THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 0C

A DRAFT OF SECTION 7.1.2: BOOLEAN EVALUATION

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

Internet
Stanford GraphBase
MMIX

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also `http://www-cs-faculty.stanford.edu/~knuth/mmixware.html` for downloadable software to simulate the MMIX computer.

Zeroth printing (revision 8), 30 January 2008

SENECA
Internet
Stanford GraphBase

PREFACE

Your mind should break free of custom, furiously seizing the bit
and recklessly choosing its own path,
where you would fear to ascend by yourself.
— SENECA, *De Tranquillitate Animi* (c. 50)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, and 3 were at the time of their first printings. And those carefully-checked volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make the text both interesting and authoritative, as far as it goes. But the field is so vast, I cannot hope to have surrounded it enough to corral it completely. Therefore I beg you to let me know about any deficiencies that you discover.

To put the material in context, this pre-fascicle contains Section 7.1.2 of a long, long chapter on combinatorial algorithms. Chapter 7 will eventually fill at least three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in the Stanford GraphBase, from which I will be drawing many examples. Then comes Section 7.1: Zeros and Ones, beginning with basic material in Section 7.1.1 (see pre-fascicle 0b). Section 7.1.2, which you're about to read here, is concerned with the study of efficient Boolean function evaluation. Section 7.1.3 will deal with tricks and techniques of bitwise calculation; Section 7.1.4 will discuss data structures for Boolean functions.

The next section, 7.2, is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns. Fascicles for this section have already appeared on the Web and/or in print. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

Liang
Paterson
Schroeppel
footprints
Knuth

The topic of Boolean functions and bit manipulation can of course be interpreted so broadly that it encompasses the entire subject of computer programming. My original title for Section 7.1 — "Bit Fiddling" — was much more modest; I decided, however, that those words were a bit too low-brow. The real goal of this fascicle is to focus on concepts that appear at the lowest levels, on which we can erect significant superstructures. And even these apparently lowly notions turn out to be surprisingly rich, with explicit ties to Sections 2.3.4.4, 4.3.1, 4.6.4, and 5.3.4 of the first three volumes. I strongly believe in building up a firm foundation, so I have discussed Boolean topics much more thoroughly than I will be able to do with material that is newer or less basic. After typing the manuscript I was astonished to discover that I had come up with 88 exercises, even though — believe it or not — I had to eliminate quite a lot of the interesting material that appears in my files.

My notes on combinatorial algorithms have been accumulating for more than forty years, so I fear that in several respects my knowledge is woefully behind the times. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 21 and 24; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises 17, 40, 55, 61, 63, 70, and 80. Are those problems still open? Please let me know if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises 11, 14, 16, 27, 29, 30, 34, 40, and 88, and/or the answer to exercise 38. Furthermore I've cited unpublished results of Frank Liang, Mike Paterson, and Rich Schroeppel; do you know of any related publications?

The text presents an approach to synthesis based on so-called "footprints" of Boolean functions, which I haven't seen in the literature. Is this method new, or did I overlook some relevant papers?

I shall happily pay a finder's fee of $2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00'; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
14 March 2006

Helpful hints. Readers of Section 7.1.2 should ideally have already read (or at least skimmed) Section 7.1.1. In particular, they should not be shocked or puzzled by notations such as

i) $\nu(x_1x_2\ldots x_n)$ or $\nu((x_1x_2\ldots x_n)_2)$ for the sum $x_1+x_2+\cdots+x_n$;

ii) $S_{k_1,k_2,\ldots,k_t}(x)$ for the symmetric function that is true when $\nu x = k_1$ or k_2 or $\ldots$ or k_t;

iii) $\langle x_1x_2\ldots x_{2k-1}\rangle$ for the threshold function $S_{\geq k}$ that equals the median value of $\{x_1,x_2,\ldots,x_{2k-1}\}$ (which is also the majority value).

notations
sideways sum
symmetric function
median
majority
Stanford University InfoLab
Sun Microsystems Laboratories
five variables

♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡ ♡

I thank the Stanford University InfoLab and Sun Microsystems Laboratories for generously donating many hours of computer time on large, fast machines, thereby allowing me to investigate the Boolean functions of five variables.

By and large the minimization of switching components
outweighs all other engineering considerations
in designing economical logic circuits.
— H. A. CURTIS, *A New Approach to the Design of Switching Circuits* (1962)

He must be a great calculator indeed who succeeds.
Simplify, simplify.
— HENRY D. THOREAU, *Walden; or, Life in the Woods* (1854)

7.1.2. Boolean Evaluation

Our next goal is to study the efficient evaluation of Boolean functions, much as we studied the evaluation of polynomials in Section 4.6.4. One natural way to investigate this topic is to consider chains of basic operations, analogous to the polynomial chains discussed in that section.

A *Boolean chain*, for functions of n variables $(x_1, \ldots, x_n)$, is a sequence $(x_{n+1}, \ldots, x_{n+r})$ with the property that each step combines two of the preceding steps:

$$x_i \;=\; x_{j(i)} \circ_i x_{k(i)}, \qquad \text{for } n+1 \le i \le n+r, \tag{1}$$

where $1 \le j(i) < i$ and $1 \le k(i) < i$, and where $\circ_i$ is one of the sixteen binary operators of Table 7.1.1–1. For example, when $n = 3$ the two chains

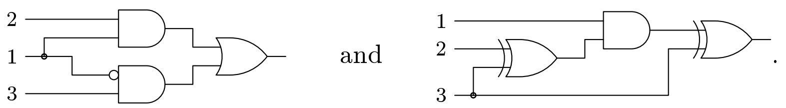
$$\begin{array}{lcl} x_4 = x_1 \wedge x_2 & & x_4 = x_2 \oplus x_3 \\ x_5 = \bar{x}_1 \wedge x_3 & \text{and} & x_5 = x_1 \wedge x_4 \\ x_6 = x_4 \vee x_5 & & x_6 = x_3 \oplus x_5 \end{array} \tag{2}$$

both evaluate the "mux" or "if-then-else" function $x_6 = (x_1?\ x_2\colon x_3)$, which takes the value x_2 or x_3 depending on whether x_1 is 1 (true) or 0 (false).

NOT-BUT
circuits
gate
engineers
fan-out and fan-in
binary trees
overlapping subtrees
BUT-NOT
topological sorting
combinational complexity
DNF
CNF

(Notice that the left-hand example in (2) uses the simplified notation '$x_5 = \bar{x}_1 \wedge x_3$' to specify the NOT-BUT operation, instead of the form '$x_5 = x_1 \mathbin{\overline{\subset}} x_3$' that appears in Table 7.1.1–1. The main point is that, regardless of notation, every step of a Boolean chain is a Boolean combination of two prior results.)

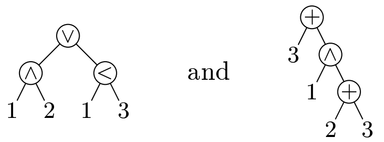
Boolean chains correspond naturally to electronic circuits, with each step in the chain corresponding to a "gate" that has two inputs and one output. Electrical engineers traditionally represent the Boolean chains of (2) by circuit diagrams such as



(3)

They need to design economical circuits that are subject to various technological constraints; for example, some gates might be more expensive than others, some outputs might need to be amplified if reused, the layout might need to be planar or nearly so, some paths might need to be short. But our chief concern in this book is software, not hardware, so we don't have to worry about such things. For our purposes, all gates have equal cost, and all outputs can be reused as often as desired. (Jargonwise, our Boolean chains boil down to circuits in which all gates have fan-in 2 and unlimited fan-out.)

Furthermore we shall depict Boolean chains as binary trees such as



(4)

instead of using circuit diagrams like (3). Such binary trees will have overlapping subtrees when intermediate steps of the chain are used more than once. Every internal node is labeled with a binary operator; external nodes are labeled with an integer k, representing the variable x_k. The label '$\bigcirc\!\!\!\!<$' in the left tree of (4) stands for the NOT-BUT operator, since $\bar{x} \wedge y = [x < y]$; similarly, the BUT-NOT operator, $x \wedge \bar{y}$, can be represented by the node label '$\bigcirc\!\!\!\!>$'.

Several different Boolean chains might have the same tree diagram. For example, the left-hand tree of (4) also represents the chain

$$x_4 = \bar{x}_1 \wedge x_3, \qquad x_5 = x_1 \wedge x_2, \qquad x_6 = x_5 \vee x_4.$$

Any topological sorting of the tree nodes yields an equivalent chain.

Given a Boolean function f of n variables, we often want to find a Boolean chain such that $x_{n+r} = f(x_1, \ldots, x_n)$, where r is as small as possible. The *combinational complexity* $C(f)$ of a function f is the length of the shortest chain that computes it. To save excess verbiage, we will simply call $C(f)$ the "cost of f." The mux function in our examples above has cost 3, because one can show by exhaustive trials that it can't be produced by any Boolean chain of length 2.

The DNF and CNF representations of f, which we studied in Section 7.1.1, rarely tell us much about $C(f)$, since substantially more efficient schemes of

pi, random
symmetric functions–
Truth tables
parity
binary trees

calculation are usually possible. For example, in the discussion following 7.1.1–(30) we found that the more-or-less random function of four variables whose truth table is 1100 1001 0000 1111 has no DNF expression shorter than

$$(\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3) \vee (\bar{x}_1 \wedge \bar{x}_3 \wedge \bar{x}_4) \vee (x_2 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_2). \tag{5}$$

This formula corresponds to a Boolean chain of 10 steps. But that function can also be expressed more cleverly as

$$\bigl(((x_2 \wedge \bar{x}_4) \oplus \bar{x}_3) \wedge \bar{x}_1\bigr) \oplus x_2, \tag{6}$$

so its complexity is at most 4.

How can nonobvious formulas like (6) be discovered? We will see that a computer can find the best chains for functions of four variables without doing an enormous amount of work. Still, the results can be quite startling, even for people who have had considerable experience with Boolean algebra. Typical examples of this phenomenon can be seen in Fig. 9, which illustrates the four-variable functions that are perhaps of greatest general interest, namely the functions that are symmetric under all permutations of their variables.

Consider, for example, the function $S_2(x_1, x_2, x_3, x_4)$, for which we have

$$\begin{array}{rll} x_1 & & 0000\ 0000\ 1111\ 1111 \\ x_2 & & 0000\ 1111\ 0000\ 1111 \\ x_3 & & 0011\ 0011\ 0011\ 0011 \\ x_4 & & 0101\ 0101\ 0101\ 0101 \\ x_5 = x_1 \oplus x_3 & & 0011\ 0011\ 1100\ 1100 \\ x_6 = x_1 \oplus x_2 & & 0000\ 1111\ 1111\ 0000 \\ x_7 = x_3 \oplus x_4 & & 0110\ 0110\ 0110\ 0110 \\ x_8 = x_5 \vee x_6 & & 0011\ 1111\ 1111\ 1100 \\ x_9 = x_6 \oplus x_7 & & 0110\ 1001\ 1001\ 0110 \\ x_{10} = x_8 \wedge \bar{x}_9 & & 0001\ 0110\ 0110\ 1000 \end{array} \tag{7}$$

according to Fig. 9. Truth tables are shown here so that we can easily verify each step of the calculation. Step x_8 yields a function that is true whenever $x_1 \neq x_2$ or $x_1 \neq x_3$; and $x_9 = x_1 \oplus x_2 \oplus x_3 \oplus x_4$ is the parity function $(x_1 + x_2 + x_3 + x_4) \bmod 2$. Therefore the final result, x_{10}, is true precisely when exactly two of $\{x_1, x_2, x_3, x_4\}$ are 1; these are the cases that satisfy x_8 and have even parity.

Several of the other computational schemes of Fig. 9 can also be justified intuitively. But some of the chains, like the one for $S_{1,4}$, are quite amazing.

Notice that the intermediate result x_6 is used twice in (7). In fact, no six-step chain for the function $S_2(x_1, x_2, x_3, x_4)$ is possible without making double use of some intermediate subexpression; the shortest algebraic formulas for S_2, including nice symmetrical ones like

$$\bigl((x_1 \wedge x_2) \vee (x_3 \wedge x_4)\bigr) \oplus \bigl((x_1 \vee x_2) \wedge (x_3 \vee x_4)\bigr), \tag{8}$$

all have cost 7. But Fig. 9 shows that the other symmetric functions of four variables can all be evaluated optimally via "pure" binary trees, without overlapping subtrees except at external nodes (which represent the variables).

length
formula complexity, see length
depth
functions of four variables–complement

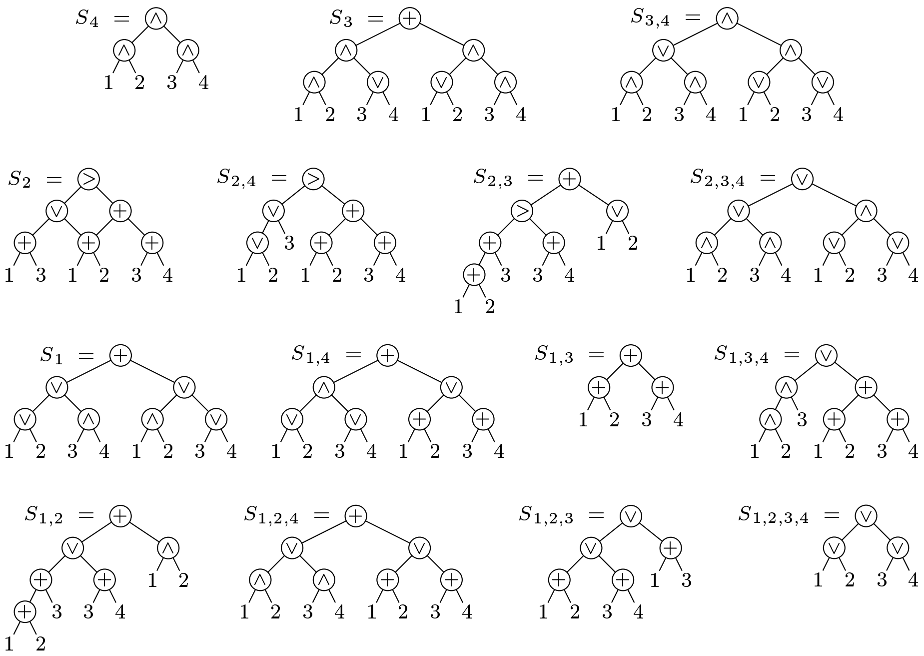


Fig. 9. Optimum Boolean chains for the symmetric functions of four variables.

In general, if $f(x_1,\ldots,x_n)$ is any Boolean function, we say that its *length* $L(f)$ is the number of binary operators in the shortest formula for f. Obviously $L(f) \geq C(f)$; and we can easily verify that $L(f) = C(f)$ whenever $n \leq 3$, by considering the fourteen basic types of 3-variable functions in 7.1.1–(95). But we have just seen that $L(S_2) = 7$ exceeds $C(S_2) = 6$ when $n = 4$, and in fact $L(f)$ is almost always substantially larger than $C(f)$ when n is large (see exercise 49).

The *depth* $D(f)$ of a Boolean function f is another important measure of its inherent complexity: We say that the depth of a Boolean chain is the length of the longest downward path in its tree diagram, and $D(f)$ is the minimum achievable depth when all Boolean chains for f are considered. All of the chains illustrated in Fig. 9 have not only the minimum cost but also the minimum depth — except in the cases $S_{2,3}$ and $S_{1,2}$, where we cannot simultaneously achieve cost 6 and depth 3. The formula

$$S_{2,3}(x_1,x_2,x_3,x_4) = \bigl((x_1 \wedge x_2) \oplus (x_3 \wedge x_4)\bigr) \vee \bigl((x_1 \vee x_2) \wedge (x_3 \oplus x_4)\bigr) \qquad (9)$$

shows that $D(S_{2,3}) = 3$, and a similar formula works for $S_{1,2}$.

Optimum chains for $n = 4$. Exhaustive computations for 4-variable functions are feasible because such functions have only $2^{16} = 65{,}536$ possible truth tables. In fact we need only consider half of those truth tables, because the complement $\bar{f}$ of any function f has the same cost, length, and depth as f itself.

normal
normalization
bitwise
truth tables
depths

Let's say that $f(x_1,\ldots,x_n)$ is *normal* if $f(0,\ldots,0)=0$, and in general that

$$f(x_1,\ldots,x_n)\oplus f(0,\ldots,0) \qquad (10)$$

is the "normalization" of f. Any Boolean chain can be normalized by normalizing each of its steps and by making appropriate changes to the operators; for if $(\hat{x}_1,\ldots,\hat{x}_{i-1})$ are the normalizations of $(x_1,\ldots,x_{i-1})$ and if $x_i=x_{j(i)}\circ_i x_{k(i)}$ as in (1), then $\hat{x}_i$ is clearly a binary function of $\hat{x}_{j(i)}$ and $\hat{x}_{k(i)}$. (Exercise 7 presents an example.) Therefore we can restrict consideration to normal Boolean chains, without loss of generality.

Notice that a Boolean chain is normal if and only if each of its binary operators $\circ_i$ is normal. And there are only eight normal binary operators — three of which, namely $\perp$, $\llcorner$, and $\urcorner$, are trivial. So we can assume that all Boolean chains of interest are formed from the five operators $\wedge$, $\overline{\subset}$, $\overline{\supset}$, $\vee$, and $\oplus$, which are denoted respectively by Ⓐ, ⊲, ⊳, Ⓥ, and ⊕ in Fig. 9. Furthermore we can assume that $j(i)<k(i)$ in each step.

There are $2^{15}=32{,}768$ normal functions of four variables, and we can compute their lengths without difficulty by systematically enumerating all functions of length 0, 1, 2, etc. Indeed, $L(f)=r$ implies that $f=g\circ h$ for some g and h, where $L(g)+L(h)=r-1$ and $\circ$ is one of the five nontrivial normal operators; so we can proceed as follows:

Algorithm L (*Find normal lengths*). This algorithm determines $L(f)$ for all normal truth tables $0\le f<2^{2^n-1}$, by building lists of all nonzero normal functions of length r for $r\ge 0$.

L1. [Initialize.] Let $L(0)\leftarrow 0$ and $L(f)\leftarrow\infty$ for $1\le f<2^{2^n-1}$. Then, for $1\le k\le n$, set $L(x_k)\leftarrow 0$ and put x_k into list 0, where

$$x_k \;=\; (2^{2^n}-1)/(2^{2^{n-k}}+1) \qquad (11)$$

is the truth table for x_k. (See exercise 8.) Finally, set $c\leftarrow 2^{2^n-1}-n-1$; c is the number of places where $L(f)=\infty$.

L2. [Loop on r.] Do step L3 for $r=1, 2, \ldots$; eventually the algorithm will terminate when c becomes 0.

L3. [Loop on j and k.] Do step L4 for $j=0, 1, \ldots$, and $k=r-1-j$, while $j\le k$.

L4. [Loop on g and h.] Do step L5 for all g in list j and all h in list k. (If $j=k$, it suffices to restrict h to functions that *follow* g in list k.)

L5. [Loop on f.] Do step L6 for $f=g\,\&\,h$, $f=\bar{g}\,\&\,h$, $f=g\,\&\,\bar{h}$, $f=g\mid h$, and $f=g\oplus h$. (Here $g\,\&\,h$ denotes the bitwise AND of the integers g and h; we are representing truth tables by integers in binary notation.)

L6. [Is f new?] If $L(f)=\infty$, set $L(f)\leftarrow r$, $c\leftarrow c-1$, and put f in list r. Terminate the algorithm if $c=0$. ▮

Exercise 10 shows that a similar procedure will compute all depths $D(f)$.

With a little more work, we can in fact modify Algorithm L so that it finds better upper bounds on $C(f)$, by computing a heuristic bit vector $\phi(f)$ called

footprint
length stats
cost stats
depth stats
minimum memory–registers

Table 1

THE NUMBER OF FOUR-VARIABLE FUNCTIONS WITH GIVEN COMPLEXITY

$C(f)$	Classes	Functions	$L(f)$	Classes	Functions	$D(f)$	Classes	Functions
0	2	10	0	2	10	0	2	10
1	2	60	1	2	60	1	2	60
2	5	456	2	5	456	2	17	1458
3	20	2474	3	20	2474	3	179	56456
4	34	10624	4	34	10624	4	22	7552
5	75	24184	5	75	24184	5	0	0
6	72	25008	6	68	24640	6	0	0
7	12	2720	7	16	3088	7	0	0

the "footprint" of f. A normal Boolean chain can begin in only $5\binom{n}{2}$ different ways, since the first step x_{n+1} must be either $x_1 \wedge x_2$ or $\bar{x}_1 \wedge x_2$ or $x_1 \wedge \bar{x}_2$ or $x_1 \vee x_2$ or $x_1 \oplus x_2$ or $x_1 \wedge x_3$ or $\cdots$ or $x_{n-1} \oplus x_n$. Suppose $\phi(f)$ is a bit vector of length $5\binom{n}{2}$ and $U(f)$ is an upper bound on $C(f)$, with the following property: Every 1 bit in $\phi(f)$ corresponds to the first step of some Boolean chain that computes f in $U(f)$ steps.

Such pairs $(U(f), \phi(f))$ can be computed by extending the basic strategy of Algorithm L. Initially we set $U(f) \leftarrow 1$ and we set $\phi(f)$ to an appropriate vector $0\ldots010\ldots0$, for all functions f of cost 1. Then, for $r = 2, 3, \ldots,$ we proceed to look for functions $f = g \circ h$ where $U(g) + U(h) = r - 1$, as before, but with two changes: (1) If the footprints of g and h have at least one element in common, namely if $\phi(g) \mathbin{\&} \phi(h) \neq 0$, then we know that $C(f) \le r - 1$, so we can decrease $U(f)$ if it was $\ge r$. (2) If the cost of $g \circ h$ is equal to (but not less than) our current upper bound $U(f)$, we can set $\phi(f) \leftarrow \phi(f) \mid (\phi(g) \mid \phi(h))$ if $U(f) = r$, $\phi(f) \leftarrow \phi(f) \mid (\phi(g) \mathbin{\&} \phi(h))$ if $U(f) = r - 1$. Exercise 11 works out the details.

It turns out that this footprint heuristic is powerful enough to find chains of optimum cost $U(f) = C(f)$ for all functions f, when $n = 4$. Moreover, we'll see later that footprints also help us solve more complicated evaluation problems.

According to Table 7.1.1–5, the $2^{16} = 65{,}536$ functions of four variables belong to only 222 distinct classes when we ignore minor differences due to permutation of variables and/or complementation of values. Algorithm L and its variants lead to the overall statistics shown in Table 1.

***Evaluation with minimum memory.** Suppose the Boolean values $x_1, \ldots, x_n$ appear in n registers, and we want to evaluate a function by performing a sequence of operations having the form

$$x_{j(i)} \leftarrow x_{j(i)} \circ_i x_{k(i)}, \qquad \text{for } 1 \le i \le r, \tag{12}$$

where $1 \le j(i) \le n$ and $1 \le k(i) \le n$ and $\circ_i$ is a binary operator. At the end of the computation, the desired function value should appear in one of the registers. When $n = 3$, for example, the four-step sequence

$$\begin{array}{ll} & (x_1 = 00001111 \quad x_2 = 00110011 \quad x_3 = 01010101) \\ x_1 \leftarrow x_1 \oplus x_2 & (x_1 = 00111100 \quad x_2 = 00110011 \quad x_3 = 01010101) \\ x_3 \leftarrow x_3 \wedge x_1 & (x_1 = 00111100 \quad x_2 = 00110011 \quad x_3 = 00010100) \\ x_2 \leftarrow x_2 \wedge \bar{x}_1 & (x_1 = 00111100 \quad x_2 = 00000011 \quad x_3 = 00010100) \\ x_3 \leftarrow x_3 \vee x_2 & (x_1 = 00111100 \quad x_2 = 00000011 \quad x_3 = 00010111) \end{array} \tag{13}$$

computes the median $\langle x_1x_2x_3\rangle$ and puts it into the original position of x_3. (All eight possibilities for the register contents are shown here as truth tables, before and after each operation.)

median
truth tables
bitwise operations
backward-computation principle
normal functions

In fact we can check the calculation by working with only one truth table at a time, instead of keeping track of all three, if we analyze the situation backwards. Let $f_l(x_1,\ldots,x_n)$ denote the function computed by steps l, $l+1$, $\ldots$, r of the sequence, omitting the first $l-1$ steps; thus, in our example, $f_2(x_1,x_2,x_3)$ would be the result in x_3 after the three steps $x_3 \leftarrow x_3\wedge x_1$, $x_2 \leftarrow x_2\wedge\bar{x}_1$, $x_3 \leftarrow x_3\vee x_2$. Then the function computed in register x_3 by all four steps is

$$f_1(x_1,x_2,x_3) \;=\; f_2(x_1\oplus x_2,x_2,x_3). \tag{14}$$

Similarly $f_2(x_1,x_2,x_3)=f_3(x_1,x_2,x_3\wedge x_1)$, $f_3(x_1,x_2,x_3)=f_4(x_1,x_2\wedge\bar{x}_1,x_3)$, $f_4(x_1,x_2,x_3)=f_5(x_1,x_2,x_3\vee x_2)$, and $f_5(x_1,x_2,x_3)=x_3$. We can therefore go back from f_5 to f_4 to $\cdots$ to f_1 by operating on truth tables in an appropriate way.

For example, suppose $f(x_1,x_2,x_3)$ is a function whose truth table is

$$t \;=\; a_0a_1a_2a_3a_4a_5a_6a_7;$$

then the truth table for $g(x_1,x_2,x_3)=f(x_1\oplus x_2,x_2,x_3)$ is

$$u \;=\; a_0a_1a_6a_7a_4a_5a_2a_3,$$

obtained by replacing a_x by $a_{x'}$, where

$$x=(x_1x_2x_3)_2 \qquad\text{implies}\qquad x'=((x_1\oplus x_2)x_2x_3)_2.$$

Similarly the truth table for, say, $h(x_1,x_2,x_3)=f(x_1,x_2,x_3\wedge x_1)$ is

$$v \;=\; a_0a_0a_2a_2a_4a_5a_6a_7.$$

And we can use bitwise operations to compute u and v from t:

$$u=t\oplus\bigl((t\oplus(t\gg 4)\oplus(t\ll 4))\mathbin{\&}(00110011)_2\bigr), \tag{15}$$
$$v=t\oplus\bigl((t\oplus(t\gg 1))\mathbin{\&}(01010000)_2\bigr). \tag{16}$$

Let $C_m(f)$ be the length of a shortest minimum-memory computation for f. The backward-computation principle tells us that, if we know the truth tables of all functions f with $C_m(f)<r$, we can readily find all the truth tables of functions with $C_m(f)=r$. Namely, we can restrict consideration to normal functions as before. Then, for all normal g such that $C_m(g)=r-1$, we can construct the $5n(n-1)$ truth tables for

$$g(x_1,\ldots,x_{j-1},x_j\circ x_k,x_{j+1},\ldots,x_n) \tag{17}$$

and mark them with cost r if they haven't previously been marked. Exercise 14 shows that those truth tables can all be computed by performing simple bitwise operations on the truth table for g.

When $n=4$, all but 13 of the 222 basic function types turn out to have $C_m(f)=C(f)$, so they can be evaluated in minimum memory without increasing the cost. In particular, all of the symmetric functions have this property—although that fact is not at all obvious from Fig. 9. Five classes of functions have $C(f) = 5$ but $C_m(f) = 6$; eight classes have $C(f) = 6$ but $C_m(f) = 7$. The most interesting example of the latter type is probably the function $(x_1 \vee x_2) \oplus (x_3 \vee x_4) \oplus (x_1 \wedge x_2 \wedge x_3 \wedge x_4)$, which has cost 6 because of the formula

$$x_1 \oplus (x_3 \vee x_4) \oplus \bigl(x_2 \wedge (\bar{x}_1 \vee (x_3 \wedge x_4))\bigr), \qquad (18)$$

but it has no minimum-memory chain of length less than 7. (See exercise 15.)

optimum Boolean evaluation–
top-down
formula length
minimum-memory

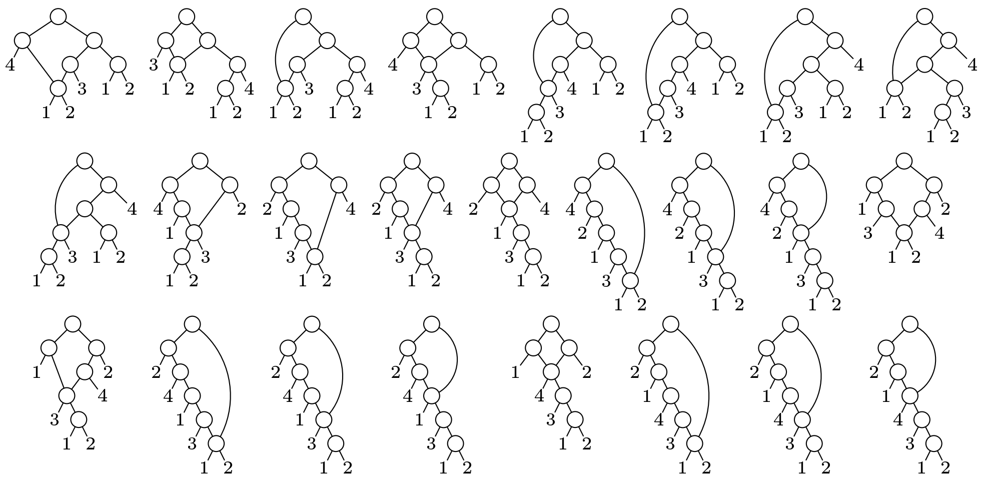
***Determining the minimum cost.** The exact value of $C(f)$ can be found by observing that all optimum Boolean chains $(x_{n+1}, \ldots, x_{n+r})$ for f obviously satisfy at least one of three conditions:

i) $x_{n+r} = x_j \circ x_k$, where x_j and x_k use no common intermediate results;
ii) $x_{n+1} = x_j \circ x_k$, where either x_j or x_k is not used in steps $x_{n+2}, \ldots, x_{n+r}$;
iii) Neither of the above, even when the intermediate steps are renumbered.

In case (i) we have $f = g \circ h$, where $C(g) + C(h) = r - 1$, and we can call this a "top-down" construction. In case (ii) we have $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_{j-1}, x_j \circ x_k, x_{j+1}, \ldots, x_n)$, where $C(g) = r - 1$; we call this construction "bottom-up."

The best chains that recursively use only top-down constructions correspond to minimum formula length, $L(f)$. The best chains that recursively use only bottom-up constructions correspond to minimum-memory calculations, of length $C_m(f)$. We can do better yet, by mixing top-down constructions with bottom-up constructions; but we still won't know that we've found $C(f)$, because a special chain belonging to case (iii) might be shorter.

Fortunately such special chains are rare, because they must satisfy rather strong conditions, and they can be exhaustively listed when n and r aren't too large. For example, exercise 19 proves that no special chains exist when $r < n+2$; and when $n = 4$, $r = 6$, there are only 25 essentially different special chains that cannot obviously be shortened:



By systematically trying 5^r possibilities in every special chain, one for each way to assign a normal operator to the internal nodes of the tree, we will find at least one function f in every equivalence class for which the minimum cost $C(f)$ is achievable only in case (iii).

In fact, when $n = 4$ and $r = 6$, these $25 \cdot 5^6 = 390{,}625$ trials yield only one class of functions that can't be computed in 6 steps by any top-down-plus-bottom-up chain. The missing class, typified by the partially symmetric function $(\langle x_1x_2x_3\rangle \vee x_4) \oplus (x_1 \wedge x_2 \wedge x_3)$, can be reached in six steps by appropriately specializing any of the first five chains illustrated above; for example, one way is

$$x_5 = x_1 \wedge x_2, \quad x_6 = x_1 \vee x_2, \quad x_7 = x_3 \oplus x_5,$$
$$x_8 = x_4 \wedge \bar{x}_5, \quad x_9 = x_6 \wedge x_7, \quad x_{10} = x_8 \vee x_9, \qquad (19)$$

corresponding to the first special chain. Since all other functions have $L(f) \le 7$, these trial calculations have established the true minimum cost in all cases.

Historical notes: The first concerted attempts to evaluate all Boolean functions $f(w,x,y,z)$ optimally were reported in *Annals of the Computation Laboratory of Harvard University* **27** (1951), where Howard Aiken's staff presented heuristic methods and extensive tables of the best switching circuits they were able to construct. Their cost measure $V(f)$ was different from the cost $C(f)$ that we've been considering, because it was based on "control grids" of vacuum tubes: They had three kinds of gates, NOR, OR, and NAND, each of which could take k inputs with cost k. Every input to such a gate could be either a variable, or the complement of a variable, or the result of a previous gate. Furthermore the function being evaluated was represented at the top level as an AND of any number of gates, with no additional cost.

With those cost criteria, a function might not have the same cost as its complement, because AND gates were possible only at the top level. One could evaluate $x \wedge y$ as $\mathrm{NOR}(\bar{x}, \bar{y})$, with cost 2; but the cost of $\bar{x} \vee (\bar{y} \wedge \bar{z}) = \mathrm{NAND}(x, \mathrm{OR}(y,z))$ was 4 while its complement $x \wedge (y \vee z) = \mathrm{AND}\bigl(\mathrm{NOR}(\bar{x}), \mathrm{OR}(y,z)\bigr)$ cost only 3. Therefore the Harvard researchers needed to consider 402 essentially different classes of 4-variable functions instead of 222 (see the answer to exercise 7.1.1–125). Of course in those days they were working by hand. They found $V(f) < 20$ in all cases, except for the 64 functions equivalent to $S_{0,1}(w,x,y,z) \vee \bigl(S_2(w,x,y) \wedge z\bigr)$, which they evaluated with 20 control grids as follows:

$$g_1 = \mathrm{NOR}(\bar{w}, \bar{x}), \ g_2 = \mathrm{NAND}(\bar{y}, z), \ g_3 = \mathrm{NOR}(w, x),$$
$$f = \mathrm{AND}\bigl(\mathrm{NAND}(g_1, g_2), \mathrm{NAND}(g_3, \mathrm{NOR}(\bar{y}, \bar{z})),$$
$$\mathrm{NOR}(\mathrm{NOR}(g_3, \bar{y}, z), \mathrm{NOR}(g_1, g_2, g_3))\bigr). \qquad (20)$$

The first computer program to find provably optimum circuits was written by Leo Hellerman [*IEEE Transactions* **EC-12** (1963), 198–223], who determined the fewest NOR gates needed to evaluate any given function $f(x,y,z)$. He required every input of every gate to be either an uncomplemented variable or the output of a previous gate; fan-in and fan-out were limited to at most 3. When two circuits had the same gate count, he preferred the one with smallest sum-of-inputs. For example, he computed $\bar{x} = \mathrm{NOR}(x)$ with cost 1; $x \vee y \vee z = \mathrm{NOR}(\mathrm{NOR}(x,y,z))$ with cost 2; $\langle xyz\rangle = \mathrm{NOR}(\mathrm{NOR}(x,y), \mathrm{NOR}(x,z), \mathrm{NOR}(y,z))$

Aiken
control grids
vacuum tubes
gates
NOR
OR
NAND
AND
symmetric funcs
Hellerman
fan-in
fan-out
majority
symmetric+

Table 2

THE NUMBER OF FIVE-VARIABLE FUNCTIONS WITH GIVEN COMPLEXITY

$C(f)$	Classes	Functions	$L(f)$	Classes	Functions	$D(f)$	Classes	Functions
0	2	12	0	2	12	0	2	12
1	2	100	1	2	100	1	2	100
2	5	1140	2	5	1140	2	17	5350
3	20	11570	3	20	11570	3	1789	6702242
4	93	109826	4	93	109826	4	614316	4288259592
5	389	995240	5	366	936440	5	0	0
6	1988	8430800	6	1730	7236880	6	0	0
7	11382	63401728	7	8782	47739088	7	0	0
8	60713	383877392	8	40297	250674320	8	0	0
9	221541	1519125536	9	141422	955812256	9	0	0
10	293455	2123645248	10	273277	1945383936	10	0	0
11	26535	195366784	11	145707	1055912608	11	0	0
12	1	1920	12	4423	31149120	12	0	0

parity function
engineers
Liang
bottom-up
functions of five variables–
Knuth
five variables
hexadecimal truth tables

with cost 4; $S_1(x,y,z) = \mathrm{NOR}\bigl(\mathrm{NOR}(x,y,z), \langle xyz\rangle\bigr)$ with cost 6; etc. Since he limited the fan-out to 3, he found that every function of three variables could be evaluated with cost 7 or less, except for the parity function $x\oplus y\oplus z = (x\equiv y)\equiv z$, where $x\equiv y$ has cost 4 because it is $\mathrm{NOR}(\mathrm{NOR}(x,\mathrm{NOR}(x,y)),\mathrm{NOR}(y,\mathrm{NOR}(x,y)))$.

Electrical engineers continued to explore other cost criteria; but four-variable functions seemed out of reach until 1977, when Frank M. Liang established the values of $C(f)$ shown in Table 1. Liang's unpublished derivation was based on a study of all chains that cannot be reduced by the bottom-up construction.

The case $n = 5$. There are 616,126 classes of essentially different functions $f(x_1,x_2,x_3,x_4,x_5)$, according to Table 7.1.1–5. Computers are now fast enough that this number is no longer frightening; so the author decided while writing this section to investigate $C(f)$ for all Boolean functions of five variables. Thanks to a bit of good luck, complete results could indeed be obtained, leading to the statistics shown in Table 2.

For this calculation Algorithm L and its variants were modified to deal with class representatives, instead of with the entire set of 2^{31} normal truth tables. The method of exercise 7.2.1.2–20 made it easy to generate all functions of a class, given any one of them, resulting in a thousand-fold speedup. The bottom-up method was enhanced slightly, allowing it to deduce for example that $f(x_1\wedge x_2, x_1\vee x_2, x_3, x_4, x_5)$ has cost $\le r$ if $C(f) = r-2$. After all classes of cost 10 had been found, the top-down and bottom-up methods were able to find chains of length ≤ 11 for all but seven classes of functions. Then the time-consuming part of the computation began, in which approximately 53 million special chains with $n = 5$ and $r = 11$ were generated; every such chain led to $5^{11} = 48{,}828{,}125$ functions, some of which would hopefully fall into the seven remaining mystery classes. But only six of those classes were found to have 11-step solutions. The lone survivor, whose truth table is `169ae443` in hexadecimal notation, is the unique class for which $C(f) = 12$, and it also has $L(f) = 12$.

The resulting constructions of symmetric functions are shown in Fig. 10. Some of them are astonishingly beautiful; some of them are beautifully simple;

monotonic
minimum memory

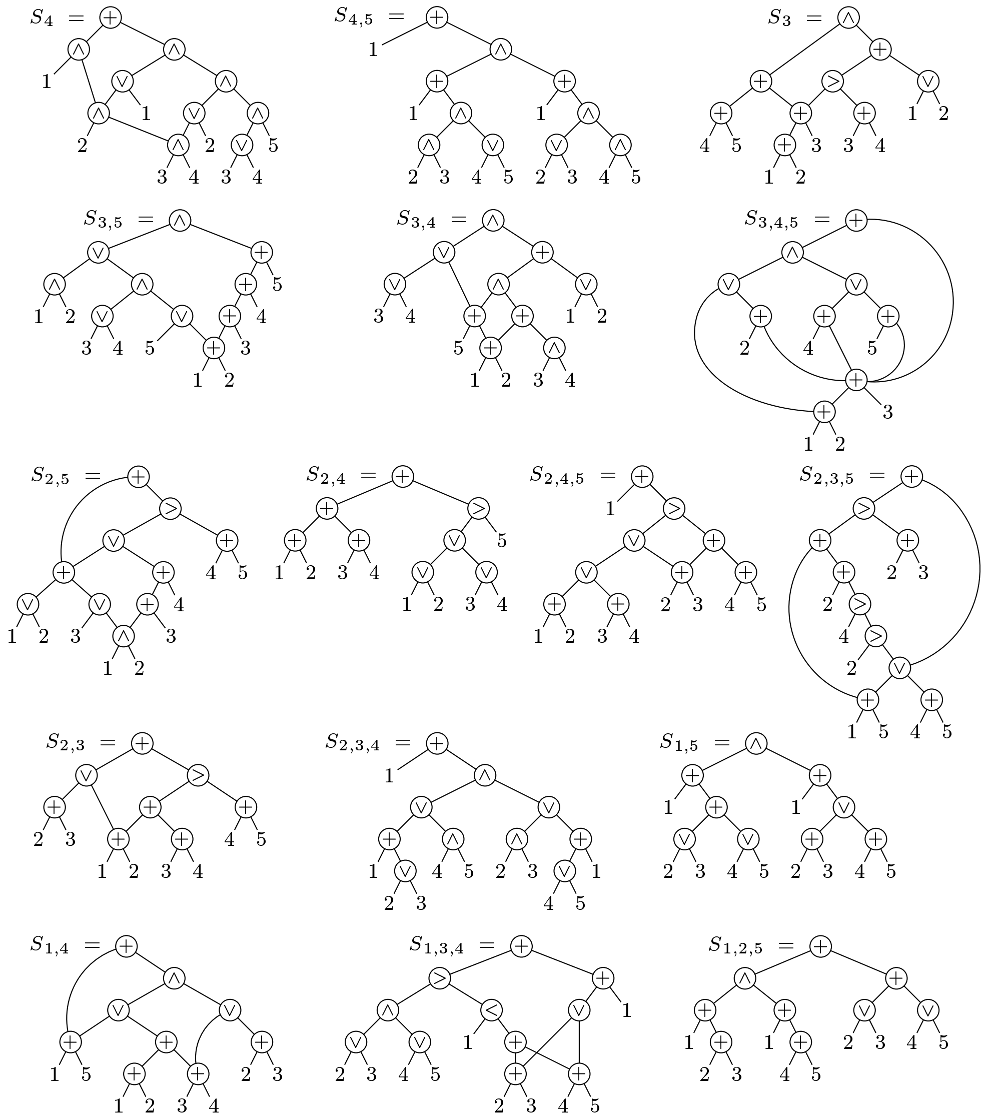


Fig. 10. Boolean chains of minimum cost for symmetric functions of five variables.

and others are simply astonishing. (Look, for example, at the 8-step computation of $S_{2,3}(x_1, x_2, x_3, x_4, x_5)$, or the elegant formula for $S_{2,3,4}$, or the nonmonotonic chains for $S_{4,5}$ and $S_{3,4,5}$.) Incidentally, Table 2 shows that all 5-variable functions have depth ≤ 4, but no attempt to minimize depth has been made in Fig. 10.

It turns out that all of these symmetric functions can be evaluated in minimum memory without increasing the cost. But no simple proof of that fact is known.

multiple outputs– vector engineers full adder half adder binary addition carry bits Red'kin

Multiple outputs. We often want to evaluate several different Boolean functions $f_1(x_1, \ldots, x_n)$, $\ldots$, $f_m(x_1, \ldots, x_n)$ at the same input values $x_1, \ldots, x_n$; in other words, we often want to evaluate a multibit function $y = f(x)$, where $y = f_1 \ldots f_m$ is a binary vector of length m and $x = x_1 \ldots x_n$ is a binary vector of length n. With luck, much of the work involved in the computation of one component value $f_j(x_1, \ldots, x_n)$ can be shared with the operations that are needed to evaluate the other component values $f_k(x_1, \ldots, x_n)$.

Let $C(f) = C(f_1 \ldots f_m)$ be the length of a shortest Boolean chain that computes all of the nontrivial functions f_j. More precisely, the chain $(x_{n+1}, \ldots, x_{n+r})$ should have the property that, for $1 \le j \le m$, either $f_j(x_1, \ldots, x_n) = x_{l(j)}$ or $f_j(x_1, \ldots, x_n) = \bar{x}_{l(j)}$, for some $l(j)$ with $0 \le l(j) \le n+r$, where $x_0 = 0$. Clearly $C(f) \le C(f_1) + \cdots + C(f_m)$, but we might be able to do much better.

For example, suppose we want to compute the functions z_1 and z_0 defined by

$$(z_1 z_0)_2 \;=\; x_1 + x_2 + x_3, \tag{21}$$

the two-bit binary sum of three Boolean variables. We have

$$z_1 \;=\; \langle x_1 x_2 x_3 \rangle \qquad \text{and} \qquad z_0 \;=\; x_1 \oplus x_2 \oplus x_3, \tag{22}$$

so the individual costs are $C(z_1) = 4$ and $C(z_0) = 2$. But it's easy to see that the combined cost $C(z_1 z_0)$ is at most 5, because $x_1 \oplus x_2$ is a suitable first step in the evaluation of each bit z_j:

$$\begin{aligned} &x_4 = x_1 \oplus x_2, \quad z_0 = x_5 = x_3 \oplus x_4; \\ &x_6 = x_3 \wedge x_4, \quad x_7 = x_1 \wedge x_2, \quad z_1 = x_8 = x_6 \vee x_7. \end{aligned} \tag{23}$$

Furthermore, exhaustive calculations show that $C(z_1 z_0) > 4$; hence $C(z_1 z_0) = 5$.

Electrical engineers traditionally call a circuit for (21) a *full adder*, because n such building blocks can be hooked together to add two n-bit numbers. The special case of (22) in which $x_3 = 0$ is also important, although it boils down simply to

$$z_1 \;=\; x_1 \wedge x_2 \qquad \text{and} \qquad z_0 \;=\; x_1 \oplus x_2 \tag{24}$$

and has complexity 2; engineers call it a "half adder" in spite of the fact that the cost of a full adder exceeds the cost of two half adders.

The general problem of radix-2 addition

$$\begin{array}{r} (x_{n-1} \ldots x_1 x_0)_2 \\ (y_{n-1} \ldots y_1 y_0)_2 \\ \hline (z_n\, z_{n-1} \ldots z_1\, z_0)_2 \end{array} \tag{25}$$

is to compute $n+1$ Boolean outputs $z_n \ldots z_1 z_0$ from the $2n$ Boolean inputs $x_{n-1} \ldots x_1 x_0 y_{n-1} \ldots y_1 y_0$; and it is readily solved by the formulas

$$c_{j+1} \;=\; \langle x_j y_j c_j \rangle, \qquad z_j \;=\; x_j \oplus y_j \oplus c_j, \qquad \text{for } 0 \le j < n, \tag{26}$$

where the c_j are "carry bits" and we have $c_0 = 0$, $z_n = c_n$. Therefore we can use a half adder to compute c_1 and z_0, followed by $n-1$ full adders to compute the other c's and z's, accumulating a total cost of $5n - 3$. And in fact N. P. Red'kin [*Problemy Kibernetiki* **38** (1981), 181–216] has proved that $5n-3$ steps

parallel computation
sideways sum
binary recurrence
symmetric functions

are actually necessary, by constructing an elaborate 35-page proof by induction, which concludes with Case 2.2.2.3.1.2.3.2.4.3(!). But the depth of this circuit, $2n-1$, is far too large for practical parallel computation, so a great deal of effort has gone into the task of devising circuits for addition that have depth $O(\log n)$ as well as reasonable cost. (See exercises 41–44.)

Now let's extend (21) and try to compute a general "sideways sum"

$$(z_{\lfloor \lg n \rfloor} \ldots z_1 z_0)_2 \;=\; x_1 + x_2 + \cdots + x_n. \tag{27}$$

If $n = 2k+1$, we can use k full adders to reduce the sum to $(x_1 + \cdots + x_n) \bmod 2$ plus k bits of weight 2, because each full adder decreases the number of weight-1 bits by 2. For example, if $n = 9$ and $k = 4$ the computation is

$$\begin{array}{llll} x_{10} = x_1 \oplus x_2 \oplus x_3, & x_{11} = x_4 \oplus x_5 \oplus x_6, & x_{12} = x_7 \oplus x_8 \oplus x_9, & x_{13} = x_{10} \oplus x_{11} \oplus x_{12}, \\ y_1 = \langle x_1 x_2 x_3 \rangle, & y_2 = \langle x_4 x_5 x_6 \rangle, & y_3 = \langle x_7 x_8 x_9 \rangle, & y_4 = \langle x_{10} x_{11} x_{12} \rangle, \end{array}$$

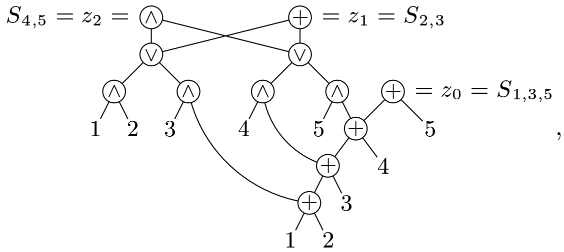
and we have $x_1 + \cdots + x_9 = x_{13} + 2(y_1 + y_2 + y_3 + y_4)$. If $n = 2k$ is even, a similar reduction applies but with a half adder at the end. The bits of weight 2 can then be summed in the same way; so we obtain the recurrence

$$s(n) \;=\; 5\lfloor n/2 \rfloor - 3[n \text{ even}] + s(\lfloor n/2 \rfloor), \qquad s(0) = 0, \tag{28}$$

for the total number of gates needed to compute $z_{\lfloor \lg n \rfloor} \ldots z_1 z_0$. (A closed formula for $s(n)$ appears in exercise 30.) We have $s(n) < 5n$, and the first values

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$s(n) =$	0	2	5	9	12	17	20	26	29	34	37	44	47	52	55	63	66	71	74	81

show that the method is quite efficient even for small n. For example, when $n = 5$ it produces



, (29)

which computes three different symmetric functions $z_2 = S_{4,5}(x_1, \ldots, x_5)$, $z_1 = S_{2,3}(x_1, \ldots, x_5)$, $z_0 = S_{1,3,5}(x_1, \ldots, x_5)$ in just 12 steps. The 10-step computation of $S_{4,5}$ is optimum, according to Fig. 10; of course the 4-step computation of $S_{1,3,5}$ is also optimum. Furthermore, although $C(S_{2,3}) = 8$, the function $S_{2,3}$ is computed here in a clever 10-step way that shares all but one gate with $S_{4,5}$.

Notice that we can now compute *any* symmetric function efficiently, because every symmetric function of $\{x_1, \ldots, x_n\}$ is a Boolean function of $z_{\lfloor \lg n \rfloor} \ldots z_1 z_0$. We know, for example, that any Boolean function of four variables has complexity ≤ 7; therefore any symmetric function $S_{k_1,\ldots,k_t}(x_1, \ldots, x_{15})$ costs at most $s(15) + 7 = 62$. Surprise: The symmetric functions of n variables were among the hardest of all to evaluate, when n was small, but they're among the easiest when $n \ge 10$.

We can also compute *sets* of symmetric functions efficiently. If we want, say, to evaluate all $n+1$ symmetric functions $S_k(x_1,\ldots,x_n)$ for $0 \le k \le n$ with a single Boolean chain, we simply need to evaluate the first $n{+}1$ *minterms* of z_0, z_1, $\ldots$, $z_{\lfloor \lg n \rfloor}$. For example, when $n=5$ the minterms that give us all functions S_k are respectively $S_0 = \bar z_0 \wedge \bar z_1 \wedge \bar z_2$, $S_1 = \bar z_0 \wedge \bar z_1 \wedge z_2$, $\ldots$, $S_5 = z_0 \wedge \bar z_1 \wedge z_2$.

How hard is it to compute all 2^n minterms of n variables? Electrical engineers call this function an n-to-2^n *binary decoder*, because it converts n bits $x_1 \ldots x_n$ into a sequence of 2^n bits $d_0 d_1 \ldots d_{2^n-1}$, exactly one of which is 1. The principle of "divide and conquer" suggests that we first evaluate all minterms on the first $\lceil n/2 \rceil$ variables, as well as all minterms on the last $\lfloor n/2 \rfloor$; then 2^n AND gates will finish the job. The cost of this method is $t(n)$, where

$$t(0) = t(1) = 0; \qquad t(n) = 2^n + t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) \quad \text{for } n \ge 2. \tag{30}$$

So $t(n) = 2^n + O(2^{n/2})$; there's roughly one gate per minterm. (See exercise 32.)

Functions with multiple outputs often help us build larger functions with single outputs. For example, we've seen that the sideways adder (27) allows us to compute symmetric functions; and an n-to-2^n decoder also has many applications, in spite of the fact that 2^n can be huge when n is large. A case in point is the 2^m*-way multiplexer* $M_m(x_1,\ldots,x_m; y_0, y_1, \ldots, y_{2^m-1})$, also known as the m-bit *storage access function*, which has $n = m + 2^m$ inputs and takes the value y_k when $(x_1 \ldots x_m)_2 = k$. By definition we have

$$M_m(x_1,\ldots,x_m; y_0, y_1, \ldots, y_{2^m-1}) \;=\; \bigvee_{k=0}^{2^m-1} (d_k \wedge y_k), \tag{31}$$

where d_k is the kth output of an m-to-2^m binary decoder; thus, by (30), we can evaluate M_m with $2^m + (2^m{-}1) + t(m) = 3n + O(\sqrt{n}\,)$ gates. But exercise 39 shows that we can actually reduce the cost to only $2n + O(\sqrt{n}\,)$. (See also exercise 79.)

Asymptotic facts. When the number of variables is small, our exhaustive-search methods have turned up lots of cases where Boolean functions can be evaluated with stunning efficiency. So it's natural to expect that, when more variables are present, even more opportunities for ingenious evaluations will arise. But the truth is exactly the opposite, at least from a statistical standpoint:

Theorem S. *The cost of almost every Boolean function $f(x_1,\ldots,x_n)$ exceeds $2^n/n$. More precisely, if $c(n,r)$ Boolean functions have complexity $\le r$, we have*

$$(r-1)!\,c(n,r) \;\le\; 2^{2r+1}(n+r-1)^{2r}. \tag{32}$$

Proof. If a function can be computed in $r-1$ steps, it is also computable by an r-step chain. (This statement is obvious when $r=1$; otherwise we can let $x_{n+r} = x_{n+r-1} \wedge x_{n+r-1}$.) We will show that there aren't very many r-step chains, hence we can't compute very many different functions with cost $\le r$.

Let π be a permutation of $\{1,\ldots,n+r\}$ that takes $1 \mapsto 1$, $\ldots$, $n \mapsto n$, and $n{+}r \mapsto n{+}r$; there are $(r{-}1)!$ such permutations. Suppose $(x_{n+1},\ldots,x_{n+r})$ is a

minterms
engineers
binary decoder
divide and conquer
binary recurrence
2^m-way multiplexer
storage access function
asymptotic methods–
lower bounds–

normal
upper bound++
Shannon
Lupanov
prime numbers
truth table

Boolean chain in which each of the intermediate steps $x_{n+1}, \ldots, x_{n+r-1}$ is used in at least one subsequent step. Then the permuted chains defined by the rule

$$x_i \;=\; x_{j'(i)} \circ_i' x_{k'(i)} \;=\; x_{j(i\pi)\pi^-} \circ_{i\pi} x_{k(i\pi)\pi^-}, \qquad \text{for } n < i \le n+r, \tag{33}$$

are distinct for different π. (If π takes $a \mapsto b$, we write $b = a\pi$ and $a = b\pi^-$.) For example, if π takes $5 \mapsto 6 \mapsto 7 \mapsto 8 \mapsto 9 \mapsto 5$, the chain (7) becomes

$$\begin{array}{rl@{\qquad\qquad}rl}
\multicolumn{2}{c}{\text{Original}} & \multicolumn{2}{c}{\text{Permuted}}\\
x_5 &= x_1 \oplus x_3, & x_5 &= x_1 \oplus x_2,\\
x_6 &= x_1 \oplus x_2, & x_6 &= x_3 \oplus x_4,\\
x_7 &= x_3 \oplus x_4, & x_7 &= x_9 \vee x_5,\\
x_8 &= x_5 \vee x_6, & x_8 &= x_5 \oplus x_6,\\
x_9 &= x_6 \oplus x_7, & x_9 &= x_1 \oplus x_3,\\
x_{10} &= x_8 \wedge \bar{x}_9; & x_{10} &= x_7 \wedge \bar{x}_8.
\end{array} \tag{34}$$

Notice that we might have $j'(i) \ge k'(i)$ or $j'(i) > i$ or $k'(i) > i$, contrary to our usual rules. But the permuted chain computes the same function x_{n+r} as before, and it doesn't have any cycles by which an entry is defined indirectly in terms of itself, because the permuted x_i is the original $x_{i\pi}$.

We can restrict consideration to *normal* Boolean chains, as remarked earlier. So the $c(n,r)/2$ normal Boolean functions of cost $\le r$ lead to $(r-1)!\,c(n,r)/2$ different permuted chains, where the operator $\circ_i$ in each step is either $\wedge$, $\vee$, $\overline{\supset}$, or $\oplus$. And there are at most $4^r(n+r-1)^{2r}$ such chains, because there are four choices for $\circ_i$ and $n+r-1$ choices for each of $j(i)$ and $k(i)$, for $n < i \le n+r$. Equation (32) follows; and we obtain the opening statement of the theorem by setting $r = \lfloor 2^n/n \rfloor$. (See exercise 46.) ▮

On the other hand, there's also good news for infinity-minded people: We can actually evaluate every Boolean function of n variables with only slightly more than $2^n/n$ steps of computation, even if we avoid $\oplus$ and $\equiv$, using a technique devised by C. E. Shannon and improved by O. B. Lupanov [*Bell System Tech. J.* **28** (1949), 59–98, Theorem 6; *Isvestiia VUZov, Radiofizika* **1** (1958), 120–140].

In fact, the Shannon–Lupanov approach leads to useful results even when n is small, so let's get acquainted with it by studying a small example. Consider

$$f(x_1,x_2,x_3,x_4,x_5,x_6) \;=\; \bigl[(x_1x_2x_3x_4x_5x_6)_2 \text{ is prime}\bigr], \tag{35}$$

a function that identifies all 6-bit prime numbers. Its truth table has $2^6 = 64$ bits, and we can work with it conveniently by using a 4×16 array to look at those bits instead of confining ourselves to one dimension:

$$\begin{array}{r|cccccccccccccccc|l}
\multicolumn{1}{r}{x_3 =} & 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 & \multicolumn{1}{l}{}\\
\multicolumn{1}{r}{x_4 =} & 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 & \multicolumn{1}{l}{}\\
\multicolumn{1}{r}{x_5 =} & 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 & \multicolumn{1}{l}{}\\
\multicolumn{1}{r}{x_6 =} & 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 & \multicolumn{1}{l}{}\\
\cline{2-17}
x_1x_2 = 00 & 0&0&1&1&0&1&0&1&0&0&0&1&0&1&0&0 & \multirow{2}{*}{$\Big\}$ Group 1}\\
x_1x_2 = 01 & 0&1&0&1&0&0&0&1&0&0&0&0&0&1&0&1 & \\
x_1x_2 = 10 & 0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&1 & \multirow{2}{*}{$\Big\}$ Group 2}\\
x_1x_2 = 11 & 0&0&0&0&0&1&0&0&0&0&0&1&0&1&0&0 & \\
\cline{2-17}
\end{array} \tag{36}$$

The rows have been divided into two groups of two rows each; and each group of rows has 16 columns, which are of four basic types, namely ${0 \atop 0}$, ${0 \atop 1}$, ${1 \atop 0}$, or ${1 \atop 1}$. Thus we see that the function can be expressed as minterms

$$\begin{aligned} f(x_1,\ldots,x_6) \;=\; & \bigl([x_1x_2 \in \{00\}] \quad\;\; \wedge [x_3x_4x_5x_6 \in \{0010, 0101, 1011\}]\bigr) \\ \vee\; & \bigl([x_1x_2 \in \{01\}] \quad\;\; \wedge [x_3x_4x_5x_6 \in \{0001, 1111\}]\bigr) \\ \vee\; & \bigl([x_1x_2 \in \{00, 01\}] \wedge [x_3x_4x_5x_6 \in \{0011, 0111, 1101\}]\bigr) \\ \vee\; & \bigl([x_1x_2 \in \{10\}] \quad\;\; \wedge [x_3x_4x_5x_6 \in \{1001, 1111\}]\bigr) \\ \vee\; & \bigl([x_1x_2 \in \{11\}] \quad\;\; \wedge [x_3x_4x_5x_6 \in \{1101\}]\bigr) \\ \vee\; & \bigl([x_1x_2 \in \{10, 11\}] \wedge [x_3x_4x_5x_6 \in \{0101, 1011\}]\bigr). \end{aligned} \tag{37}$$

(The first line corresponds to group 1, type ${1 \atop 0}$, then comes group 1, type ${0 \atop 1}$, etc.; the last line corresponds to group 2 and type ${1 \atop 1}$.) A function like $\bigl[x_3x_4x_5x_6 \in \{0010, 0101, 1011\}\bigr]$ is the OR of three minterms of $\{x_3, x_4, x_5, x_6\}$.

In general we can view the truth table as a $2^k \times 2^{n-k}$ array, with l groups of rows having either $\lfloor 2^k/l \rfloor$ or $\lceil 2^k/l \rceil$ rows in each group. A group of size m will have columns of 2^m basic types. We form a conjunction $(g_{it}(x_1,\ldots,x_k) \wedge h_{it}(x_{k+1},\ldots,x_n))$ for each group i and each nonzero type t, where g_{it} is the OR of all minterms of $\{x_1,\ldots,x_k\}$ for the rows of the group where t has a 1, while h_{it} is the OR of all minterms of $\{x_{k+1},\ldots,x_n\}$ for the columns having type t in group i. The OR of all these conjunctions $(g_{it} \wedge h_{it})$ gives $f(x_1,\ldots,x_n)$.

Once we've chosen the parameters k and l, with $1 \le k \le n-2$ and $1 \le l \le 2^k$, the computation starts by computing all the minterms of $\{x_1,\ldots,x_k\}$ and all the minterms of $\{x_{k+1},\ldots,x_n\}$, in $t(k) + t(n-k)$ steps (see (30)). Then, for $1 \le i \le l$, we let group i consist of rows for the values of $(x_1,\ldots,x_k)$ such that $(i-1)2^k/l \le (x_1 \ldots x_k)_2 < i2^k/l$; it contains $m_i = \lceil i2^k/l \rceil - \lceil (i-1)2^k/l \rceil$ rows. We form all functions g_{it} for $t \in S_i$, the family of $2^{m_i} - 1$ nonempty subsets of those rows; $2^{m_i} - m_i - 1$ ORs of previously computed minterms will accomplish that task. We also form all functions h_{it} representing columns of nonzero type t; for this purpose we'll need at most 2^{n-k} OR operations in each group i, since we can OR each minterm into the h function of the appropriate type t. Finally we compute $f = \bigvee_{i=1}^{l} \bigvee_{t \in S_i} (g_{it} \wedge h_{it})$; each AND operation is compensated by an unnecessary first OR into h_{it}. So the total cost is at most

$$t(k) + t(n-k) + (l-1) + \sum_{i=1}^{l} \bigl((2^{m_i} - m_i - 1) + 2^{n-k} + (2^{m_i} - 2)\bigr); \tag{38}$$

we want to choose k and l so that this upper bound is minimized. Exercise 52 discusses the best choice when n is small. And when n is large, a good choice yields a provably near-optimum chain, at least for most functions:

Theorem L. *Let $C(n)$ denote the cost of the most expensive Boolean functions of n variables. Then as $n \to \infty$ we have*

$$C(n) \;\ge\; \frac{2^n}{n}\Bigl(1 + \frac{\lg n}{n} + O\Bigl(\frac{1}{n}\Bigr)\Bigr); \tag{39}$$

$$C(n) \;\le\; \frac{2^n}{n}\Bigl(1 + 3\frac{\lg n}{n} + O\Bigl(\frac{1}{n}\Bigr)\Bigr). \tag{40}$$

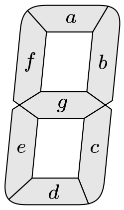
Lupanov
seven-segment display
Wood
Patent

Proof. Exercise 48 shows that the lower bound (39) is a consequence of Theorem S. For the upper bound, we set $k = \lfloor 2 \lg n \rfloor$ and $l = \lceil 2^k/(n - 3 \lg n) \rceil$ in Lupanov's method; see exercise 53. ▌

Synthesizing a good chain. Formula (37) isn't the best way to implement a 6-bit prime detector, but it does suggest a decent strategy. For example, we needn't let variables x_1 and x_2 govern the rows: Exercise 51 shows that a better chain results if the rows are based on $x_5 x_6$ while the columns come from $x_1 x_2 x_3 x_4$, and in general there are many ways to partition a truth table by playing k of the variables against the other $n - k$.

Furthermore, we can improve on (37) by using our complete knowledge of all 4-variable functions; there's no need to evaluate a function like $[x_3 x_4 x_5 x_6 \in \{0010, 0101, 1011\}]$ by first computing the minterms of $\{x_3, x_4, x_5, x_6\}$, if we know the best way to evaluate every such function from scratch. On the other hand, we do need to evaluate several 4-variable functions simultaneously, so the minterm approach might not be such a bad idea after all. Can we really improve on it?

Let's try to find a good way to synthesize a Boolean chain that computes a given set of 4-variable functions. The six functions of $x_3 x_4 x_5 x_6$ in (37) are rather tame (see exercise 54), so we'll learn more by considering a more interesting example chosen from everyday life.



A *seven-segment display* is a now-ubiquitous way to represent a 4-bit number $(x_1 x_2 x_3 x_4)_2$ in terms of seven cleverly positioned segments that are either visible or invisible. The segments are traditionally named (a, b, c, d, e, f, g) as shown; we get a '0' by turning on segments (a, b, c, d, e, f), but a '1' uses only segments (b, c). (Incidentally, the idea for such displays was invented by F. W. Wood, *U.S. Patent 974943* (1910), although Wood's original design used eight segments because he thought that a '4' requires a diagonal stroke.) Seven-segment displays usually support only the decimal digits '0', '1', ..., '9'; but of course a computer scientist's digital watch should display also hexadecimal digits. So we shall design seven-segment logic that displays the sixteen digits

(41)

when given the respective inputs $x_1 x_2 x_3 x_4 = 0000, 0001, 0010, \ldots, 1111$.

In other words, we want to evaluate seven Boolean functions whose truth tables are respectively

$$
\begin{aligned}
a &= 1011\ 0111\ 1110\ 0011,\\
b &= 1111\ 1001\ 1110\ 0100,\\
c &= 1101\ 1111\ 1111\ 0100,\\
d &= 1011\ 0110\ 1101\ 1110,\\
e &= 1010\ 0010\ 1011\ 1111,\\
f &= 1000\ 1111\ 1111\ 0011,\\
g &= 0011\ 1110\ 1111\ 1111.
\end{aligned}
\qquad (42)
$$

footprint
greedy
normal
partial functions–

If we simply wanted to evaluate each function separately, several methods that we've already discussed would tell us how to do it with minimum costs $C(a) = 5$, $C(b) = C(c) = C(d) = 6$, $C(e) = C(f) = 5$, and $C(g) = 4$; the total cost for all seven functions would then be 37. But we want to find a single Boolean chain that contains them all, and the shortest such chain is presumably much more efficient. How can we discover it?

Well, the task of finding a truly optimum chain for $\{a, b, c, d, e, f, g\}$ is probably infeasible from a computational standpoint. But a surprisingly good solution can be found with the help of the "footprint" idea explained earlier. Namely, we know how to compute not only a function's minimum cost, but also the set of all first steps consistent with that minimum cost in a normal chain. Function e, for example, has cost 5, but only if we evaluate it by starting with one of the instructions

$$x_5 = x_1 \oplus x_4 \qquad \text{or} \qquad x_5 = x_2 \wedge \bar{x}_3 \qquad \text{or} \qquad x_5 = x_2 \vee x_3.$$

Fortunately, one of the desirable first steps belongs to four of the seven footprints: Functions c, d, f, and g can all be evaluated optimally by starting with $x_5 = x_2 \oplus x_3$. So that is a natural choice; it essentially saves us three steps, because we know that at most 33 of the original 37 steps will be needed to finish.

Now we can recompute the costs and footprints of all 2^{16} functions, proceeding as before but also initializing the cost of the new function x_5 to zero. The costs of functions c, d, f, and g decrease by 1 as a result, and the footprints change too. For example, function a still has cost 5, but its footprint has increased from $\{x_1 \oplus x_3, x_2 \wedge x_3\}$ to $\{x_1 \oplus x_3, x_1 \wedge x_4, \bar{x}_1 \wedge x_4, x_2 \wedge x_3, \bar{x}_2 \wedge x_4, x_2 \oplus x_4, x_4 \wedge x_5, x_4 \oplus x_5\}$ when the function $x_5 = x_2 \oplus x_3$ is available for free.

In fact, $x_6 = \bar{x}_1 \wedge x_4$ is common to four of the new footprints, so again we have a natural way to proceed. And when everything is recalculated with zero cost given to both x_5 and x_6, the subsequent step $x_7 = x_3 \wedge \bar{x}_6$ turns out to be desirable in five of the newest footprints. Continuing in this "greedy" fashion, we aren't always so lucky, but a remarkable chain of only 22 steps does emerge:

$$\begin{array}{lll}
x_5 = x_2 \oplus x_3, & x_{13} = x_1 \oplus x_7, & \bar{a} = x_{20} = x_{14} \wedge \bar{x}_{19}, \\
x_6 = \bar{x}_1 \wedge x_4, & x_{14} = x_5 \oplus x_6, & \bar{b} = x_{21} = x_7 \oplus x_{12}, \\
x_7 = x_3 \wedge \bar{x}_6, & x_{15} = x_7 \vee x_{12}, & \bar{c} = x_{22} = \bar{x}_8 \wedge x_{15}, \\
x_8 = x_1 \oplus x_2, & x_{16} = x_1 \vee x_5, & \bar{d} = x_{23} = x_9 \wedge \bar{x}_{13}, \\
x_9 = x_4 \oplus x_5, & x_{17} = x_5 \vee x_6, & \bar{e} = x_{24} = x_6 \vee x_{18}, \\
x_{10} = \bar{x}_7 \wedge x_8, & x_{18} = x_9 \wedge x_{10}, & \bar{f} = x_{25} = \bar{x}_8 \wedge x_{17}, \\
x_{11} = x_9 \oplus x_{10}, & x_{19} = x_3 \wedge x_9, & g = x_{26} = x_7 \vee x_{16}. \\
x_{12} = x_5 \wedge x_{11}, & &
\end{array} \tag{43}$$

(This is a *normal* chain, so it contains the normalizations $\{\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{f}, g\}$ instead of $\{a, b, c, d, e, f, g\}$. Simple changes will produce the unnormalized functions without changing the cost.)

Partial functions. In practice the output value of a Boolean function is often specified only at certain inputs $x_1 \ldots x_n$, and the outputs in other cases don't really matter. We might know, for example, that some of the input combinations will never arise. In such cases, we place an asterisk into the corresponding positions of the truth table, instead of specifying 0 or 1 everywhere.

The seven-segment display provides a case in point, because most of its applications involve only the ten binary-coded decimal inputs for which we have $(x_1x_2x_3x_4)_2 \le 9$. We don't care what segments are visible in the other six cases. So the truth tables of (42) actually become

$$\begin{aligned} a &= 1011\ 0111\ 11{*}{*}\ {*}{*}{*}{*},\\ b &= 1111\ 1001\ 11{*}{*}\ {*}{*}{*}{*},\\ c &= 1101\ 1111\ 11{*}{*}\ {*}{*}{*}{*},\\ d &= 1011\ 0110\ 11{*}{*}\ {*}{*}{*}{*},\\ e &= 1010\ 0010\ 10{*}{*}\ {*}{*}{*}{*},\\ f &= 1000\ 111{*}\ 11{*}{*}\ {*}{*}{*}{*},\\ g &= 0011\ 1110\ 11{*}{*}\ {*}{*}{*}{*}. \end{aligned} \tag{44}$$

(Function f here has an asterisk also in position $x_1x_2x_3x_4 = 0111$, because a '7' can be displayed as either or . Both of these styles appeared about equally often in the display units available to the author when this section was written. Truncated variants of the and the were sometimes seen in olden days, but they have thankfully disappeared.)

Asterisks in truth tables are generally known as *don't-cares* — a quaint term that could only have been invented by an electrical engineer. Table 3 shows that the freedom to choose arbitrary outputs is advantageous. For example, there are $\binom{16}{3}2^{13} = 4{,}587{,}520$ truth tables with 3 don't-cares; 69% of them cost 4 or less, even though only 21% of the asterisk-free truth tables permit such economy. On the other hand, don't-cares don't save us as much as we might hope; exercise 63 proves that a random function with, say, 30% don't-cares in its truth table tends to save only about 30% of the cost of a fully specified function.

What is the shortest Boolean chain that evaluates the seven partially specified functions in (44)? Our greedy-footprint method adapts itself readily to the presence of don't-cares, because we can OR together the footprints of all 2^d functions that match a pattern with d asterisks. The initial costs to evaluate each function separately are now reduced to $C(a)=3$, $C(b)=C(c)=2$, $C(d)=5$, $C(e)=2$, $C(f)=3$, $C(g)=4$, totalling just 21 instead of 37. Function g hasn't gotten cheaper, but it does have a larger footprint. Proceeding as before, but taking advantage of the don't-cares, we now can find a suitable chain of length only 13 — a chain with fewer than two operations per output(!):

$$\begin{array}{lll} x_5 = x_1 \oplus x_2, & \bar e = x_{10} = x_4 \vee x_8, & \bar b = x_{15} = x_2 \wedge \bar x_{13},\\ x_6 = x_3 \wedge \bar x_4, & g = x_{11} = x_7 \oplus x_8, & \bar c = x_{16} = \bar x_2 \wedge x_6,\\ x_7 = x_1 \oplus x_3, & \quad x_{12} = x_4 \oplus x_{11}, & \bar f = x_{17} = \bar x_5 \wedge x_9.\\ x_8 = x_2 \wedge \bar x_6, & \bar d = x_{13} = x_{10} \wedge x_{12}, & \\ x_9 = x_3 \vee x_4, & \bar a = x_{14} = \bar x_3 \wedge x_{13}, & \end{array} \tag{45}$$

Tic-tac-toe. Let's turn now to a slightly larger problem, based on a popular children's game. Two players take turns filling the cells of a 3×3 grid. One player writes X's and the other writes O's, continuing until there either are three

truth table
seven-segment display
binary-coded decimal
don't-cares
engineer
greedy-footprint
game–
noughts and crosses, see tic-tac-toe
tic-tac-toe–

cat's game
Knuth
Museum of Science and Industry
Bell Telephone Laboratories
truth table

Table 3

THE NUMBER OF 4-VARIABLE FUNCTIONS WITH d DON'T-CARES AND COST c

	$c=0$	$c=1$	$c=2$	$c=3$	$c=4$	$c=5$	$c=6$	$c=7$
$d=0$	10	60	456	2474	10624	24184	25008	2720
$d=1$	160	960	7296	35040	131904	227296	119072	2560
$d=2$	1200	7200	52736	221840	700512	816448	166144	
$d=3$	5600	33600	228992	831232	2045952	1381952	60192	
$d=4$	18200	108816	666528	2034408	3505344	1118128	3296	
$d=5$	43680	257472	1367776	3351488	3491648	433568	32	
$d=6$	80080	455616	2015072	3648608	1914800	86016		
$d=7$	114400	606944	2115648	2474688	533568	12032		
$d=8$	128660	604756	1528808	960080	71520	896		
$d=9$	114080	440960	707488	197632	4160			
$d=10$	78960	224144	189248	20160				
$d=11$	41440	72064	25472	800				
$d=12$	15480	12360	1280					
$d=13$	3680	800						
$d=14$	480							
$d=15$	32							
$d=16$	1							

X's or three O's in a straight line (in which case that player wins) or all nine cells are filled without a winner (in which case it's a "cat's game" or tie). For example, the game might proceed thus:


; (46)

X has won. Our goal is to design a machine that plays tic-tac-toe optimally — making a winning move from each position in which a forced victory is possible, and never making a losing move from a position in which defeat is avoidable.

More precisely, we will set things up so that there are 18 Boolean variables $x_1, \ldots, x_9, o_1, \ldots, o_9$, which govern lamps to illuminate cells of the current position. The cells are numbered $\begin{smallmatrix}1&2&3\\4&5&6\\7&8&9\end{smallmatrix}$ as on a telephone dial. Cell j displays an X if $x_j = 1$, an O if $o_j = 1$, or remains blank if $x_j = o_j = 0$.* We never have $x_j = o_j = 1$, because that would display '$\otimes$'. We shall assume that the variables $x_1 \ldots x_9 o_1 \ldots o_9$ have been set to indicate a legal position in which nobody has won; the computer plays the X's, and it is the computer's turn to move. For this purpose we want to define nine functions $y_1, \ldots, y_9$, where y_j means "change x_j from 0 to 1." If the current position is a cat's game, we should make $y_1 = \cdots = y_9 = 0$; otherwise exactly one y_j should be equal to 1, and of course the output value $y_j = 1$ should occur only if $x_j = o_j = 0$.

With 18 variables, each of our nine functions y_j will have a truth table of size $2^{18} = 262{,}144$. It turns out that only 4520 legal inputs $x_1 \ldots x_9 o_1 \ldots o_9$ are

* This setup is based on an exhibit from the early 1950s at the Museum of Science and Industry in Chicago, where the author was first introduced to the magic of switching circuits. The machine in Chicago, designed circa 1940 by W. Keister at Bell Telephone Laboratories, allowed me to go first; yet I soon discovered that there was no way to defeat it. Therefore I decided to move as stupidly as possible, hoping that the designer had not anticipated such bizarre behavior. In fact I allowed the machine to reach a position where it had two winning moves; and it seized *both* of them! Moving twice is of course a flagrant violation of the rules, so I had won a moral victory even though the machine announced that I had lost.

Babbage
symmetric functions

> *I commenced an examination of a game called "tit-tat-to" ...*
> *to ascertain what number of combinations were required*
> *for all the possible variety of moves and situations.*
> *I found this to be comparatively insignificant.*
> *... A difficulty, however, arose of a novel kind.*
> *When the automaton had to move, it might occur that there were*
> *two different moves, each equally conducive to his winning the game.*
> *... Unless, also, some provision were made,*
> *the machine would attempt two contradictory motions.*
>
> — CHARLES BABBAGE, *Passages from the Life of a Philosopher* (1864)

possible, so those truth tables are 98.3% filled with don't-cares. Still, 4520 is uncomfortably large if we hope to design and understand a Boolean chain that makes sense intuitively. Section 7.1.4 will discuss alternative ways to represent Boolean functions, by which it is often possible to deal with hundreds of variables even though the associated truth tables are impossibly large.

Most functions of 18 variables require more than $2^{18}/18$ gates, but let's hope we can do better. Indeed, a plausible strategy for making suitable moves in tic-tac-toe suggests itself immediately, in terms of several conditions that aren't hard to recognize:

w_j, an X in cell j will win, completing a line of X's;
b_j, an O in cell j would lose, completing a line of O's;
f_j, an X in cell j will give X two ways to win;
d_j, an O in cell j would give O two ways to win.

For example, X's move to the center in (46) was needed to block O, so it was of type b_5; fortunately it was also of type f_5, forcing a win on the next move.

Let $L = \{\{1,2,3\},\{4,5,6\},\{7,8,9\},\{1,4,7\},\{2,5,8\},\{3,6,9\},\{1,5,9\},\{3,5,7\}\}$ be the set of winning lines. Then we have

$$m_j = \bar{x}_j \wedge \bar{o}_j; \qquad \text{[moving in cell } j \text{ is legal]} \quad (47)$$
$$w_j = m_j \wedge \textstyle\bigvee_{\{i,j,k\}\in L}(x_i \wedge x_k); \qquad \text{[moving in cell } j \text{ wins]} \quad (48)$$
$$b_j = m_j \wedge \textstyle\bigvee_{\{i,j,k\}\in L}(o_i \wedge o_k); \qquad \text{[moving in cell } j \text{ blocks]} \quad (49)$$
$$f_j = m_j \wedge S_2\bigl(\{\alpha_{ik} \mid \{i,j,k\}\in L\}\bigr); \qquad \text{[moving in cell } j \text{ forks]} \quad (50)$$
$$d_j = m_j \wedge S_2\bigl(\{\beta_{ik} \mid \{i,j,k\}\in L\}\bigr); \qquad \text{[moving in cell } j \text{ defends]} \quad (51)$$

here α_{ik} and β_{ik} denote a single X or O together with a blank, namely

$$\alpha_{ik} = (x_i \wedge m_k) \vee (m_i \wedge x_k), \qquad \beta_{ik} = (o_i \wedge m_k) \vee (m_i \wedge o_k). \quad (52)$$

For example, $b_1 = m_1 \wedge \bigl((o_2 \wedge o_3) \vee (o_4 \wedge o_7) \vee (o_5 \wedge o_9)\bigr)$; $f_2 = m_2 \wedge S_2(\alpha_{13}, \alpha_{58}) = m_2 \wedge \alpha_{13} \wedge \alpha_{58}$; $d_5 = m_5 \wedge S_2(\beta_{19}, \beta_{28}, \beta_{37}, \beta_{46})$.

With these definitions we might try rank-ordering our moves thus:

$$\{w_1,\ldots,w_9\} > \{b_1,\ldots,b_9\} > \{f_1,\ldots,f_9\} > \{d_1,\ldots,d_9\} > \{m_1,\ldots,m_9\}. \quad (53)$$

"Win if you can; otherwise block if you can; otherwise fork if you can; otherwise defend if you can; otherwise make a legal move." Furthermore, when choosing

cat's game
decomposition of functions–
disjoint decomposition
Ashenhurst

between legal moves it seems sensible to use the ordering

$$m_5 > m_1 > m_3 > m_9 > m_7 > m_2 > m_6 > m_8 > m_4, \qquad (54)$$

because 5, the middle cell, occurs in four winning lines, while a corner move to 1, 3, 9, or 7 occurs in three, and a side cell 2, 6, 8, or 4 occurs in only two. We might as well adopt this ordering of subscripts within all five groups of moves $\{w_j\}$, $\{b_j\}$, $\{f_j\}$, $\{d_j\}$, and $\{m_j\}$ in (53).

To ensure that at most one move is chosen, we define w'_j, b'_j, f'_j, d'_j, m'_j to mean "a prior choice is better." Thus, $w'_5 = 0$, $w'_1 = w_5$, $w'_3 = w_1 \vee w'_1$, …, $w'_4 = w_8 \vee w'_8$, $b'_5 = w_4 \vee w'_4$, $b'_1 = b_5 \vee b'_5$, …, $m'_4 = m_8 \vee m'_8$. Then we can complete the definition of a tic-tac-toe automaton by letting

$$y_j = (w_j \wedge \bar{w}'_j) \vee (b_j \wedge \bar{b}'_j) \vee (f_j \wedge \bar{f}'_j) \vee (d_j \wedge \bar{d}'_j) \vee (m_j \wedge \bar{m}'_j), \quad \text{for } 1 \le j \le 9. \qquad (55)$$

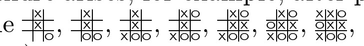
So we've constructed 9 gates for the m's, 48 for the w's, 48 for the b's, 144 for the α's and β's, 35 for the f's (with the help of Fig. 9), 35 for the d's, 43 for the primed variables, and 80 for the y's. Furthermore we can use our knowledge of partial 4-variable functions to reduce the six operations in (52) to only four,

$$\alpha_{ik} = (x_i \oplus x_k) \vee \overline{(o_i \oplus o_k)}, \qquad \beta_{ik} = \overline{(x_i \oplus x_k)} \vee (o_i \oplus o_k). \qquad (56)$$

This trick saves 48 gates; so our design has cost 396 gates altogether.

The strategy for tic-tac-toe in (47)–(56) works fine in most cases, but it also has some glaring glitches. For example, it loses ignominiously in the game

 ; (57)

the second × move is d_3, defending against a fork by ○, yet it actually forces ○ to fork in the opposite corner! Another failure arises, for example, after position , when move m_5 leads to the cat's game , instead of to the victory for × that appeared in (46). Exercise 65 patches things up and obtains a fully correct Boolean tic-tac-toe player that needs just 445 gates.

***Functional decomposition.** If the function $f(x_1, \ldots, x_n)$ can be written in the form $g(x_1, \ldots, x_k, h(x_{k+1}, \ldots, x_n))$, it's usually a good idea to evaluate $y = h(x_{k+1}, \ldots, x_n)$ first and then to compute $g(x_1, \ldots, x_k, y)$. Robert L. Ashenhurst inaugurated the study of such decompositions in 1952 [see *Annals Computation Lab. Harvard University* **29** (1957), 74–116], and observed that there's an easy way to recognize when f has this special property: If we write the truth table for f in a $2^k \times 2^{n-k}$ array as in (36), with rows for each setting of $x_1 \ldots x_k$ and columns for each setting of $x_{k+1} \ldots x_n$, then the desired subfunctions g and h exist if and only if the columns of this array have at most two different values. For example, the truth table for the function $\langle x_1 x_2 \langle x_3 x_4 x_5 \rangle\rangle$ is

0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
1	1	1	1	1	1	1	1

when expressed in this two-dimensional form. One type of column corresponds to the case $h(x_{k+1},\ldots,x_n)=0$; the other corresponds to $h(x_{k+1},\ldots,x_n)=1$.

In general the variables $X=\{x_1,\ldots,x_n\}$ might be partitioned into any two disjoint subsets $Y=\{y_1,\ldots,y_k\}$ and $Z=\{z_1,\ldots,z_{n-k}\}$, and we might have $f(x)=g(y,h(z))$. We could test for a (Y,Z) decomposition by looking at the columns of the $2^k\times 2^{n-k}$ truth table whose rows correspond to values of y. But there are 2^n such ways to partition X; and all of them are potential winners, except for trivial cases when $|Y|=0$ or $|Z|\le 1$. How can we avoid examining such a humungous number of possibilities?

A practical way to proceed was discovered by V. Y.-S. Shen, A. C. McKellar, and P. Weiner [*IEEE Transactions* **C-20** (1971), 304–309], whose method usually needs only $O(n^2)$ steps to identify any potentially useful partition (Y,Z) that may exist. The basic idea is simple: Suppose $x_i\in Z$, $x_j\in Z$, and $x_m\in Y$. Define eight binary vectors δ_l for $l=(l_1l_2l_3)_2$, where δ_l has (l_1,l_2,l_3) respectively in components (i,j,m), and zeros elsewhere. Consider any randomly chosen vector $x=x_1\ldots x_n$, and evaluate $f_l=f(x\oplus\delta_l)$ for $0\le l\le 7$. Then the four pairs

$$\binom{f_0}{f_1}\qquad\binom{f_2}{f_3}\qquad\binom{f_4}{f_5}\qquad\binom{f_6}{f_7}\tag{58}$$

will appear in a 2×4 submatrix of the $2^k\times 2^{n-k}$ truth table. So a decomposition is impossible if these pairs are distinct, or if they contain three different values.

Let's call the pairs "good" if they're all equal, or if they have only two different values. Otherwise they're "bad." If f has essentially random behavior, we'll soon find bad pairs if we do this experiment with several different randomly chosen vectors x, because only 88 of the 256 possibilities for $f_0f_1\ldots f_7$ correspond to a good set of pairs; the probability of finding good pairs ten times in a row is only $(\frac{88}{256})^{10}\approx .00002$. And when we do discover bad pairs, we can conclude that

$$x_i\in Z\quad\text{and}\quad x_j\in Z\implies x_m\in Z,\tag{59}$$

because the alternative $x_m\in Y$ is impossible.

Suppose, for example, that $n=9$ and that f is the function whose truth table $11001001000011\ldots00101$ consists of the 512 most significant bits of π, in binary notation. (This is the "more-or-less random function" that we studied for $n=4$ in (5) and (6) above.) Bad pairs for this π function are quickly found in each of the cases (i,j,m) for which $m\ne i<j\ne m$. Indeed, in the author's experiments, 170 of those 252 cases were decided immediately; the average number of random x vectors per case was only 1.52; and only one case needed as many as eight x's before bad pairs appeared. Thus (59) holds for all relevant (i,j,m), and the function is clearly indecomposable. In fact, exercise 73 points out that we needn't make 252 tests to establish the indecomposability of this π function; only $\binom{n}{2}=36$ of them would have been sufficient.

Turning to a less random function, let $f(x_1,\ldots,x_9)=(\det X)\bmod 2$, where

$$X=\begin{pmatrix}x_1&x_2&x_3\\x_4&x_5&x_6\\x_7&x_8&x_9\end{pmatrix}.\tag{60}$$

Shen
McKellar
Weiner
randomization
good
bad
π
π function

Horn clauses
permanent
matching

This function does not satisfy condition (59) when $i = 1$, $j = 2$, and $m = 3$, because there are no bad pairs in that case. But it does satisfy (59) for $4 \le m \le 9$ when $\{i,j\} = \{1,2\}$. We can denote this behavior by the convenient abbreviation '12⇒456789'; the full set of implications, for all pairs $\{i,j\}$, is

12⇒456789	18⇒34569	27⇒34569	37⇒24568	48⇒12369	67⇒12358
13⇒456789	19⇒24568	28⇒134679	38⇒14567	49⇒12358	68⇒12347
14⇒235689	23⇒456789	29⇒14567	39⇒124578	56⇒123789	69⇒124578
15⇒36789	24⇒36789	34⇒25789	45⇒123789	57⇒12369	78⇒123456
16⇒25789	25⇒134679	35⇒14789	46⇒123789	58⇒134679	79⇒123456
17⇒235689	26⇒14789	36⇒124578	47⇒235689	59⇒12347	89⇒123456

(see exercise 69). Bad pairs are a little more difficult to find when we probe this function at random: The average number of x's needed in the author's experiments rose to about 3.6, when bad pairs did exist. And of course there was a need to limit the testing, by choosing a tolerance threshold t and then giving up when t consecutive trials failed to find any bad pairs. Choosing $t = 10$ would have found all but 8 of the 198 implications listed above.

Implications like (59) are Horn clauses, and we know from Section 7.1.1 that it's easy to make further deductions from Horn clauses. Indeed, the method of exercise 74 will deduce that the only possible partition with $|Z| > 1$ is the trivial one ($Y = \emptyset$, $Z = \{x_1, \ldots, x_9\}$), after looking at fewer than 50 cases (i,j,m).

Similar results occur when $f(x_1, \ldots, x_9) = [\operatorname{per} X > 0]$, where per denotes the *permanent* function. (In this case f tells us if there is a matching in the bipartite subgraph of $K_{3,3}$ whose edges are specified by the variables $x_1 \ldots x_9$.) Now there are just 180 implications,

12⇒456789	18⇒3459	27⇒3459	37⇒2468	48⇒1269	67⇒1358
13⇒456789	19⇒2468	28⇒134679	38⇒1567	49⇒1358	68⇒2347
14⇒235689	23⇒456789	29⇒1567	39⇒124578	56⇒123789	69⇒124578
15⇒3678	24⇒3678	34⇒2579	45⇒123789	57⇒1269	78⇒123456
16⇒2579	25⇒134679	35⇒1489	46⇒123789	58⇒134679	79⇒123456
17⇒235689	26⇒1489	36⇒124578	47⇒235689	59⇒2347	89⇒123456,

only 122 of which would have been discovered with $t = 10$ as the cutoff threshold. (The best choice of t is not clear; perhaps it should vary dynamically.) Still, those 122 Horn clauses were more than enough to establish indecomposability.

What about a decomposable function? With $f = \langle x_2x_3x_6x_9\langle x_1x_4x_5x_7x_8\rangle\rangle$ we get $i \wedge j \Rightarrow m$ for all $m \notin \{i,j\}$, except when $\{i,j\} \subseteq \{1,4,5,7,8\}$; in the latter case, m must also belong to $\{1,4,5,7,8\}$. Although only 185 of these 212 implications were discovered with tolerance $t = 10$, the partition $Y = \{x_2, x_3, x_6, x_9\}$, $Z = \{x_1, x_4, x_5, x_7, x_8\}$ emerged quickly as a strong possibility.

Whenever a potential decomposition is supported by the evidence, we need to verify that the corresponding $2^k \times 2^{n-k}$ truth table does indeed have only one or two distinct columns. But we're happy to spend 2^n units of time on that verification, because we've greatly simplified the evaluation of f.

The comparison function $f = \bigl[(x_1x_2x_3x_4)_2 \geq (x_5x_6x_7x_8)_2 + x_9\bigr]$ is another interesting case. Its 184 potentially deducible implications are

12⇒3456789	18⇒2345679	27⇒34689	37⇒489	48⇒9	67⇒23489
13⇒2456789	19⇒2345678	28⇒34679	38⇒479	49⇒8	68⇒23479
14⇒2356789	23⇒46789	29⇒34678	39⇒478	56⇒1234789	69⇒23478
15⇒2346789	24⇒36789	34⇒789	45⇒1236789	57⇒1234689	78⇒349
16⇒2345789	25⇒1346789	35⇒1246789	46⇒23789	58⇒1234679	79⇒348
17⇒2345689	26⇒34789	36⇒24789	47⇒389	59⇒1234678	89⇒4,

and 145 of them were found when $t = 10$. Three decompositions reveal themselves in this case, having $Z = \{x_4, x_8, x_9\}$, $Z = \{x_3, x_4, x_7, x_8, x_9\}$, and $Z = \{x_2, x_3, x_4, x_6, x_7, x_8, x_9\}$, respectively. Ashenhurst proved that we can reduce f immediately as soon as we find a nontrivial decomposition; the other decompositions will show up later, when we try to reduce the simpler functions g and h.

comparison function
Ashenhurst
decomposition of partial functions–
don't-cares
incompatible
bipartite
generalized consensus
Hight
Boros
Gurvich
Hammer
Ibaraki
Kogan
Shen
McKellar
Weiner
bad
Ashenhurst

***Decomposition of partial functions.** When the function f is only partially specified, a decomposition with partition (Y, Z) hinges on being able to assign values to the don't-cares so that at most two different columns appear in the corresponding $2^k \times 2^{n-k}$ truth table.

Two vectors $u_1 \ldots u_m$ and $v_1 \ldots v_m$ consisting of 0s, 1s, and $*$s are said to be *incompatible* if either $u_j = 0$ and $v_j = 1$ or $u_j = 1$ and $v_j = 0$, for some j — equivalently, if the subcubes of the m-cube specified by u and v have no points in common. Consider the graph whose vertices are the columns of a truth table with don't-cares, where $u \longrightarrow v$ if and only if u and v are incompatible. We can assign values to the $*$s to achieve at most two distinct columns if and only if this graph is *bipartite*. For if $u_1, \ldots, u_l$ are mutually compatible, their generalized consensus $u_1 \sqcup \cdots \sqcup u_l$, defined in exercise 7.1.1–32, is compatible with all of them. [See S. L. Hight, *IEEE Trans.* **C-22** (1973), 103–110; E. Boros, V. Gurvich, P. L. Hammer, T. Ibaraki, and A. Kogan, *Discrete Applied Math.* **62** (1995), 51–75.] Since a graph is bipartite if and only if it contains no odd cycles, we can easily test this condition with a depth-first search (see Section 7.4.1).

Consequently the method of Shen, McKellar, and Weiner works also when don't-cares are present: The four pairs in (58) are considered bad if and only if three of them are mutually incompatible. We can operate almost as before, although bad pairs will naturally be harder to find when there are lots of $*$s (see exercise 72). However, Ashenhurst's theorem no longer applies. When several decompositions exist, they all should be explored further, because they might use different settings of the don't-cares, and some might be better than the others.

Although most functions $f(x)$ have no simple decomposition $g(y, h(z))$, we needn't give up hope too quickly, because other forms like $g(y, h_1(z), h_2(z))$ might well lead to an efficient chain. If, for example, f is symmetric in three of its variables $\{z_1, z_2, z_3\}$, we can always write $f(x) = g\bigl(y, S_{1,2}(z_1, z_2, z_3), S_{1,3}(z_1, z_2, z_3)\bigr)$, since $S_{1,2}(z_1, z_2, z_3)$ and $S_{1,3}(z_1, z_2, z_3)$ characterize the value of $z_1 + z_2 + z_3$. (Notice that just four steps will suffice to compute both $S_{1,2}$ and $S_{1,3}$.)

In general, as observed by H. A. Curtis [*JACM* **8** (1961), 484–496], $f(x)$ can be expressed in the form $g(y, h_1(z), \ldots, h_r(z))$ if and only if the $2^k \times 2^{n-k}$ truth table corresponding to Y and Z has at most 2^r different columns. And when don't-cares are present, the same result holds if and only if the incompatibility graph for Y and Z can be colored with at most 2^r colors.

For example, the function $f(x) = (\det X) \bmod 2$ considered above turns out to have eight distinct columns when $Z = \{x_4, x_5, x_6, x_7, x_8, x_9\}$; that's a surprisingly small number, considering that the truth table has 8 rows and 64 columns. From this fact we might be led to discover how to expand a determinant by cofactors of the first row,

$$f(x) \;=\; x_1 \wedge h_1(x_4, \ldots, x_9) \;\oplus\; x_2 \wedge h_2(x_4, \ldots, x_9) \;\oplus\; x_3 \wedge h_3(x_4, \ldots, x_9),$$

if we didn't already know such a rule.

When there are $d \le 2^r$ different columns, we can think of $f(x)$ as a function of y and $h(z)$, where h takes each binary vector $z_1 \ldots z_{n-k}$ into one of the values $\{0, 1, \ldots, d-1\}$. Thus $(h_1, \ldots, h_r)$ is essentially an encoding of the different column types, and we hope to find very simple functions h_1, …, h_r that provide such an encoding. Moreover, if d is strictly less than 2^r, the function $g(y, h_1, \ldots, h_r)$ will have many don't-cares that may well decrease its cost.

The distinct columns might also suggest a function g for which the h's have don't-cares. For example, we can use $g(y_1, y_2, h_1, h_2) = (y_1 \oplus (h_1 \wedge y_2)) \wedge h_2$ when all columns are either $(0,0,0,0)^T$ or $(0,0,1,1)^T$ or $(0,1,1,0)^T$; then the value of $h_1(z)$ is arbitrary when z corresponds to an all-zero column. H. A. Curtis has explained how to exploit this idea when $|Y| = 1$ and $|Z| = n - 1$ [see *IEEE Transactions* **C-25** (1976), 1033–1044].

For a comprehensive discussion of decomposition techniques, see Richard M. Karp, *J. Society for Industrial and Applied Math.* **11** (1963), 291–335.

Larger values of n. We've been considering only rather tiny examples of Boolean functions. Theorem S tells us that large, random examples are inherently difficult; but practical examples might well be highly nonrandom. So it makes sense to search for simplifications using heuristic methods.

When n grows, the best ways currently known for dealing with Boolean functions generally start with a Boolean chain — not with a huge truth table — and they try to improve that chain via "local changes." The chain can be specified by a set of equations. Then, if an intermediate result is used in comparatively few subsequent steps, we can try to eliminate it, temporarily making those subsequent steps into functions of three variables, and reformulating those functions in order to make a better chain when possible.

For example, suppose the gate $x_i = x_j \circ x_k$ is used only once, in the gate $x_l = x_i \mathbin{\square} x_m$, so that $x_l = (x_j \circ x_k) \mathbin{\square} x_m$. Other gates might already exist, by which we have computed other functions of x_j, x_k, and x_m; and the definitions of x_j, x_k, and x_m may imply that some of the joint values of (x_j, x_k, x_m) are impossible. Thus we might be able to compute x_l from other gates by doing just one further operation. For example, if $x_i = x_j \wedge x_k$ and $x_l = x_i \vee x_m$, and if the values $x_j \vee x_m$ and $x_k \vee x_m$ appear elsewhere in the chain, we can set $x_l = (x_j \vee x_m) \wedge (x_k \vee x_m)$; this eliminates x_i and reduces the cost by 1. Or if,

coloring of graphs
determinant
Curtis
Karp
functions of many variables–local changes

say, $x_j \wedge (x_k \oplus x_m)$ appears elsewhere and we know that $x_j x_k x_m \neq 101$, we can set $x_l = x_m \oplus (x_j \wedge (x_k \oplus x_m))$.

If x_i is used only in x_l and x_l is used only in x_p, then gate x_p depends on four variables, and we might be able to reduce the cost by using our total knowledge of four-variable functions, obtaining x_p in a better way while eliminating x_i and x_l. Similarly, if x_i appears only in x_l and x_p, we can eliminate x_i if we find a better way to evaluate two different functions of four variables, possibly with don't-cares and with other functions of those four variables available for free. Again, we know how to solve such problems, using the footprint method discussed above.

When no local changes are able to decrease the cost, we can also try local changes that preserve or even increase the cost, in order to discover different kinds of chains that might simplify in other ways. We shall discuss such local search methods extensively in Section 7.10.

Excellent surveys of techniques for Boolean optimization, which electrical engineers call the problem of "multilevel logic synthesis," have been published by R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, *Proceedings of the IEEE* **78** (1990), 264–300, and in the book *Synthesis and Optimization of Digital Circuits* by G. De Micheli (McGraw–Hill, 1994).

Lower bounds. Theorem S tells us that nearly every Boolean function of $n \geq 12$ variables is hard to evaluate, requiring a chain whose length exceeds $2^n/n$. Yet modern computers, which are built from logic circuits involving electric signals that represent thousands of Boolean variables, happily evaluate zillions of Boolean functions every microsecond. Evidently there are plenty of important functions that can be evaluated quickly, in spite of Theorem S. Indeed, the proof of that theorem was indirect; we simply counted the cases of low cost, so we learned absolutely nothing about any particular examples that might arise in practice. When we want to compute a given function and we can only think of a laborious way to do the job, how can we be sure that there's no tricky shortcut?

The answer to that question is almost scandalous: After decades of concentrated research, computer scientists have been unable to find *any* explicit family of functions $f(x_1, \ldots, x_n)$ whose cost is inherently nonlinear, as n increases. The true behavior is $2^n/n$, but no lower bound as strong as $n \log\log\log n$ has yet been proved! Of course we could rig up artificial examples, such as "the lexicographically smallest truth table of length 2^n that isn't achievable by any Boolean chain of length $\lfloor 2^n/n \rfloor - 1$"; but such functions are surely not explicit. The truth table of an explicit function $f(x_1, \ldots, x_n)$ should be computable in at most, say, 2^{cn} units of time for some constant c; that is, the time needed to specify all of the function values should be polynomial in the length of the truth table. Under those ground rules, no family of single-output functions is currently known to have a combinational complexity that exceeds $3n + O(1)$ as $n \to \infty$. [See N. Blum, *Theoretical Computer Science* **28** (1984), 337–345.]

The picture is not totally bleak, because several interesting *linear* lower bounds have been proved for functions of practical importance. A basic way to obtain such results was introduced by N. P. Red'kin in 1970: Suppose we have

four-variable functions
footprint
electrical engineers
multilevel logic synthesis
Brayton
Hachtel
Sangiovanni-Vincentelli
De Micheli
lower bounds on combinational complexity–
Blum
Red'kin

canalizing
adjacency matrix
Stockmeyer
Meyer
quantified formulas
logic

an optimum chain of cost r for $f(x_1,\ldots,x_n)$. By setting $x_n \leftarrow 0$ or $x_n \leftarrow 1$, we obtain reduced chains for the functions $g(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},0)$ and $h(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},1)$, having cost $r-u$ if x_n was used as an input to u different gates. Moreover, if x_n is used in a "canalizing" gate $x_i = x_n \circ x_k$, where the operator $\circ$ is neither $\oplus$ nor $\equiv$, some setting of x_n will force x_i to be constant, thereby further reducing the chain for g or h. Lower bounds on g and/or h therefore lead to a lower bound on f. (See exercises 77–81.)

But where are the proofs of nonlinear lower bounds? Almost every problem with a yes-no answer can be formulated as a Boolean function, so there's no shortage of explicit functions that we don't know how to evaluate in linear time, or even in polynomial time. For example, any directed graph G with vertices $\{v_1,\ldots,v_m\}$ can be represented by its adjacency matrix X, where $x_{ij} = [v_i \to v_j]$; then

$$f(x_{12},\ldots,x_{1m},\ldots,x_{m1},\ldots,x_{m(m-1)}) \;=\; [G \text{ has a Hamiltonian path}] \tag{61}$$

is a Boolean function of $n = m(m-1)$ variables. We would dearly love to be able to evaluate this function in, say, n^4 steps. We do know how to compute the truth table for f in $O(m!\,2^n) = 2^{n+O(\sqrt{n}\,\log n)}$ steps, since only $m!$ potential Hamiltonian paths exist; thus f is indeed "explicit." But nobody knows how to evaluate f in polynomial time, or how to prove that there isn't a $4n$-step chain.

For all we know, short Boolean chains for f might exist, for each n. After all, Figs. 9 and 10 reveal the existence of fiendishly clever chains even in the cases of 4 and 5 variables. Efficient chains for all of the larger problems that we ever will need to solve might well be "out there" — yet totally beyond our grasp, because we don't have time to find them. Even if an omniscient being revealed the simple chains to us, we might find them incomprehensible, because the shortest proof of their correctness might be longer than the number of cells in our brains.

Theorem S rules out such a scenario for most Boolean functions. But fewer than 2^{100} Boolean functions will ever be of practical importance in the entire history of the world, and Theorem S tells us zilch about them.

In 1974, Larry Stockmeyer and Albert Meyer were, however, able to construct a Boolean function f whose complexity is provably huge. Their f isn't "explicit," in the precise sense described above, but it isn't artificial either; it arises naturally in mathematical logic. Consider symbolic statements such as

$$\mathtt{048+1015\ne 1063;} \tag{62}$$
$$\forall\mathtt{m}\exists\mathtt{n(m<n+1);} \tag{63}$$
$$\forall\mathtt{n}\exists\mathtt{m(m+1<n);} \tag{64}$$
$$\forall\mathtt{a}\forall\mathtt{b(b\ge a+2}\Rightarrow\exists\mathtt{ab(a<ab}\land\mathtt{ab<b));} \tag{65}$$
$$\forall\mathtt{A}\forall\mathtt{B(A\equiv B}\Leftrightarrow\neg\exists\mathtt{n(n\in A}\land\mathtt{n\notin B}\lor\mathtt{n\in B}\land\mathtt{n\notin A));} \tag{66}$$
$$\forall\mathtt{A(}\exists\mathtt{n(n\in A)}\Rightarrow\exists\mathtt{m(m\in A}\land\forall\mathtt{n(n\in A}\Rightarrow\mathtt{m\le n)));} \tag{67}$$
$$\forall\mathtt{A(}\exists\mathtt{n(n\in A)}\Rightarrow\exists\mathtt{m(m\in A}\land\forall\mathtt{n(n\in A}\Rightarrow\mathtt{m\ge n)));} \tag{68}$$
$$\exists\mathtt{P}\forall\mathtt{a((a\in P}\Leftrightarrow\mathtt{a+3\notin P)}\Leftrightarrow\mathtt{a<1000);} \tag{69}$$
$$\forall\mathtt{A}\forall\mathtt{B(}\forall\mathtt{C}\forall\mathtt{c(C\equiv A}\land\mathtt{c=1}\lor\mathtt{C\equiv B}\land\mathtt{c=0}\Rightarrow\mathtt{(}\forall\mathtt{n(n\in C}\Leftrightarrow\mathtt{n+1\in C)}\Leftrightarrow\mathtt{c=1))}\Rightarrow\neg\mathtt{A\equiv B).} \tag{70}$$

Stockmeyer and Meyer defined a language L by using the 63-character alphabet

```
∀∃¬()≡∈∉+∧∨⇒⇔<≤=≠≥>abcdefghijklmnopqABCDEFGHIJKLMNOPQ0123456789
```

and giving conventional meanings to these symbols. Strings of lowercase letters within the sentences of L, like '`ab`' in (65), represent numeric variables, restricted to nonnegative integers; strings of uppercase letters represent set variables, restricted to finite sets of such numbers. For example, (66) means, "For all finite sets A and B, we have $A = B$ if and only if there doesn't exist a number n that is in A but not in B, or in B but not in A." Some of these statements are true; others are false. (See exercise 82.)

All of the strings (62)–(70) belong to L, but the language is actually quite restricted: The only arithmetic operation allowed on a number is to add a constant; we can write '`a+13`' but not '`a+b`'. The only relation allowed between a number and a set is elementhood (∈ or ∉). The only relation allowed between sets is equality (≡). Furthermore all variables must be quantified by ∃ or ∀.*

Every sentence of L that has length $k \le n$ can be represented by a binary vector of length $6n$, with zeros in the last $6(n-k)$ bits. Let $f(x)$ be a Boolean function of $6n$ variables such that $f(x) = 1$ whenever x represents a true sentence of L, and $f(x) = 0$ whenever x represents a sentence that is false; the value of $f(x)$ is unspecified when x doesn't represent a meaningful sentence. The truth table for such a function f can be constructed in a finite number of steps, according to theorems of Büchi and Elgot [*Zeitschrift für math. Logik und Grundlagen der Math.* **6** (1960), 66–92; *Transactions of the Amer. Math. Soc.* **98** (1961), 21–51]. But "finite" does not mean "feasible": Stockmeyer and Meyer proved that

$$C(f) > 2^{r-5} \qquad \text{whenever } n \ge 460 + .302r + 5.08\ln r \text{ and } r > 36. \tag{71}$$

In particular, we have $C(f) > 2^{426} > 10^{128}$ when $n = 621$. *A Boolean chain with that many gates could never be built*, since 10^{128} is a generous upper bound on the number of protons in the universe. So this is a fairly small, finite problem that will never be solved.

Details of Stockmeyer and Meyer's proof appear in *JACM* **49** (2002), 753–784. The basic idea is that the language L, though limited, is rich enough to describe truth tables and the complexity of Boolean chains, using fairly short sentences; hence f has to deal with inputs that essentially refer to themselves.

***For further reading.** Thousands of significant papers have been written about networks of Boolean gates, because such networks underlie so many aspects of theory and practice. We have focused in this section chiefly on topics that are relevant to computer programming for sequential machines. But other topics have also been extensively investigated, of primary relevance to parallel computation, such as the study of small-depth circuits in which gates can have any number of inputs ("unlimited fan-in"). Ingo Wegener's book *The Complexity of*

---

* Technically speaking, the sentences of L belong to "weak second-order monadic logic with one successor." Weak second-order logic allows quantification over finite sets; monadic logic with k successors is the theory of unlabeled k-ary trees.

weak second-order logic
second-order logic
k-ary trees
WS1S
Monadic logic: Logic with only unary operators
Büchi
Elgot
Stockmeyer
Meyer
self-reference
diagonalization
depth
fan-in
Wegener

Boolean Functions (Teubner and Wiley, 1987) provides a good introduction to the entire subject.

We have mostly considered Boolean chains in which all binary operators have equal importance. For our purposes, gates such as $\oplus$ or $\overline{\subset}$ are neither more nor less desirable than gates such as $\wedge$ or $\vee$. But it's natural to wonder if we can get by with only the monotone operators $\wedge$ and $\vee$ when we are computing a monotone function. Alexander Razborov has developed striking proof techniques to show that, in fact, monotone operators by themselves have inherently limited capabilities. He proved, for example, that all AND-OR chains to determine whether the permanent of an $n \times n$ matrix of 0s and 1s is zero or nonzero must have cost $n^{\Omega(\log n)}$. [See *Doklady Akademii Nauk SSSR* **281** (1985), 798–801; *Matematicheskie Zametki* **37** (1985), 887–900.] By contrast, we will see in Section 7.5.1 that this problem, equivalent to "bipartite matching," is solvable in only $O(n^{2.5})$ steps. Furthermore, the efficient methods in that section can be implemented as Boolean chains of only slightly larger cost, when we allow negation or other Boolean operations in addition to $\wedge$ and $\vee$. (Vaughan Pratt has called this "the power of negative thinking.") An introduction to Razborov's methods appears in exercises 85 and 86.

EXERCISES

1. [*24*] The "random" function in formula (6) corresponds to a Boolean chain of cost 4 and depth 4. Find a formula of depth 3 that has the same cost.

2. [*21*] Show how to compute (a) $w \oplus \langle xyz\rangle$ and (b) $w \wedge \langle xyz\rangle$ with formulas that have depth 3 and cost 5.

3. [*M23*] (B. I. Finikov, 1957.) If the Boolean function $f(x_1,\ldots,x_n)$ is true at exactly k points, prove that $L(f) < 2n+(k-2)2^{k-1}$. *Hint:* Think of $k=3$ and $n=10^6$.

4. [*M26*] (P. M. Spira, 1971.) Prove that the minimum depth and formula length of a Boolean function satisfy $\lg L(f) < D(f) \le \alpha \lg L(f)+1$, where $\alpha = 2/\lg(\frac{3}{2}) \approx 3.419$. *Hint:* Every binary tree with $r \ge 3$ internal nodes contains a subtree with s internal nodes, where $\frac{1}{3}r \le s < \frac{2}{3}r$.

▸ **5.** [*21*] The Fibonacci threshold function $F_n(x_1,\ldots,x_n) = \langle x_1^{F_1}x_2^{F_2}\ldots x_{n-1}^{F_{n-1}}x_n^{F_{n-2}}\rangle$ was analyzed in exercise 7.1.1–101, when $n \ge 3$. Is there an efficient way to evaluate it?

6. [*20*] True or false: A Boolean function $f(x_1,\ldots,x_n)$ is normal if and only if it satisfies the general distributive law $f(x_1,\ldots,x_n)\wedge y = f(x_1\wedge y,\ldots,x_n\wedge y)$.

7. [*20*] Convert the Boolean chain '$x_5 = x_1 \,\overline{\vee}\, x_4$, $x_6 = \bar{x}_2 \vee x_5$, $x_7 = \bar{x}_1 \wedge \bar{x}_3$, $x_8 = x_6 \equiv x_7$' to an equivalent chain $(\hat{x}_5,\hat{x}_6,\hat{x}_7,\hat{x}_8)$ in which every step is normal.

▸ **8.** [*20*] Explain why (11) is the truth table of variable x_k.

9. [*20*] Algorithm L determines the lengths of shortest formulas for all functions f, but it gives no further information. Extend the algorithm so that it also provides actual minimum-length formulas like (6).

▸ **10.** [*20*] Modify Algorithm L so that it computes $D(f)$ instead of $L(f)$.

▸ **11.** [*22*] Modify Algorithm L so that, instead of lengths $L(f)$, it computes upper bounds $U(f)$ and footprints $\phi(f)$ as described in the text.

12. [*15*] What Boolean chain is equivalent to the minimum-memory scheme (13)?

monotone function
Razborov
AND-OR
permanent
0–1 matrices
bipartite matching
Pratt
depth
pi, random
median
Finikov
Spira
Fibonacci threshold function
distributive law
normal
footprints
minimum-memory

13. [*16*] What are the truth tables of f_1, f_2, f_3, f_4, and f_5 in example (13)?

14. [*22*] What's a convenient way to compute the $5n(n-1)$ truth tables of (17), given the truth table of g? (Use bitwise operations as in (15) and (16).)

15. [*28*] Find short-as-possible ways to evaluate the following functions using minimum memory: (a) $S_2(x_1, x_2, x_3, x_4)$; (b) $S_1(x_1, x_2, x_3, x_4)$; (c) the function in (18).

16. [*HM33*] Prove that fewer than 2^{118} of the 2^{128} Boolean functions $f(x_1, \ldots, x_7)$ are computable in minimum memory.

▶ **17.** [*25*] (M. S. Paterson, 1977.) Although Boolean functions $f(x_1, \ldots, x_n)$ cannot always be evaluated in n registers, prove that $n+1$ registers are always sufficient. In other words, show that there is always a sequence of operations like (13) to compute $f(x_1, \ldots, x_n)$ if we allow $0 \le j(i), k(i) \le n$.

▶ **18.** [*35*] Investigate optimum minimum-memory computations for $f(x_1, x_2, x_3, x_4, x_5)$: How many classes of five-variable functions have $C_m(f) = r$, for $r = 0, 1, 2, \ldots$?

19. [*M22*] If a Boolean chain uses n variables and has length $r < n+2$, prove that it must be either a "top-down" or a "bottom-up" construction.

▶ **20.** [*40*] (R. Schroeppel, 2004.) A Boolean chain is *canalizing* if it does not use the operators $\oplus$ or $\equiv$. Find the optimum cost, length, and depth of all 4-variable functions under this constraint. Does the footprint heuristic still give optimum results?

21. [*46*] For how many four-variable functions did the Harvard researchers discover an optimum vacuum-tube circuit in 1951?

22. [*21*] Explain the chain for S_3 in Fig. 10, by noting that it incorporates the chain for $S_{2,3}$ in Fig. 9. Find a similar chain for $S_2(x_1, x_2, x_3, x_4, x_5)$.

▶ **23.** [*23*] Figure 10 illustrates only 16 of the 64 symmetric functions on five elements. Explain how to write down optimum chains for the others.

24. [*47*] Does every symmetric function f have $C_m(f) = C(f)$?

▶ **25.** [*17*] Suppose we want a Boolean chain that includes *all* functions of n variables: Let $f_k(x_1, \ldots, x_n)$ be the function whose truth table is the binary representation of k, for $0 \le k < m = 2^{2^n}$. What is $C(f_0 f_1 \ldots f_{m-1})$?

26. [*25*] True or false: If $f(x_0, \ldots, x_n) = (x_0 \wedge g(x_1, \ldots, x_n)) \oplus h(x_1, \ldots, x_n)$, where g and h are nontrivial Boolean functions whose joint cost is $C(gh)$, then $C(f) = 2 + C(gh)$.

▶ **27.** [*23*] Can a full adder (22) be implemented in five steps using only minimum memory (that is, completely inside three one-bit registers)?

28. [*26*] Prove that $C(u'v') = C(u''v'') = 5$ for the two-output functions defined by

$$(u'v')_2 = (x + y - (uv)_2) \bmod 4, \qquad (u''v'')_2 = (-x - y - (uv)_2) \bmod 4.$$

Use these functions to evaluate $[(x_1 + \cdots + x_n) \bmod 4 = 0]$ in fewer than $2.5n$ steps.

29. [*M28*] Prove that the text's circuit for sideways addition (27) has depth $O(\log n)$.

30. [*M25*] Solve the binary recurrence (28) for the cost $s(n)$ of sideways addition.

31. [*21*] If $f(x_1, \ldots, x_n)$ is symmetric, prove that $C(f) \le 5n + O(n/\log n)$.

32. [*HM16*] Why does the solution to (30) satisfy $t(n) = 2^n + O(2^{n/2})$?

33. [*HM22*] True or false: If $1 \le N \le 2^n$, the first N minterms of $\{x_1, \ldots, x_n\}$ can all be evaluated in $N + O(\sqrt{N})$ steps, as $n \to \infty$ and $N \to \infty$.

bitwise operations
symmetric functions
minimum memory
Paterson
five-variable functions
top-down
bottom-up
Schroeppel
canalizing
canalizing functions
AND/OR/NOT circuits
footprint
four-variable functions
Harvard
vacuum-tube circuit
symmetric functions
full adder
minimum memory
sideways addition
binary recurrence
minterms

▶ **34.** [*22*] A *priority encoder* has $n = 2^m - 1$ inputs $x_1 \ldots x_n$ and m outputs $y_1 \ldots y_m$, where $(y_1 \ldots y_m)_2 = k$ if and only if $k = \max\{j \mid j = 0 \text{ or } x_j = 1\}$. Design a priority encoder that has cost $O(n)$ and depth $O(m)$.

35. [*23*] If $n > 1$, show that the conjunctions $x_1 \wedge \cdots \wedge x_{k-1} \wedge x_{k+1} \wedge \cdots \wedge x_n$ for $1 \le k \le n$ can all be computed from $(x_1, \ldots, x_n)$ with total cost $\le 3n - 6$.

▶ **36.** [*M28*] (R. E. Ladner and M. J. Fischer, 1980.) Let y_k be the "prefix" $x_1 \wedge \cdots \wedge x_k$ for $1 \le k \le n$. Clearly $C(y_1 \ldots y_n) = n - 1$ and $D(y_1 \ldots y_n) = \lceil \lg n \rceil$; but we can't simultaneously minimize both cost and depth. Find a chain of optimum depth $\lceil \lg n \rceil$ that has cost $< 4n$.

37. [*M28*] (Marc Snir, 1986.) Given $n \ge m \ge 1$, consider the following algorithm:

S1. [Upward loop.] For $t \leftarrow 1, 2, \ldots, \lceil \lg m \rceil$, set $x_{\min(m,2^t k)} \leftarrow x_{2^t(k-1/2)} \wedge x_{\min(m,2^t k)}$ for $k \ge 1$ and $2^t(k - 1/2) < m$.

S2. [Downward loop.] For $t \leftarrow \lceil \lg m \rceil - 1, \lceil \lg m \rceil - 2, \ldots, 1$, set $x_{2^t(k+1/2)} \leftarrow x_{2^t k} \wedge x_{2^t(k+1/2)}$ for $k \ge 1$ and $2^t(k + 1/2) < m$.

S3. [Extension.] For $k \leftarrow m + 1, m + 2, \ldots, n$, set $x_k \leftarrow x_{k-1} \wedge x_k$. ▮

a) Prove that this algorithm solves the prefix problem of exercise 36: It transforms $(x_1, x_2, \ldots, x_n)$ into $(x_1, x_1 \wedge x_2, \ldots, x_1 \wedge x_2 \wedge \cdots \wedge x_n)$.

b) Let $c(m, n)$ and $d(m, n)$ be the cost and depth of the corresponding Boolean chain. Prove that, if n is sufficiently large, $c(m, n) + d(m, n) = 2n - 2$.

c) Given n, what is $d(n) = \min_{1 \le m \le n} d(m, n)$? Show that $d(n) < 2 \lg n$.

d) Prove that there's a Boolean chain of cost $2n - 2 - d$ and depth d for the prefix problem whenever $d(n) \le d < n$. (This cost is optimum, by exercise 81.)

38. [*25*] In Section 5.3.4 we studied *sorting networks*, by which $\hat{S}(n)$ comparator modules are able to sort n numbers $(x_1, x_2, \ldots, x_n)$ into ascending order. If the inputs x_j are 0s and 1s, each comparator module is equivalent to two gates $(x \wedge y, x \vee y)$; so a sorting network corresponds to a certain kind of Boolean chain, which evaluates n particular functions of $(x_1, x_2, \ldots, x_n)$.

a) What are the n functions $f_1 f_2 \ldots f_n$ that a sorting network computes?

b) Show that those functions $\{f_1, f_2, \ldots, f_n\}$ can be computed in $O(n)$ steps with a chain of depth $O(\log n)$. (Hence sorting networks aren't asymptotically optimal, Booleanwise.)

▶ **39.** [*M21*] (M. S. Paterson and P. Klein, 1980.) Implement the 2^m-way multiplexer $M_m(x_1, \ldots, x_m; y_0, y_1, \ldots, y_{2^m-1})$ of (31) with an efficient chain that simultaneously establishes the upper bounds $C(M_m) \le 2n + O(\sqrt{n})$ and $D(M_m) \le m + O(\log m)$.

40. [*25*] If $n \ge k \ge 1$, let $f_{nk}(x_1, \ldots, x_n)$ be the "k in a row" function,

$$(x_1 \wedge \cdots \wedge x_k) \vee (x_2 \wedge \cdots \wedge x_{k+1}) \vee \cdots \vee (x_{n+1-k} \wedge \cdots \wedge x_n).$$

Show that the cost $C(f_{nk})$ of this function is less than $4n - 3k$.

41. [*M23*] (*Conditional-sum adders.*) One way to accomplish binary addition (25) with depth $O(\log n)$ is based on the multiplexer trick of exercise 4: If $(xx')_2 + (yy')_2 = (zz')_2$, where $|x'| = |y'| = |z'|$, we have either $(x)_2 + (y)_2 = (z)_2$ and $(x')_2 + (y')_2 = (z')_2$, or $(x)_2 + (y)_2 + 1 = (z)_2$ and $(x')_2 + (y')_2 = (1z')_2$. To save time, we can compute *both* $(x)_2 + (y)_2$ and $(x)_2 + (y)_2 + 1$ simultaneously as we compute $(x')_2 + (y')_2$. Afterwards, when we know whether or not the less significant part $(x')_2 + (y')_2$ produces a carry, we can use multiplexers to select the correct bits for the most significant part.

priority encoder
Ladner
Fischer
prefix
Snir
prefix problem
sorting networks
comparator modules
Paterson
Klein
2^m-way multiplexer
k in a row
consecutive 1s
Conditional-sum adders
multiplexer
carry

If this method is used recursively to build $2n$-bit adders from n-bit adders, how many gates are needed when $n = 2^m$? What is the corresponding depth?

42. [*25*] In the binary addition (25), let $u_k = x_k \wedge y_k$ and $v_k = x_k \oplus y_k$ for $0 \le k < n$.

a) Show that $z_k = v_k \oplus c_k$, where the carry bits c_k satisfy

$$c_k = u_{k-1} \vee (v_{k-1} \wedge (u_{k-2} \vee (v_{k-2} \wedge (\cdots (u_1 \wedge v_0) \cdots)))).$$

b) Let $U_k^k = 0$, $V_k^k = 1$, and $U_j^{k+1} = u_k \vee (v_k \wedge U_j^k)$, $V_j^{k+1} = v_k \wedge V_j^k$, for $k \ge j$. Prove that $c_k = U_0^k$, and that $U_i^k = U_j^k \vee (V_j^k \wedge U_i^j)$, $V_i^k = V_j^k \wedge V_i^j$ for $i \le j \le k$.

c) Let $h(m) = 2^{m(m-1)/2}$. Show that when $n = h(m)$, the carries $c_1, \ldots, c_n$ can all be evaluated with depth $(m+1)m/2 \approx \lg n + \sqrt{2 \lg n}$ and with total cost $O(2^m n)$.

▸ **43.** [*28*] A *finite state transducer* is an abstract machine with a finite input alphabet A, a finite output alphabet B, and a finite set of internal states Q. One of those states, q_0, is called the "initial state." Given a string $\alpha = a_1 \ldots a_n$, where each $a_j \in A$, the machine computes a string $\beta = b_1 \ldots b_n$, where each $b_j \in B$, as follows:

T1. [Initialize.] Set $j \leftarrow 1$ and $q \leftarrow q_0$.

T2. [Done?] Terminate the algorithm if $j > n$.

T3. [Output b_j.] Set $b_j \leftarrow c(q, a_j)$.

T4. [Advance j.] Set $q \leftarrow d(q, a_j)$, $j \leftarrow j + 1$, and return to step T2. ▌

The machine has built-in instructions that specify $c(q, a) \in B$ and $d(q, a) \in Q$ for every state $q \in Q$ and every character $a \in A$. The purpose of this exercise is to show that, if the alphabets A and B of any finite state transducer are encoded in binary, the string β can be computed from α by a Boolean chain of size $O(n)$ and depth $O(\log n)$.

a) Consider the problem of changing a binary vector $a_1 \ldots a_n$ to $b_1 \ldots b_n$ by setting

$$b_j \leftarrow a_j \oplus [a_j = a_{j-1} = \cdots = a_{j-k} = 1 \text{ and } a_{j-k-1} = 0, \text{ where } k \text{ is odd}],$$

assuming that $a_0 = 0$. For example, $\alpha = 1100100100011111101101010 \mapsto \beta = 1000100100010101001001010$. Prove that this transformation can be carried out by a finite state transducer with $|A| = |B| = |Q| = 2$.

b) Suppose a finite state transducer is in state q_j after reading $a_1 \ldots a_{j-1}$. Explain how to compute the sequence $q_1 \ldots q_n$ with a Boolean chain of cost $O(n)$ and depth $O(\log n)$, using the construction of Ladner and Fischer in exercise 36. (From this sequence $q_1 \ldots q_n$ it is easy to compute $b_1 \ldots b_n$, since $b_j = c(q_j, a_j)$.)

c) Apply the method of (b) to the problem in (a).

▸ **44.** [*26*] (R. E. Ladner and M. J. Fischer, 1980.) Show that the problem of binary addition (25) can be viewed as a finite state transduction. Describe the Boolean chain that results from the construction of exercise 43 when $n = 2^m$, and compare it to the conditional-sum adder of exercise 41.

45. [*HM20*] Why doesn't the proof of Theorem S simply argue that the number of ways to choose $j(i)$ and $k(i)$ so that $1 \le j(i), k(i) < i$ is $n^2(n+1)^2 \ldots (n+r-1)^2$?

▸ **46.** [*HM21*] Let $\alpha(n) = c(n, \lfloor 2^n/n \rfloor)/2^{2^n}$ be the fraction of n-variable Boolean functions $f(x_1, \ldots, x_n)$ for which $C(f) \le 2^n/n$. Prove that $\alpha(n) \to 0$ rapidly as $n \to \infty$.

47. [*M23*] Extend Theorem S to functions with n inputs and m outputs.

48. [*HM23*] Find the smallest integer $r = r(n)$ such that $(r-1)!\, 2^{2^n} \le 2^{2r+1}(n+r-1)^{2r}$, (a) exactly when $1 \le n \le 16$; (b) asymptotically when $n \to \infty$.

carry bits
finite state transducer
pi
Ladner
Fischer
Ladner
Fischer

49. [*HM25*] Prove that, as $n \to \infty$, almost all Boolean functions $f(x_1, \ldots, x_n)$ have minimum formula length $L(f) > 2^n/\lg n - 2^{n+2}/(\lg n)^2$.

50. [*24*] What are the prime implicants and prime clauses of the prime-number function (35)? Express that function in (a) DNF (b) CNF of minimum length.

51. [*20*] What representation of the prime-number detector replaces (37), if rows of the truth table are based on x_5x_6 instead of x_1x_2?

52. [*23*] What choices of k and l minimize the upper bound (38) when $5 \le n \le 16$?

53. [*HM22*] Estimate (38) when $k = \lfloor 2\lg n \rfloor$ and $l = \lceil 2^k/(n - 3\lg n) \rceil$ and $n \to \infty$.

54. [*29*] Find a short Boolean chain to evaluate all six of the functions $f_j(x) = [x_1x_2x_3x_4 \in A_j]$, where $A_1 = \{0010, 0101, 1011\}$, $A_2 = \{0001, 1111\}$, $A_3 = \{0011, 0111, 1101\}$, $A_4 = \{1001, 1111\}$, $A_5 = \{1101\}$, $A_6 = \{0101, 1011\}$. (These six functions appear in the prime-number detector (37).) Compare your chain to the minterm-first evaluation scheme of Lupanov's general method.

55. [*34*] Show that the cost of the 6-bit prime-detecting function is at most 14.

▶ **56.** [*16*] Explain why all functions with 14 or more don't-cares in Table 3 have cost 0.

57. [*19*] What seven-segment "digits" are displayed when $(x_1x_2x_3x_4)_2 > 9$ in (45)?

▶ **58.** [*30*] A 4×4-bit *S-box* is a permutation of the 4-bit vectors $\{0000, 0001, \ldots, 1111\}$; such permutations are used as components of well-known cryptographic systems such as the Russian standard GOST 28147 (1989). Every 4×4-bit S-box corresponds to a sequence of four functions $f_1(x_1, x_2, x_3, x_4)$, ..., $f_4(x_1, x_2, x_3, x_4)$, which transform $x_1x_2x_3x_4 \mapsto f_1f_2f_3f_4$.

Find all 4×4-bit S-boxes for which $C(f_1) = C(f_2) = C(f_3) = C(f_4) = 7$.

59. [*29*] One of the S-boxes satisfying the conditions of exercise 58 takes $(\mathtt{0}, \ldots, \mathtt{f}) \mapsto (\mathtt{0}, \mathtt{6}, \mathtt{5}, \mathtt{b}, \mathtt{3}, \mathtt{9}, \mathtt{f}, \mathtt{e}, \mathtt{c}, \mathtt{4}, \mathtt{7}, \mathtt{8}, \mathtt{d}, \mathtt{2}, \mathtt{a}, \mathtt{1})$; in other words, the truth tables of (f_1, f_2, f_3, f_4) are respectively $(\mathtt{179a}, \mathtt{63e8}, \mathtt{5b26}, \mathtt{3e29})$. Find a Boolean chain that evaluates these four "maximally difficult" functions in fewer than 20 steps.

60. [*23*] (Frank Ruskey.) Suppose $z = (x+y) \bmod 3$, where $x = (x_1x_2)_2$, $y = (y_1y_2)_2$, $z = (z_1z_2)_2$, and each two-bit value is required to be either 00, 01, or 10. Compute z_1 and z_2 from x_1, x_2, y_1, and y_2 in six Boolean steps.

61. [*34*] Continuing exercise 60, find a good way to compute $z = (x+y) \bmod 5$, using the three-bit values 000, 001, 010, 011, 100.

62. [*HM23*] Consider a random Boolean partial function of n variables that has 2^nc "cares" and 2^nd "don't-cares," where $c + d = 1$. Prove that the cost of almost all such partial functions exceeds $2^nc/n$.

63. [*HM35*] (L. A. Sholomov, 1969.) Continuing exercise 62, prove that all such functions have cost $\le 2^nc/n(1 + O(n^{-1}\log n))$. *Hint:* There is a set of $2^m(1+k)$ vectors $x_1 \ldots x_k$ that intersects every $(k-m)$-dimensional subcube of the k-cube.

64. [*25*] (*Magic Fifteen.*) Two players alternately select digits from 1 to 9, using no digit twice; the winner, if any, is the first to get three digits that sum to 15. What's a good strategy for playing this game?

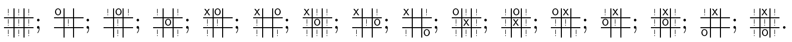
▶ **65.** [*35*] Modify the tic-tac-toe strategy of (47)–(56) so that it always plays correctly.

66. [*20*] Criticize the moves chosen in exercise 65. Are they always optimum?

prime implicants
prime clauses
prime-number function
Lupanov
don't-cares
S-box
GOST
Ruskey
addition modulo 3 and 5
cares
don't-cares
Sholomov
subcube
k-cube
Magic Fifteen

π function
multilinear representation
good
bad
analysis of algorithms

▸ **67.** [*40*] Instead of simply finding one correct move for each position in tic-tac-toe, we might prefer to find them all. In other words, given $x_1 \ldots x_9 o_1 \ldots o_9$, we could try to compute nine outputs $g_1 \ldots g_9$, where $g_j = 1$ if and only if a move into cell j is among X's best. For example, exclamation marks indicate all of the right moves for X in the following typical positions:



A machine that chooses randomly among these possibilities is more fun to play against than a machine that has only one fixed strategy.

One attractive way to solve the all-good-moves problem is to use the fact that tic-tac-toe has eight symmetries. Imagine a chip that has 18 inputs $x_1 \ldots x_9 o_1 \ldots o_9$ and three outputs (c, s, m), for "corner," "side," and "middle," with the property that the desired functions g_j can be computed by hooking together eight of the chips appropriately:

$$\begin{aligned}
g_1 &= c(x_1x_2x_3x_4x_5x_6x_7x_8x_9o_1o_2o_3o_4o_5o_6o_7o_8o_9)\\
&\qquad \vee\, c(x_1x_4x_7x_2x_5x_8x_3x_6x_9o_1o_4o_7o_2o_5o_8o_3o_6o_9),\\
g_2 &= s(x_1x_2x_3x_4x_5x_6x_7x_8x_9o_1o_2o_3o_4o_5o_6o_7o_8o_9)\\
&\qquad \vee\, s(x_3x_2x_1x_6x_5x_4x_9x_8x_7o_3o_2o_1o_6o_5o_4o_9o_8o_7),\\
g_3 &= c(x_3x_2x_1x_6x_5x_4x_9x_8x_7o_3o_2o_1o_6o_5o_4o_9o_8o_7)\\
&\qquad \vee\, c(x_3x_6x_9x_2x_5x_8x_1x_4x_7o_3o_6o_9o_2o_5o_8o_1o_4o_7),\\
g_4 &= s(x_1x_4x_7x_2x_5x_8x_3x_6x_9o_1o_4o_7o_2o_5o_8o_3o_6o_9)\\
&\qquad \vee\, s(x_7x_4x_1x_8x_5x_2x_9x_6x_3o_7o_4o_1o_8o_5o_2o_9o_6o_3), \qquad \ldots\\
g_9 &= c(x_9x_8x_7x_6x_5x_4x_3x_2x_1o_9o_8o_7o_6o_5o_4o_3o_2o_1)\\
&\qquad \vee\, c(x_9x_6x_3x_8x_5x_2x_7x_4x_1o_9o_6o_3o_8o_5o_2o_7o_4o_1),
\end{aligned}$$

and g_5 is the OR of the m outputs from all eight chips.

Design such a chip, using fewer than 2000 gates.

68. [*M25*] Consider the n-bit π function $\pi_n(x_1 \ldots x_n)$, whose value is the $(x_1 \ldots x_n)_2$th bit to the right of the most significant bit in the binary representation of π. Does the method of exercise 4.3.1–39, which describes an efficient way to compute arbitrary bits of π, prove that $C(\pi_n) < 2^n/n$ for sufficiently large n?

69. [*M24*] Let the multilinear representation of f be

$$\alpha_{000} \oplus \alpha_{001}x_m \oplus \alpha_{010}x_j \oplus \alpha_{011}x_jx_m \oplus \alpha_{100}x_i \oplus \alpha_{101}x_ix_m \oplus \alpha_{110}x_ix_j \oplus \alpha_{111}x_ix_jx_m,$$

where each coefficient α_l is a function of the variables $\{x_1, \ldots, x_n\} \setminus \{x_i, x_j, x_m\}$.

a) Prove that the pairs (58) are "good" if and only if the coefficients satisfy

$$\alpha_{010}\alpha_{101} = \alpha_{011}\alpha_{100}, \quad \alpha_{101}\alpha_{110} = \alpha_{100}\alpha_{111}, \quad \text{and} \quad \alpha_{110}\alpha_{011} = \alpha_{111}\alpha_{010}.$$

b) For which values (i, j, m) are the pairs bad, when $f = (\det X) \bmod 2$? (See (60).)

▸ **70.** [*M27*] Let X be the 3×3 Boolean matrix (60). Find efficient chains for the Boolean functions (a) $(\det X) \bmod 2$; (b) $[\operatorname{per} X > 0]$; (c) $[\det X > 0]$.

▸ **71.** [*M26*] Suppose $f(x)$ is equal to 0 with probability p at each point $x = x_1 \ldots x_n$, independent of its value at other points.

a) What is the probability that the pairs (58) are good?
b) What is the probability that bad pairs (58) exist?
c) What is the probability that bad pairs (58) are found in at most t random trials?
d) What is the expected time to test case (i, j, m), as a function of p, t, and n?

72. [*M24*] Extend the previous exercise to the case of partial functions, where $f(x) = 0$ with probability p, $f(x) = 1$ with probability q, and $f(x) = *$ with probability r.

▶ **73.** [*20*] If bad pairs (58) exist for all (i, j, m) with $m \neq i \neq j \neq m$, show that the indecomposability of f can be deduced after testing only $\binom{n}{2}$ well-chosen triples (i, j, m).

74. [*25*] Extend the idea in the previous exercise, suggesting a strategy for choosing successive triples (i, j, m) when using the method of Shen, McKellar, and Weiner.

75. [*20*] What happens when the text's decomposition procedure is applied to the "all-equal" function $S_{0,n}(x_1, \ldots, x_n)$?

▶ **76.** [*M25*] (D. Uhlig, 1974.) The purpose of this exercise is to prove the amazing fact that, for certain functions f, the best chain to evaluate the Boolean function

$$F(u_1, \ldots, u_n, v_1, \ldots, v_n) \;=\; f(u_1, \ldots, u_n) \vee f(v_1, \ldots, v_n)$$

costs *less* than $2C(f)$; hence functional decomposition is *not* always a good idea.

We let $n = m + 2^m$ and write $f(i_1, \ldots, i_m, x_0, \ldots, x_{2^m-1}) = f_i(x)$, where i is regarded as the number $(i_1 \ldots i_m)_2$. Then $(u_1, \ldots, u_n) = (i_1, \ldots, i_m, x_0, \ldots, x_{2^m-1})$, $(v_1, \ldots, v_n) = (j_1, \ldots, j_m, y_0, \ldots, y_{2^m-1})$, and $F(u, v) = f_i(x) \vee f_j(y)$.

a) Prove that a chain of cost $O(n/\log n)^2$ suffices to evaluate the $2^m + 1$ functions

$$z_l \;=\; x \oplus \big(([l \le i] \oplus [i \le j]) \wedge (x \oplus y)\big), \qquad 0 \le l \le 2^m,$$

from given vectors i, j, x, and y; each z_l is a vector of length 2^m.

b) Let $g_i(x) = f_i(x) \oplus f_{i-1}(x)$ for $0 \le i \le 2^m$, where $f_{-1}(x) = f_{2^m}(x) = 0$. Estimate the cost of computing the $2^m + 1$ values $c_l = g_l(z_l)$, given the vectors z_l, for $0 \le l \le 2^m$.

c) Let $c'_l = c_l \wedge ([i \le j] \equiv [l \le i])$ and $c''_l = c_l \wedge ([i \le j] \equiv [j > l])$. Prove that

$$f_i(x) = c'_0 \oplus c'_1 \oplus \cdots \oplus c'_{2^m}, \qquad f_j(y) = c''_0 \oplus c''_1 \oplus \cdots \oplus c''_{2^m}.$$

d) Conclude that $C(F) \le 2^n/n + O(2^n(\log n)/n^2)$. (When n is sufficiently large, this cost is definitely less than $2^{n+1}/n$, but functions f exist with $C(f) > 2^n/n$.)

e) For clarity, write out the chain for F when $m = 1$ and $f(i, x_0, x_1) = (i \wedge x_0) \vee x_1$.

▶ **77.** [*35*] (N. P. Red'kin, 1970.) Suppose a Boolean chain uses only the operations AND, OR, or NOT; thus, every step is either $x_i = x_{j(i)} \wedge x_{k(i)}$ or $x_i = x_{j(i)} \vee x_{k(i)}$ or $x_i = \bar{x}_{j(i)}$. Prove that if such a chain computes either the "odd parity" function $f_n(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$ or the "even parity" function $\bar{f}_n(x_1, \ldots, x_n) = 1 \oplus x_1 \oplus \cdots \oplus x_n$, where $n \ge 2$, the length of the chain is at least $4(n-1)$.

78. [*26*] (W. J. Paul, 1977.) Let $f(x_1, \ldots, x_m, y_0, \ldots, y_{2^m-1})$ be any Boolean function that equals y_k whenever $(x_1 \ldots x_m)_2 = k \in S$, for some given set $S \subseteq \{0, 1, \ldots, 2^m - 1\}$; we don't care about the value of f at other points. Show that $C(f) \ge 2\|S\| - 2$ whenever S is nonempty. (In particular, when $S = \{0, 1, \ldots, 2^m - 1\}$, the multiplexer chain of exercise 39 is asymptotically optimum.)

79. [*32*] (C. P. Schnorr, 1976.) Say that variables u and v are "mates" in a Boolean chain if there is exactly one simple path between them in the corresponding binary tree diagram. Two variables can be mates only if they are each used only once in the chain; but this necessary condition is not sufficient. For example, variables 2 and 4 are mates in the chain for $S_{1,2,3}$ in Fig. 9, but they are not mates in the chain for S_2.

a) Prove that a Boolean chain on n variables with no mates has cost $\ge 2n - 2$.

b) Prove that $C(f) = 2n - 3$ when f is the all-equal function $S_{0,n}(x_1, \ldots, x_n)$.

partial functions
don't-cares
Shen
McKellar
Weiner
all-equal
Uhlig
decomposition
Red'kin
AND-OR-NOT chains
parity
Paul
don't care
partially specified
multiplexer
Schnorr
mates
all-equal function

▶ **80.** [*M27*] (L. J. Stockmeyer, 1977.) Another notation for symmetric functions is sometimes convenient: If $\alpha = a_0 a_1 \ldots a_n$ is any binary string, let $S_\alpha(x) = a_{\nu x}$. For example, $\langle x_1 x_2 x_3 \rangle = S_{0011}$ and $x_1 \oplus x_2 \oplus x_3 = S_{0101}$ in this notation. Notice that $S_\alpha(0, x_2, \ldots, x_n) = S_{\alpha'}(x_2, \ldots, x_n)$ and $S_\alpha(1, x_2, \ldots, x_n) = S_{'\alpha}(x_2, \ldots, x_n)$, where α' and $'\alpha$ stand respectively for α with its last or first element deleted. Also,

$$S_\alpha(f(x_3, \ldots, x_n), \bar{f}(x_3, \ldots, x_n), x_3, \ldots, x_n) \;=\; S_{'\alpha'}(x_3, \ldots, x_n)$$

when f is any Boolean function of $n-2$ variables.

a) A parity function has $a_0 \neq a_1 \neq a_2 \neq \cdots \neq a_n$. Assume that $n \geq 2$. Prove that if S_α is not a parity function and $S_{'\alpha'}$ isn't constant, then

$$C(S_\alpha) \;\geq\; \max\bigl(C(S_{\alpha'})+2, C(S_{'\alpha})+2, \min\bigl(C(S_{\alpha'})+3, C(S_{'\alpha})+3, C(S_{'\alpha'})+5\bigr)\bigr).$$

b) What lower bounds on $C(S_k)$ and $C(S_{\geq k})$ follow from this result, when $0 \leq k \leq n$?

81. [*23*] (M. Snir, 1986.) Show that any chain of cost c and depth d for the prefix problem of exercise 36 has $c + d \geq 2n - 2$.

▶ **82.** [*M23*] Explain the logical sentences (62)–(70). Which of them are true?

83. [*21*] If there's a Boolean chain for $f(x_1, \ldots, x_n)$ that contains p canalizing operations, show that $C(f) < (p+1)(n + p/2)$.

84. [*M20*] A *monotone Boolean chain* is a Boolean chain in which every operator $\circ_i$ is monotone. The length of a shortest monotone chain for f is denoted by $C^+(f)$. If there's a monotone Boolean chain for $f(x_1, \ldots, x_n)$ that contains p occurrences of $\wedge$ and q occurrences of $\vee$, show that $C^+(f) < \min\bigl((p+1)(n+p/2), (q+1)(n+q/2)\bigr)$.

▶ **85.** [*M28*] Let M_n be the set of all monotone functions of n variables. If L is a family of functions contained in M_n, let

$$x \sqcup y = \bigwedge\{z \in L \mid z \supseteq x \vee y\} \qquad \text{and} \qquad x \sqcap y = \bigvee\{z \in L \mid z \subseteq x \wedge y\}.$$

We call L "legitimate" if it includes the constant functions 0 and 1 as well as the projection functions x_j for $1 \leq j \leq n$, and if $x \sqcup y \in L$, $x \sqcap y \in L$ whenever $x, y \in L$.

a) When $n = 3$ we can write $M_3 = \{$`00`, `01`, `03`, `05`, `11`, `07`, `13`, `15`, `0f`, `33`, `55`, `17`, `1f`, `37`, `57`, `3f`, `5f`, `77`, `7f`, `ff`$\}$, representing each function by its hexadecimal truth table. There are 2^{15} families L such that $\{\texttt{00}, \texttt{0f}, \texttt{33}, \texttt{55}, \texttt{ff}\} \subseteq L \subseteq M_3$; how many of them are legitimate?

b) If A is a subset of $\{1, \ldots, n\}$, let $\lceil A \rceil = \bigvee_{a \in A} x_a$; also let $\lceil \infty \rceil = 1$. Suppose $\mathcal{A}$ is a family of subsets of $\{1, \ldots, n\}$ that contains all sets of size ≤ 1 and is closed under intersection; in other words, $A \cap B \in \mathcal{A}$ whenever $A \in \mathcal{A}$ and $B \in \mathcal{A}$. Prove that the family $L = \{\lceil A \rceil \mid A \in \mathcal{A} \cup \{\infty\}\}$ is legitimate.

c) Let $(x_{n+1}, \ldots, x_{n+r})$ be a monotone Boolean chain (1). Suppose $(\hat{x}_{n+1}, \ldots, \hat{x}_{n+r})$ is obtained from the same Boolean chain, but with every operator $\wedge$ changed to $\sqcap$ and with every operator $\vee$ changed to $\sqcup$, with respect to some legitimate family L. Prove that, for $n+1 \leq l \leq n+r$, we must have

$$\hat{x}_l \;\subseteq\; x_l \vee \bigvee_{i=n+1}^{l} \{\hat{x}_i \oplus (\hat{x}_{j(i)} \vee \hat{x}_{k(i)}) \mid \circ_i = \vee\};$$

$$x_l \;\subseteq\; \hat{x}_l \vee \bigvee_{i=n+1}^{l} \{\hat{x}_i \oplus (\hat{x}_{j(i)} \wedge \hat{x}_{k(i)}) \mid \circ_i = \wedge\}.$$

Stockmeyer
notation
symmetric functions
prefixes of strings
suffixes of strings
parity
Snir
prefix problem
logic
monotone Boolean chain
monotone functions
legitimate
hexadecimal
truth table
closed under intersection

86. [*HM37*] A graph G on vertices $\{1,\ldots,n\}$ can be defined by $N = \binom{n}{2}$ Boolean variables x_{uv} for $1 \le u < v \le n$, where $x_{uv} = [u \mathrel{—} v \text{ in } G]$. Let f be the function $f(x) = [G \text{ contains a triangle}]$; for example, when $n = 4$, $f(x_{12}, x_{13}, x_{14}, x_{23}, x_{24}, x_{34}) = (x_{12} \wedge x_{13} \wedge x_{23}) \vee (x_{12} \wedge x_{14} \wedge x_{24}) \vee (x_{13} \wedge x_{14} \wedge x_{34}) \vee (x_{23} \wedge x_{24} \wedge x_{34})$. The purpose of this exercise is to prove that the monotone complexity $C^+(f)$ is $\Omega(n/\log n)^3$.

a) If $u_j \mathrel{—} v_j$ for $1 \le j \le r$ in a graph G, call $S = \{\{u_1, v_1\}, \ldots, \{u_r, v_r\}\}$ an *r-family*, and let $\Delta(S) = \bigcup_{1 \le i < j \le r}(\{u_i, v_i\} \cap \{u_j, v_j\})$ be the elements of its pairwise intersections. Say that G is *r-closed* if we have $u \mathrel{—} v$ whenever $\Delta(S) \subseteq \{u, v\}$ for some r-family S. It is *strongly* r-closed if, in addition, we have $|\Delta(S)| \ge 2$ for all r-families S. Prove that a strongly r-closed graph is also strongly $(r+1)$-closed.

b) Prove that the complete bigraph $K_{m,n}$ is strongly r-closed when $r > \max(m, n)$.

c) Prove that a strongly r-closed graph has at most $(r-1)^2$ edges.

d) Let L be the family of functions $\{1\} \cup \{\lceil G \rceil \mid G \text{ is a strongly } r\text{-closed graph on } \{1, \ldots, n\}\}$. (See exercise 85(b); we regard G as a set of edges. For example, when the edges are $1 \mathrel{—} 3$, $1 \mathrel{—} 4$, $2 \mathrel{—} 3$, $2 \mathrel{—} 4$, we have $\lceil G \rceil = x_{13} \vee x_{14} \vee x_{23} \vee x_{24}$.) Is L legitimate?

e) Let $x_{N+1}, \ldots, x_{N+p+q} = f$ be a monotone Boolean chain with p $\wedge$-steps and q $\vee$-steps, and consider the modified chain $\hat{x}_{N+1}, \ldots, \hat{x}_{N+p+q} = \hat{f}$ based on the family L in (d). If $\hat{f} \ne 1$, show that $2(r-1)^3 p + (r-1)^2 (n-2) \ge \binom{n}{3}$. *Hint:* Use the second formula in exercise 85(c).

f) Furthermore, if $\hat{f} = 1$ we must have $r^2 q \ge 2^{r-1}$.

g) Therefore $p = \Omega(n/\log n)^3$. *Hint:* Let $r \approx 6 \lg n$ and apply exercise 84.

87. [*M20*] Show that when nonmonotonic operations are permitted, the triangle function of exercise 86 has cost $C(f) = O(n^{\lg 7}(\log n)^2) = O(n^{2.81})$. *Hint:* A graph has a triangle if and only if the cube of its adjacency matrix has a nonzero diagonal.

88. [*40*] A *median chain* is analogous to a Boolean chain, but it uses median-of-three steps $x_i = \langle x_{j(i)} x_{k(i)} x_{l(i)} \rangle$ for $n+1 \le i \le n+r$, instead of the binary operations in (1).

Study the optimum length, depth, and cost of median chains, for all self-dual monotone Boolean functions of 7 variables. What is the shortest chain for $\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 \rangle$?

monotone complexity
r-family
r-closed
complete bigraph
legitimate
monotone Boolean chain
triangle function
adjacency matrix
median chain
self-dual monotone
median of seven

mux
pi
normalization
recursive procedure

SECTION 7.1.2

1. $((x_1 \vee x_4) \wedge x_2) \equiv (x_1 \vee x_3)$.

2. (a) $(w \oplus (x \wedge y)) \oplus ((x \oplus y) \wedge z)$; (b) $(w \wedge (x \vee y)) \wedge ((x \wedge y) \vee z)$.

3. [*Doklady Akademii Nauk SSSR* **115** (1957), 247–248.] Construct a $k \times n$ matrix whose rows are the vectors x where $f(x) = 1$. By permuting and/or complementing variables, we may assume that the top row is $1 \ldots 1$ and that the columns are sorted. Suppose there are l distinct columns. Then $f = g \wedge h$, where g is the AND of the expressions $(x_{j-1} \equiv x_j)$ over all $1 < j \le n$ such that column $j-1$ equals column j, and h is the OR of k minterms of length l, using one variable from each group of equal columns. For example, if $n = 8$ and if f is 1 at the $k = 3$ points 11111111, 00001111, 00110111, then $l = 4$ and $f(x)$ equals $(x_1 \equiv x_2) \wedge (x_3 \equiv x_4) \wedge (x_6 \equiv x_7) \wedge (x_7 \equiv x_8) \wedge ((x_1 \wedge x_3 \wedge x_5 \wedge x_6) \vee (\bar{x}_1 \wedge \bar{x}_3 \wedge x_5 \wedge x_6) \vee (\bar{x}_1 \wedge x_3 \wedge \bar{x}_5 \wedge x_6))$. The length of this formula in general is $2n + (k-2)l - 1$, and we have $l \le 2^{k-1}$.

Notice that, if k is large, we get shorter formulas by writing $f(x)$ as a disjunction $f_1(x) \vee \cdots \vee f_r(x)$, where each f_j has at most $\lceil k/r \rceil$ 1s. Thus

$$L(f) \;\le\; \min_{r \ge 1}\bigl(r - 1 + (2n + \lceil k/r - 2\rceil\, 2^{\lceil k/r - 1\rceil})r\bigr).$$

4. The first inequality is obvious, because a binary tree of depth d has at most $1 + 2 + \cdots + 2^{d-1} = 2^d - 1$ internal nodes.

The hint follows because we can find a minimal subtree of size $\ge \lfloor r/3 \rfloor$. Its size s is at most $1 + 2(\lfloor r/3 \rfloor - 1)$. Therefore we can write $f = (g?\ f_1\colon f_0)$, where g is a subformula of size s; f_0 and f_1 are the formulas of size $r - s - 1$ obtained when that subformula is replaced by 0 and 1, respectively.

Let $d(r) = \max\{\, D(f) \mid L(f) = r \,\}$. Since the mux function has depth 2, and since $\max(s, r-s-1) < \lceil \frac{2r}{3} \rceil$, we have $d(r) \le 2 + d(\lceil \frac{2r}{3} \rceil - 1)$ for $r \ge 3$, and the result follows by induction on r. [*Hawaii International Conf. System Sci.* **4** (1971), 525–527.]

5. Let $g_0 = 0$, $g_1 = x_1$, and $g_j = x_j \wedge (x_{j-1} \vee g_{j-2})$ for $j \ge 2$. Then $F_n = g_n \vee g_{n-1}$, with cost $2n - 2$ and depth n. [These functions g_j also play a prominent role in binary addition; see exercises 42 and 44 for ways to compute them with depth $O(\log n)$.]

6. True: Consider the cases $y = 0$ and $y = 1$.

7. $\hat{x}_5 = x_1 \vee x_4$, $\hat{x}_6 = x_2 \wedge \hat{x}_5$, $\hat{x}_7 = x_1 \vee x_3$, $\hat{x}_8 = \hat{x}_6 \oplus \hat{x}_7$. (The original chain computes the "random" function (6); see exercise 1. The new chain computes the normalization of that function, namely its complement.)

8. The desired truth table consists of blocks of 2^{n-k} 0s alternating with blocks of 2^{n-k} 1s, as in (7). Therefore, if we multiply by $2^{2^{n-k}} + 1$ we get $x_k + (x_k \ll 2^{n-k})$, which is all 1s.

9. When finding $L(f) = \infty$ in step L6, we can store g and h in a record associated with f. Then a recursive procedure will be able to construct a minimum-length formula for f from the respective formulas for g and h.

10. In step L3, use $k = r - 1$ instead of $k = r - 1 - j$. Also change L to D everywhere.

11. The only subtle point is that j should *decrease* in step U3; then we'll never have $\phi(g) \,\&\, \phi(h) \ne 0$ when $j = 0$, so all cases of cost $r - 1$ will be discovered before we begin to look at list $r - 1$.

U1. [Initialize.] Set $U(0) \leftarrow \phi(0) \leftarrow 0$ and $U(f) \leftarrow \infty$ for $1 \le f < 2^{2^n - 1}$. Then set $U(x_k) \leftarrow \phi(x_k) \leftarrow 0$ and put x_k into list 0, as in step L1. Also set

$U(x_j \circ x_k) \leftarrow 1$, set $\phi(x_j \circ x_k)$ to an appropriate bit vector of weight 1, and put $x_j \circ x_k$ into list 1, for $1 \le j < k \le n$ and all five normal operators $\circ$. Finally set $c \leftarrow 2^{2^n-1} - 5\binom{n}{2} - n - 1$.

affine functions

U2. [Loop on r.] Do step U3 for $r = 2, 3, \ldots,$ while $c > 0$.

U3. [Loop on j and k.] Do step U4 for $j = \lfloor (r-1)/2 \rfloor, \lfloor (r-1)/2 \rfloor - 1, \ldots,$ and $k = r - 1 - j$, while $j \ge 0$.

U4. [Loop on g and h.] Do step U5 for all g in list j and all h in list k; if $j = k$, restrict h to functions that *follow* g in list k.

U5. [Loop on f.] If $\phi(g) \,\&\, \phi(h) \ne 0$, set $u \leftarrow r-1$ and $v \leftarrow \phi(g) \,\&\, \phi(h)$; otherwise set $u \leftarrow r$ and $v \leftarrow \phi(g) \mid \phi(h)$. Then do step U6 for $f = g \,\&\, h$, $f = \bar{g} \,\&\, h$, $f = g \,\&\, \bar{h}$, $f = g \mid h$, and $f = g \oplus h$.

U6. [Update $U(f)$ and $\phi(f)$.] If $U(f) = \infty$, set $c \leftarrow c - 1$, $\phi(f) \leftarrow v$, and put f into list u. Otherwise if $U(f) > u$, set $\phi(f) \leftarrow v$ and move f from list $U(f)$ to list u. Otherwise if $U(f) = u$, set $\phi(f) \leftarrow \phi(f) \mid v$. ▌

12. $x_4 = x_1 \oplus x_2$, $x_5 = x_3 \wedge x_2$, $x_6 = x_2 \wedge \bar{x}_4$, $x_7 = x_5 \vee x_6$.

13. $f_5 = 01010101$ (x_3); $f_4 = 01110111$ ($x_2 \vee x_3$); $f_3 = 01110101$ ($(\bar{x}_1 \wedge x_2) \vee x_3$); $f_2 = 00110101$ ($x_1?\ x_3\colon x_2$); $f_1 = 00010111$ ($\langle x_1 x_2 x_3 \rangle$).

14. For $1 \le j \le n$, first compute $t \leftarrow (g \oplus (g \gg 2^{n-j})) \,\&\, x_j$, $t \leftarrow t \oplus (t \ll 2^{n-j})$, where x_j is the truth table (11); then for $1 \le k \le n$ and $k \ne j$, the desired truth table corresponding to $x_j \leftarrow x_j \circ x_k$ is $g \oplus (t \,\&\, ((x_j \circ x_k) \oplus x_j))$.

(The $5n(n-1)$ masks $(x_j \circ x_k) \oplus x_j$ are independent of g and can be computed in advance. The same idea applies if we allow more general computations of the form $x_{j(i)} \leftarrow x_{k(i)} \circ_i x_{l(i)}$, with $5n^2(n-1)$ masks $(x_k \circ x_l) \oplus x_j$.)

15. Remarkably asymmetrical ways to compute symmetrical functions:

(a)	(b)	(c)
$x_1 \leftarrow x_1 \oplus x_2$,	$x_1 \leftarrow x_1 \oplus x_2$,	$x_1 \leftarrow x_1 \oplus x_2$,
$x_3 \leftarrow x_3 \oplus x_4$,	$x_2 \leftarrow x_2 \wedge \bar{x}_1$,	$x_2 \leftarrow x_2 \oplus x_3$,
$x_1 \leftarrow x_1 \oplus x_3$,	$x_3 \leftarrow x_3 \oplus x_4$,	$x_2 \leftarrow x_2 \vee x_1$,
$x_2 \leftarrow x_2 \oplus x_4$,	$x_4 \leftarrow x_4 \wedge x_1$,	$x_1 \leftarrow x_1 \oplus x_4$,
$x_3 \leftarrow x_3 \vee x_2$,	$x_2 \leftarrow \bar{x}_2 \wedge x_3$,	$x_1 \leftarrow x_1 \wedge x_3$,
$x_3 \leftarrow x_3 \wedge \bar{x}_1$.	$x_2 \leftarrow x_2 \oplus x_1$,	$x_2 \leftarrow x_2 \wedge \bar{x}_1$,
	$x_2 \leftarrow x_2 \wedge \bar{x}_4$.	$x_2 \leftarrow x_2 \oplus x_4$.

16. A computation that uses only $\oplus$ and complementation produces nothing but affine functions (see exercise 7.1.1–132). Suppose $f(x) = f(x_1, \ldots, x_n)$ is a non-affine function computable in minimum memory. Then $f(x)$ has the form $g(Ax + c)$ where $g(y_1, y_2, \ldots, y_n) = g(y_1 \wedge y_2, y_2, \ldots, y_n)$, for some nonsingular $n \times n$ matrix A of 0s and 1s, where x and c are column vectors and the vector operations are performed modulo 2; in this formula the matrix A and vector c account for all operations $x_i \leftarrow x_i \oplus x_j$ and/or permutations and complementations of coordinates that occur after the most recent non-affine operation that was performed. We will exploit the fact that $g(0, 0, y_3, \ldots, y_n) = g(1, 0, y_3, \ldots, y_n)$.

Let α and β be the first two rows of A; also let a and b be the first two elements of c. Then if $Ax + c \equiv y$ (modulo 2) we have $y_1 = y_2 = 0$ if and only if $\alpha \cdot x \equiv a$ and $\beta \cdot x \equiv b$. Exactly 2^{n-2} vectors x satisfy this condition, and for all such vectors we have $f(x) = f(x \oplus w)$, where $Aw \equiv (1, 0, \ldots, 0)^T$.

Given α, β, a, b, and w, with $\alpha \ne (0, \ldots, 0)$, $\beta \ne (0, \ldots, 0)$, $\alpha \ne \beta$, and $\alpha \cdot w \equiv 1$ (modulo 2), there are $2^{2^n - 2^{n-2}}$ functions f with the property that $f(x) = f(x \oplus w)$

CNF
canalizing
NAND

whenever $\alpha \cdot x \bmod 2 = a$ and $\beta \cdot x \bmod 2 = b$. Therefore the total number of functions computable in minimum memory is at most 2^{n+1} (for affine functions) plus

$$(2^n-1)(2^n-2)2^2(2^{n-1})(2^{2^n-2^{n-2}}) < 2^{2^n-2^{n-2}+3n+1}.$$

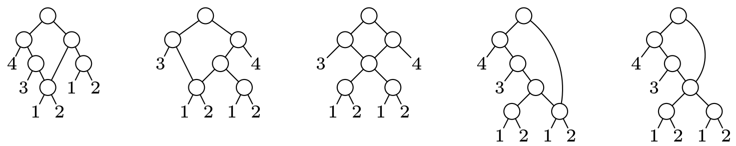
17. Let $f(x_1,\ldots,x_n) = g(x_1,\ldots,x_{n-1}) \oplus (h(x_1,\ldots,x_{n-1}) \wedge x_n)$ as in 7.1.1–(16). Representing h in CNF, form the clauses one by one in x_0 and AND them into x_n, obtaining $h \wedge x_n$. Representing g as a sum (mod 2) of conjunctions, form the successive conjunctions in x_0 and XOR them into x_n when ready.

(It appears to be impossible to evaluate all functions inside of $n+1$ registers if we disallow the non-canalizing operators $\oplus$ and $\equiv$. But $n+2$ registers clearly do suffice, even if we restrict ourselves to the single operator $\overline{\wedge}$.)

18. As mentioned in answer 14, we should extend the text's definition of minimum-memory computation to allow also steps like $x_{j(i)} \leftarrow x_{k(i)} \circ_i x_{l(i)}$, with $k(i) \neq j(i)$ and $l(i) \neq j(i)$, because that will give better results for certain functions that depend on only four of the five variables. Then we find $C_m(f) = (0, 1, \ldots, 13, 14)$ for respectively (2, 2, 5, 20, 93, 389, 1960, 10459, 47604, 135990, 198092, 123590, 21540, 472, 0) classes of functions … leaving 75,908 classes (and 575,963,136 functions) for which $C_m(f) = \infty$ because they cannot be evaluated *at all* in minimum memory. The most interesting function of that kind is probably $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) \vee (x_4 \wedge x_5) \vee (x_5 \wedge x_1)$, which has $C(f) = 7$ but $C_m(f) = \infty$. Another interesting case is $\bigl(((x_1 \vee x_2) \oplus x_3) \vee ((x_2 \vee \bar{x}_4) \wedge x_5)\bigr) \wedge \bigl((x_1 \equiv x_2) \vee x_3 \vee x_4\bigr)$, for which $C(f) = 8$ and $C_m(f) = 13$. One way to evaluate that function in eight steps is $x_6 = x_1 \vee x_2$, $x_7 = x_1 \vee x_4$, $x_8 = x_2 \oplus x_7$, $x_9 = x_3 \oplus x_6$, $x_{10} = x_4 \oplus x_9$, $x_{11} = x_5 \vee x_9$, $x_{12} = x_8 \wedge x_{10}$, $x_{13} = x_{11} \wedge \bar{x}_{12}$.

19. If not, the left and right subtrees of the root must overlap, since case (i) fails. Each variable must occur at least once as a leaf, by hypothesis. At least two variables must occur at least twice as leaves, since case (ii) fails. But we can't have $n+2$ leaves with $r \le n+1$ internal nodes, unless the subtrees fail to overlap.

20. Now Algorithm L (with '$f = g \oplus h$' omitted in step L5) shows that some formulas must have length 15; and even the footprint method of exercise 11 does no better than 14. To get truly minimum chains, the 25 special chains for $r = 6$ in the text must be supplemented by five others that can no longer be ruled out, namely



;

and when $r = (7, 8, 9)$ we must also consider respectively (653, 12387, 225660) additional potential chains that are not special cases of the top-down and bottom-up constructions. Here are the resulting statistics, for comparison with Table 1:

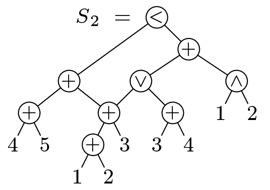
$C_c(f)$	Classes	Functions	$U_c(f)$	Classes	Functions	$L_c(f)$	Classes	Functions	$D_c(f)$	Classes	Functions
0	2	10	0	2	10	0	2	10	0	2	10
1	1	48	1	1	48	1	1	48	1	1	48
2	2	256	2	2	256	2	2	256	2	7	684
3	7	940	3	7	940	3	7	940	3	59	17064
4	9	2336	4	9	2336	4	7	2048	4	151	47634
5	24	6464	5	21	6112	5	20	5248	5	2	96
6	30	10616	6	28	9664	6	23	8672	6	0	0

7	61	18984	7	45	15128	7	37	11768	7	0	0
8	45	17680	8	40	14296	8	27	10592	8	0	0
9	37	7882	9	23	8568	9	33	11536	9	0	0
10	4	320	10	28	5920	10	16	5472	10	0	0
11	0	0	11	6	1504	11	30	6304	11	0	0
12	0	0	12	5	576	12	3	960	12	0	0
13	0	0	13	3	144	13	8	1472	13	0	0
14	0	0	14	2	34	14	2	96	14	0	0
15	0	0	15	0	0	15	4	114	15	0	0

The two function classes of depth 5 are represented by $S_{2,4}(x_1,x_2,x_3,x_4)$ and $x_1 \oplus S_2(x_2,x_3,x_4)$; and those two functions, together with $S_2(x_1,x_2,x_3,x_4)$ and the parity function $S_{1,3}(x_1,x_2,x_3,x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$, have length 15. Also $U_c(S_{2,4}) = U_c(S_{1,3}) = 14$. The four classes of cost 10 are represented by $S_{1,4}(x_1,x_2,x_3,x_4)$, $S_{2,4}(x_1,x_2,x_3,x_4)$, $(x_4?\ x_1 \oplus x_2 \oplus x_3\colon \langle x_1x_2x_3\rangle)$, and $[(x_1x_2x_3x_4)_2 \in \{0,1,4,7,10,13\}]$. (The third of these, incidentally, is the complement of (20), "Harvard's hardest case.")

21. (The authors stated that their table entries "should be regarded only as the most economical operators known to the present writers.")

22. $\nu(x_1x_2x_3x_4x_5) = 3$ if and only if $\nu(x_1x_2x_3x_4) \in \{2,3\}$ and $\nu(x_1x_2x_3x_4x_5)$ is odd. Similarly, $S_2(x_1,x_2,x_3,x_4,x_5) = S_3(\bar{x}_1,\bar{x}_2,\bar{x}_3,\bar{x}_4,\bar{x}_5)$ incorporates $S_{1,2}(x_1,x_2,x_3,x_4)$:



23. We need only consider the 32 normal cases, as in Fig. 9, since the complement of a symmetric function is symmetric. Then we can use reflection, like $S_{1,2}(x) = S_{3,4}(\bar{x})$, possibly together with complementation, like $S_{2,3,4,5}(x) = \bar{S}_{0,1}(x) = \bar{S}_{4,5}(\bar{x})$, to deduce most of the remaining cases. Of course S_1, $S_{1,3,5}$, and $S_{1,2,3,4,5}$ trivially have cost 4. That leaves only $S_{1,2,3,4}(x_1,x_2,x_3,x_4,x_5) = (x_1\oplus x_2)\vee(x_2\oplus x_3)\vee(x_3\oplus x_4)\vee(x_4\oplus x_5)$, which is discussed for general n in exercise 79.

24. As noted in the text, this conjecture holds for $n \le 5$.

25. It is $2^{2^n-1}-n-1$, the number of nontrivial normal functions. (In any normal chain of length r that doesn't include all of these functions, $x_j \circ x_k$ will be a new function for some j and k in the range $1 \le j,k \le n+r$ and some normal binary operator $\circ$; so we can compute a new function with every new step, until we've got them all.)

26. False. For example, if $g = S_{1,3}(x_1,x_2,x_3)$ and $h = S_{2,3}(x_1,x_2,x_3)$, then $C(gh) = 5$ is the cost of a full adder; but $f = S_{2,3}(x_0,x_1,x_2,x_3)$ has cost 6 by Fig. 9.

27. Yes: The operations '$x_2 \leftarrow x_2 \oplus x_1$, $x_1 \leftarrow x_1 \oplus x_3$, $x_1 \leftarrow x_1 \wedge \bar{x}_2$, $x_1 \leftarrow x_1 \oplus x_3$, $x_2 \leftarrow x_2 \oplus x_3$' transform (x_1,x_2,x_3) into (z_1,z_0,x_3).

28. Let $v' = v'' = v \oplus (x\oplus y)$; $u' = ((v\oplus y)\,\overline{\subset}\,(x\oplus y))\oplus u$, $u'' = ((v\oplus y)\vee(x\oplus y))\oplus u$. Thus we can set $u_0 = 0$, $v_0 = x_1$, $u_j = ((v_{j-1}\oplus x_{2j+1})\vee(x_{2j}\oplus x_{2j+1}))\oplus u_{j-1}$ if j is odd, $u_j = ((v_{j-1}\oplus x_{2j+1})\,\overline{\subset}\,(x_{2j}\oplus x_{2j+1}))\oplus u_{j-1}$ if j is even, and $v_j = v_{j-1}\oplus(x_{2j}\oplus x_{2j+1})$, obtaining $(u_jv_j)_2 = (x_1+\cdots+x_{2j+1}) \bmod 4$ for $1 \le j \le \lfloor n/2\rfloor$. Set $x_{n+1} = 0$ if n is even. The function $[(x_1+\cdots+x_n) \bmod 4 = 0] = \bar{u}_{\lfloor n/2\rfloor} \wedge \bar{v}_{\lfloor n/2\rfloor}$ is thereby computed in $\lfloor 5n/2\rfloor - 2$ steps.

symmetric functions
parity function
Harvard
normal
trivial
normal
full adder

This construction is due to L. J. Stockmeyer, who proved that it is nearly optimal. In fact, the result of exercise 80 together with Figs. 9 and 10 shows that it is at most one step longer than a best possible chain, for all $n \geq 5$.

Incidentally, the analogous formula $u''' = ((v \oplus y) \wedge (x \oplus y)) \oplus u$ yields $(u'''v')_2 = ((uv)_2 + x - y) \bmod 4$. The simpler-looking function $((uv)_2 + x + y) \bmod 4$ costs 6, not 5.

29. To get an upper bound, assume that each full adder or half adder increases the depth by 3. If there are a_{jd} bits of weight 2^j and depth $3d$, we schedule at most $\lceil a_{jd}/3 \rceil$ subsequent bits of weights $\{2^j, 2^{j+1}\}$ and depth $3(d+1)$. It follows by induction that $a_{jd} \leq \binom{d}{j} 3^{-d} n + 4$. Hence $a_{jd} \leq 5$ when $d \geq \log_{3/2} n$, and the overall depth is at most $3 \log_{3/2} n + 3$. (Curiously, the actual depth turns out to be exactly 100 when $n = 10^7$.)

30. As usual, let νn denote the sideways addition of the bits in the binary representation of n itself. Then $s(n) = 5n - 2\nu n - 3\lfloor \lg n \rfloor - 3$.

31. After sideways addition in $s(n) < 5n$ steps, an arbitrary function of $(z_{\lfloor \lg n \rfloor}, \ldots, z_0)$ can be evaluated in $\sim 2n/\lg n$ steps at most, by Theorem L. [See O. B. Lupanov, *Doklady Akademii Nauk SSSR* **140** (1961), 322–325.]

32. Bootstrap: First prove by induction on n that $t(n) \leq 2^{n+1}$.

33. False, on a technicality: If, say, $N = \sqrt{n}$, at least n steps are needed. A correct asymptotic formula $N + O(\sqrt{N}\,) + O(n)$ can, however, be proved by first noting that the text's method gives $N + O(\sqrt{N}\,)$ when $N \geq 2^{n-1}$; otherwise, if $\lfloor \lg N \rfloor = n - k - 1$, we can use $O(n)$ operations to AND the quantity $\bar{x}_1 \wedge \cdots \wedge \bar{x}_k$ to the other variables $x_{k+1}, \ldots, x_n$, then proceed with n reduced by k.

(One consequence is that we can compute the symmetric functions $\{S_1, S_2, \ldots, S_n\}$ with cost $s(n) + n + O(\sqrt{n}\,) = 6n + O(\sqrt{n}\,)$ and depth $O(\log n)$.)

34. Say that an *extended* priority encoder has $n + 1 = 2^m$ inputs $x_0 x_1 \ldots x_n$ and $m + 1$ outputs $y_0 y_1 \ldots y_m$, where $y_0 = x_0 \vee x_1 \vee \cdots \vee x_n$. If Q'_m and Q''_m are extended encoders for $x'_0 \ldots x'_n$ and $x''_0 \ldots x''_n$, then Q_{m+1} works for $x'_0 \ldots x'_n x''_0 \ldots x''_n$ if we define $y_0 = y'_0 \vee y''_0$, $y_1 = y''_0$, $y_2 = y_1 ?\ y''_1 \colon y'_1$, $\ldots$, $y_{m+1} = y_1 ?\ y''_m \colon y'_m$. If P'_m is an ordinary priority encoder for $x'_1 \ldots x'_n$, we get P_{m+1} for $x'_1 \ldots x'_n x''_0 \ldots x''_n$ in a similar way.

Starting with $m = 2$ and $y_2 = x_3 \vee (x_1 \wedge \bar{x}_2)$, $y_1 = x_2 \vee x_3$, $y_0 = x_0 \vee x_1 \vee y_1$, this construction yields P_m and Q_m of costs p_m and q_m, where $p_2 = 3$, $q_2 = 5$, and $p_{m+1} = 3m + p_m + q_m$, $q_{m+1} = 3m + 1 + 2q_m$ for $m \geq 2$. Consequently $p_m = q_m - m$ and $q_m = 15 \cdot 2^{m-2} - 3m - 4 \approx 3.75n$.

35. If $n = 2m$, compute $x_1 \wedge x_2$, $\ldots$, $x_{n-1} \wedge x_n$, then recursively form $x_1 \wedge \cdots \wedge x_{2k-2} \wedge x_{2k+1} \wedge \cdots \wedge x_n$ for $1 \leq k \leq m$, and finish in n more steps. If $n = 2m - 1$, use this chain for $n + 1$ elements; three steps can be eliminated by setting $x_{n+1} \leftarrow 1$. [I. Wegener, *The Complexity of Boolean Functions* (1987), exercise 3.25. The same idea can be used with *any* associative and commutative operator in place of $\wedge$.]

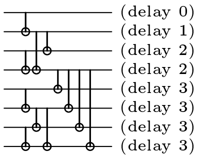
36. Recursively construct $P_n(x_1, \ldots, x_n)$ and $Q_n(x_1, \ldots, x_n)$ as follows, where P_n has optimum depth and Q_n has depth $\leq \lceil \lg n \rceil + 1$: The case $n = 1$ is trivial; otherwise P_n is obtained from $Q'_r(x_1, \ldots, x_r)$ and $P''_s(x_{r+1}, \ldots, x_n)$, where $r = \lceil n/2 \rceil$ and $s = \lfloor n/2 \rfloor$, by setting $y_j = y'_j$ for $1 \leq j \leq r$, $y_j = y'_r \wedge y''_{j-r}$ for $r < j \leq n$. And Q_n is obtained from either $P'_r(x_1 \wedge x_2, \ldots, x_{n-1} \wedge x_n)$ or $P'_r(x_1 \wedge x_2, \ldots, x_{n-2} \wedge x_{n-1}, x_n)$ by setting $y_1 = x_1$, $y_{2j} = y'_j$, $y_{2j+1} = y'_j \wedge x_{2j+1}$ for $1 \leq j < s$, and $y_{2s} = y'_s$, $y_n = y'_r$.

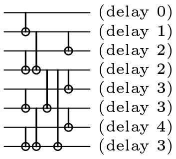
To prove validity we must show also that output y_n of Q_n has depth $\lceil \lg n \rceil$; notice that Q_{2m+1} would fail if we began it with $P'_m(x_1 \wedge x_2, \ldots, x_{2m-1} \wedge x_{2m})$ instead of with $P'_{m+1}(x_1 \wedge x_2, \ldots, x_{2m-1} \wedge x_{2m}, x_{2m+1})$, *except* when m is a power of 2.

Stockmeyer
sideways addition function nu
Lupanov
Bootstrap
symmetric functions
recurrence
Wegener
associative
commutative
Recursively

minimum memory
sorting networks
binary recurrence
Fibonacci number
associative
Gray binary code
threshold function
divide-and-conquer
binary decoder

These calculations can be performed in *minimum memory*, setting $x_{k(i)} \leftarrow x_{j(i)} \wedge x_{k(i)}$ at step i for some indices $j(i) < k(i)$. Thus we can illustrate the construction with diagrams analogous to the diagrams for sorting networks. For example,

P_8 = 

; Q_8 = 

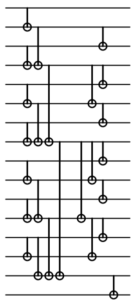
.

The costs p_n and q_n satisfy $p_n = \lfloor n/2 \rfloor + q_{\lceil n/2 \rceil} + p_{\lfloor n/2 \rfloor}$, $q_n = 2\lfloor n/2 \rfloor - 1 + p_{\lceil n/2 \rceil}$ when $n > 1$; for example, $(p_1, \ldots, p_7) = (q_1, \ldots, q_7) = (0, 1, 2, 4, 5, 7, 9)$. Setting $\bar{p}_n = 4n - p_n$ and $\bar{q}_n = 3n - q_n$ leads to simpler formulas, which prove that $p_n < 4n$ and $q_n < 3n$: $\bar{q}_n = \bar{p}_{\lceil n/2 \rceil} + [n \text{ even}]$; $\bar{p}_{4n} = \bar{p}_{2n} + \bar{p}_n + 1$, $\bar{p}_{4n+1} = \bar{p}_{2n} + \bar{p}_{n+1} + 1$, $\bar{p}_{4n+2} = \bar{p}_{2n+1} + \bar{p}_{n+1}$, $\bar{p}_{4n+3} = \bar{p}_{4n+2} + 2$. In particular, $1 + \bar{p}_{2^m} = F_{m+5}$ is a Fibonacci number.

[See *JACM* **27** (1980), 831–834. Slightly better chains are obtained if we use the otherwise-forbidden $P'_{\lfloor n/2 \rfloor}$ construction for Q_n when $n = 2^m + 1$, if we replace P_5 and P_6 by Q_5 and Q_6, and if we then replace $(P_9, P_{10}, P_{11}, P_{17})$ by $(Q_9, Q_{10}, Q_{11}, Q_{17})$.]

Notice that this construction works in general if we replace '$\wedge$' by *any* associative operator. In particular, the sequence of prefixes $x_1 \oplus \cdots \oplus x_k$ for $1 \le k \le n$ defines the conversion from Gray binary code to radix-2 integers, Eq. 7.2.1.1–(10).

37. The case $m = 15$, $n = 16$ is illustrated at the right.



(a) Let $x_{i..j}$ denote the original value of $x_i \wedge \cdots \wedge x_j$. Whenever the algorithm sets $x_k \leftarrow x_j \wedge x_k$, one can show that the previous value of x_k was $x_{j+1..k}$. After step S1, x_k is $x_{f(k)+1..k}$ where $f(k) = k \mathbin{\&} (k-1)$ for $1 \le k < m$ and $f(m) = 0$. After step S2, x_k is $x_{1..k}$ for $1 \le k \le m$.

(b) The cost of S1 is $m - 1$, the cost of S2 is $m - 1 - \lceil \lg m \rceil$, and the cost of S3 is $n - m$. The final delay of x_k is $\lfloor \lg k \rfloor + \nu k - 1$ for $1 \le k < m$, and it is $\lceil \lg m \rceil + k - m$ for $m \le k \le n$. So the maximum delay for $\{x_1, \ldots, x_{m-1}\}$ turns out to be $g(m) = m - 1$ for $m < 4$, $g(m) = \lfloor \lg m \rfloor + \lfloor \lg \frac{m}{3} \rfloor$ for $m \ge 4$. We have $c(m,n) = m + n - 2 - \lceil \lg m \rceil$, $d(m,n) = \max(g(m), \lceil \lg m \rceil + n - m)$. Hence $c(m,n) + d(m,n) = 2n - 2$ whenever $n \ge m + g(m) - \lceil \lg m \rceil$.

(c) A table of values reveals that $d(n) = \lceil \lg n \rceil$ for $n < 8$, and $d(n) = \lfloor \lg(n - \lfloor \lg n \rfloor + 3) \rfloor + \lfloor \lg \frac{2}{3}(n - \lfloor \lg n \rfloor + 3) \rfloor - 1$ for $n \ge 8$. Stating this another way, we have $d(n) > d(n-1) > 0$ if and only if $n = 2^k + k - 3$ or $2^k + 2^{k-1} + k - 3$ for some $k > 1$. The minimum occurs for $m = n$ when $n < 8$; otherwise it occurs for $m = n - \lfloor \frac{2}{3}(n - \lfloor \lg n \rfloor + 3) \rfloor + 2 - [n = 2^k + k - 3 \text{ for some } k]$.

(d) Set $m \leftarrow m(n,d)$, where $m(n, d(n))$ is defined in the previous sentence and $m(n,d) = m(n-1, d-1)$ when $d > d(n)$. [See *J. Algorithms* **7** (1986), 185–201.]

38. (a) From top to bottom, $f_k(x_1, \ldots, x_n)$ is an elementary symmetric function also called the threshold function $S_{\ge k}(x_1, \ldots, x_n)$. (See exercise 5.3.4–28, Eq. 7.1.1–(90).)

(b) After calculating $\{S_1, \ldots, S_n\}$ in $\approx 6n$ steps as in answer 33, we can apply the method of exercise 37 to finish in $2n$ further steps.

But it is more interesting to design a Boolean chain specifically for the computation of the $2^m + 1$ threshold functions $g_k(x_1, \ldots, x_m) = [(x_1 \ldots x_m)_2 \ge k]$ for $0 \le k \le 2^m$. Since $[(x'x'')_2 \ge (y'y'')_2] = [(x')_2 \ge (y')_2 + 1] \vee ([(x')_2 \ge (y')_2] \wedge [(x'')_2 \ge (y'')_2])$, a divide-and-conquer construction analogous to a binary decoder solves this problem with a cost at most $2t(m)$.

Furthermore, if $2^{m-1} \le n < 2^m$, the cost $u(n)$ of computing $\{g_1, \ldots, g_n\}$ by this method turns out to be $2n + O(\sqrt{n}\,)$, and it is quite reasonable when n is small:

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$u(n)$ =	0	1	2	4	7	7	8	12	15	17	19	19	20	21	22	27	32	34	36	36

Starting with sideways addition, we can sort n Boolean values in $s(n) + u(n) \approx 7n$ steps. A sorting network, which costs $2\hat{S}(n)$, is better when $n = 4$ but loses when $n \ge 8$. [See 5.3.4–(11); D. E. Muller and F. P. Preparata, *JACM* **22** (1975), 195–201.]

sideways addition
Muller
Preparata
recursion
recurrence
Sklansky

39. [*IEEE Transactions* **C-29** (1980), 737–738.] The identity

$$M_{r+s}(x_1, \ldots, x_r, x_{r+1}, \ldots, x_{r+s}; y_0, \ldots, y_{2^{r+s}-1}) = M_r(x_1, \ldots, x_r; y'_0, \ldots, y'_{2^r-1}),$$

where $y'_j = \bigvee_{k=0}^{2^s-1}(d_k \wedge y_{2^s j+k})$ and d_k is the kth output of an s-to-2^s decoder applied to $(x_{r+1}, \ldots, x_{r+s})$, shows that $C(M_{r+s}) \le C(M_r) + 2^{r+s} + 2^r(2^s - 1) + t(s)$, where $t(s)$ is the cost (30) of the decoder. The depth is $D(M_{r+s}) = \max\bigl(D_x(M_{r+s}), D_y(M_{r+s})\bigr)$, where D_x and D_y denote the maximum depth of the x and y variables; we have $D_x(M_{r+s}) \le \max\bigl(D_x(M_r), 1 + s + \lceil \lg s \rceil + D_y(M_r)\bigr)$ and $D_y(M_{r+s}) \le 1 + s + D_y(M_r)$.

Taking $r = \lceil m/2 \rceil$ and $s = \lfloor m/2 \rfloor$ yields $C(M_m) \le 2^{m+1} + O(2^{m/2})$, $D_y(M_m) \le m + 1 + \lceil \lg m \rceil$, and $D_x(M_m) \le D_y(M_m) + \lceil \lg m \rceil$.

40. We can, for example, let $f_{nk}(x) = \bigvee_{j=1}^{n+1-k}(l_j(x) \wedge r_{j+k-1}(x))$, where

$$l_j(x) = \begin{cases} x_j, & \text{if } j \bmod k = 0, \\ x_j \wedge l_{j+1}(x), & \text{if } j \bmod k \ne 0, \end{cases} \quad \text{for } 1 \le j \le n - (n \bmod k);$$

$$r_j(x) = \begin{cases} 1, & \text{if } j \bmod k = 0, \\ x_j \wedge r_{j-1}(x), & \text{if } j \bmod k \ne 0, \end{cases} \quad \text{for } k \le j \le n.$$

The cost is $4n - 3k - 3\lfloor \frac{n}{k} \rfloor - \lfloor \frac{n-1}{k} \rfloor + 2 - (n \bmod k)$.

A recursive solution is preferable when n is small or k is small: Observe that

$$f_{nk}(x) = \begin{cases} x_{n-k+1} \wedge \cdots \wedge x_k \wedge \\ \qquad f_{(2n-2k)(n-k)}(x_1, \ldots, x_{n-k}, x_{k+1}, \ldots, x_n), & \text{for } k < n < 2k; \\ f_{\lfloor (n+k)/2 \rfloor k}(x_1, \ldots, x_{\lfloor (n+k)/2 \rfloor}) \vee \\ \qquad f_{\lfloor (n+k-1)/2 \rfloor k}(x_{\lfloor (n-k)/2 \rfloor + 1}, \ldots, x_n), & \text{for } n \ge 2k. \end{cases}$$

The cost of this solution can be shown to equal $n - 1 + \sum_{j=1}^{n-k} \lfloor \lg j \rfloor$ when $k \le n < 2k$, and it lies asymptotically between $(m + \alpha_k - 1)n + O(km)$ and $(m + 2 - 2/\alpha_k)n + O(km)$ as $n \to \infty$, where $m = \lfloor \lg k \rfloor$ and $1 < \alpha_k = (k+1)/2^m \le 2$.

A marriage of these methods is better yet; the optimum cost is unknown.

41. Let $c(m)$ be the cost of computing both $(x)_2 + (y)_2$ and $(x)_2 + (y)_2 + 1$ by the conditional-sum method when x and y have $n = 2^m$ bits, and let $c'(m)$ be the cost of the simpler problem of computing just $(x)_2 + (y)_2$. Then $c(m+1) = 2c(m) + 6 \cdot 2^m + 2$, $c'(m+1) = c(m) + c'(m) + 3 \cdot 2^m + 1$. (Bit z_n of the sum costs 1; but bits z_k for $n < k \le 2n + 1$ cost 3, because they have the form $c?\ a_k\colon b_k$ where c is a carry bit.) If we start with $n = 1$ and $c(0) = 3$, $c'(0) = 2$, the solution is $c(m) = (3m + 5)2^m - 2$, $c'(m) = (3m + 2)2^m - m$. But improved constructions for the case $n = 2$ allow us to start with $c(1) = 11$ and $c'(1) = 7$; then the solution is $c(m) = (3m + \frac{7}{2})2^m - 2$, $c'(m) = (3m + \frac{1}{2})2^m - m + 1$. In either case the depth is $2m + 1$. [See J. Sklansky, *IRE Transactions* **EC-9** (1960), 226–231.]

42. (a) Since $\langle x_k y_k c_k\rangle = u_k \vee (v_k \wedge c_k)$, we can use (26) and induction.

(b) Notice that $U_k^{k+1} = u_k$ and $V_k^{k+1} = v_k$; use induction on $j - i$. [See A. Weinberger and J. L. Smith, *IRE Transactions* **EC-5** (1956), 65–73; R. P. Brent and H. T. Kung, *IEEE Transactions* **C-31** (1982), 260–264.]

(c) First, for $l = 1, 2, \ldots, m-1$, and for $1 \le k \le n$, compute V_i^k for all multiples i of $h(l)$ in the range $k_l \ge i \ge k_{l+1}$, where $k_l = h(l)\lfloor (k-1)/h(l)\rfloor$ denotes the largest multiple of $h(l)$ that is less than k. For example, when $l = 2$ and $k = 99$, we compute V_{96}^{99}, $V_{88}^{99} = V_{96}^{99} \wedge V_{88}^{96}$, $V_{80}^{99} = V_{88}^{99} \wedge V_{80}^{88}$, $\ldots$, $V_{64}^{99} = V_{72}^{99} \wedge V_{64}^{72}$; this is a prefix computation using the values V_{96}^{99}, V_{88}^{96}, V_{80}^{88}, $\ldots$, V_{64}^{72} that were computed when $l = 2$. Using the method of exercise 36, step l adds at most l levels to the depth, and it requires a total of $(p_1 + p_2 + \cdots + p_{2^l})n/2^l = O(2^l n)$ gates.

Then, again for $l = 1, 2, \ldots, m-1$, and for $1 \le k \le n$, compute U_i^k for $i = k_{l+1}$, using the "unrolled" formula

$$U_{k_{l+1}}^k \;=\; U_{k_l}^k \vee \bigvee_{\substack{k_l > j \ge k_{l+1} \\ h(l)\backslash j}} (V_{j+h(l)}^k \wedge U_j^{j+h(l)}).$$

For example, the unrolled formula when $l = 3$ and $k = 99$ is

$$U_{64}^{99} \;=\; U_{96}^{99} \vee (V_{96}^{99} \wedge U_{88}^{96}) \vee (V_{88}^{99} \wedge U_{80}^{88}) \vee (V_{80}^{99} \wedge U_{72}^{80}) \vee (V_{72}^{99} \wedge U_{64}^{72}).$$

Every such U_i^k is a union of at most 2^l terms, so it can be computed with depth $\le l$ in addition to the depth of each term. The total cost of this phase for $1 \le k \le n$ is $(0 + 2 + 4 + \cdots + (2^l - 2))n/2^l = O(2^l n)$.

The overall cost to compute all necessary U's and V's is therefore $\sum_{l=1}^{m-1} O(2^l n) = O(2^m n)$. (Furthermore the quantities V_0^k aren't actually needed, so we save the cost of $\sum_{l=1}^{m-1} h(l)p_{2^l}$ gates.) For example, when $m = (2, 3, 4, 5)$ we obtain Boolean chains for the addition of $(2, 8, 64, 1024)$-bit numbers, respectively, with overall depths $(3, 7, 11, 16)$ and costs $(7, 64, 1254, 48470)$.

[This construction is due to V. M. Khrapchenko, *Problemy Kibernetiki* **19** (1967), 107–122, who also showed how to combine it with other methods so that the overall cost will be $O(n)$ while still achieving depth $\lg n + O(\sqrt{\log n}\,)$. However, his combined method is purely of theoretical interest, because it requires $n > 2^{64}$ before the depth becomes less than $2\lg n$. Another way to achieve small depth using the recurrences in (b) can be based on the Fibonacci numbers: The Fibonacci method computes the carries with depth $\log_\phi n + O(1) \approx 1.44 \lg n$ and cost $O(n \log n)$. For example, it yields chains for binary addition with the following characteristics:

n =	4	8	16	32	64	128	256	512	1024
depth	6	7	9	10	12	13	15	16	18
cost	24	71	186	467	1125	2648	6102	13775	30861

See D. E. Knuth, *The Stanford GraphBase* (1994), 276–279.

Charles Babbage found an ingenious mechanical solution to the analogous problem for addition in radix 10, claiming that his design would be able to add numbers of arbitrary precision in constant time; for this to work he would have needed idealized, rigid components with vanishing clearances. See H. P. Babbage, *Babbage's Calculating Engines* (1889), 334–335. Curiously, an equivalent idea works fine with physical transistors, although it cannot be expressed in terms of Boolean chains; see P. M. Fenwick, *Comp. J.* **30** (1987), 77–79.]

Weinberger
Smith
Brent
Kung
Khrapchenko
Fibonacci numbers
Knuth
Stanford GraphBase
Babbage
Babbage, HP
Fenwick

43. (a) Let $A = B = Q = \{0,1\}$ and $q_0 = 0$. Define $c(q,a) = d(q,a) = \bar{q} \wedge a$.

(b) The key idea is to construct the functions $d_1(q) \ldots d_{n-1}(q)$, where $d_1(q) = d(q,a_1)$ and $d_{j+1}(q) = d(d_j(q), a_j)$. In other words, $d_1 = d^{(a_1)}$ and $d_{j+1} = d_j \circ d^{(a_j)}$, where $d^{(a)}$ is the function that takes $q \mapsto d(q,a)$ and where $\circ$ denotes composition of functions. Each function d_j can be encoded in binary notation, and $\circ$ is an associative operation on these binary representations. Hence the functions $d_1 d_2 \ldots d_{n-1}$ are the prefixes $d^{(a_1)}$, $d^{(a_1)} \circ d^{(a_2)}$, $\ldots$, $d^{(a_1)} \circ \cdots \circ d^{(a_{n-1})}$; and $q_1 q_2 \ldots q_n = q_0 d_1(q_0) \ldots d_{n-1}(q_0)$.

(c) Represent a function $f(q)$ by its truth table $f_0 f_1$. Then the composition $f_0 f_1 \circ g_0 g_1$ is $h_0 h_1$, where the functions $h_0 = f_0?\, g_1\colon g_0$ and $h_1 = f_1?\, g_1\colon g_0$ are muxes that can each be computed with cost 3 and depth 2. (The combined cost $C(h_0 h_1)$ is only 5, but we are trying to keep the depth small.) The truth table for $d^{(a)}$ is $a0$. Using exercise 36, we can therefore compute the truth tables $d_{10} d_{11} d_{20} d_{21} \ldots d_{(n-1)0} d_{(n-1)1}$ with cost $\le 6p_{n-1} < 24n$ and depth $\le 2\lceil \lg(n-1) \rceil$; then $b_j = \bar{q}_j \wedge a_j = \bar{d}_{(j-1)0} \wedge a_j$. (These cost estimates are quite conservative; substantial simplifications arise because of the 0s in the initial truth tables of $d^{(a_j)}$ and because many of the intermediate values d_{j1} are never used. For example, when $n = 5$ the actual cost is only 10, not $6p_4 + 4 = 28$; the actual depth is 4, not $1 + 2\lceil \lg 4 \rceil = 5$.)

44. The inputs may be regarded as the string $x_0 y_0\, x_1 y_1\, \ldots\, x_{n-1} y_{n-1}$ whose elements belong to the four-letter alphabet $A = \{00, 01, 10, 11\}$; there are two states $Q = \{0,1\}$, representing a possible carry bit, with $q_0 = 0$; the output alphabet is $B = \{0,1\}$; and we have $c(q,xy) = q \oplus x \oplus y$, $d(q,xy) = \langle qxy \rangle$. In this case, therefore, the finite state transducer is essentially described by a full adder.

Only three of the four possible functions of q occur when we compose the mappings $d^{(xy)}$. We can encode them as $u \vee (q \wedge v)$. The initial functions $d^{(xy)}$ have $u = x \wedge y$, $v = x \oplus y$; and the composition $(uv) \circ (u'v')$ is $u''v''$, where $u'' = u' \vee (v' \wedge u)$ and $v'' = v \wedge v'$.

When $n = 4$, for example, the chain has the following form, using the notation of exercise 42: $U_k^{k+1} = x_k \wedge y_k$, $V_k^{k+1} = x_k \oplus y_k$, for $0 \le k < 4$; $U_0^2 = U_1^2 \vee (V_1^2 \wedge U_0^1)$, $U_2^4 = U_3^4 \vee (V_3^4 \wedge U_2^3)$, $V_2^4 = V_2^3 \wedge V_3^4$; $U_0^3 = U_2^3 \vee (V_2^3 \wedge U_0^2)$, $U_0^4 = U_2^4 \vee (V_2^4 \wedge U_0^2)$; $z_0 = V_0^1$, $z_1 = U_0^1 \oplus V_1^2$, $z_2 = U_0^2 \oplus V_2^3$, $z_3 = U_0^3 \oplus V_3^4$, $z_4 = U_0^4$. The total cost is 20, and the maximum depth is 5.

In general the cost will be $2n + 3p_n$ in the notation of exercise 36, because we need $2n$ gates for the initial u's and v's, then $3p_n$ gates for the prefix computation; the $n-1$ additional gates needed to form z_j for $0 < j < n$ are compensated by the fact that we need not compute V_0^j for $1 < j \le n$. Therefore the total cost is $14 \cdot 2^m - 3F_{m+5} + 3$, clearly superior to the conditional-sum method (which has the same depth $2m+1$):

$n =$	2	4	8	16	32	64	128	256	512	1024
cost of conditional-sum chain	7	25	74	197	492	1179	2746	6265	14072	31223
cost of Ladner–Fischer chain	7	20	52	125	286	632	1363	2888	6040	12509

[George Boole introduced his Algebra in order to show that logic can be understood in terms of arithmetic. Eventually logic became so well understood, the situation was reversed: People like Shannon and Zuse began in the 1930s to design circuits for arithmetic in terms of logic, and since then many approaches to the problem of parallel addition have been discovered. The first Boolean chains of cost $O(n)$ and depth $O(\log n)$ were devised by Yu. P. Ofman, *Doklady Akademii Nauk SSSR* **145** (1962), 48–51. His chains were similar to the construction above, but the depth was approximately $4m$.]

45. That argument would indeed be simpler, but it wouldn't be strong enough to prove the desired result. (Many chains with steps of fan-out 0 inflate the simpler estimate.)

composition of functions
associative
truth table
mux
multiplex operation
carry
full adder
Boole
Shannon
Zuse
parallel addition
Ofman
fan-out

The text's permutation-enhanced proof technique was introduced by J. E. Savage in his book *The Complexity of Computing* (New York: Wiley, 1976), Theorem 3.4.1.

46. When $r = 2^n/n + O(1)$ we have $\ln(2^{2r+1}(n+r-1)^{2r}/(r-1)!) = r\ln r + (1+\ln 4)r + O(n) = (2^n/n)(n\ln 2 - \ln n + 1 + \ln 4) + O(n)$. So $\alpha(n) \le (n/(4e))^{-2^n/n+O(n/\log n)}$, which approaches zero quite rapidly indeed when $n > 4e$.

(In fact, (32) gives $\alpha(11) < 7.6\times 10^7$, $\alpha(12) < 4.2\times 10^{-6}$, $\alpha(13) < 1.2\times 10^{-38}$.)

47. Restrict permutations to the $(r-m)!$ cases where $i\pi = i$ for $1 \le i \le n$ and $(n+r+1-k)\pi$ is the kth output. Then we get $(r-m)!\,c(m,n,r) \le 2^{2r+1}(n+r-1)^{2r}$ in place of (32). Hence, as in exercise 46, almost all such functions have cost exceeding $2^n m/(n+\lg m)$ when $m = O(2^n/n^2)$.

48. (a) Not surprisingly, this lower bound on $C(n)$ is rather crude when n is small:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$r(n)$	1	1	2	3	5	9	16	29	54	99	184	343	639	1196	2246	4229

(b) The bootstrap method (see *Concrete Mathematics* §9.4) yields

$$r(n) = \frac{2^n}{n}\Bigl(1 + \frac{\lg n - 2 - 1/\ln 2}{n} + O\Bigl(\frac{\log n}{n^2}\Bigr)\Bigr).$$

49. The number of normal Boolean functions that can be represented by a formula of length $\le r$ is at most $5^r n^{r+1} g_r$, where g_r is the number of oriented binary trees with r internal nodes. Set $r = 2^n/\lg n - 2^{n+2}/(\lg n)^2$ in this formula and divide by 2^{2^n-1} to get an upper bound on the fraction of functions with $L(f) \le r$. The result rapidly approaches zero, by exercise 2.3.4.4–7, because it is $O((5\alpha/16)^{2^n/\lg n})$ where $\alpha \approx 2.483$.

[J. Riordan and C. E. Shannon obtained a similar lower bound for series-parallel switching networks in *J. Math. and Physics* **21** (1942), 83–93; such networks are equivalent to formulas in which only canalizing operators are used. R. E. Krichevsky obtained more general results in *Problemy Kibernetiki* **2** (1959), 123–138, and O. B. Lupanov gave an asymptotically matching upper bound in *Prob. Kibernetiki* **3** (1960), 61–80.]

50. (a) Using subcube notation as in exercise 7.1.1–30, the prime implicants are 00001∗, (0001∗1), 0100∗1, 0111∗1, 1010∗1, 101∗11, 00∗011, 00∗101, (01∗111), 11∗101, (0∗1101), (1∗0101), 1∗1011, 0∗0∗11, ∗00101, (∗01011), (∗11101), where the parenthesized subcubes are omitted in a shortest DNF. (b) Similarly, the prime clauses and a shortest CNF are given by 00111∗, 01010∗, 10110∗, 0110∗∗, 00∗00∗, 11∗00∗, 11∗11∗, (0∗100∗), (1∗00∗∗), 1∗0∗1∗, (1∗∗∗∗0), ∗0000∗, (∗1100∗), ∗1∗∗∗0, ∗∗1∗∗0, ∗∗∗1∗0, and (∗∗∗∗00). (Thus the CNF is $(x_1\vee x_2\vee \bar x_3\vee \bar x_4\vee \bar x_5)\wedge(x_1\vee \bar x_2\vee x_3\vee \bar x_4\vee x_5)\wedge\cdots\wedge(\bar x_4\vee x_6)$.)

51. $f = \bigl([x_5x_6\in\{01\}]\wedge[(x_1x_2x_3x_4)_2\in\{1,3,4,7,9,10,13,15\}]\bigr)\vee\bigl([x_5x_6\in\{10,11\}]\wedge [x_1x_2x_3x_4=0000]\bigr)\vee\bigl([x_5x_6\in\{11\}]\wedge[(x_1x_2x_3x_4)_2\in\{1,2,4,5,7,10,11,14\}]\bigr)$.

52. The small-n results are quite different from those that work asymptotically:

n	k	l	(38)	n	k	l	(38)	n	k	l	(38)	n	k	l	(38)
5	2	2	39	8	3	2	175	11	4	4	803	14	5	5	4045
6	2	2	67	9	3	2	279	12	4	3	1329	15	5	5	7141
7	2	1	109	10	4	4	471	13	5	6	2355	16	5	4	12431

(Optimizations like the fact that $[x_1x_2\in\{00,01\}] = \bar x_1$ usually reduce the cost further.)

53. First note that $2^k/l \le n - 3\lg n$, hence $m_i \le n - 3\lg n + 1$ and $2^{m_i} = O(2^n/n^3)$. Also $l = O(n)$ and $t(n-k) = O(2^n/n^2)$. So (38) reduces to $l\cdot 2^{n-k} + O(2^n/n^2) = 2^n/(n-3\lg n) + O(2^n/n^2)$.

Savage
bootstrap method
oriented binary trees
Riordan
Shannon
series-parallel switching networks
canalizing operators
Krichevsky
Lupanov
subcube

greedy-footprint heuristic
DNF
CNF
hexadecimal notation

54. The greedy-footprint heuristic gives a chain of length 15:

$$\begin{array}{lll}
x_5 = x_1 \oplus x_2, & x_{10} = x_4 \oplus x_5, & f_2 = x_{15} = \bar{x}_5 \wedge x_8, \\
x_6 = x_2 \oplus x_3, & x_{11} = x_7 \vee x_{10}, & f_3 = x_{16} = x_4 \wedge \bar{x}_{12}, \\
x_7 = x_1 \wedge \bar{x}_3, & x_{12} = x_6 \oplus x_{11}, & f_4 = x_{17} = x_1 \wedge x_8, \\
x_8 = x_4 \wedge \bar{x}_6, & x_{13} = \bar{x}_{10} \wedge x_{12}, & f_5 = x_{18} = x_7 \wedge x_9, \\
x_9 = x_4 \wedge \bar{x}_5, & f_1 = x_{14} = x_6 \wedge \bar{x}_{11}, & f_6 = x_{19} = \bar{x}_3 \wedge x_{13}.
\end{array}$$

The minterm-first method corresponds to a chain of length 22, after we remove steps that are never used:

$$\begin{array}{lll}
x_5 = \bar{x}_1 \wedge \bar{x}_2, & x_{13} = x_5 \wedge x_{10}, & x_{20} = x_8 \wedge x_{11}, \\
x_6 = \bar{x}_1 \wedge x_2, & x_{14} = x_5 \wedge x_{11}, & f_6 = x_{21} = x_{15} \vee x_{18}, \\
x_7 = x_1 \wedge \bar{x}_2, & x_{15} = x_6 \wedge x_9, & f_1 = x_{22} = x_{13} \vee x_{21}, \\
x_8 = x_1 \wedge x_2, & x_{16} = x_6 \wedge x_{11}, & f_2 = x_{23} = x_{12} \vee x_{20}, \\
x_9 = \bar{x}_3 \wedge x_4, & x_{17} = x_7 \wedge x_9, & x_{24} = x_{14} \vee x_{16}, \\
x_{10} = x_3 \wedge \bar{x}_4, & x_{18} = x_7 \wedge x_{11}, & f_3 = x_{25} = x_{24} \vee x_{19}, \\
x_{11} = x_3 \wedge x_4, & f_5 = x_{19} = x_8 \wedge x_9, & f_4 = x_{26} = x_{17} \vee x_{20}. \\
x_{12} = x_5 \wedge x_9, & &
\end{array}$$

(The distributive law could replace the computation of x_{14}, x_{16}, and x_{24} by two steps.)

Incidentally, the three functions in the answer to exercise 51 can be computed in only ten steps:

$$\begin{array}{lll}
x_5 = x_2 \vee x_4, & f_3 = x_9 = x_6 \oplus x_8, & x_{12} = x_2 \oplus x_3, \\
x_6 = \bar{x}_1 \wedge x_5, & x_{10} = x_1 \oplus x_8, & x_{13} = \bar{x}_{10} \wedge x_{12}, \\
x_7 = x_2 \wedge x_4, & \bar{f}_2 = x_{11} = x_9 \vee x_{10}, & f_1 = x_{14} = x_4 \oplus x_{13}. \\
x_8 = x_3 \wedge \bar{x}_7, & &
\end{array}$$

55. The optimum two-level DNF and CNF representations in answer 50 cost 53 and 43, respectively. Formula (37) costs 30, when optimized as in exercise 54. The alternative in exercise 51 costs only 17. But the catalog of optimum five-variable chains suggests

$$\begin{array}{llll}
x_7 = \bar{x}_1 \wedge x_2, & x_{11} = x_5 \wedge x_{10}, & x_{15} = x_{13} \oplus x_{14}, & x_{18} = \bar{x}_4 \wedge x_{17}, \\
x_8 = x_3 \oplus x_7, & x_{12} = x_5 \vee x_{10}, & x_{16} = x_5 \wedge \bar{x}_{10}, & x_{19} = x_6 \wedge x_{15}, \\
x_9 = x_2 \wedge x_8, & x_{13} = x_4 \wedge \bar{x}_{11}, & x_{17} = \bar{x}_3 \wedge x_{16}, & x_{20} = x_{18} \vee x_{19}, \\
x_{10} = x_1 \oplus x_9, & x_{14} = x_7 \wedge x_{12}, & &
\end{array}$$

for this six-variable function. Is there a better way?

56. If we care about at most two values, the function can be either constant or x_j or $\bar{x}_j$.

57. The truth tables for x_5 through x_{17}, in hexadecimal notation, are respectively `0ff0`, `2222`, `33cc`, `0d0d`, `7777`, `5d5d`, `3ec1`, `6b94`, `4914`, `4804`, `060b`, `2020`, `7007`. So we get

1010 = , 1011 = , 1012 = , 1013 = , 1014 = , 1015 = .

58. The truth tables of all cost-7 functions with exactly eight 1s in their truth tables are equivalent to either `0779`, `169b`, or `179a`. Combining these in all possible ways yields 9656 solutions that are distinct under permutation and/or complementation of $\{x_1, x_2, x_3, x_4\}$ as well as under permutation and/or complementation of $\{f_1, f_2, f_3, f_4\}$.

Lee
Lupanov
Nechiporuk
Stein
Pippenger
tic-tac-toe
magic square
Lo Shu magic sq
Berlekamp
Conway
Guy

59. The greedy-footprint heuristic produces the following 17-step chain:

$$\begin{array}{lll} x_5 = x_1 \vee x_4, & x_{11} = x_8 \vee x_9, & x_{17} = \bar{x}_2 \wedge x_3, \\ x_6 = x_1 \oplus x_3, & x_{12} = x_1 \oplus x_{11}, & f_1 = x_{18} = x_{13} \oplus x_{15}, \\ x_7 = x_2 \oplus x_4, & x_{13} = x_5 \wedge \bar{x}_9, & f_2 = x_{19} = x_{11} \wedge \bar{x}_{16}, \\ x_8 = \bar{x}_4 \wedge x_6, & x_{14} = x_5 \wedge x_{12}, & f_3 = x_{20} = x_{12} \oplus x_{17}, \\ x_9 = x_3 \oplus x_7, & x_{15} = x_2 \wedge x_6, & f_4 = x_{21} = x_{10} \wedge \bar{x}_{14}. \\ x_{10} = x_2 \vee x_3, & x_{16} = x_2 \wedge \bar{x}_6, & \end{array}$$

The initial functions all have large footprints, so we can't achieve $C(f_1f_2f_3f_4) = 28$; but a slightly more difficult S-box probably does exist.

60. One way is $u_1 = x_1 \oplus y_1$, $u_2 = x_2 \oplus y_2$, $v_1 = y_2 \oplus u_1$, $v_2 = y_1 \oplus u_2$, $z_1 = v_1 \wedge \bar{u}_2$, $z_2 = v_2 \wedge \bar{u}_1$.

61. Viewing these partial functions of six variables as 4×16 truth tables, with rows governed by x_1y_1, our knowledge of 4-bit functions suggests good ways to compute the rows and leads to the following 25-step solution: $t_1 = x_2 \wedge y_2$, $t_2 = x_3 \wedge y_3$, $t_3 = x_2 \vee y_2$, $t_4 = x_3 \vee y_3$, $t_5 = t_1 \oplus t_2$, $t_6 = t_1 \vee t_2$, $t_7 = t_4 \wedge \bar{t}_5$, $t_8 = t_3 \oplus t_6$, $t_9 = x_2 \oplus y_2$, $t_{10} = t_4 \oplus t_9$, $t_{11} = t_5 \wedge \bar{t}_{10}$, $t_{12} = t_3 \oplus t_4$, $t_{13} = x_1 \vee y_1$, $t_{14} = t_8 \oplus t_{12}$, $t_{15} = t_{13} \wedge \bar{t}_{14}$, $t_{16} = t_4 \oplus t_7$, $t_{17} = t_{13} \wedge \bar{t}_{16}$, $t_{18} = t_3 \vee t_4$, $t_{19} = x_1 \oplus y_1$, $t_{20} = t_{19} \wedge \bar{t}_{18}$, $t_{21} = t_8 \oplus t_{15}$, $t_{22} = t_7 \oplus t_{17}$, $z_1 = t_{11} \vee t_{20}$, $z_2 = t_{21} \wedge \bar{t}_{20}$, $z_3 = t_{22} \wedge \bar{t}_{20}$. (Is there a better way? Gilbert Lee has found a 17-step solution if the inputs are represented by 000, 001, 011, 101, and 111.)

62. There are $\binom{2^n}{2^n d} 2^{2^n c}$ such functions, at most $\binom{2^n}{2^n d} t(n,r)$ of which have cost $\le r$. So we can argue as in exercise 46 to conclude from (32) that the fraction with cost $\le r = \lfloor 2^n c/n \rfloor$ is at most $2^{2r+1-2^n c}(n+r-1)^{2r}/(r-1)! = 2^{-r \lg n + O(r)}$.

63. [*Problemy Kibernetiki* **21** (1969), 215–226.] Put the truth table in a $2^k \times 2^{n-k}$ array as in Lupanov's method, and suppose there are c_j cares in column j, for $0 \le j < 2^{n-k}$. Break that column into $\lfloor c_j/m \rfloor$ subcolumns that each have m cares, plus a possibly empty subcolumn at the bottom that contains fewer than m of them. The hint tells us that at most 2^{m+k} column vectors suffice to match the 0s and 1s of every subcolumn that has a specified top row i_0 and bottom row i_1. With $O(m2^{m+3k})$ operations we can therefore construct $O(2^{m+3k})$ functions $g_t(x_1, \ldots, x_k)$ from the minterms of $\{x_1, \ldots, x_k\}$, so that every subcolumn matches some type t. And for every type t we can construct functions $h_t(x_{k+1}, \ldots, x_n)$ from the minterms of $\{x_{k+1}, \ldots, x_n\}$, specifying the columns that match t; the cost is at most $\sum_j(\lfloor c_j/m \rfloor + 1) \le 2^n c/m + 2^{n-k}$. Finally, $f = \bigvee_t (g_t \wedge h_t)$ requires $O(2^{m+3k})$ additional steps. Choosing $k = \lfloor 2 \lg n \rfloor$ and $m = \lceil n - 9 \lg n \rceil$ makes the total cost at most $(2^n c/n)(1 + 9n^{-1} \lg n + O(n^{-1}))$.

Of course we need to prove the hint, which is due to E. I. Nechiporuk [*Doklady Akad. Nauk SSSR* **163** (1965), 40–42]. In fact, $2^m(1 + \lceil k \ln 2 \rceil)$ vectors suffice (see S. K. Stein, *J. Combinatorial Theory* **A16** (1974), 391–397): If we choose $q = 2^m \lceil k \ln 2 \rceil$ vectors at random, not necessarily distinct, the expected number of untouched subcubes is $\binom{k}{m} 2^m (1 - 2^{-m})^q < \binom{k}{m} 2^m e^{-q2^{-m}} < 2^m$. (An explicit construction would be nicer.)

For extensive generalizations — tolerating a percentage of errors and specifying the density of 1s — see N. Pippenger, *Mathematical Systems Theory* **10** (1977), 129–167.

64. It's exactly the game of tic-tac-toe, if we number the cells $\begin{array}{|c|c|c|}6&1&8\\7&5&3\\2&9&4\end{array}$ as in an ancient Chinese magic square. [Berlekamp, Conway, and Guy use this numbering scheme to present a complete analysis of tic-tac-toe in their book *Winning Ways* **3** (2003), 732–736.]

65. One solution is to replace the "defending" moves d_j by "attacking" moves a_j and "counterattacking" moves c_j, and to include them only for corner cells $j \in \{1, 3, 9, 7\}$.

Let $j \cdot k = (jk) \bmod 10$; then

$$\begin{matrix} j\cdot 1 & j\cdot 2 & j\cdot 3 \\ j\cdot 4 & j\cdot 5 & j\cdot 6 \\ j\cdot 7 & j\cdot 8 & j\cdot 9 \end{matrix}$$

gives us another way to look at the tic-tac-toe diagram, when j is a corner, because $j \perp 10$. The precise definition of a_j and c_j is then

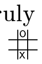
$$\begin{aligned} a_j &= m_j \wedge \bigl(\bigl(x_{j\cdot 3} \wedge \beta_{(j\cdot 8)(j\cdot 9)} \wedge (o_{j\cdot 4} \oplus o_{j\cdot 6})\bigr) \vee \bigl(x_{j\cdot 7} \wedge \beta_{(j\cdot 6)(j\cdot 9)} \wedge (o_{j\cdot 2} \oplus o_{j\cdot 8})\bigr) \\ &\qquad\qquad \vee \bigl(m_{j\cdot 9} \wedge \bigl(\bigl(m_{j\cdot 8} \wedge x_{j\cdot 2} \wedge \overline{(o_{j\cdot 3} \oplus o_{j\cdot 6})}\bigr) \vee \bigl(m_{j\cdot 6} \wedge x_{j\cdot 4} \wedge \overline{(o_{j\cdot 7} \oplus o_{j\cdot 8})}\bigr)\bigr)\bigr)\bigr); \\ c_j &= d_j \wedge \overline{(x_{j\cdot 6} \wedge o_{j\cdot 7})} \wedge \overline{(x_{j\cdot 8} \wedge o_{j\cdot 3})} \wedge \bar{d}_{j\cdot 9}; \end{aligned}$$

here $d_j = m_j \wedge \beta_{(j\cdot 2)(j\cdot 3)} \wedge \beta_{(j\cdot 4)(j\cdot 7)}$ takes the place of (51). We also define

$$\begin{aligned} u &= (x_1 \oplus x_3) \oplus (x_7 \oplus x_9), \\ v &= (o_1 \oplus o_3) \oplus (o_7 \oplus o_9), \\ t &= m_2 \wedge m_6 \wedge m_8 \wedge m_4 \wedge (u \vee \bar{v}), \end{aligned} \qquad z_j = \begin{cases} m_j \wedge \bar{t}, & \text{if } j = 5, \\ m_j \wedge \bar{d}_{j\cdot 9}, & \text{if } j \in \{1,3,9,7\}, \\ m_j, & \text{if } j \in \{2,6,8,4\}, \end{cases}$$

in order to cover a few more exceptional cases. Finally the sequence of rank-ordered moves $d_5 d_1 d_3 d_9 d_7 d_2 d_6 d_8 d_4 m_5 m_1 m_3 m_9 m_7 m_2 m_6 m_8 m_4$ in (53) is replaced by the sequence $a_1 a_3 a_9 a_7 c_1 c_3 c_9 c_7 z_5 z_1 z_3 z_9 z_7 z_2 z_6 z_8 z_4$; and we replace $(d_j \wedge \bar{d}'_j) \vee (m_j \wedge \bar{m}'_j)$ in (55) by $(a_j \wedge \bar{a}'_j) \vee (c_j \wedge \bar{c}'_j) \vee (z_j \wedge \bar{z}'_j)$ when j is a corner cell, otherwise simply by $(z_j \wedge \bar{z}'_j)$.

(Notice that this machine is required to move correctly from *all* legal positions, even when those positions couldn't arise after the machine had made X's earlier moves. We essentially allow humans to play the game until they ask the machine for advice. Otherwise great simplifications would be possible. For example, if X always goes first, it could grab the center cell and eliminate a huge number of future possibilities; fewer than $8 \times 6 \times 4 \times 2 = 384$ games could arise. Even if O goes first, there are fewer than $9 \times 7 \times 5 \times 3 = 945$ possible scenarios against a fixed strategy. In fact, the actual number of different games with the strategy defined here turns out to be $76 + 457$, of which $72 + 328$ are won by the machine and the rest belong to the cat.)

66. The Boolean chain in the previous answer fulfills its mission of making correct moves from all 4520 legal positions, where correctness was essentially defined to mean that the worst-case final outcome is maximized. But a truly great tic-tac-toe player would do things differently. For example, from position  the machine takes the center, , and O probably draws by playing in a corner. But moving to  would give O only two chances to avoid defeat. [See Martin Gardner, *Hexaflexagons and Other Mathematical Diversions*, Chapter 4.]

Furthermore the best move from a position like  is to  instead of winning immediately; then if the reply is , move to . That way you still win, but without humiliating your opponent so badly.

Finally, even the concept of a single "best move" is flawed, because a good player will choose different moves in different games (as Babbage observed).

It might be thought that programing a digital computer to play ticktacktoe, or designing special circuits for a ticktacktoe machine, would be simple. This is true unless your aim is to construct a master robot that will win the maximum number of games against inexperienced players.

— MARTIN GARDNER, *The Scientific American Book of Mathematical Puzzles & Diversions* (1959)

cat
Gardner
Babbage
GARDNER

Sholomov
Wakerly
Babbage
full adder
comparator
don't-cares

67. The author's best effort, with 1734 gates, was constructed by adapting the method of Sholomov in answer 63: First divide the truth tables into 64 rows for $o_5x_5o_2o_6o_8o_4$ and 4096 columns for the other 12 input variables. Then place appropriate 1s into "care" positions, in such a way that the columns have relatively few 1s. Then find a small number of column types that match the cares in all columns; 23 types suffice for the c function, 20 types for s, and 6 for m. We can then compute each output as $\bigvee(g_t \wedge h_t)$, sharing much of the work of the minterm calculations within g_t and h_t.

[This exercise was inspired by a discussion in John Wakerly's book *Digital Design* (Prentice–Hall, 3rd edition, 2000), §6.2.7. Incidentally, Babbage planned to choose among k possible moves by looking at $N \bmod k$, where N was the number of games won so far; he didn't realize that successive moves would tend to be highly correlated until N changed. Much better would have been to let N be the number of *moves made* so far.]

68. No. That method yields a "uniform" chain with a comprehensible structure, but its cost is 2^n times a polynomial in n. A circuit with approximately $2^n/n$ gates, constructed by Theorem L, exists but is more difficult to fabricate. (Incidentally, $C(\pi_5) = 10$.)

69. (a) One can, for example, verify this result by trying all 64 cases.

(b) If x_m lies in the same row or column as x_i, and also in the same row or column as x_j, we have $\alpha_{111} = \alpha_{101} = \alpha_{110} = 0$, so the pairs are good. Otherwise there are essentially three different possibilities, all bad: If $(i,j,m) = (1,2,4)$ then $\alpha_{101} = 0$, $\alpha_{100} = x_5x_9 \oplus x_6x_8$, $\alpha_{011} = x_9$; if $(i,j,m) = (1,2,6)$ then $\alpha_{010} = x_4x_9$, $\alpha_{011} = x_7$, $\alpha_{100} = x_5x_9$, $\alpha_{101} = x_8$; if $(i,j,m) = (1,5,9)$ then $\alpha_{111} = 1$, $\alpha_{110} = 0$, $\alpha_{010} = x_3x_7$.

70. (a) $x_1\wedge((x_5\wedge x_9)\oplus(x_6\wedge x_8)) \oplus x_2\wedge((x_6\wedge x_7)\oplus(x_4\wedge x_9)) \oplus x_3\wedge((x_4\wedge x_8)\oplus(x_5\wedge x_7))$.

(b) $x_1\wedge((x_5\wedge x_9)\vee(x_6\wedge x_8)) \vee x_2\wedge((x_6\wedge x_7)\vee(x_4\wedge x_9)) \vee x_3\wedge((x_4\wedge x_8)\vee(x_5\wedge x_7))$.

(c) Let $y_1 = x_1\wedge x_5\wedge x_9$, $y_2 = x_1\wedge x_6\wedge x_8$, $y_3 = x_2\wedge x_6\wedge x_7$, $y_4 = x_2\wedge x_4\wedge x_9$, $y_5 = x_3\wedge x_4\wedge x_8$, $y_6 = x_3\wedge x_5\wedge x_7$. The function $f(y_1,\ldots,y_6) = [y_1+y_2+y_3 > y_4+y_5+y_6]$ can be evaluated in 15 further steps with two full adders and a comparator; but there is a 14-step solution: Let $z_1 = (y_1\oplus y_2)\oplus y_3$, $z_2 = (y_1\oplus y_2)\vee(y_1\oplus y_3)$, $z_3 = (y_4\oplus y_5)\oplus y_6$, $z_4 = (y_4\oplus y_5)\vee(y_4\oplus y_6)$. Then $f = (z_1\oplus(z_2\wedge(\bar z_4\oplus(z_1\vee z_3))))\wedge(\bar z_3\vee z_4)$. Furthermore $y_1y_2y_3 = 111 \iff y_4y_5y_6 = 111$; so there are don't-cares, leading to an 11-step solution: $f = ((\bar z_1\wedge z_3)\vee\bar z_4)\wedge z_2$. The total cost is $12+11 = 23$.

(The author knows of no way by which a computer could discover such an efficient chain in a reasonable amount of time, given only the truth table of f. But perhaps an even better chain exists.)

71. (a) $P(p) = 1 - 12p^2 + 24p^3 + 12p^4 - 96p^5 + 144p^6 - 96p^7 + 24p^8$, which is $\frac{11}{32} + \frac{9}{2}\epsilon^2 - 3\epsilon^4 - 24\epsilon^6 + 24\epsilon^8$ when $p = \frac12 + \epsilon$.

(b) There are $N = 2^{n-3}$ sets of eight values $(f_0,\ldots,f_7)$, each of which yields good pairs with probability $P(p)$. So the answer is $1 - P(p)^N$.

(c) The probability is $\binom{N}{r}P(p)^r(1-P(p))^{N-r}$ that exactly r sets succeed; and in such a case t trials will find good pairs with probability $(r/N)^t$. The answer is therefore $1 - \sum_{r=0}^{N}\binom{N}{r}P(p)^r(1-P(p))^{N-r}(r/N)^t = 1 - P(p)^t + O(t^2/N)$.

(d) $\sum_{r=0}^{N}\binom{N}{r}P(p)^r(1-P(p))^{N-r}\sum_{j=0}^{t-1}(r/N)^j = (1-P(p)^t)/(1-P(p)) + O(t^3/N)$.

72. The probability in exercise 71(a) becomes $P(p) + (72p^3 - 264p^4 + 432p^5 - 336p^6 + 96p^7)r + (60p^2 - 240p^3 + 456p^4 - 432p^5 + 144p^6)r^2 + (-48p^2 + 144p^3 - 216p^4 + 96p^5)r^3 + (-36p^2 + 24p^3 + 12p^4)r^4 + (48p^2 - 24p^3)r^5 - 12p^2r^6$. If $p = q = (1-r)/2$, this is $(11 + 48r + 36r^2 - 144r^3 - 30r^4 + 336r^5 - 348r^6 + 144r^7 - 21r^8)/32$; for example, it's $7739/8192 \approx 0.94$ when $r = 1/2$.

73. Consider the Horn clauses $1\wedge 2\Rightarrow 3$, $1\wedge 3\Rightarrow 4$, $\ldots$, $1\wedge(n-1)\Rightarrow n$, $1\wedge n\Rightarrow 2$, and $i\wedge j\Rightarrow 1$ for $1 < i < j \leq n$. Suppose $|Z| > 1$ in a decomposition, and let i be minimum such that $x_i \in Z$. Also let j be minimum such that $j > i$ and $x_j \in Z$. We cannot have $i > 1$, since $i\wedge j\Rightarrow 1$ in that case. Thus $i = 1$, and $x_j \in Z$ for $2 \leq j \leq n$.

74. Suppose we know that no nontrivial decomposition exists with $x_1 \in Z$ or $\cdots$ or $x_{i-1} \in Z$; initially $i = 1$. We hope to rule out $x_i \in Z$ too, by choosing j and m cleverly. The Horn clauses $i\wedge j\Rightarrow m$ reduce to Krom clauses $j\Rightarrow m$ when i is asserted. So we essentially want to use Tarjan's depth-first search for strong components, in a digraph with arcs $j\Rightarrow m$ that may or may not exist.

When exploring from vertex j, first try $m = 1$, $\ldots$, $m = i-1$; if any such implication $i\wedge j\Rightarrow m$ succeeds, we can eliminate j and all its predecessors from the digraph for i. Otherwise, test if $j\Rightarrow m$ for any such eliminated vertex m. Otherwise test unexplored vertices m. Otherwise try vertices m that have already been seen, favoring those near the root of the depth-first tree.

In the example $f(x) = (\det X) \bmod 2$, we would successively find $1\wedge 2\not\Rightarrow 3$, $1\wedge 2\Rightarrow 4$, $1\wedge 4\Rightarrow 3$, $1\wedge 3\Rightarrow 5$, $1\wedge 5\Rightarrow 6$, $1\wedge 6\Rightarrow 7$, $1\wedge 7\Rightarrow 8$, $1\wedge 8\Rightarrow 9$, $1\wedge 9\Rightarrow 2$ (now $i \leftarrow 2$); $2\wedge 3\not\Rightarrow 1$, $2\wedge 3\Rightarrow 4$, $2\wedge 4\not\Rightarrow 1$, $2\wedge 4\not\Rightarrow 5$, $2\wedge 4\Rightarrow 6$, $2\wedge 6\Rightarrow 1$ (now 3, 4, and 6 are eliminated from the digraph for 2), $2\wedge 5\Rightarrow 1$ (and 5 is eliminated), $2\wedge 7\not\Rightarrow 1$, $2\wedge 7\Rightarrow 3$ (7 is eliminated), $2\wedge 8\Rightarrow 1$, $2\wedge 9\Rightarrow 1$ (now $i \leftarrow 3$); $3\wedge 4\not\Rightarrow 1$, $3\wedge 4\Rightarrow 2$, $3\wedge 5\Rightarrow 1$, etc.

75. This function is 1 at only two points, which are complementary. So it is indecomposable; yet the pairs (58) are *never* bad when $n > 3$. Every partition (Y, Z) will therefore be a candidate for decomposition.

Similarly, if f is decomposable with respect to (Y, Z), the indecomposable function $f(x) \oplus S_{0,n}(x)$ will act essentially like f in the tests. (A method to deal with *approximately decomposable functions* should probably be provided in a general-purpose decomposability tester.)

76. (a) Let $a_l = [i \geq l]$ for $0 \leq l \leq 2^m$. The cost is $\leq 2t(m)$, as observed in answer 38(b); and in fact, the cost can be reduced to $2^{m+1} - 2m - 2$ with $\Theta(m)$ depth. Furthermore the function $[i \leq j] = (\bar{\imath}_1 \wedge j_1) \vee ((i_1 \equiv j_1) \wedge [i_2 \ldots i_m \leq j_2 \ldots j_m])$ can be evaluated with $4m-3$ gates. After computing $x\oplus y$, each z_l costs $2^{m+1}+1 = O(n/\log n)$.

(b) Here the cost is at most $C(g_0) + \cdots + C(g_{2^m}) \leq (2^m+1)(2^{2^m}/(2^m + O(m)))$ by Theorem L, because each g_l is a function of 2^m inputs.

(c) If $i \leq j$ we have $z_l = x$ for $l \leq i$ and $z_l = y$ for $l > i$; hence $f_i(x) = c_0 \oplus \cdots \oplus c_i$ and $f_j(y) = c_{j+1} \oplus \cdots \oplus c_{2^m}$. If $i > j$ we have $z_l = y$ for $l \leq i$ and $z_l = x$ for $l > i$; hence $f_j(y) = c_0 \oplus \cdots \oplus c_j$ and $f_i(x) = c_{i+1} \oplus \cdots \oplus c_{2^m}$.

(d) The functions $b_l = [j < l]$ can be computed for $0 \leq l \leq 2^m$ in $O(2^m)$ steps, as in (a). So we can compute F from $(c_0, \ldots, c_{2^m})$ with $O(2^m)$ further gates. Step (b) therefore dominates the cost, for large m.

(e) $a_0 = 1$, $a_1 = i$, $a_2 = 0$; $b_0 = 0$, $b_1 = j$, $b_2 = 1$; $d = [i \leq j] = \bar{\imath} \vee j$; $m_l = a_l \oplus d$, $z_{l0} = x_0 \oplus (m_l \wedge (x_0 \oplus y_0))$, $z_{l1} = x_1 \oplus (m_l \wedge (x_1 \oplus y_1))$, for $l = 0, 1, 2$; $c_0 = z_{01}$; $c_1 = z_{10} \wedge \bar{z}_{11}$; $c_2 = z_{20} \vee z_{21}$; $c'_l = c_l \wedge (d \equiv a_l)$, $c''_l = c_l \wedge (d \equiv b_l)$, for $l = 0, 1, 2$; and finally $F = (c'_0 \oplus c'_1 \oplus c'_2) \vee (c''_0 \oplus c''_1 \oplus c''_2)$.

The net cost (29 after obvious simplifications) is, of course, outrageous in such a small example. But one wonders if a state-of-the-art automatic optimizer would be able to reduce this chain to just 5 gates.

[This result is a special case of more general theorems in *Matematicheskie Zametki* **15** (1974), 937–944; *London Math. Soc. Lecture Note Series* **169** (1992), 165–173.]

Horn clauses
Krom clauses
Tarjan
depth-first search
strong components
approximately decomposable functions
optimizer

Red'kin
comparison function
normal
Red'kin

77. Given a shortest such chain for f_n or $\bar{f}_n$, let $U_l = \{i \mid l = j(i) \text{ or } l = k(i)\}$ be the "uses" of x_l, and let $u_l = |U_l|$. Let $t_i = 1$ if $x_i = x_{j(i)} \vee x_{k(i)}$, otherwise $t_i = 0$. We will show that there's a chain of length $\le r-4$ that computes either f_{n-1} or $\bar{f}_{n-1}$, by using the following idea: If variable x_m is set to 0 or 1, for any m, we can obtain a chain for f_{n-1} or $\bar{f}_{n-1}$ by deleting all steps of U_m and modifying other steps appropriately. Furthermore, if $x_i = x_{j(i)} \circ x_{k(i)}$ and if either $x_{j(i)}$ or $x_{k(i)}$ is known to equal t_i when x_m has been set to 0 or 1, then we can also delete the steps U_i. (Throughout this argument, the letter m will stand for an index in the range $1 \le m \le n$.)

Case 1: $u_m = 1$ for some m. This case cannot occur in a shortest chain. For if the only use of x_m is $x_i = \bar{x}_m$, eliminating this step would change $f_n \leftrightarrow \bar{f}_n$; and otherwise we could set the values of $x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n$ to make x_i independent of x_m, contradicting $x_{n+r} = f_n$ or $\bar{f}_n$. Thus every variable must be used at least twice.

Case 2: $x_l = \bar{x}_m$ for some l and m, where $u_m > 1$. Then $x_i = x_l \circ x_k$ for some i and k, and we can set $x_m \leftarrow \bar{t}_i$ to make x_i independent of x_k. Eliminating steps U_m, U_l, and U_i then removes at least 4 steps, except when $u_l = u_i = 1$ and $u_m = 2$ and $x_j = x_m \circ x_i$; but in that case we can also eliminate U_j.

Case 3: $u_m \ge 3$ for some m, and not Case 2. If $i, j, k \in U_m$ and $i < j < k$, set $x_m \leftarrow t_k$ and remove steps i, j, k, U_k.

Case 4: $u_1 = u_2 = \cdots = u_n = 2$, and not Case 2. We may assume that the first step is $x_1 = x_1 \circ x_2$, and that $x_l = x_1 \circ x_k$ for some $k < l$.

Case 4.1: $k > 0$. Then $k > 1$. If $u_k = 1$, set $x_1 \leftarrow t_l$ and remove steps 1, k, l, U_l. Otherwise set $x_2 \leftarrow t_1$; this forces $x_k = \bar{t}_l$, and we can remove steps 1, k, l, U_k.

Case 4.2: $x_l = x_1 \circ x_m$. Then we must have $m = 2$; for if $m > 2$ we could set $x_2 \leftarrow t_1$, $x_m \leftarrow t_l$, and make x_r independent of x_1. Hence we may assume that $x_1 = x_1 \wedge x_2$, $x_2 = x_1 \vee x_2$. Setting $x_1 \leftarrow 0$ allows us to remove U_0 and U_1; setting $x_1 \leftarrow 1$ allows us to remove U_0 and U_2. Thus we're done unless $u_1 = u_2 = 1$.

If $x_p = \bar{x}_1$, set $x_1 \leftarrow 0$ and remove 1, 2, p, U_p; if $x_q = \bar{x}_2$, set $x_1 \leftarrow 1$ and remove 1, 2, q, U_q. Otherwise $x_p = x_1 \circ x_u$ and $x_q = x_2 \circ x_v$, where x_u and x_v do not depend on x_1 or x_2. But that's impossible; it would allow us to set $x_3, \ldots, x_n$ to make $x_u = t_p$, then $x_2 \leftarrow 1$ to make x_r independent of x_1.

[*Problemy Kibernetiki* **23** (1970), 83–101; **28** (1974), 4. With similar proofs, Red'kin showed that the shortest AND-OR-NOT chains for the functions '$x_1 \ldots x_n < y_1 \ldots y_n$' and '$x_1 \ldots x_n = y_1 \ldots y_n$' have lengths $5n-3$ and $5n-1$, respectively.]

78. [*SICOMP* **6** (1977), 427–430.] Say that y_k is *active* if $k \in S$. We may assume that the chain is normal and that $\|S\| > 1$; the proof is like Red'kin's in answer 77:

Case 1: Some active y_k is used more than once. Setting $y_k \leftarrow 0$ saves at least two steps and yields a chain for a function with $\|S\| - 1$ active values.

Case 2: Some active y_k appears only in an AND gate. Setting $y_k \leftarrow 0$ eliminates at least two steps, unless this AND is the final step. But it can't be the final step, because $y_k = 0$ makes the result independent of every other active y_j.

Case 3: Like Case 2 but with an OR or NOT-BUT or BUT-NOT gate. Setting $y_k \leftarrow c$ for some appropriate constant c has the desired effect.

Case 4: Like Case 2 but with XOR. The gate can't be final, since the result should be independent of y_k when $(x_1 \ldots x_m)_2$ addresses a different active value y_j. So we can eliminate two steps by setting y_k to the function defined by the *other* input to XOR.

79. (a) Suppose the cost is $r < 2n-2$; then $n > 1$. If each variable is used exactly once, two leaves must be mates. Therefore some variable is used at least twice. Pruning it away produces a chain of cost $\le r-2$ on $n-1$ variables, having no mates.

free tree
prime implicants
canalizing

(Incidentally, the cost is at least $2n-1$ if every variable is used at least twice, because at least $2n$ uses of variables must be connected together in the chain.)

(b) Notice that $S_{0,n} = \bigwedge_{u \,—\, v}(u \equiv v)$ whenever the edges $u \,—\, v$ form a free tree on $\{x_1, \ldots, x_n\}$. So there are many ways to achieve cost $2n-3$.

Any chain of cost $r < 2n-3$ must have $n > 2$ and must contain mates u and v. By renaming and possibly complementing intermediate results, we can assume that $u = 1$, $v = 2$, and that $f(x_1, \ldots, x_n) = g(x_1 \circ h(x_3, \ldots, x_n), x_2, \ldots, x_n)$, where $\circ$ is $\wedge$ or $\oplus$.

Case 1: $\circ$ is AND. We must have $h(0, \ldots, 0) = h(1, \ldots, 1) = 1$, for otherwise $f(x_1, x_2, y, \ldots, y)$ wouldn't depend on x_1. Therefore $f(x_1, \ldots, x_n) = h(x_3, \ldots, x_n) \wedge g(x_1, x_2, \ldots, x_n)$ can be computed by a chain of the same cost in which 1 and 2 are mates and in which the path between them has gotten shorter.

Case 2: $\circ$ is XOR. Then $f = f_0 \vee f_1$, where $f_0(x_1, \ldots, x_n) = (x_1 \equiv h(x_3, \ldots, x_n)) \wedge g(0, x_2, \ldots, x_n)$ and $f_1(x_1, \ldots, x_n) = (x_1 \oplus h(x_3, \ldots, x_n)) \wedge g(1, x_2, \ldots, x_n)$. But $f = S_{0,n}$ has only two prime implicants; so there are only four possibilities:

Case 2a: $f_0 = f$. Then we can replace $x_1 \oplus h$ by 0, to get a chain of cost $\le r-2$ for the function $g(0, x_2, \ldots, x_n) = S_{0,n-1}(x_2, \ldots, x_n)$.

Case 2b: $f_1 = f$. Similar to Case 2a.

Case 2c: $f_0(x) = x_1 \wedge \cdots \wedge x_n$ and $f_1(x) = \bar{x}_1 \wedge \cdots \wedge \bar{x}_n$. In this case we must have $g(0, x_2, \ldots, x_n) = x_2 \wedge \cdots \wedge x_n$ and $g(1, x_2, \ldots, x_n) = \bar{x}_2 \wedge \cdots \wedge \bar{x}_n$. Replacing h by 1 therefore yields a chain that computes f in $< r$ steps.

Case 2d: $f_0(x) = \bar{x}_1 \wedge \cdots \wedge \bar{x}_n$ and $f_1(x) = x_1 \wedge \cdots \wedge x_n$. Similar to Case 2c.

Applying these reductions repeatedly will lead to a contradiction. Similarly, one can show that $C(S_0 S_n) = 2n-2$. [*Theoretical Computer Science* **1** (1976), 289–295.]

80. [*Mathematical Systems Theory* **10** (1977), 323–336.] Without loss of generality, $a_0 = 0$ and the chain is normal. Define U_l and u_l as in answer 77. We may assume by symmetry that $u_1 = \max(u_1, \ldots, u_n)$.

We must have $u_1 \ge 2$. For if $u_1 = 1$, we could assume further that $x_{n+1} = x_1 \circ x_2$; hence two of the three functions $S_\alpha(0, 0, x_3, \ldots, x_n) = S_{\alpha''}$, $S_\alpha(0, 1, x_3, \ldots, x_n) = S_{'\alpha'}$, $S_\alpha(1, 1, x_3, \ldots, x_n) = S_{''\alpha}$ would be equal. But then S_α would be a parity function, or $S_{'\alpha'}$ would be constant.

Therefore setting $x_1 = 0$ allows us to eliminate the gates of U_1, giving a chain for $S_{\alpha'}$ with at least 2 fewer gates. It follows that $C(S_\alpha) \ge C(S_{\alpha'}) + 2$. Similarly, setting $x_1 = 1$ proves that $C(S_\alpha) \ge C(S_{'\alpha}) + 2$.

Three cases arise when we explore the situation further:

Case 1: $u_1 \ge 3$. Setting $x_1 = 0$ proves that $C(S_\alpha) \ge C(S_{\alpha'}) + 3$.

Case 2: $U_1 = \{i, j\}$ and operator $\circ_j$ is canalizing (namely, AND, BUT-NOT, NOT-BUT, or OR). Setting x_1 to an appropriate constant forces the value of x_j and allows us to eliminate $U_1 \cup U_j$; notice that $i \notin U_j$ in an optimum chain. So either $C(S_\alpha) \ge C(S_{\alpha'}) + 3$ or $C(S_\alpha) \ge C(S_{'\alpha}) + 3$.

Case 3: $U_1 = \{i, j\}$ and $\circ_i = \circ_j = \oplus$. We may assume that $x_i = x_1 \oplus x_2$ and $x_j = x_1 \oplus x_k$. If $u_j = 1$ and $x_l = x_j \oplus x_p$, we can restructure the chain by letting $x_j = x_k \oplus x_p$, $x_l = x_1 \oplus x_j$; therefore we can assume that either $u_j \ne 1$ or $x_l = x_j \circ x_p$ for some canalizing operator $\circ$. If $U_2 = \{i, j'\}$, we can assume similarly that $x_{j'} = x_2 \oplus x_{k'}$ and that either $u_{j'} = 1$ or $x_{l'} = x_{j'} \circ' x_{p'}$ for some canalizing operator $\circ'$. Furthermore we can assume by symmetry that x_j does not depend on $x_{j'}$.

If x_k does not depend on x_i, let $f(x_3, \ldots, x_n) = x_k$; otherwise let $f(x_3, \ldots, x_n)$ be the value of x_k when $x_i = 1$. By setting $x_1 = f(x_3, \ldots, x_n)$ and $x_2 = \bar{f}(x_3, \ldots, x_n)$, or vice versa, we make x_i and x_j constant, and we obtain a chain for the nonconstant function $S_{'\alpha'}$. We can, in fact, ensure that x_l is constant in the case $u_j = 1$. We claim that at least five gates of this chain (including x_i and x_j) can be eliminated; hence $C(S_\alpha) \ge C(S_{'\alpha'}) + 5$. The claim is clearly true if $|U_i \cup U_j| \ge 3$.

quantified formulas
Stockmeyer
Meyer
Alon
Boppana

We must have $|U_i \cup U_j| > 1$. Otherwise we'd have $p = i$, and x_k would not depend on x_i, so S_α would be independent of x_1 with our choice of x_2. Therefore $|U_i \cup U_j| = 2$.

Case 3a: $U_j = \{l\}$. Then x_l is constant; we can eliminate x_i, x_j, and $U_i \cup U_j \cup U_l$. If the latter set contains only two elements, then $x_q = x_i \circ x_l$ is also constant and we eliminate U_q. Since $S_{'\alpha'}$ isn't constant, we won't eliminate the output gate.

Case 3b: $U_i \subseteq U_j$, $|U_j| = 2$. Then $x_q = x_i \circ x_j$ for some q; we can eliminate x_i, x_j, and $U_j \cup U_q$. The claim has been proved.

(b) By induction, $C(S_k) \ge 2n + \min(k, n-k) - 3 - [n = 2k]$, for $0 < k < n$; $C(S_{\ge k}) \ge 2n + \min(k, n+1-k) - 4$, for $1 < k < n$. The easy cases are $C(S_0) = C(S_n) = C(S_{\ge 1}) = C(S_{\ge n}) = n - 1$; $C(S_{\ge 0}) = 0$. (According to Figs. 9 and 10, these bounds are optimum for $k = \lceil n/2 \rceil$ when $n \le 5$. All known results are consistent with the conjecture that $C(S_k) = C(S_{\ge k})$ for $k \ge n/2$.)

81. If some variable is used more than once, we can set it to a constant, decreasing n by 1 and decreasing c by ≥ 2. Otherwise the first operation must involve x_1, because $y_1 = x_1$ is the only output that doesn't need computation; making x_1 constant decreases n by 1, c by ≥ 1, and d by ≥ 1. [*J. Algorithms* **7** (1986), 185–201.]

82. (62) is false.

(63) reads, "For all numbers m there's a number n such that $m < n + 1$"; it is true because we can take $m = n$.

(64) fails when $n = 0$ or $n = 1$, because the numbers in these formulas are required to be nonnegative integers.

(65) says that, if b exceeds a by 2 or more, there's a number ab between them. Of course it's true, because we can let $ab = a + 1$.

(66) was explained in the text, and it too is true. Notice that '$\wedge$' takes precedence over '$\vee$' and '$\equiv$' takes precedence over '$\Leftrightarrow$', just as '$+$' takes precedence over '$\ge$' and '$<$' over '$\wedge$' in (65); these conventions reduce the need for parentheses in sentences of L.

(67) says that, if A contains at least one element n, it must contain a minimum element m (an element that's less than or equal to all of its elements). True.

(68) is similar, but m is now a maximum element. Again true, because all sets are assumed to be finite.

(69) asks for a set P with the property that $[0 \in P] = [3 \notin P]$, $[1 \in P] = [4 \notin P]$, $\ldots$, $[999 \in P] = [1002 \notin P]$, $[1000 \in P] \ne [1003 \notin P]$, $[1001 \in P] \ne [1004 \notin P]$, etc. It's true if (and only if) $P = \{x \mid x \bmod 6 \in \{1, 2, 3\} \text{ and } 0 \le x < 1000\}$.

Finally, the subformula $\forall n\,(n \in C \Leftrightarrow n + 1 \in C)$ in (70) is another way of saying that $C = \emptyset$, because C is finite. Hence the parenthesized formula after $\forall A\,\forall B$ is a tricky way to say that $A = \emptyset$ and $B \ne \emptyset$. (Stockmeyer and Meyer used this trick to abbreviate statements in L that involve long subformulas more than once.) Statement (70) is true because an empty set doesn't equal a nonempty set.

83. We can assume that the chain is normal. Let the canalizing steps be $y_1, \ldots, y_p$. Then $y_k = \alpha_k \circ \beta_k$ and $f = \alpha_{p+1}$, where α_k and β_k are $\oplus$'s of some subsets of $\{x_1, \ldots, x_n, y_1, \ldots, y_{k-1}\}$; at most $n+k-2$ $\oplus$'s are needed to compute them, combining common terms first. Hence $C(f) \le p + \sum_{k=1}^{p+1}(n + k - 2) = (p + 1)(n + p/2) - 1$.

84. Argue as in the previous answer, with $\vee$ or $\wedge$ in place of $\oplus$. [N. Alon and R. B. Boppana, *Combinatorica* **7** (1987), 15–16.]

pigeonhole principle
complete bigraphs

85. (a) A simple computer program shows that 13744 are legitimate and 19024 aren't. (An illegitimate family of this kind has at least 8 members; one such is $\{\texttt{00},\texttt{0f},\texttt{33},\texttt{55},\texttt{ff},\texttt{15},\texttt{3f},\texttt{77}\}$. Indeed, if the functions $x_1 \vee x_2$ ($\texttt{3f}$), $x_2 \vee x_3$ ($\texttt{77}$), and $(x_1 \vee x_2) \wedge x_3$ ($\texttt{15}$) are present in a legitimate family L, then $x_2 \sqcup \texttt{15} = \texttt{33} \mid \texttt{15} = \texttt{37}$ must also be in L.)

(b) The projection and constant functions are obviously present. Define $A^* = \bigcap\{B \mid B \supseteq A \text{ and } B \in \mathcal{A}\}$, or $A^* = \infty$ if no such set B exists. Then we have $\lceil A\rceil \sqcap \lceil B\rceil = \lceil A \cap B\rceil$ and $\lceil A\rceil \sqcup \lceil B\rceil = \lceil (A \cup B)^*\rceil$.

(c) Abbreviate the formulas as $\hat{x}_l \subseteq x_l \vee \bigvee_{i=n+1}^{l} \delta_i$, $x_l \subseteq \hat{x}_l \vee \bigvee_{i=n+1}^{l} \epsilon_i$, and argue by induction: If step l is an AND step, $\hat{x}_l = \hat{x}_j \sqcap \hat{x}_k \subseteq \hat{x}_j \wedge \hat{x}_k \subseteq (x_j \vee \bigvee_{i=n+1}^{l} \delta_i) \wedge (x_k \vee \bigvee_{i=n+1}^{l} \delta_i) = x_l \vee \bigvee_{i=n+1}^{l} \delta_i$; $x_l = x_j \wedge x_k \subseteq (\hat{x}_j \vee \bigvee_{i=n+1}^{l-1} \epsilon_i) \wedge (\hat{x}_k \vee \bigvee_{i=n+1}^{l-1} \epsilon_i) = (\hat{x}_j \wedge \hat{x}_k) \vee \bigvee_{i=n+1}^{l-1} \epsilon_i$, and $\hat{x}_j \wedge \hat{x}_k = \hat{x}_l \vee \epsilon_l$. Argue similarly if step l is an OR step.

86. (a) If S is an r-family contained in the $(r+1)$-family S', clearly $\Delta(S) \subseteq \Delta(S')$.

(b) By the pigeonhole principle, $\Delta(S)$ contains elements u and v of each part, whenever S is an r-family. And if $\Delta(S) = \{u, v\}$, we certainly have $u \mathrel{-\!\!-} v$.

(c) The result is obvious when $r = 1$. There are at most $r - 1$ edges containing any given vertex u, by the "strong" property. And if $u \mathrel{-\!\!-} v$, the edges *disjoint* from $\{u, v\}$ are strongly $(r-1)$-closed; so there are at most $(r-2)^2$ of them, by induction. Thus there are at most $1 + 2(r-2) + (r-2)^2$ edges altogether.

(d) Yes, by exercise 85(b), if $r > 1$, because strongly r-closed graphs are closed under intersection. All graphs with ≤ 1 edges are strongly r-closed when $r > 1$, because they have no r-families containing distinct edges.

(e) There are $\binom{n}{3}$ triangles $x_{ij} \wedge x_{ik} \wedge x_{jk}$, only $n-2$ of which are contained in any term x_{uv} of $\hat{f}$. Hence the minterms for at most $(r-1)^2(n-2)$ triangles are contained in $\hat{f}$, and the others must be contained in one of the functions $\epsilon_i = \hat{x}_i \oplus (\hat{x}_{j(i)} \wedge x_{k(i)})$. Such a term has the form $T = (\lceil G\rceil \sqcap \lceil H\rceil) \oplus (\lceil G\rceil \wedge \lceil H\rceil) = (\lceil G\rceil \wedge \lceil H\rceil) \wedge \overline{\lceil G \cap H\rceil}$, where G and H are strongly r-closed; we will prove that T contains at most $2(r-1)^3$ triangles.

A triangle $x_{ij} \wedge x_{ik} \wedge x_{jk}$ in T must involve some variable (say x_{ij}) of $\lceil G\rceil$ and some variable (say x_{ik}) of $\lceil H\rceil$, but no variable of $\lceil G \cap H\rceil$. There are at most $(r-1)^2$ choices for ij; and then there are at most $2(r-1)$ choices for k, since H has at most $r-1$ edges touching i and at most $r-1$ edges touching j.

(f) There are 2^{n-1} complete bigraphs obtained by coloring 1 red, coloring other vertices either red or blue, and letting $u \mathrel{-\!\!-} v$ if and only if u and v have opposite colors. By the first formula in exercise 85(c), the minterms B for every such graph must be contained in one of the terms $T = \delta_i = \hat{x}_i \oplus (\hat{x}_{j(i)} \vee x_{k(i)}) = \lceil (G \cup H)^*\rceil \wedge \overline{\lceil G \cup H\rceil}$. (For example, if $n = 4$ and vertices $(2, 3, 4)$ are (red, blue, blue), then $B = \bar{x}_{12} \wedge x_{13} \wedge x_{14} \wedge x_{23} \wedge x_{24} \wedge \bar{x}_{34}$.) A minterm B is contained in T if and only if, in the coloring for B, some edge of $(G \cup H)^*$ has vertices of opposite colors, but all edges of $G \cup H$ are monochromatic. We will prove that T includes at most $2^{n-r}r^2$ such B.

Let G be any graph, and $T = \lceil G^*\rceil \wedge \overline{\lceil G\rceil}$. The following (inefficient) algorithm can be used to find G^*: If there's an r-family S with $|\Delta(S)| < 2$, stop with $G^* = \infty$. Otherwise, if $\Delta(S) = \{u, v\}$ and $u \not\mathrel{-\!\!-} v$, add the edge $u \mathrel{-\!\!-} v$ to G and repeat.

At most 2^{n-r} bipartite minterms B have monochromatic $\{u_j, v_j\}$ for $1 \le j \le r$ when $|\Delta(S)| < 2$. And when $\Delta(S) = \{u, v\}$ there are 2^{n-r-1} with monochromatic $\{u_j, v_j\}$ and bichromatic $\{u, v\}$. So we want to show that the algorithm for G takes fewer then $2r^2$ iterations when G is strongly r-closed.

For $k \ge 1$, let $u_k \mathrel{-\!\!-} v_k$ be the first new edge added to G that is disjoint from $\{u_j, v_j\}$ for $1 \le j < k$. At most r such edges exist, by "strongness"; and each of them is followed by at most $2r-3$ new edges that touch u_j or v_j. So the total number of steps to find G^* is at most $r(2r-2)+1 < 2r^2$.

(g) Exercise 84 tells us that $q < \binom{p}{2} + (p+1)\binom{n}{2}$. Thus we have either $2(r-1)^3 p \geq \binom{n}{3} - (r-1)^2(n-2)$ or $\binom{p}{2} + (p+1)\binom{n}{2} > 2^{r-1}/r^2$. Both lower bounds for p are

$$\frac{1}{12}\Bigl(\frac{n}{6\lg n}\Bigr)^3\Bigl(1+O\Bigl(\frac{\log\log n}{\log n}\Bigr)\Bigr) \qquad \text{when} \qquad r = \Bigl\lceil \lg\Bigl(\frac{n^6}{186624(\lg n)^4}\Bigr)\Bigr\rceil.$$

[Noga Alon and Ravi B. Boppana, *Combinatorica* **7** (1987), 1–22, proceeded in this way to prove, among other things, the lower bound $\Omega(n/\log n)^s$ for the number of $\wedge$'s in any monotone chain that decides whether or not G has a clique of fixed size $s \geq 3$.]

87. The entries of X^3 are at most n^2 when X is a 0–1 matrix. A Boolean chain with $O(n^{\lg 7}(\log n)^2)$ gates can implement Strassen's matrix multiplication algorithm 4.6.4–(36), on integers modulo $2^{\lfloor \lg n^2 \rfloor + 1}$.

88. There are 1,422,564 such functions, in 716 classes with respect to permutation of variables. Algorithm L and the other methods of this section extend readily to ternary operations, and we obtain the following results for optimum median-only computation:

$C(f)$	Classes	Functions	$C_m(f)$	Classes	Functions	$L(f)$	Classes	Functions	$D(f)$	Classes	Functions
0	1	7	0	1	7	0	1	7	0	1	7
1	1	35	1	1	35	1	1	35	1	1	35
2	2	350	2	2	350	2	2	350	2	13	5670
3	9	3885	3	9	3885	3	8	3745	3	700	1416822
4	48	42483	4	48	42483	4	38	35203	4	1	30
5	201	406945	5	188	391384	5	139	270830	5	0	0
6	353	798686	6	253	622909	6	313	699377	6	0	0
7	99	169891	7	69	134337	7	176	367542	7	0	0
8	2	282	8	2	2520	8	34	43135	8	0	0
9	0	0	9	0	0	9	3	2310	9	0	0
10	0	0	10	0	0	10	0	0	10	0	0
11	0	0	∞	143	224654	11	1	30	11	0	0

S. Amarel, G. E. Cooke, and R. O. Winder [*IEEE Trans.* **EC-13** (1964), 4–13, Fig. 5b] conjectured that the 9-operation formula

$$\langle x_1x_2x_3x_4x_5x_6x_7\rangle = \langle x_1\langle\langle x_2x_3x_5\rangle\langle x_2x_4x_6\rangle\langle x_3x_4x_7\rangle\rangle\langle\langle x_2x_5x_6\rangle\langle x_3x_5x_7\rangle\langle x_4x_6x_7\rangle\rangle\rangle$$

is the best way to compute medians-of-7 via medians-of-3. But the "magic" formula

$$\langle x_1\langle\langle x_2\langle x_3x_4x_5\rangle\langle x_3x_6x_6\rangle\rangle\langle x_4\langle x_2x_6x_7\rangle\langle x_3x_5\langle x_5x_6x_7\rangle\rangle\rangle$$

needs only 8 operations; and in fact the shortest chain needs just seven steps:

$$\langle x_1x_2x_3x_4x_5x_6x_7\rangle = \langle x_1\langle x_2\langle x_5x_6x_7\rangle\langle x_3\langle x_5x_6x_7\rangle x_4\rangle\langle x_5\langle x_2x_3x_4\rangle\langle x_6\langle x_2x_3x_4\rangle x_7\rangle\rangle.$$

The interesting function $f(x_1,\ldots,x_7) = (x_1\wedge x_2\wedge x_4) \vee (x_2\wedge x_3\wedge x_5) \vee (x_3\wedge x_4\wedge x_6) \vee (x_4\wedge x_5\wedge x_7) \vee (x_5\wedge x_6\wedge x_1) \vee (x_6\wedge x_7\wedge x_2) \vee (x_7\wedge x_1\wedge x_3)$, whose prime implicants correspond to the projective plane with 7 points, is the toughest of all: Its minimum length $L(f) = 11$ and minimum depth $D(f) = 4$ are achieved by the remarkable formula

$$\langle\langle x_1x_4\langle x_4x_5x_6\rangle\rangle\langle x_3x_6\langle x_1\langle x_2x_3x_7\rangle\langle x_2x_5x_6\rangle\rangle\rangle\langle x_2x_7\langle x_1\langle x_5x_2x_4\rangle\langle x_5x_3x_7\rangle\rangle\rangle\rangle.$$

And the following even more astonishing chain computes it optimally:

$$x_8 = \langle x_1x_2x_3\rangle,\quad x_9 = \langle x_1x_4x_6\rangle,\quad x_{10} = \langle x_1x_5x_8\rangle,\quad x_{11} = \langle x_2x_7x_8\rangle,$$
$$x_{12} = \langle x_3x_9x_{10}\rangle,\quad x_{13} = \langle x_4x_5x_{12}\rangle,\quad x_{14} = \langle x_6x_{11}x_{12}\rangle,\quad x_{15} = \langle x_7x_{13}x_{14}\rangle.$$

Alon
Boppana
clique
0–1 matrix
Strassen
matrix multiplication
ternary operations
Amarel
Cooke
Winder
prime implicants
projective plane

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.