Part III: Objects, algorithms, programs

Computing with numbers and other objects

Since the introduction of computers four or five decades ago the meaning of the word *computation* has kept expanding. Whereas "computation" traditionally implied "numbers", today we routinely compute pictures, texts, and many other types of objects. When classified according to the types of objects being processed, three types of computer applications stand out prominently with respect to the influence they had on the development of computer science.

The first generation involved *numerical computing*, applied mainly to scientific and technical problems. Data to be processed consisted almost exclusively of numbers, or sets of numbers with a simple structure, such as vectors and matrices. Programs were characterized by long execution times but small sets of input and output data. Algorithms were more important than data structures, and many new numerical algorithms were invented. Lasting achievements of this first phase of computer applications include systematic study of numerical algorithms, error analysis, the concept of program libraries, and the first high-level programming languages, Fortran and Algol.

The second generation, hatched by the needs of commercial data processing, leads to the development of many new data structures. Business applications thrive on record keeping and updating, text and form processing, and report generation: there is not much computation in the numeric sense of the word, but a lot of reading, storing, moving, and printing of data. In other words, these applications are data intensive rather than computation intensive. By focusing attention on the problem of efficient management of large, dynamically varying data collections, this phase created one of the core disciplines of computer science: data structures, and corresponding algorithms for managing data, such as searching and sorting.

We are now in a third generation of computer applications, dominated by computing with geometric and pictorial objects. This change of emphasis was triggered by the advent of computers with bitmap graphics. In turn, this leads to the widespread use of sophisticated user interfaces that depend on graphics, and to a rapid increase in applications such as computer-aided design (CAD) and image processing and pattern recognition (in medicine, cartography, robot control). The young discipline of computational geometry has emerged in response to the growing importance of processing geometric and pictorial objects. It has created novel data structures and algorithms, some of which are presented in Parts V and VI.

Our selection of algorithms in Part III reflects the breadth of applications whose history we have just sketched. We choose the simplest types of objects from each of these different domains of computation and some of the most concise and elegant algorithms designed to process them. The study of typical small programs is an essential part of programming. A large part of computer science consists of the knowledge of how typical problems can be solved; and the best way to gain such knowledge is to study the main ideas that make standard programs work.

Algorithms and programs

Theoretical computer science treats *algorithm* as a formal concept, rigorously defined in a number of ways, such as Turing machines or lambda calculus. But in the context of programming, *algorithm* is typically used as an intuitive concept designed to help people express solutions to their problems. The formal counterpart of an algorithm is a procedure or program (fragment) that expresses the algorithm in a formally defined programming language. The process of formalizing an algorithm as a program typically requires many decisions: some superficial (e.g. what type of statement is chosen to set up a loop), some of great practical consequence (e.g. for a given range of values of n, is the algorithm's asymptotic complexity analysis relevant or misleading?).

We present algorithms in whatever notation appears to convey the key ideas most clearly, and we have a clear preference for pictures. We present programs in an extended version of Pascal; readers should have little difficulty translating this into any programming language of their choice. Mastery of interesting small programs is the best way to get started in computer science. We encourage the reader to work the examples in detail.

The literature on algorithms. The development of new algorithms has been proceeding at a very rapid pace for several decades, and even a specialist can only stay abreast with the state of the art in some subfield, such as graph algorithms, numerical algorithms, or geometric algorithms. This rapid development is sure to continue unabated, particularly in the increasingly important field of parallel algorithms. The cutting edge of algorithm research is published in several journals that specialize in this research topic, including the *Journal of Algorithms* and *Algorithmica*. This literature is generally accessible only after a student has studied a few textbooks on algorithms, such as [AHU 75], [Baa 88], [BB 88], [CLR 90], [GB 91], [HS 78], [Knu 73a], [Knu 81], [Knu 73b], [Man 89], [Meh 84a], [Meh 84b], [Meh 84c], [RND 77], [Sed 88], [Wil 86], and [Wir 86].

8. Truth values, the data type 'set', and bit acrobatics

Learning objectives:

- truth values, bits
- boolean variables and functions
- bit sum: four clever algorithms compared
- trade-off between time and space

Bits and boolean functions

The English mathematician George Boole (1815–1864) became one of the founders of symbolic logic when he endeavored to express logical arguments in mathematical form. The goal of his 1854 book *The Laws Of Thought* was "to investigate the laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolic language of calculus. ..."

Truth values or *boolean values*, named in Boole's honor, possess the smallest possible useful domain: the binary domain, represented by yes/no, 1/0, true/false, T/F. In the late 1940s, as the use of binary arithmetic became standard and as information theory came to regard a two-valued quantity as the natural unit of information, the concise term *bit* was coined as an abbreviation of "binary digit". A bit, by any other name, is truly a primitive data element—at a sufficient level of detail, (almost) everything that happens in today's computers is bit manipulation. Just because bits are simple data quantities does not mean that processing them is necessarily simple, as we illustrate in this section by presenting some clever and efficient bit manipulation algorithms.

Boolean variables range over boolean values, and *boolean functions* take boolean arguments and produce boolean results. There are only four distinct boolean functions of a single boolean variable, among which 'not' is the most useful: It yields the complement of its argument (i.e. turns 0 into 1, and vice versa). The other three are the identity and the functions that yield the constants 0 and 1. There are 16 distinct boolean functions of two boolean variables, of which several are frequently used, in particular: 'and', 'or'; their negations 'nand', 'nor'; the exclusive-or 'xor'; and the implication '⊃'. These functions are defined as follows:

a	b	a and b	a or b	a nand b	a nor b	a xor b	a ⊃ b
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	1
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

Bits are the atomic data elements of today's computers, and most programming languages provide a data type 'boolean' and built-in operators for 'and', 'or', 'not'. To avoid the necessity for boolean expressions to be fully

parenthesized, precedence relations are defined on these operators: 'not' takes precedence over 'and', which takes precedence over 'or'. Thus

x and not y or not x and y ⇔ ((x and (not y)) or ((not x) and y)).

What can you compute with boolean variables? Theoretically everything, since large finite domains can always be represented by a sufficient number of boolean variables: 16-bit integers, for example, use 16 boolean variables to represent the integer domain -2^{15} .. $2^{15}-1$. Boolean variables are often used for program optimization in practical problems where efficiency is important.

Swapping and crossovers: the versatile exclusive-or

Consider the swap statement x :=: y, which we use to abbreviate the cumbersome triple: t := x; x := y; y := t. On computers that provide bitwise boolean operations on registers, the swap operator :=: can be implemented efficiently without the use of a temporary variable.

The operator *exclusive-or*, often abbreviated as 'xor', is defined as

x xor y = x and not y or not x and y.

It yields true iff exactly one of its two arguments is true.

The bitwise boolean operation z:= x op y on n-bit registers: x[1 .. n], y[1 .. n], z[1 .. n], is defined as

for i := 1 to n do z[i] := x[i] op y[i]

With a bitwise exclusive-or, the swap x :=: y can be programmed as

```
x := x xor y;  y := x xor y;  x := x xor y;
```

It still takes three statements, but no temporary variable. Given that registers are usually in short supply, and that a logical operation on registers is typically just as fast as an assignment, the latter code is preferable. Exhibit 8.1 traces the execution of this code on two 4-bit registers and shows exhaustively that the swap is performed correctly for all possible values of x and y.

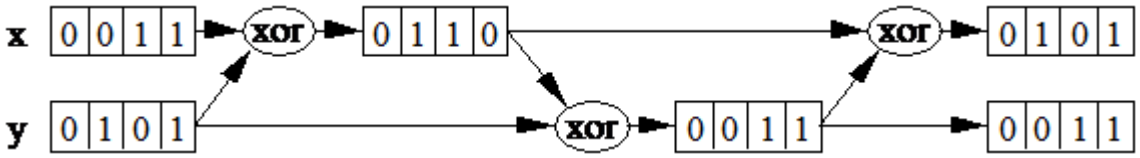

Exhibit 8.1: Trace of registers x and y under repeated exclusive-or operations.

Exercise: planar circuits without crossover of wires

The code above has yet another interpretation: How should we design a logical circuit that effects a logical crossover of two wires x and y while avoiding any physical crossover? If we had an 'xor' gate, the circuit diagram shown in Exhibit 8.2 would solve the problem. 'xor' gates must typically be realized as circuits built from simpler primitives, such as 'and', 'or', 'not'. Design a circuit consisting of 'and', 'or', 'not' gates only, which has the effect of crossing wires x and y while avoiding physical crossover.

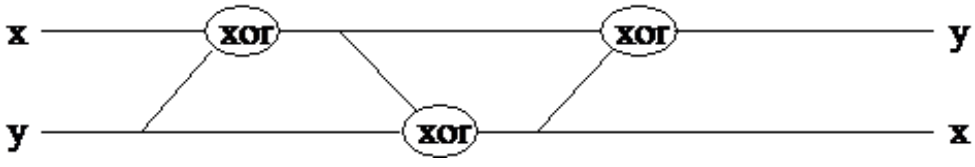

Exhibit 8.2: Three exclusive-or gates in series interchange values on two wires.

The bit sum or "population count"

A computer word is a fixed-length sequence of bits, call it a bit vector. Typical word lengths are 16, 32, or 64, and most instructions in most computers operate on all the bits in a word at the same time, in parallel. When efficiency is of great importance, it is worth exploiting to the utmost the bit parallelism built into the hardware of most computers. Today's programming languages often fail to refer explicitly to hardware features such as registers or words in memory, but it is usually possible to access individual bits if one knows the representation of integers or other data types. In this section we take the freedom to drop the constraint of *strong typing* built into Pascal and other modern languages. We interpret the content of a register or a word in memory as it suits the need of the moment: a bit string, an integer, or a set.

We are well aware of the dangers of such ambiguous interpretations: Programs become system and compiler dependent, and thus lose portability. If such ambiguity is localized in a single, small procedure, the danger may be kept under control, and the gain in efficiency may outweigh these drawbacks. In Pascal, for example, the type 'set' is especially well suited to operate at the bit level. 'type s = set of (a, b, c)' consists of the 2^3 sets that can be formed from the three elements a, b, c. If the basic set M underlying the declaration of

type S = set of M

consists of n elements, then S has 2^n elements. Usually, a value of type S is internally represented by a vector of n contiguously allocated bits, one bit for each element of the set M. When computing with values of type S we operate on single bits using the boolean operators. The union of two sets of type S is obtained by applying bitwise 'or', the intersection by applying bitwise 'and'. The complement of a set is obtained by applying bitwise 'not'.

Example

M = {0, 1, … , 7}

Set	Bit vector 7 6 5 4 3 2 1 0	Elements
s_1	{0, 3, 4, 6}	0 1 0 1 1 0 0 1
s_2	{0, 1, 4, 5}	0 0 1 1 0 0 1 1
$s_1 \cup s_2$	{0, 1, 3, 4, 5, 6}	0 1 1 1 1 0 1 1
$s_1 \cap s_2$	{0, 4}	0 0 0 1 0 0 0 1
$\neg s_1$	{1, 2, 5, 7}	1 0 1 0 0 1 1 0

Integers are represented on many small computers by 16 bits. We assume that a type 'w16', for "word of length 16", can be defined. In Pascal, this might be

```
type  w16 = set of 0 .. 15;
```

A variable of type 'w16' is a set of at most 16 elements represented as a vector of 16 bits.

Asking for the number of elements in a set s is therefore the same as asking for the number of 1's in the bit pattern that represents s. The operation that counts the number of elements in a set, or the number of 1's in a word, is called the *population count* or *bit sum*. The bit sum is frequently used in inner loops of combinatorial calculations, and many a programmer has tried to make it as fast as possible. Let us look at four of these tries, beginning with the obvious.

Inspect every bit

```
function bitsum0(w: w16): integer;
var  i, c: integer;
begin
   c := 0;
   for  i := 0  to  15  do  { inspect every bit }
      if  i ∈ w {w[i] = 1}  then  c := c + 1;  { count the ones}
   return(c)
end;
```

Skip the zeros

Is there a faster way? The following algorithm looks mysterious and tricky. The expression w ∩ (w – 1) contains both an intersection operation '∩', which assumes that its operands are sets, and a subtraction, which assumes that w is an integer:

```
c := 0;
while  w ≠ 0  do  { c := c + 1;  w := w ∩ (w - 1) } ;
```

Such mixing makes sense only if we can rely on an implicit assumption on how sets and integers are represented as bit vectors. With the usual binary number representation, an example shows that when the body of the loop is executed once, the rightmost 1 of w is replaced by 0:

```
w                          1000100011001000

w - 1                      1000100011000111

w ∩ (w - 1)                1000100011000000
```

This clever code seems to look at the 1's only and skip over all the 0's: Its loop is executed only as many times as there are 1's in the word. This savings is worthwhile for long, sparsely populated words (few 1's and many 0's).

In the statement w := w ∩ (w – 1), w is used both as an integer (in w – 1) and as a set (as an operand in the intersection operation '∩'). Strongly typed languages, such as Pascal, do not allow such mixing of types. In the following function 'bitsum$_1$', the conversion routines 'w16toi' and 'itow16' are introduced to avoid this double interpretation of w. However, 'bitsum$_1$' is of interest only if such a type conversion requires no extra time (i.e. if one knows how sets and integers are represented internally).

```
function bitsum1(w: w16): integer;
var  c, i: integer;  w0, w1: w16;
begin
   w0 := w;  c := 0;
   while  w0 ≠ Ø  { empty set }  do  begin
      i := w16toi(w0);  { w16toi converts type w16 to integer }
      i := i - 1;
      w1 := itow16(i);  { itow16 converts type integer to w16 }
      w0 := w0 ∩ w1;  { intersection of two sets }
      c := c + 1
   end;
   return(c)
end;
```

Most languages provide some facility for permitting purely formal type conversions that result in no work: 'EQUIVALENCE' statements in Fortran, 'UNSPEC' in PL/1, variant records in Pascal. Such "conversions" are done merely by interpreting the contents of a given storage location in different ways.

Logarithmic bit sum

For a computer of word length n, the following algorithm computes the bit sum of a word w running through its loop only $\lceil \log_2 n \rceil$ times, as opposed to n times for '$bitsum_0$' or up to n times for '$bitsum_1$'. The following description holds for arbitrary n but is understood most easily if $n = 2h$.

The logarithmic bit sum works on the familiar principle of divide-and-conquer. Let w denote a word consisting of $n = 2h$ bits, and let S(w) be the bit sum of the bit string w. Split w into two halves and denote its left part by wL and its right part by wR. The bit sum obviously satisfies the recursive equation S(w) = S(wL) + S(wR). Repeating the same argument on the substrings wL and wR, and, in turn, on the substrings they create, we arrive at a process to compute S(w). This process terminates when we hit substrings of length 1 [i.e. substrings consisting of a single bit b; in this case we have S(b) = b]. Repeated halving leads to a recursive decomposition of w, and the bit sum is computed by a tree of n – 1 additions as shown below for n = 4 (Exhibit 8.3).

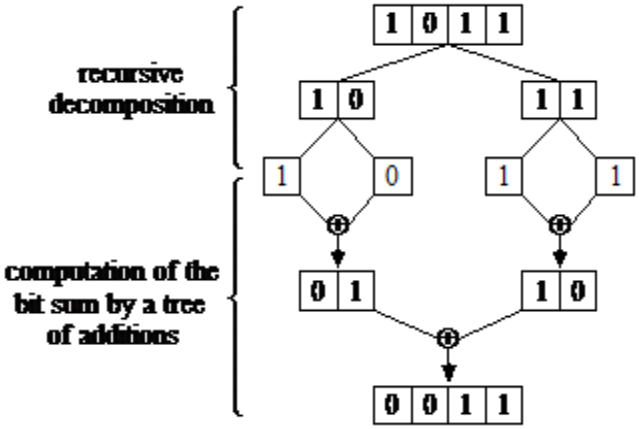

Exhibit 8.3: Logarithmic bit sum algorithm as a result of divide-and-conquer.

This approach of treating both parts of w symmetrically and repeated halving leads to a computation of depth $h = \lceil \log_2 n \rceil$. To obtain a logarithmic bit sum, we apply the additional trick of performing many additions in parallel. Notice that the total length of all operands on the same level is always n. Thus we can pack them into a single word and, if we arrange things cleverly, perform all the additions at the same level in one machine operation, an addition of two n-bit words.

Exhibit 8.4 shows how a number of the additions on short strings are carried out by a *single* addition on long strings. S(w) now denotes not only the bit sum but also its binary representation, padded with zeros to the left so as to have the appropriate length. Since the same algorithm is being applied to w_L and w_R, and since w_L and w_R are of equal length, exactly the same operations are performed at each stage on wL and its parts as on wR and its corresponding parts. Thus if the operations of addition and shifting operate on words of length n, a single one of these operations can be interpreted as performing many of the same operations on the shorter parts into which w has been split. This logarithmic speedup works up to the word length of the computer. For n = 64, for example, recursive splitting generates six levels and translates into six iterations of the loop below.

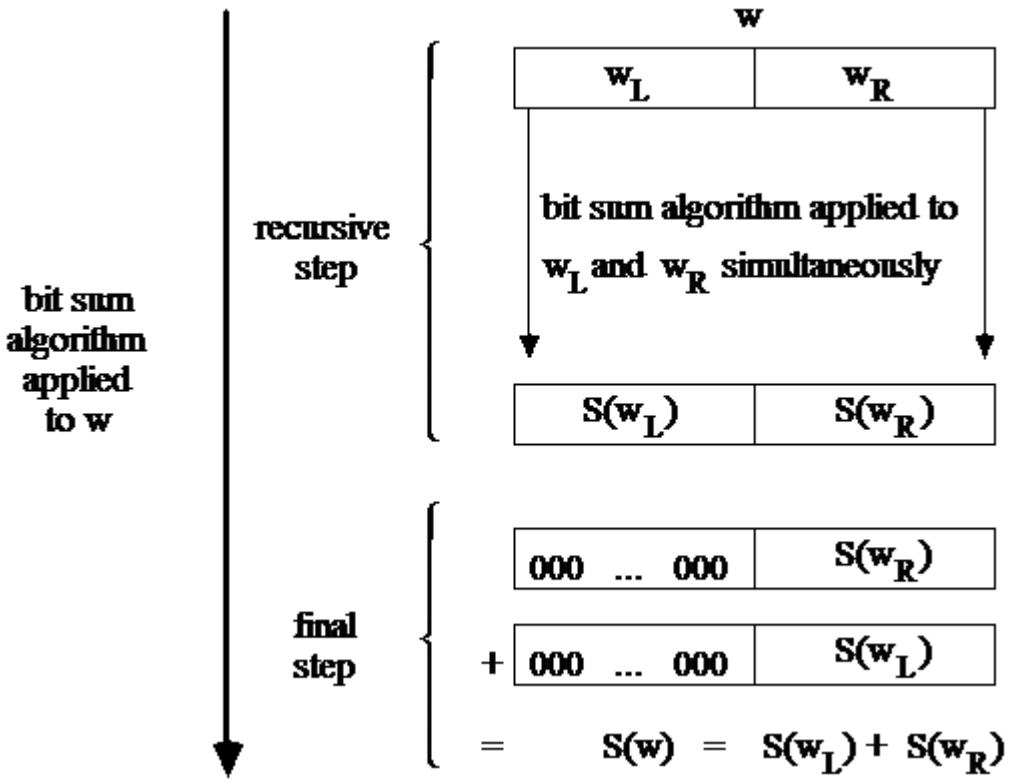

Exhibit 8.4: All processes generated by divide-and-conquer are performed in parallel on shared data registers.

The algorithm is best explained with an example; we use n = 8.

	w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0
w	1	1	0	1	0	0	0	1

First, extract the even-indexed bits w_6 w_4 w_2 w_0 and place a zero to the left of each bit to obtain w_{even}. The newly inserted zeros are shown in small type.

		w_6		w_4		w_2		w_0
w_{even}	0	1	0	1	0	0	0	1

Next, extract the odd-indexed bits w_7 w_5 w_3 w_1, shift them right by one place into bit positions w_6 w_4 w_2 w_0, and place a zero to the left of each bit to obtain w_{odd}.

		w_7		w_5		w_3		w_1
w_{odd}	0	1	0	0	0	0	0	0

Then, numerically add w_{even} and w_{odd}, considered as integers written in base 2, to obtain w'.

	w'_7	w'_6	w'_5	w'_4	w'_3	w'_2	w'_1	w'_0
w_{even}	0	1	0	1	0	0	0	1
w_{odd}	0	1	0	0	0	0	0	0
w'	1	0	0	1	0	0	0	1

Next, we index not bits, but pairs of bits, from right to left: (w'_1 w'_0) is the zeroth pair, (w'_5 w'_4) is the second pair. Extract the even-indexed pairs w'_5 w'_4 and w'_1 w'_0, and place a pair of zeros to the left of each pair to obtain w'_{even}.

			w'_5	w'_4			w'_1	w'_0
w'_{even}	0	0	0	1	0	0	0	1

Next, extract the odd-indexed pairs $w'_7\ w'_6$ and $w'_3\ w'_2$, shift them right by two places into bit positions $w'_5\ w'_4$ and $w'_1\ w'_0$, respectively, and insert a pair of zeros to the left of each pair to obtain w'_{odd}.

			w'_7	w'_6			w'_3	w'_2
w'_{odd}	0	0	1	0	0	0	0	0

Numerically, add w'_{even} and w'_{odd} to obtain w".

	w''_7	w''_6	w''_5	w''_4	w''_3	w''_2	w''_1	w''_0
w"	0	0	1	1	0	0	0	1

Next, we index quadruples of bits, extract the quadruple $w''_3\ w''_2\ w''_1\ w''_0$, and place four zeros to the left to obtain w''_{even}.

					w''_3	w''_2	w''_1	w''_0
w''_{even}	0	0	0	0	0	0	0	1

Extract the quadruple $w''_7\ w''_6\ w''_5\ w''_4$, shift it right four places into bit positions $w''_3\ w''_2\ w''_1\ w''_0$, and place four zeros to the left to obtain w''_{odd}.

					w''_7	w''_6	w''_5	w''_4
w''_{odd}	0	0	0	0	0	0	1	1

Finally, numerically add w''_{even} and w''_{odd} to obtain w''' = (00000100), which is the representation in base 2 of the bit sum of w (4 in this example). The following function implements this algorithm.

Logarithmic bit sum implemented for a 16-bit computer:

In 'bitsum$_2$' we apply addition and division operations directly to variables of type 'w16' without performing the type conversions that would be necessary in a strongly typed language such as Pascal.

```
function bitsum2(w: w16): integer;
const  mask[0] = '0101010101010101';
       mask[1] = '0011001100110011';
       mask[2] = '0000111100001111';
       mask[3] = '0000000011111111';
var  i, d: integer;  weven, wodd: w16;
begin
  d := 2;
  for  i := 0  to  3  do  begin
    weven := w ∩ mask[i];
    w := w / d;  { shift w right 2^i bits }
    d := d^2;
    wodd := w ∩ mask[i];
    w := weven + wodd
  end;
  return(w)
end;
```

Trade-off between time and space: the fastest algorithm

Are there still faster algorithms for computing the bit sum of a word? Is there an *optimal* algorithm? The question of optimality of algorithms is important, but it can be answered only in special cases. To show that an algorithm is optimal, one must specify precisely the class of algorithms allowed and the criterion of optimality. In the case of bit sum algorithms, such specifications would be complicated and largely arbitrary, involving specific details of how computers work.

However, we can make a plausible argument that the following bit sum algorithm is the fastest possible, since it uses a table lookup to obtain the result in essentially one operation. The penalty for this speed is an extravagant use of memory space (2^n locations), thereby making the algorithm impractical except for small values of n. The choice of an algorithm almost always involves trade-offs among various desirable properties, and the better an algorithm is from one aspect, the worse it may be from another.

The algorithm is based on the idea that we can precompute the solutions to all possible questions, store the results, and then simply look them up when needed. As an example, for n = 3, we would store the information

```
Word      Bit sum

0  0  0       0
0  0  1       1
0  1  0       1
0  1  1       2
1  0  0       1
1  0  1       2
1  1  0       2
1  1  1       3
```

What is the fastest way of looking up a word w in this table? Under assumptions similar to those used in the preceding algorithms, we can interpret w as an address of a memory cell that contains the bit sum of w, thus giving us an algorithm that requires only one memory reference.

Table lookup implemented for a 16-bit computer:

```
function bitsum3(w: w16): integer;
const  c: array[0 .. 65535] of integer = [0, 1, 1, 2, 1, 2, 2, 3,
… , 15, 16];
begin  return(c[w])  end;
```

In concluding this example, we notice the variety of algorithms that exist for computing the bit sum, each one based on entirely different principles, giving us a different trade-off between space and time. 'bitsum$_0$' and 'bitsum$_3$' solve the problem by "brute force" and are simple to understand: 'bitsum$_0$' looks at each bit and so requires much time; 'bitsum$_3$' stores the solution for each separate case and thus requires much space. The logarithmic bit sum algorithm is an elegant compromise: efficient with respect to both space and time, it merely challenges the programmer's wits.

Exercises

1. Show that there are exactly 16 distinct boolean functions of two variables.
2. Show that each of the boolean functions 'nand' and 'nor' is universal in the following sense: Any boolean function f(x, y) can be written as a nested expression involving only 'nands', and it can also be written using only 'nors'. Show that no other boolean function of two variables is universal.

3. Consider the logarithmic bit sum algorithm, and show that *any* strategy for splitting w (not just the halving split) requires n – 1 additions.