SECTION 3.6 Integers and Algorithms

In addition to calculation exercises on the Euclidean algorithm, the base conversion algorithm, and algorithms for the basic arithmetic operations, this exercise set introduces other forms of representing integers. These are **balanced ternary expansion, Cantor expansion, binary coded decimal** *(or* **BCD***) representation, and* **one's** *and* **two's complement** *representations. Each has practical and/or theoretical importance in mathematics or computer science. If all else fails, one can carry out an algorithm by "playing computer" and mechanically following the pseudocode step by step.*

1. We divide repeatedly by 2, noting the remainders. The remainders are then arranged from right to left to obtain the binary representation of the given nnmber.

a) We begin by dividing 231 by 2, obtaining a quotient of 115 and a remainder of 1. Therefore $a_0 = 1$. Next $115/2 = 57$, remainder 1. Therefore $a_1 = 1$. Similarly $57/2 = 28$, remainder 1. Therefore $a_2 = 1$. Then $28/2 = 14$, remainder 0, so $a_3 = 0$. Similarly $a_4 = 0$, after we divide 14 by 2, obtaining 7 with remainder 0. Three more divisions yield quotients of 3, 1, and 0, with remainders of 1, 1, and 1, respectively, so $a_5 = a_6 = a_7 = 1$. Putting all this together, we see that the binary representation is $(a_7a_6a_5a_4a_3a_2a_1a_0)_2 = (1110\,0111)_2$. As a check we can compute that $2^0 + 2^1 + 2^2 + 2^5 + 2^6 + 2^7 = 231$.

b) Following the same procedure as in part (a), we obtain successive remainders 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1. Therefore $4532 = (1\ 0001\ 1011\ 0100)_2$.

c) By the same method we obtain $97644 = (1\ 0111\ 1101\ 0110\ 1100)_2$.

3. a) $(1\ 1111)_2 = 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 16 + 8 + 4 + 2 + 1 = 31$. An easier way to get the answer is to note that $(1\ 1111)_2 = (10\ 0000)_2 - 1 = 2^5 - 1 = 31$.

b) $(10\ 0000\ 0001)_2 = 2^9 + 2^0 = 513$

c) $(1\ 0101\ 0101)_2 = 2^8 + 2^6 + 2^4 + 2^2 + 2^0 = 256 + 64 + 16 + 4 + 1 = 341$

d) $(110\ 1001\ 0001\ 0000)_2 = 2^{14} + 2^{13} + 2^{11} + 2^8 + 2^4 = 16384 + 8192 + 2048 + 256 + 16 = 26896$

5. Following Example 6, we simply write the binary equivalents of each digit: $(A)_{16} = (1010)_2$, $(B)_{16} = (1011)_2$, $(C)_{16} = (1100)_2$, $(D)_{16} = (1101)_2$, $(E)_{16} = (1110)_2$, and $(F)_{16} = (1111)_2$. Note that the blocking by groups of four binary digits is just for readability by humans.

a) $(80E)_{16} = (1000\ 0000\ 1110)_2$

b) $(135AB)_{16} = (0001\ 0011\ 0101\ 1010\ 1011)_2$

c) $(ABBA)_{16} = (1010\ 1011\ 1011\ 1010)_2$

d) $(DEFACED)_{16} = (1101\ 1110\ 1111\ 1010\ 1100\ 1110\ 1101)_2$

7. Following Example 6, we simply write the binary equivalents of each digit. Since $(A)_{16} = (1010)_2$, $(B)_{16} = (1011)_2$, $(C)_{16} = (1100)_2$, $(D)_{16} = (1101)_2$, $(E)_{16} = (1110)_2$, and $(F)_{16} = (1111)_2$, we see that $(ABCDEF)_{16} = (101010111100110111101111)_2$. Following the convention shown in Exercise 3 of grouping binary digits by fours, we can write this in a more readable form as 1010 1011 1100 1101 1110 1111.

9. Following Example 6, we simply write the hexadecimal equivalents of each group of four binary digits. Thus we have $(1011\ 0111\ 1011)_2 = (B7B)_{16}$.

11. We adopt a notation that will help with the explanation. Adding up to three leading 0's if necessary, write the binary expansion as $(\ldots b_{23}b_{22}b_{21}b_{20}b_{13}b_{12}b_{11}b_{10}b_{03}b_{02}b_{01}b_{00})_2$. The value of this numeral is $b_{00} + 2b_{01} + 4b_{02} + 8b_{03} + 2^4b_{10} + 2^5b_{11} + 2^6b_{12} + 2^7b_{13} + 2^8b_{20} + 2^9b_{21} + 2^{10}b_{22} + 2^{11}b_{23} + \cdots$, which we can rewrite as $b_{00} + 2b_{01} + 4b_{02} + 8b_{03} + (b_{10} + 2b_{11} + 4b_{12} + 8b_{13}) \cdot 2^4 + (b_{20} + 2b_{21} + 4b_{22} + 8b_{23}) \cdot 2^8 + \cdots$. Now $(b_{i3}b_{i2}b_{i1}b_{i0})_2$ translates into the hexadecimal digit h_i. So our number is $h_0 + h_1 \cdot 2^4 + h_2 \cdot 2^8 + \cdots = h_0 + h_1 \cdot 16 + h_2 \cdot 16^2 + \cdots$, which is the hexadecimal expansion $(\ldots h_1h_1h_0)_{16}$.

13. This is exactly the same as what we can do with hexadecimal expansion, replacing groups of four with groups of three. Specifically, group together blocks of three binary digits, adding up to two initial 0's if necessary, and translate each block of three binary digits into a single octal digit. For example, $(011\ 000\ 110)_2 = (306)_8$

15. In each case we follow the method of Example 6, blocking by threes instead of fours. We replace each octal digit of the given numeral by its 3-digit binary equivalent and string the digits together. The first digit is $(7)_8 = (111)_2$, the next is $(3)_8 = (011)_2$, and so on, so we obtain $(111011100101011010001)_2$. For the other direction, we split the given binary numeral into blocks of three digits, adding initial 0's to fill it out: $001\ 010\ 111\ 011$. Then we replace each block by its octal equivalent, obtaining the answer $(1273)_8$.

17. Since we have procedures for converting both octal and hexadecimal to and from binary (Example 6 and Exercises 13–15), to convert from octal to hexadecimal, we first convert from octal to binary and then convert from binary to hexadecimal.

19. In effect, this algorithm computes $7 \textbf{ mod } 645$, $7^2 \textbf{ mod } 645$, $7^4 \textbf{ mod } 645$, $7^8 \textbf{ mod } 645$, $7^{16} \textbf{ mod } 645$, …, and then multiplies (modulo 645) the required values. Since $644 = (1010000100)_2$, we need to multiply together $7^4 \textbf{ mod } 645$, $7^{128} \textbf{ mod } 645$, and $7^{512} \textbf{ mod } 645$, reducing modulo 645 at each step. We compute by repeatedly squaring: $7^2 \textbf{ mod } 645 = 49$, $7^4 \textbf{ mod } 645 = 49^2 \textbf{ mod } 645 = 2401 \textbf{ mod } 645 = 466$, $7^8 \textbf{ mod } 645 = 466^2 \textbf{ mod } 645 = 217156 \textbf{ mod } 645 = 436$, $7^{16} \textbf{ mod } 645 = 436^2 \textbf{ mod } 645 = 190096 \textbf{ mod } 645 = 466$. At this point we see a pattern with period 2, so we have $7^{32} \textbf{ mod } 645 = 436$, $7^{64} \textbf{ mod } 645 = 466$, $7^{128} \textbf{ mod } 645 = 436$, $7^{256} \textbf{ mod } 645 = 466$, and $7^{512} \textbf{ mod } 645 = 436$. Thus our final answer will be the product of 466, 436, and 436, reduced modulo 645. We compute these one at a time: $466 \cdot 436 \textbf{ mod } 645 = 203176 \textbf{ mod } 645 = 1$, and $1 \cdot 436 \textbf{ mod } 645 = 436$. So $7^{644} \textbf{ mod } 645 = 436$. A computer algebra system will verify this: use the command "`7 &^ 644 mod 645;`" in *Maple*, for example. The ampersand here tells *Maple* to use modular exponentiation, rather than first computing the integer 7^{644}, which has over 500 digits, although it could certainly handle this if asked. The point is that modular exponentiation is much faster and avoids having to deal with such large numbers.

21. In effect, this algorithm computes $3 \textbf{ mod } 99$, $3^2 \textbf{ mod } 99$, $3^4 \textbf{ mod } 99$, $3^8 \textbf{ mod } 99$, $3^{16} \textbf{ mod } 99$, …, and then multiplies (modulo 99) the required values. Since $2003 = (11111010011)_2$, we need to multiply together $3 \textbf{ mod } 99$, $3^2 \textbf{ mod } 99$, $3^{16} \textbf{ mod } 99$, $3^{64} \textbf{ mod } 99$, $3^{128} \textbf{ mod } 99$, $3^{256} \textbf{ mod } 99$, $3^{512} \textbf{ mod } 99$, and $3^{1024} \textbf{ mod } 99$, reducing modulo 99 at each step. We compute by repeatedly squaring: $3^2 \textbf{ mod } 99 = 9$, $3^4 \textbf{ mod } 99 = 81$, $3^8 \textbf{ mod } 99 = 81^2 \textbf{ mod } 99 = 6561 \textbf{ mod } 99 = 27$, $3^{16} \textbf{ mod } 99 = 27^2 \textbf{ mod } 99 = 729 \textbf{ mod } 99 = 36$, $3^{32} \textbf{ mod } 99 = 36^2 \textbf{ mod } 99 = 1296 \textbf{ mod } 99 = 9$, and then the pattern repeats, so $3^{64} \textbf{ mod } 99 = 81$, $3^{128} \textbf{ mod } 99 = 27$, $3^{256} \textbf{ mod } 99 = 36$, $3^{512} \textbf{ mod } 99 = 9$, and $3^{1024} \textbf{ mod } 99 = 81$. Thus our final answer will be the product of 3, 9, 36, 81, 27, 36, 9, and 81. We compute these one at a time modulo 99: $3 \cdot 9$ is 27, $27 \cdot 36$ is 81, $81 \cdot 81$ is 27, $27 \cdot 27$ is 36, $36 \cdot 36$ is 9, $9 \cdot 9$ is 81, and finally $81 \cdot 81$ is 27. So $3^{2003} \textbf{ mod } 99 = 27$.

23. **a)** By Lemma 1, $\gcd(12, 18)$ is the same as the gcd of the smaller of these two numbers (12) and the remainder when the larger (18) is divided by the smaller. In this case the remainder is 6, so $\gcd(12, 18) = \gcd(12, 6)$. Now $\gcd(12, 6)$ is the same as the gcd of the smaller of *these* two numbers (6) and the remainder when the larger (12) is divided by the smaller, namely 0. This gives $\gcd(12, 6) = \gcd(6, 0)$. But $\gcd(x, 0) = x$ for all positive integers, so $\gcd(6, 0) = 6$. Thus the answer is 6. In brief (the form we will use for the remaining parts), $\gcd(12, 18) = \gcd(12, 6) = \gcd(6, 0) = 6$.

b) $\gcd(111, 201) = \gcd(111, 90) = \gcd(90, 21) = \gcd(21, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$

c) $\gcd(1001, 1331) = \gcd(1001, 330) = \gcd(330, 11) = \gcd(11, 0) = 11$

d) $\gcd(12345, 54321) = \gcd(12345, 4941) = \gcd(4941, 2463) = \gcd(2463, 15) = \gcd(15, 3) = \gcd(3, 0) = 3$

e) $\gcd(1000, 5040) = \gcd(1000, 40) = \gcd(40, 0) = 40$

f) $\gcd(9888, 6060) = \gcd(6060, 3828) = \gcd(3828, 2232) = \gcd(2232, 1596) = \gcd(1596, 636) = \gcd(636, 324)$ $= \gcd(324, 312) = \gcd(312, 12) = \gcd(12, 0) = 12$

25. In carrying out the Euclidean algorithm on this data, we divide successively by 34, 21, 13, 8, 5, 3, 2, and 1, so eight divisions are required.

27. The binary expansion of an integer represents the integer as a sum of distinct powers of 2. For example, since $21 = (1\ 0101)_2$, we have $21 = 2^4 + 2^2 + 2^0$. Since binary expansions are unique, each integer can be so represented uniquely.

29. Let the decimal expansion of the integer a be given by $a = (a_{n-1}a_{n-2}\ldots a_1a_0)_{10}$. Thus $a = 10^{n-1}a_{n-1} + 10^{n-2}a_{n-2} + \cdots + 10a_1 + a_0$. Since $10 \equiv 1 \pmod 3$, we have $a \equiv a_{n-1} + a_{n-2} + \cdots + a_1 + a_0 \pmod 3$. Therefore $a \equiv 0 \pmod 3$ if and only if the sum of the digits is congruent to $0 \pmod 3$. Since being divisible by 3 is the same as being congruent to $0 \pmod 3$, we have proved that a positive integer is divisible by 3 if and only if the sum of its decimal digits is divisible by 3.

31. Let the binary expansion of the positive integer a be given by $a = (a_{n-1}a_{n-2}\ldots a_1a_0)_2$. Thus $a = a_0 + 2a_1 + 2^2a_2 + \cdots + 2^{n-1}a_{n-1}$. Since $2^2 \equiv 1 \pmod 3$, we see that $2^k \equiv 1 \pmod 3$ when k is even, and $2^k \equiv 2 \equiv -1 \pmod 3$ when k is odd. Therefore we have $a \equiv a_0 - a_1 + a_2 - a_3 + \cdots \pm a_{n-1} \pmod 3$. Thus $a \equiv 0 \pmod 3$ if and only if the sum of the binary digits in the even-numbered positions minus the sum of the binary digits in the odd-numbered positions is congruent to 0 modulo 3. Since being divisible by 3 is the same as being congruent to $0 \pmod 3$, our proof is complete.

33. a) Since the leading bit is a 1, this represents a negative number. The binary expansion of the absolute value of this number is the complement of the rest of the expansion, namely the complement of 1001, or 0110. Since $(0110)_2 = 6$, the answer is -6.

b) Since the leading bit is a 0, this represents a positive number, namely the number whose binary expansion is the rest of this string, 1101. Since $(1101)_2 = 13$, the answer is 13.

c) The answer is the negative of the complement of 0001, namely $-(1110)_2 = -14$.

d) $-(0000)_2 = 0$; note that 0 has two different representations, 0000 and 1111

35. We must assume that the sum actually represents a number in the appropriate range. Assume that n bits are being used, so that numbers strictly between -2^{n-1} and 2^{n-1} can be represented. The answer is almost, but not quite, that to obtain the one's complement representation of the sum of two numbers, we simply add the two strings representing these numbers using Algorithm 3. Instead, after performing this operation, there may be a carry out of the left-most column; in such a case, we then add 1 more to the answer. For example, suppose that $n = 4$; then numbers from -7 to 7 can be represented. To add -5 and 3, we add 1010 and 0011, obtaining 1101; there was no carry out of the left-most column. Since 1101 is the one's complement representation of -2, we have the correct answer. On the other hand, to add -4 and -3, we add 1011 and 1100, obtaining 1 0111. The 1 that was carried out of the left-most column is instead added to 0111, yielding 1000, which is the one's complement representation of -7. A proof that this method works entails considering the various cases determined by the signs and magnitudes of the addends.

37. If m is positive (or 0), then the leading bit (a_{n-1}) is 0, so the formula reads simply $m = \sum_{i=0}^{n-2} a_i2^i$, which is clearly correct, since this is the binary expansion of m. (See Section 2.4 for the meaning of summation notation. This symbolism is a shorthand way of writing $a_0 + 2a_1 + 4a_2 + \cdots + 2^{n-2}a_{n-2}$.) Now suppose that m is negative. The one's complement expansion for m has its leading bit equal to 1. By the definition of one's complement, we can think of obtaining the remaining $n-1$ bits by subtracting $-m$, written in binary, from $111\ldots1$ (with $n-1$ 1's), since subtracting a bit from 1 is the same thing as complementing it. Equivalently, if we view the bit string $(a_{n-2}a_{n-1}\ldots a_0)$ as a binary number, then it represents $(2^{n-1}-1)-(-m)$. In symbols, this says that $(2^{n-1}-1)-(-m)=\sum_{i=0}^{n-2}a_i2^i$. Solving for m gives us the equation we are trying to prove (since $a_{n-1}=1$).

39. Following the definition, if the first bit is a 0, then we just evaluate the binary expansion. If the first bit is a 1, then we find what number x is represented by the remaining four bits in binary; the answer is then $-(2^4-x)$.
a) Since the first bit is a 1, and the remaining bits represent the number 9, this string represents the number $-(2^4-9)=-7$.
b) Since the first bit is a 0 and this is just the binary expansion of 13, the answer is 13.
c) Since the first bit is a 1, and the remaining bits represent the number 1, this string represents the number $-(2^4-1)=-15$.
d) Since the first bit is a 1, and the remaining bits represent the number 15, this string represents the number $-(2^4-15)=-1$. Note that 10000 would represent $-(2^4-0)=-16$, so in fact we can represent one extra negative nmmber than positive number with this notation.

41. The nice thing about two's complement arithmetic is that we can just work as if it were all in base 2, since $-x$ (where x is positive) is represented by 2^n-x; in other words, modulo 2^n, negative numbers represent themselves. However, if overflow occurs, then we must recognize an error. Let us look at some examples, where $n=5$ (i.e., we use five bits to represent numbers between -15 and 15). To add $5+7$, we write $00101+00111=01100$ in base 2, which gives us the correct answer, 12. However, if we try to add $13+7$ we obtain $01101+00111=10100$, which represents -12, rather than 20, so we report an overflow error. (Of course these two numbers are congruent modulo 32.) Similarly, for $5+(-7)$, we write $00101+11001=11110$ in base 2, and 11110 is the two's complement representation of -2, the right answer. For $(-5)+(-7)$, we write $11011+11001=110100$ in base 2; if we ignore the extra 1 in the left-most column (which doesn't exist), then this is the two's complement representation of -12, again the right answer. To summarize, to obtain the two's complement representation of the sum of two integers given in two's complement representation, add them as if they were binary integers, and ignore any carry out of the left-most column. However, if the left-most digits of the two addends agree and the left-most digit of the answer is different from their common value, then an overflow has occurred, and the answer is not valid.

43. If m is positive (or 0), then the leading bit (a_{n-1}) is 0, so the formula reads simply $m=\sum_{i=0}^{n-2}a_i2^i$, which is clearly correct, since this is the binary expansion of m. (See Section 2.4 for the meaning of summation notation. This symbolism is a shorthand way of writing $a_0+2a_1+4a_2+\cdots+2^{n-2}a_{n-2}$.) Now snppose that m is negative. The two's complement expansion for m has its leading bit equal to 1. By the definition of two's complement, the remaining $n-1$ bits are the binary expansion of $2^{n-1}-(-m)$. In symbols, this says that $2^{n-1}-(-m)=\sum_{i=0}^{n-2}a_i2^i$. Solving for m gives us the equation we are trying to prove (since $a_{n-1}=1$).

45. Clearly we need $4n$ digits, four for each digit of the decimal representation.

47. To find the Cantor expansion, we will work from left to right. Thus the first step will be to find the largest number n whose factorial is still less than or equal to the given positive integer x. Then we determine the digits in the expansion, starting with a_n and ending with a_1.

```
procedure Cantor(x : positive integer)
n := 1; factorial := 1
while (n + 1) · factorial ≤ x
begin
```

```
        $n := n + 1$
        $factorial := factorial \cdot n$
end {at this point we know that there are $n$ digits in the expansion}
$y := x$ {this is just so we do not destroy the original input}
while $n > 0$
begin
        $a_n := \lfloor y/factorial \rfloor$
        $y := y - a_n \cdot factorial$
        $factorial := factorial/n$
        $n := n - 1$
end
{we are done: $x = a_n n! + a_{n-1}(n-1)! + \cdots + a_2 2! + a_1 1!$ }
```

49. Note that $n = 5$. Initially the carry is $c = 0$, and we start the **for** loop with $j = 0$. Since $a_0 = 1$ and $b_0 = 0$, we set d to be $\lfloor(1+0+0)/2\rfloor = 0$; then $s_0 = 1+0+0-2\cdot 0$, which equals 1, and finally $c = 0$. At the end of the first pass, then, the right-most digit of the answer has been determined (it's a 1), and there is a carry of 0 into the next column.

Now $j = 1$, and we compute d to be $\lfloor(a_1 + b_1 + c)/2\rfloor = \lfloor(1+1+0)/2\rfloor = 1$; whereupon s_1 becomes $1+1+0-2\cdot 1 = 0$, and c is set to 1. Thus far we have determined that the last two bits of the answer are 01 (from left to right), and there is a carry of 1 into the next column.

The next three passes through the loop are similar. As a result of the pass when $j = 2$ we set $d = 1$, $s_2 = 0$, and then $c = 1$. When $j = 3$, we obtain $d = 1$, $s_3 = 0$, and then $c = 1$. Finally, when $j = 4$, we obtain $d = 1$, $s_4 = 1$, and then $c = 1$. At this point the loop is terminated, and when we execute the final step, $s_5 = 1$. Thus the answer is 11 0001.

51. We will assume that the answer is not negative, since otherwise we would need something like the one's complement representation. The algorithm is similar to the algorithm for addition, except that we need to borrow instead of carry. Rather than trying to incorporate the two cases (borrow or no borrow) into one, as was done in the algorithm for addition, we will use an **if...then** statement to treat the cases separately. The notation is the usual one: $a = (a_{n-1} \ldots a_1 a_0)_2$ and $b = (b_{n-1} \ldots b_1 b_0)_2$

```
procedure subtract($a, b$ : nonnegative integers)
$borrow := 0$
for $j := 0$ to $n - 1$
        if $a_j - borrow \geq b_j$ then
        begin
                $s_j := a_j - borrow - b_j$
                $borrow := 0$
        end
        else
        begin
                $s_j := a_j + 2 - borrow - b_j$
                $borrow := 1$
        end
{assuming $a \geq b$, we have $a - b = (s_{n-1}s_{n-2} \ldots s_1 s_0)_2$ }
```

53. To determine which of two integers (we assume they are nonnegative), given in binary as $a = (a_{n-1} \ldots a_1 a_0)_2$ and $b = (b_{n-1} \ldots b_1 b_0)_2$, is larger, we need to compare digits from the most significant end ($i = n-1$) to the least ($i = 0$), stopping if and when we find a difference. For variety here we record the answer as a character string; in most applications it would probably be better to set *compare* to one of three code values (such as -1, 1, and 0) to indicate which of the three possibilities held.

```
**procedure** $compare(a, b$ : nonnegative integers)
$i := n - 1$
**while** $i > 0$ and $a_i = b_i$
        $i := i - 1$
**if** $a_i > b_i$ **then** $answer :=$ "$a > b$"
**else if** $a_i < b_i$ **then** $answer :=$ "$a < b$"
**else** $answer :=$ "$a = b$"
{the answer is recorded in $answer$}
```

55. There is one division for each pass through the **while** loop. Also, each pass generates one digit in the base b expansion. Thus the number of divisions equals the number of digits in the base b expansion of n. This is just $\lfloor \log_b n \rfloor + 1$ (for example, numbers from 10 to 99, inclusive, have common logarithms in the interval $[1, 2)$). Therefore exactly $\lfloor \log_b n \rfloor + 1$ divisions are required, and this is $O(\log n)$. (We are counting only the actual division operation in the statement $q := \lfloor q/b \rfloor$. If we also count the implied division in the statement $a_k := q$ **mod** b, then there are twice as many as we computed here. The big-O estimate is the same, of course.)

57. The only time-consuming part of the algorithm is the **while** loop, which is iterated q times. The work done inside is a subtraction of integers no bigger than a, which has $\log a$ bits. The results now follows from Example 8.