise it is negative.

   Example: Find the decimal equivalent of the following 16-bit two's complement number

```
**1**001  1100  0000  0101
```

1. Make the two's complement number positive. Find the magnitude of the two's complement.
   1.1. First, find the one's complement by inverting all the bits.

```
1001  1100  0000  0101
0110  0011  1111  1010
```

   1.2. Next, add one (1) to find the two's complement.

```
0110  0011  1111  1010
                    +1
0110  0011  1111  1011
```

2. Convert to decimal.
   2.1. First convert to hexadecimal.

```
0110  0011  1111  1011
 6     3     F     B
```

   2.2. Next, convert to decimal.

Hex	Dec.			
6	6	$\times$	16^3	24576
3	3	$\times$	16^2	768
F	15	$\times$	16^1	240
B	11	$\times$	16^0	11
				25595

3. Remember that the number was negative. Thus
   `1001  1100  0000  0101` = **-25595**

Two's Complement Subtraction

M and S are the minuend and subtrahend respectively and both are in two's complement representation.

1. Define a field width and add M to the two's complement of S. Discard any digits that carry into positions more significant than those defined by the field width. Discard any carry out digits.

   Example Find the difference $(1010 - 0111)$.
   Find the 2's complement of $S = 0111$.

	0111	
1's complement of S	1000	
Add 1 to find 2's complement of S	+ 1	
2's complement of S	1001	

   Add M to the two's complement of S.

M	1010	10
2's complement of S	+1001	-7
$D = M - S$	10011	3
Final result	0011	3

2. Determining the sign.
   2.1. If the most significant digit is 1, then the number is negative, otherwise it is positive.
3. Finding the magnitude of a two's complement number.
   3.1. If the number is not negative – that is – if the number has a zero in its most significant bit, then convert it to decimal as we have discussed.

   Example: Find the magnitude of the 8-bit, two's complement number 01101001.
   1.1. First, convert it to hexadecimal.
   $01101001_2 = 69_{16}$
   1.2. Next, convert it to decimal.
   $69_{16} = 6 \times 16 + 9 = 105$

   3.2. If the number is negative – that is – if the number has a one in its most significant bit, then complement the number, convert it to decimal as we have discussed, and **recall** that the number was negative.

   Example: Find the magnitude of the 8-bit, two's complement number 10110110.
   1. Complement the number
      1.1. One's complement 01001001
      1.2. Two's complement 01001010
   2. Convert the number to decimal
      2.1. First, convert it to hexadecimal.
      $01001010_2 = 4A_{16}$
      2.2. Next, convert it to decimal.
      $4A_{16} = 4 \times 16 + 10 = 74$
   3. **Recall** that the number was negative
      -74

4. Determining if overflow occurred.
   4.1. Overflow occurs when adding two positive numbers, adding two negative numbers, subtracting a positive number from a negative number, or when subtracting a negative number from a positive number. When the magnitude of the result exceeds the range of values that can be represented in the field an overflow has occurred.

   Consider a 4-bit two's complement number. The range of values for a 4-bit two complement number is $-8 \le v \le 7$. Whenever the result falls outside the range, an overflow has occurred.

   Example: Find the sum of 4+5 using 4-bit two's complement numbers.

				1			
	4			0	1	0	0
+	5		+	0	1	0	1
	9			1	0	0	1
				0	1	1	0
			+				1
				0	1	1	1

   We observe the sum to be 1001, a negative number in 4-bit two's complement representation. Immediately, we see that a sign inversion has occurred. The sum of two positive numbers is negative. Clearly, this is impossible. Therefore, an overflow has occurred.

   4.2. Special case: Overflow in 2's complement numbers. Overflow occurs when the carry into the sign bit is unequal to the carry out.

   Example: consider a 4-bit 2's complement number. A number, n, ranges over the interval, $-8 \le n \le 7$, Example: Find the sum of -5 and -4.

Decimal		**Binary**						**2's Complement**				
								CO	CI			
								1	0			
-	5	0	1	0	1	2's comp →			1	0	1	1
-	4	0	1	0	0	2's comp →	+		1	1	0	0
-	9							1	0	1	1	1

In the example above, CI is an abbreviation for Carry In (to the sign bit) and CO means Carry Out (of the sign bit).

Algorithms for Integer Operations

ALGORITHM 2 Addition of Integers

procedure add(**a,b: positive integers)**
{The binary expansions of *a* and *b* are $(\boldsymbol{a_{n-1}a_{n-2}\cdots a_1a_0})_2$ and $(\boldsymbol{b_{n-1}b_{n-2}\cdots b_1b_0})_2$ respectively where *a* and *b* are both binary numbers.}
c:=0
for j**:=0 to** n**-1**
begin
$d \coloneqq a_j + b_j + c$
$s_j \coloneqq d \bmod 2$
if $d > 1$ **then** $c \coloneqq \mathbf{1}$ **else** $c \coloneqq \mathbf{0}$
end
$s_n = c$
{The binary expansion of the sum is $(\boldsymbol{s_n s_{n-1}\cdots s_0})_2$}

EXAMPLE 7 Add $a = (1110)_2$ and $b = (1011)_2$

Solution: Follow the procedure specified in algorithm 2

c:=0

$j = 0$
$d \coloneqq a_0 + b_0 + c = 0 + 1 + 0 = 1$
$s_0 \coloneqq d \bmod 2 = 1 \bmod 2 = s_0 = 1$
if $d > 1$ **then** $c \coloneqq \mathbf{1}$ **else** $c \coloneqq \mathbf{0} \Rightarrow c = \mathbf{0}$

$j = 1$
$d \coloneqq a_1 + b_1 + c = 1 + 1 + 0 = 2$
$s_1 \coloneqq d \bmod 2 = 2 \bmod 2 = s_1 = 0$
if $d > 1$ **then** $c \coloneqq \mathbf{1}$ **else** $c \coloneqq \mathbf{0} \Rightarrow c = \mathbf{1}$

$j = 2$
$d \coloneqq a_2 + b_2 + c = 1 + 0 + 1 = 2$
$s_2 \coloneqq d \bmod 2 = 2 \bmod 2 = s_2 = 0$
if $d > 1$ **then** $c \coloneqq \mathbf{1}$ **else** $c \coloneqq \mathbf{0} \Rightarrow c = \mathbf{1}$

$j = 3$
$d \coloneqq a_3 + b_3 + c = 1 + 1 + 1 = 3$
$s_3 \coloneqq d \bmod 2 = 3 \bmod 2 = s_3 = 1$
if $d > 1$ **then** $c \coloneqq \mathbf{1}$ **else** $c \coloneqq \mathbf{0} \Rightarrow c = \mathbf{1}$

$j = 4$
$s_4 = c = 1$

$s = 11001$

Check

$j =$	4	3	2	1	0	
Carry	1	1	1	0		
		1	1	1	0	14
		1	0	1	1	+11
	1	1	0	0	1	25
25	16	8			1	

ALGORITHM 5 Modular Exponentiation

procedure *modular exponentiation*
(*b*: **integer**
,$n = (n_{k-1}n_{k-2}\cdots n_1 n_0)_2$: **positive integer**
,m: **positive integer**
)
$x := 1$
$power := b \textbf{ mod } m$
for $i := \mathbf{0}$ **to** $k - \mathbf{1}$
begin
if $a_i = \mathbf{1}$ **then** $x := (x \cdot power) \textbf{mod } m$
$power := (power \cdot power) \textbf{mod } m$
end
$\{\boldsymbol{x = b^n} \textbf{ mod } \boldsymbol{m}\}$

EXAMPLE 11 Use Algorithm 5 to find 3^{644} **mod** 645.
Solution: Algorithm 5 initially sets $x = 1$ and ; $power = 3 \textbf{ mod } 645 = 3$. In the computation of $3^{644} \textbf{ mod } 645$, this algorithm determines $3^{2^j} \textbf{ mod } 645$ for $j = 1, 2, \cdots, 9$ by successively squaring the and reducing modulo 645. If $a_j = 1$ (where a_j is the bit in the jth position in the binary expansion of 644), it multiplies the current value of x by $3^{2^j} \textbf{ mod } 645$ and reduces the result modulo 645. Here are the steps used:

$\boldsymbol{i}$	$\boldsymbol{a_i}$	$\boldsymbol{x}$	$\boldsymbol{power}$
0	0	1	$power = 3^2 \textbf{mod } 645 =$ $9 \textbf{ mod } 645 = \mathbf{9}$
1	0	1	$power = 9^2 \textbf{mod } 645 =$ $81 \textbf{ mod } 645 = \mathbf{81}$
2	1	$(1 \cdot 81) \textbf{ mod } 645 = 81$	$power = 81^2 \textbf{mod } 645 =$ $6521 \textbf{ mod } 645 = \mathbf{111}$
3	0	81	$power = 111^2 \textbf{mod } 645 =$ $12{,}321 \textbf{ mod } 645 = \mathbf{66}$
4	0	81	$power = 66^2 \textbf{mod } 645 =$ $4356 \textbf{ mod } 645 = \mathbf{486}$
5	0	81	$power = 486^2 \textbf{mod } 645 =$ $236{,}196 \textbf{ mod } 645 = \mathbf{126}$
6	0	81	$power = 126^2 \textbf{mod } 645 =$ $15{,}876 \textbf{ mod } 645 = \mathbf{396}$
7	1	$(81 \cdot 396)\ mod\ 645 = 471$	$power = 396^2 \textbf{mod } 645 =$ $156{,}816 \textbf{ mod } 645 = \mathbf{81}$
8	0	471	$power = 81^2 \textbf{mod } 645 =$ $6561 \textbf{ mod } 645 = \mathbf{111}$
9	1	$(471 \cdot 111) mod\ 645 = \mathbf{36}$	

$$3^{644} \textbf{ mod } 645 = 36$$

Exponentiation by squaring:

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & if\ n\ is\ odd \\ (x^2)^{\frac{n}{2}}, & if\ n\ is\ even \end{cases}$$

Example: $x^7 = x(x^2)^{\frac{7-1}{2}} = x(x^2)^{\frac{6}{2}} = x(x^2)^3 = x(x^{2\times3}) = x(x^6) = x^7$
Example: $x^{10} = (x^2)^{\frac{10}{2}} = (x^2)^5 = (x^{2\times5}) = x^{10}$

```
double exp_by_squaring(double x,unsigned int n)
{     if (n==0) return 1.0;
      if (n==1) return x;
      if (n%2) return x*exp_by_squaring(x*x,(n-1)/2);
      else return exp_by_squaring(x*x,n/2);
}
```