

KNUTH

# THE ART OF COMPUTER PROGRAMMING

VOLUME 4    PRE-FASCICLE 1A

## A DRAFT OF SECTION 7.1.3: BITWISE TRICKS AND TECHNIQUES

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY



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.

Copyright © 2006 by Addison–Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher, except that the official electronic file may be used to print single copies for personal (not commercial) use.

Zeroth printing (revision 6), 22 December 2008

## PREFACE

*These unforeseen stoppages,  
which I own I had no conception of when I first set out;  
— but which, I am convinced now, will rather increase than diminish as I advance,  
— have struck out a hint which I am resolved to follow;  
— and that is, — not to be in a hurry;  
— but to go on leisurely, writing and publishing two volumes of my life every year;  
— which, if I am suffered to go on quietly, and can make a tolerable bargain  
with my bookseller, I shall continue to do as long as I live.*  
— LAURENCE STERNE, *The Life and Opinions of  
Tristram Shandy, Gentleman* (1760)

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 vast; I cannot hope to have surrounded it enough to corral it completely. So I beg you to let me know about any deficiencies that you discover.

To put the material in context, this pre-fascicle contains Section 7.1.3 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 about Boolean operations in Section 7.1.1 and Boolean evaluation in Section 7.1.2. Section 7.1.3, which you're about to read here, applies these ideas to make computer programs run fast. Section 7.1.4 will then discuss the representation of Boolean functions.

The next part, 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 continue, 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.

MMIX  
ASCII

This part of *The Art of Computer Programming* has probably been more fun to write than any other so far. Indeed, I've spent more than 30 years collecting material for Section 7.1.3; finally I'm able to assemble these goodies together and segue through them.

Most of Volume 4 will deal with abstract concepts, and there will be little or no need to say much about a computer's machine language. Volumes 1–3 have already dealt with most of the important ideas about programming at that level. But Section 7.1.3 is a notable exception: Here we often want to see the very pulse of the machine.

Therefore I strongly recommend that readers become familiar with the basics of the MMIX computer, explained in Volume 1 Fascicle 1, in order to fully appreciate the bitwise tricks and techniques described here. Cross references to Sections 1.3.1' and 1.3.2' in the present booklet refer to that fascicle. I've reprinted the basic MMIX opcode-and-timing chart, Table 1.3.1'–1, at the end of this booklet for convenience, together with a list of ASCII codes.

The topic of Boolean functions and bit manipulation can of course be interpreted so broadly that it encompasses the entire subject of computer programming. The real goal of this fascicle is to focus on concepts that appear at the lowest levels, concepts on which we can erect significant superstructures. And even these apparently lowly notions turn out to be surprisingly rich, with explicit ties to sections 1.2.4, 1.2.5, 1.2.8, 2.3.1, 2.3.3, 2.3.4.2, 2.3.5, 3.1, 3.2.2, 4.1, 4.4, 4.5.3, 4.5.4, 4.6.1, 4.6.2, 4.6.3, 4.6.4, 5, 5.2.2, 5.2.3, 5.2.5, 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. Section 7.1.3 presented me with an extreme embarrassment of riches: After typing the manuscript I was astonished to discover that I had come up with 217 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 61, 76, 112, 117, 126, 128, 129, 130, and 174; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises 21, 140, 141, 156, and 165. Are those problems still open? Please inform me 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 5, 6, 20, 26, 34, 39, 49, 50, 53, 57, 58(d,e), 59, 60, 72, 78, 80, 81, 82, 83,

84, 86, 90, 95, 110, 115, 116, 120, 121, 127, 146, 154, 155, 159, 168, 184, 194, and 199, and/or the answers to exercises 17, 18, and 139. Furthermore I've credited exercises 45 and 54 to unpublished work of Tom Rokicki and Bill Gosper. Have either of those results ever appeared in print, to your knowledge?

Special thanks are due to Guy Steele and Hank Warren for their comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections.

I happily offer 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 do my best to give you immortal glory, 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  
16 December 2006

D. E. K.

*[These techniques] are instances of general mathematical principles waiting to be discovered, if an appropriate setting is created.*

*Such a setting would be a calculus of bitmap operations, so one can learn to use these operations just as naturally as arithmetic operations on numbers.*

— L. J. GUIBAS and J. STOLFI, *ACM Transactions on Graphics* (1982)

*A nice mixture of boolean and numeric functions—  
a suitable exercise for biturgical acolytes.*

— R. W. GOSPER (1996)

**A note on notation.** Several formulas in Section 7.1.3 use the notation  $\langle xyz \rangle$ , for the median function (aka majority function) that is discussed extensively in Section 7.1.1. Other formulas use the notation  $x \dot{-} y$ , for the monus function (aka dot-minus or saturating subtraction), which was defined in Section 1.3.1’. Hexadecimal constants are preceded by a sharp sign:  $\#123$  means  $(123)_{16}$ . If you run across other notations that may be unfamiliar, please look at the Index to Notations at the end of Volumes 1, 2, or 3, and/or the entries under “Notation” in the index to the present booklet. Of course Volume 4 will some day contain its own Index to Notations.

Rokicki  
Gosper  
Steele  
Warren  
Knuth  
GUIBAS  
STOLFI  
GOSPER  
notation  $\langle xyz \rangle$   
median function  
majority function  
notation  $x \dot{-} y$   
monus function  
dot-minus  
saturating subtraction  
Hexadecimal constants  
Notation

Bray more, Caroline  
Rochdale, Simon  
COLMAN  
bitwise—

Lady Caroline. *Psha! that's such a hack!*  
Sir Simon. *A hack, Lady Caroline, that  
the knowing ones have warranted sound.*  
— GEORGE COLMAN, *John Bull*, Act 3, Scene 1 (1803)

### 7.1.3. Bitwise Tricks and Techniques

Now comes the fun part: We get to use Boolean operations in our programs.

People are more familiar with arithmetic operations like addition, subtraction, and multiplication than they are with bitwise operations such as “and,” “exclusive-or,” and so on, because arithmetic has a very long history. But we will see that Boolean operations on binary numbers deserve to be much better known. Indeed, they’re an important component of every good programmer’s toolkit.

Early machine designers provided fullword bitwise operations in their computers primarily because such instructions could be included in a machine’s repertoire almost for free. Binary logic seemed to be potentially useful, although

only a few applications were originally foreseen. For example, the EDSAC computer, completed in 1949, included a “collate” command that essentially performed the operation  $z \leftarrow z + (x \& y)$ , where  $z$  was the accumulator,  $x$  was the multiplier register, and  $y$  was a specified word in memory; it was used for unpacking data. The Manchester Mark I computer, built at about the same time, included not only bitwise AND, but also OR and XOR. When Alan Turing wrote the first programming manual for the Mark I in 1950, he remarked that bitwise NOT can be obtained by using XOR (denoted ‘ $\neq$ ’) in combination with a row of 1s. R. A. Brooker, who extended Turing’s manual in 1952 when the Mark II computer was being designed, remarked further that OR could be used “to round off a number by forcing 1 into its least significant digit position.” By this time the Mark II, which was to become the prototype of the Ferranti Mercury, had also acquired new instructions for sideways addition and for the position of the most significant 1.

Keith Tocher published an unusual application of AND and OR in 1954, which has subsequently been reinvented frequently (see exercise 85). And during the ensuing decades, programmers have gradually discovered that bitwise operations can be amazingly useful. Many of these tricks have remained part of the folklore; the time is now ripe to take advantage of what has been learned.

A *trick* is a clever idea that can be used once, while a *technique* is a trick that can be used at least twice. We will see in this section that tricks tend to evolve naturally into techniques.

**Enriched arithmetic.** Let’s begin by officially defining bitwise operations on integers so that, if  $x = (\dots x_2 x_1 x_0)_2$ ,  $y = (\dots y_2 y_1 y_0)_2$ , and  $z = (\dots z_2 z_1 z_0)_2$  in binary notation, we have

$$x \& y = z \iff x_k \wedge y_k = z_k, \quad \text{for all } k \geq 0; \quad (1)$$

$$x | y = z \iff x_k \vee y_k = z_k, \quad \text{for all } k \geq 0; \quad (2)$$

$$x \oplus y = z \iff x_k \oplus y_k = z_k, \quad \text{for all } k \geq 0. \quad (3)$$

(It would be tempting to write ‘ $x \wedge y$ ’ instead of  $x \& y$ , and ‘ $x \vee y$ ’ instead of  $x | y$ ; but when we study optimization problems we’ll find it better to reserve the notations  $x \wedge y$  and  $x \vee y$  for  $\min(x, y)$  and  $\max(x, y)$ , respectively.) Thus, for example,

$$5 \& 11 = 1, \quad 5 | 11 = 15, \quad \text{and} \quad 5 \oplus 11 = 14,$$

since  $5 = (0101)_2$ ,  $11 = (1011)_2$ ,  $1 = (0001)_2$ ,  $15 = (1111)_2$ , and  $14 = (1110)_2$ . Negative integers are to be thought of in this connection as infinite-precision numbers in two’s complement notation, having infinitely many 1s at the left; for example,  $-5$  is  $(\dots 1111011)_2$ . Such infinite-precision numbers are a special case of *2-adic integers*, which are discussed in exercise 4.1–31, and in fact the operators  $\&$ ,  $|$ ,  $\oplus$  make perfect sense when they are applied to arbitrary 2-adic numbers.

Mathematicians have never paid much attention to the properties of  $\&$  and  $|$  as operations on integers. But the third operation,  $\oplus$ , has a venerable history, because it describes a winning strategy in the game of nim (see exercises 8–16). For this reason  $x \oplus y$  has often been called the “nim sum” of the integers  $x$  and  $y$ .

EDSAC computer  
collation, see bitwise and  
unpacking  
Manchester Mark I computer  
AND  
OR  
XOR  
Turing  
NOT  
Brooker  
Mark II computer (Manchester/Ferranti)  
round off  
Ferranti Mercury  
sideways addition  
most significant 1  
Tocher  
tricks versus techniques  
infinite-precision numbers  
two’s complement notation  
2-adic integers  
nim  
nim sum

All three of the basic bitwise operations turn out to have many useful properties. For example, every relation between  $\wedge$ ,  $\vee$ , and  $\oplus$  that we studied in Section 7.1.1 is automatically inherited by  $\&$ ,  $|$ , and  $\oplus$  on integers, since the relation holds in every bit position. We might as well recap the main identities here:

$$x \& y = y \& x, \quad x | y = y | x, \quad x \oplus y = y \oplus x; \quad (4)$$

$$(x \& y) \& z = x \& (y \& z), \quad (x | y) | z = x | (y | z), \quad (x \oplus y) \oplus z = x \oplus (y \oplus z); \quad (5)$$

$$(x | y) \& z = (x \& z) | (y \& z), \quad (x \& y) | z = (x | z) \& (y | z); \quad (6)$$

$$(x \oplus y) \& z = (x \& z) \oplus (y \& z); \quad (7)$$

$$(x \& y) | x = x, \quad (x | y) \& x = x; \quad (8)$$

$$(x \& y) \oplus (x | y) = x \oplus y; \quad (9)$$

$$x \& 0 = 0, \quad x | 0 = x, \quad x \oplus 0 = x; \quad (10)$$

$$x \& x = x, \quad x | x = x, \quad x \oplus x = 0; \quad (11)$$

$$x \& -1 = x, \quad x | -1 = -1, \quad x \oplus -1 = \bar{x}; \quad (12)$$

$$x \& \bar{x} = 0, \quad x | \bar{x} = -1, \quad x \oplus \bar{x} = -1; \quad (13)$$

$$\overline{x \& y} = \bar{x} | \bar{y}, \quad \overline{x | y} = \bar{x} \& \bar{y}, \quad \overline{x \oplus y} = \bar{x} \oplus \bar{y} = x \oplus \bar{y}. \quad (14)$$

The notation  $\bar{x}$  in (12), (13), and (14) stands for bitwise *complementation* of  $x$ , namely  $(\dots \bar{x}_2 \bar{x}_1 \bar{x}_0)_2$ , also written  $\sim x$ . Notice that (12) and (13) aren't quite the same as 7.1.1–(10) and 7.1.1–(18); we must now use  $-1 = (\dots 1111)_2$  instead of  $1 = (\dots 0001)_2$  in order to make the formulas bitwise correct.

We say that  $x$  is *contained in*  $y$ , written  $x \subseteq y$  or  $y \supseteq x$ , if the individual bits of  $x$  and  $y$  satisfy  $x_k \leq y_k$  for all  $k \geq 0$ . Thus

$$x \subseteq y \iff x \& y = x \iff x | y = y \iff x \& \bar{y} = 0. \quad (15)$$

Of course we needn't use bitwise operations only in connection with each other; we can combine them with all the ordinary operations of arithmetic. For example, from the relation  $x + \bar{x} = (\dots 1111)_2 = -1$  we can deduce the formula

$$-x = \bar{x} + 1, \quad (16)$$

which turns out to be extremely important. Replacing  $x$  by  $x - 1$  gives also

$$-x = \overline{x - 1}; \quad (17)$$

and in general we can reduce subtraction to complementation and addition:

$$\overline{x - y} = \bar{x} + y. \quad (18)$$

We often want to shift binary numbers to the left or right. These operations are equivalent to multiplication and division by powers of 2, with appropriate rounding, but it is convenient to have special notations for them:

$$x \ll k = x \text{ shifted left } k \text{ bits} = \lfloor 2^k x \rfloor; \quad (19)$$

$$x \gg k = x \text{ shifted right } k \text{ bits} = \lfloor 2^{-k} x \rfloor. \quad (20)$$

Here  $k$  can be any integer, possibly negative. In particular we have

$$x \ll (-k) = x \gg k \quad \text{and} \quad x \gg (-k) = x \ll k, \quad (21)$$

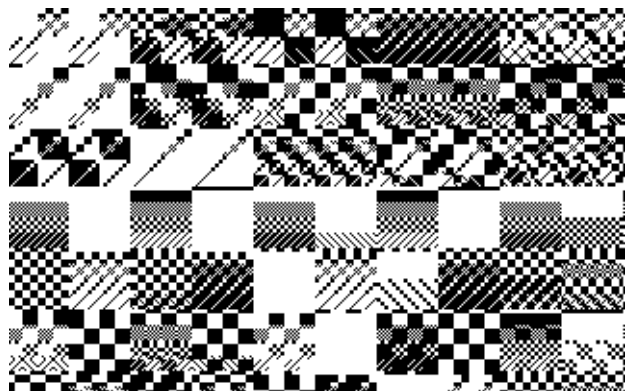
commutative laws  
associative laws  
distributive laws  
absorption laws  
complementation  
notation:  $\sim x$   
negation  
subtraction  
addition  
shift binary



for every infinite-precision number  $x$ . Also  $(x \& y) \ll k = (x \ll k) \& (y \ll k)$ , etc.

When bitwise operations are combined with addition, subtraction, multiplication, and/or shifting, extremely intricate results can arise, even when the formulas are quite short. A taste of the possibilities can be seen, for example, in Fig. 11. Furthermore, such formulas do not merely produce purposeless, chaotic behavior: A famous chain of operations known as “Gosper’s hack,” first published in 1972, opened people’s eyes to the fact that a large number of useful and nontrivial functions can be computed rapidly (see exercise 20). Our goal in this section is to explore how such efficient constructions might be discovered.

infinite-precision  
Sleator  
quilt  
pixel pattern  
black  
white  
Gosper’s hack  
packing++  
unpacking++  
Lehmer  
fractional precision  
date  
mod  
division



**Fig. 11.** A small portion of the patchwork quilt defined by the bitwise function  $f(x, y) = ((x \oplus \bar{y}) \& ((x - 350) \gg 3))^2$ ; the square cell in row  $x$  and column  $y$  is painted white or black according as the value of  $((f(x, y) \gg 12) \& 1)$  is 0 or 1. (Design by D. Sleator, 1976; see also exercise 18.)

**Packing and unpacking.** We studied algorithms for multiple-precision arithmetic in Section 4.3.1, dealing with situations where integers are too large to fit in a single word of memory or a single computer register. But the opposite situation, when integers are significantly *smaller* than the capacity of one computer word, is actually much more common; D. H. Lehmer called this “fractional precision.” We can often deal with several integers at once, by packing them into a single word.

For example, a date  $x$  that consists of a year number  $y$ , a month number  $m$ , and a day number  $d$ , can be represented by using 4 bits for  $m$  and 5 bits for  $d$ :

$$x = (((y \ll 4) + m) \ll 5) + d. \quad (22)$$

We’ll see below that many operations can be performed directly on dates in this packed form. For example,  $x < x'$  when date  $x$  precedes date  $x'$ . But if necessary the individual components  $(y, m, d)$  can readily be unpacked when  $x$  is given:

$$d = x \bmod 32, \quad m = (x \gg 5) \bmod 16, \quad y = x \gg 9. \quad (23)$$

And these “mod” operations do not require division, because of the important law

$$x \bmod 2^n = x \& (2^n - 1) \quad (24)$$

for any integer  $n \geq 0$ . We have, for instance,  $d = x \& 31$  in (22) and (23).

Such packing of data obviously saves space in memory, and it also saves time: We can more quickly move or copy items of data from one place to another when

they've been packed together. Moreover, computers run considerably faster when they operate on numbers that fit into a cache memory of limited size.

The ultimate packing density is achieved when we have 1-bit items, because we can then cram 64 of them into a single 64-bit word. Suppose, for example, that we want a table of all odd prime numbers less than 1024, so that we can easily decide the primality of a small integer. No problem; only eight 64-bit numbers are required:

$P_0 = 0111011011010011001011010010011001011001010010001011011010000001$ ,  
 $P_1 = 0100110000110010010100100110000110110000010000010110100110000100$ ,  
 $P_2 = 1001001100101100001000000101101000000100100001101001000100100101$ ,  
 $P_3 = 0010001010001000011000011001010010001011010000010001010001010010$ ,  
 $P_4 = 0000110000000010010000100100110010000100100110010010110000010000$ ,  
 $P_5 = 1101001001100000101001000100001000100001000100100101000100101000$ ,  
 $P_6 = 1010000001000010000011000011011000010000001011010000001011010000$ ,  
 $P_7 = 0000010100010000100010100100100000010100100100010010000010100110$ .

To test whether  $2k + 1$  is prime, for  $0 \leq k < 512$ , we simply compute

$$P_{\lfloor k/64 \rfloor} \ll (k \& 63) \quad (25)$$

in a 64-bit register, and see if the leftmost bit is 1. For example, the following MMIX instructions will do the job, if register **pbase** holds the address of  $P_0$ :

SRU	\$0, k, 3	\$0 $\leftarrow \lfloor k/8 \rfloor$ (i.e., $k \gg 3$ ).	
LDOU	\$1, pbase, \$0	\$1 $\leftarrow P_{\lfloor k/64 \rfloor}$ (i.e., $P_{\lfloor k/64 \rfloor}$ ).	
AND	\$0, k, #3f	\$0 $\leftarrow k \bmod 64$ (i.e., $k \& \#3f$ ).	(26)
SLU	\$1, \$1, \$0	\$1 $\leftarrow (\$1 \ll \$0) \bmod 2^{64}$ .	
BN	\$1, PRIME	Branch to PRIME if s(\$1) < 0.	■

Notice that the leftmost bit of a register is 1 if and only if the register contents are negative.

We could equally well pack the bits from right to left in each word:

$Q_0 = 1000000101101101000100101001101001100100101101001100101101101110$ ,  
 $Q_1 = 0010000110010110100000100000110110000110010010100100110000110010$ ,  
 $Q_2 = 1010010010001001011000010010000001011010000001000011010011001001$ ,  
 $Q_3 = 0100101000101000100000101101000100101001100001100001000101000100$ ,  
 $Q_4 = 0000100000110100100110010010000100110010010000100100000000110000$ ,  
 $Q_5 = 0001010010001010010010001000010001000010001001010000011001001011$ ,  
 $Q_6 = 0000101101000000101101000000100001101100001100000100001000000101$ ,  
 $Q_7 = 0110010100000100100010010010100000010010010100010000100010100000$ ;

here  $Q_j = P_j^R$ . Instead of shifting left as in (25), we now shift right,

$$Q_{\lfloor k/64 \rfloor} \gg (k \& 63), \quad (27)$$

and look at the *rightmost* bit of the result. The last two lines of (26) become

SRU	\$1, \$1, \$0	\$1 $\leftarrow \$1 \gg \$0$ .	
BOD	\$1, PRIME	Branch to PRIME if \$1 is odd.	■

(28)

(And of course we use **qbase** instead of **pbase**.) Either way, the classic *sieve of Eratosthenes* will readily set up the basic table entries  $P_j$  or  $Q_j$  (see exercise 24).

cache memory  
prime numbers  
table lookup by shifting  
sieve of Eratosthenes

**Table 1**  
THE BIG-ENDIAN VIEW OF A 32-BYTE MEMORY

big-endian++  
little-endian++  
multiple-precision

octa 0							
tetra 0				tetra 4			
wyde 0		wyde 2		wyde 4		wyde 6	
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
$a_0 \dots a_7$	$a_8 \dots a_{15}$	$a_{16} \dots a_{23}$	$a_{24} \dots a_{31}$	$a_{32} \dots a_{39}$	$a_{40} \dots a_{47}$	$a_{48} \dots a_{55}$	$a_{56} \dots a_{63}$
octa 8							
tetra 8				tetra 12			
wyde 8		wyde 10		wyde 12		wyde 14	
byte 8	byte 9	byte 10	byte 11	byte 12	byte 13	byte 14	byte 15
$a_{64} \dots a_{71}$	$a_{72} \dots a_{79}$	$a_{80} \dots a_{87}$	$a_{88} \dots a_{95}$	$a_{96} \dots a_{103}$	$a_{104} \dots a_{111}$	$a_{112} \dots a_{119}$	$a_{120} \dots a_{127}$
octa 16							
tetra 16				tetra 20			
wyde 16		wyde 18		wyde 20		wyde 22	
byte 16	byte 17	byte 18	byte 19	byte 20	byte 21	byte 22	byte 23
$a_{128} \dots a_{135}$	$a_{136} \dots a_{143}$	$a_{144} \dots a_{151}$	$a_{152} \dots a_{159}$	$a_{160} \dots a_{167}$	$a_{168} \dots a_{175}$	$a_{176} \dots a_{183}$	$a_{184} \dots a_{191}$
octa 24							
tetra 24				tetra 28			
wyde 24		wyde 26		wyde 28		wyde 30	
byte 24	byte 25	byte 26	byte 27	byte 28	byte 29	byte 30	byte 31
$a_{192} \dots a_{199}$	$a_{200} \dots a_{207}$	$a_{208} \dots a_{215}$	$a_{216} \dots a_{223}$	$a_{224} \dots a_{231}$	$a_{232} \dots a_{239}$	$a_{240} \dots a_{247}$	$a_{248} \dots a_{255}$

**Big-endian and little-endian conventions.** Whenever we pack bits or bytes into words, we must decide whether to place them from left to right or from right to left. The left-to-right convention is called “big-endian,” because the initial items go into the most significant positions; thus they will have bigger significance than their successors, when numbers are compared. The right-to-left convention is called “little-endian”; it puts the first items where little numbers go.

A big-endian approach seems more natural in many cases, because we’re accustomed to reading and writing from left to right. But a little-endian placement has advantages too. For example, let’s consider the prime number problem again; let  $a_k = [2k+1 \text{ is prime}]$ . Our table entries  $\{P_0, P_1, \dots, P_7\}$  are big-endian, and we can regard them as the representation of a single multiple-precision integer that is 512 bits long:

$$(P_0 P_1 \dots P_7)_{2^{64}} = (a_0 a_1 \dots a_{511})_2. \quad (29)$$

Similarly, our little-endian table entries represent the multiprecise integer

$$(Q_7 \dots Q_1 Q_0)_{2^{64}} = (a_{511} \dots a_1 a_0)_2. \quad (30)$$

The latter integer is mathematically nicer than the former, because it is

$$\sum_{k=0}^{511} 2^k a_k = \sum_{k=0}^{511} 2^k [2k+1 \text{ is prime}] = \left( \sum_{k=0}^{\infty} 2^k [2k+1 \text{ is prime}] \right) \bmod 2^{512}. \quad (31)$$

**Table 2**  
THE LITTLE-ENDIAN VIEW OF A 32-BYTE MEMORY

portability+

octa 24							
tetra 28				tetra 24			
wyde 30		wyde 28		wyde 26		wyde 24	
byte 31	byte 30	byte 29	byte 28	byte 27	byte 26	byte 25	byte 24
$a_{255} \dots a_{248}$	$a_{247} \dots a_{240}$	$a_{239} \dots a_{232}$	$a_{231} \dots a_{224}$	$a_{223} \dots a_{216}$	$a_{215} \dots a_{208}$	$a_{207} \dots a_{200}$	$a_{199} \dots a_{192}$
octa 16							
tetra 20				tetra 16			
wyde 22		wyde 20		wyde 18		wyde 16	
byte 23	byte 22	byte 21	byte 20	byte 19	byte 18	byte 17	byte 16
$a_{191} \dots a_{184}$	$a_{183} \dots a_{176}$	$a_{175} \dots a_{168}$	$a_{167} \dots a_{160}$	$a_{159} \dots a_{152}$	$a_{151} \dots a_{144}$	$a_{143} \dots a_{136}$	$a_{135} \dots a_{128}$
octa 8							
tetra 12				tetra 8			
wyde 14		wyde 12		wyde 10		wyde 8	
byte 15	byte 14	byte 13	byte 12	byte 11	byte 10	byte 9	byte 8
$a_{127} \dots a_{120}$	$a_{119} \dots a_{112}$	$a_{111} \dots a_{104}$	$a_{103} \dots a_{96}$	$a_{95} \dots a_{88}$	$a_{87} \dots a_{80}$	$a_{79} \dots a_{72}$	$a_{71} \dots a_{64}$
octa 0							
tetra 4				tetra 0			
wyde 6		wyde 4		wyde 2		wyde 0	
byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
$a_{63} \dots a_{56}$	$a_{55} \dots a_{48}$	$a_{47} \dots a_{40}$	$a_{39} \dots a_{32}$	$a_{31} \dots a_{24}$	$a_{23} \dots a_{16}$	$a_{15} \dots a_8$	$a_7 \dots a_0$

Notice, however, that we used  $(Q_7 \dots Q_1 Q_0)_{2^{64}}$  to get this simple result, not  $(Q_0 Q_1 \dots Q_7)_{2^{64}}$ . The other number,

$$(Q_0 Q_1 \dots Q_7)_{2^{64}} = (a_{63} \dots a_1 a_0 a_{127} \dots a_{65} a_{64} a_{191} \dots a_{385} a_{384} a_{511} \dots a_{449} a_{448})_2$$

is in fact quite weird, and it has no really nice formula. (See exercise 25.)

Endianness has important consequences, because most computers allow individual bytes of the memory to be addressed as well as register-sized units. MMIX has a big-endian architecture; therefore if register  $x$  contains the 64-bit number #0123456789abcdef, and if we use the commands ‘STOU  $x,0$ ; LDBU  $y,1$ ’ to store  $x$  into octabyte location 0 and read back the byte in location 1, the result in register  $y$  will be #23. On machines with a little-endian architecture, the analogous commands would set  $y \leftarrow \#cd$  instead; #23 would be byte 6.

Tables 1 and 2 illustrate the competing “world views” of big-endian and little-endian aficionados. The big-endian approach is basically top-down, with bit 0 and byte 0 at the top left; the little-endian approach is basically bottom-up, with bit 0 and byte 0 at the bottom right. Because of this difference, great care is necessary when transmitting data from one kind of computer to another, or when writing programs that are supposed to give equivalent results in both cases. On the other hand, our example of the  $Q$  table for primes shows that we can perfectly well use a little-endian packing convention on a big-endian computer



The elegant formula  $x \& -x$  in (37) allows us to *extract* the rightmost 1 bit very nicely, but we often want to identify exactly which bit it is. The ruler function can be computed in many ways, and the best method often depends heavily on the computer that is being used. For example, a two-instruction sequence due to J. Dallos does the job quickly and easily on **MMIX** (see (42)):

Dallos  
SADD  
magic mask  
mask: A bit pattern with 1s in key positions  
2-adic fraction  
truth tables  
projection functions  
MMIX  
CSZ  
ZSZ

$$\text{SUBU } t, x, 1; \text{ SADD } \text{rho}, t, x. \quad (46)$$

(See exercise 30 for the case  $x = 0$ .) We shall discuss here two approaches that do not rely on exotic commands like `SADD`; and later, after learning a few more techniques, we'll consider a third way.

The first general-purpose method makes use of “magic mask” constants  $\mu_k$  that prove to be useful in many other applications, namely

$$\begin{aligned}\mu_0 &= (\dots 1010101010101010101010101010101)_2 = -1/3, \\ \mu_1 &= (\dots 100110011001100110011001100110011)_2 = -1/5, \\ \mu_2 &= (\dots 100001111000011110000111100001111)_2 = -1/17,\end{aligned}\tag{47}$$

and so on. In general  $\mu_k$  is the infinite 2-adic fraction  $-1/(2^{2^k} + 1)$ , because  $(2^{2^k} + 1)\mu_k = (\mu_k \lll 2^k) + \mu_k = (\dots 11111)_2 = -1$ . On a computer that has  $2^d$ -bit registers we don't need infinite precision, of course, so we use the truncated constants

$$\mu_{d,k} = (2^{2^d} - 1)/(2^{2^k} + 1) \quad \text{for } 0 \leq k < d. \quad (48)$$

These constants are familiar from our study of Boolean evaluation, because they are the truth tables of the projection functions  $x_{d-k}$  (see, for example, 7.1.2–(7)).

When  $x$  is a power of 2, we can use these masks to compute

$$\rho x = [x \ \& \ \mu_0 = 0] + 2[x \ \& \ \mu_1 = 0] + 4[x \ \& \ \mu_2 = 0] + 8[x \ \& \ \mu_3 = 0] + \dots, \quad (49)$$

because  $[2^j \& \mu_k = 0] = j_k$  when  $j = (\dots j_3 j_2 j_1 j_0)_2$ . Thus, on a  $2^d$ -bit computer, we can start with  $\rho \leftarrow 0$  and  $y \leftarrow x \& -x$ ; then set  $\rho \leftarrow \rho + 2^k$  if  $y \& \mu_{d,k} = 0$ , for  $0 \leq k < d$ . This procedure gives  $\rho = \rho x$  when  $x \neq 0$ . (It also gives  $\rho 0 = 2^d - 1$ , an anomalous value that may need to be corrected; see exercise 30.)

For example, the corresponding MMIX program might look like this:

```

m0 GREG #5555555555555555 ;m1 GREG #3333333333333333;
m2 GREG #0f0f0f0f0f0f0f0f ;m3 GREG #00ff00ff00ff00ff;
m4 GREG #0000ffff0000ffff ;m5 GREG #00000000ffffffff;
  NEGU y,x; AND y,x,y; AND q,y,m5; ZSZ rho,q,32;
  AND q,y,m4; ADD t,rho,16; CSZ rho,q,t;
  AND q,y,m3; ADD t,rho,8; CSZ rho,q,t;
  AND q,y,m2; ADD t,rho,4; CSZ rho,q,t;
  AND q,y,m1; ADD t,rho,2; CSZ rho,q,t;
  AND q,y,m0; ADD t,rho,1; CSZ rho,q,t;

```

total time =  $19v$ . Or we could replace the last three lines by

$$\text{SRU } y, y, \text{rho}; \text{ LDB } t, \text{rhotab}, y; \text{ ADD } \text{rho}, \text{rho}, t \quad (51)$$

where `rhottab` points to the beginning of an appropriate 129-byte table (only eight of whose entries are actually used). The total time would then be  $\mu + 13v$ .

The second general-purpose approach to the computation of  $\rho x$  is quite different. On a 64-bit machine it starts as before, with  $y \leftarrow x \& -x$ ; but then it simply sets

$$\rho \leftarrow \text{decode}[(a \cdot y) \bmod 2^{64} \gg 58], \quad (52)$$

where  $a$  is a suitable multiplier and *decode* is a suitable 64-byte table. The constant  $a = (a_{63} \dots a_1 a_0)_2$  must have the property that its 64 substrings

$$a_{63}a_{62} \dots a_{58}, a_{62}a_{61} \dots a_{57}, \dots, a_5a_4 \dots a_0, a_4a_3a_2a_1a_00, \dots, a_000000$$

are distinct. Exercise 2.3.4.2–23 shows that many such “de Bruijn cycles” exist; for example, we can use M. H. Martin’s constant #03f79d71b4ca8b09, which is discussed in exercise 3.2.2–17. The decoding table *decode*[0], ..., *decode*[63] is then

$$\begin{aligned} &00, 01, 56, 02, 57, 49, 28, 03, 61, 58, 42, 50, 38, 29, 17, 04, \\ &62, 47, 59, 36, 45, 43, 51, 22, 53, 39, 33, 30, 24, 18, 12, 05, \\ &63, 55, 48, 27, 60, 41, 37, 16, 46, 35, 44, 21, 52, 32, 23, 11, \\ &54, 26, 40, 15, 34, 20, 31, 10, 25, 14, 19, 09, 13, 08, 07, 06. \end{aligned} \quad (53)$$

[This technique was devised in 1997 by M. L  uter, and independently by C. E. Leiserson, H. Prokop, and K. H. Randall a few months later (unpublished). David Seal had used a similar method in 1994, with a larger decoding table.]

**Working with the leftmost bits.** The function  $\lambda x = \lfloor \lg x \rfloor$ , which is dual to  $\rho x$  because it locates the *leftmost* 1 when  $x > 0$ , was introduced in Eq. 4.6.3–(6). It satisfies the recurrence

$$\lambda 1 = 0; \quad \lambda(2x) = \lambda(2x + 1) = \lambda(x) + 1 \quad \text{for } x > 0; \quad (54)$$

and it is undefined when  $x$  is not a positive integer. What is a good way to compute it? Once again MMIX provides a quick-but-tricky solution:

$$\text{FLOTU } y, \text{ROUND\_DOWN}, x; \text{ SUB } y, y, \text{fone}; \text{ SR } \text{lam}, y, 52 \quad (55)$$

where *fone* = #3ff0000000000000 is the floating point representation of 1.0. (Total time  $6v$ .) This code floats  $x$ , then extracts the exponent.

But if floating point conversion is not readily available, a binary reduction strategy works fairly well on a  $2^d$ -bit machine. We can start with  $\lambda \leftarrow 0$  and  $y \leftarrow x$ ; then we set  $\lambda \leftarrow \lambda + 2^k$  and  $y \leftarrow y \gg 2^k$  if  $y \gg 2^k \neq 0$ , for  $k = d - 1, \dots, 1, 0$  (or until  $k$  is reduced to the point where a short table can be used to finish up). The MMIX code analogous to (50) and (51) is now

$$\begin{aligned} &\text{SRU } y, x, 32; \text{ ZSNZ } \text{lam}, y, 32; \\ &\text{ADD } t, \text{lam}, 16; \text{ SRU } y, x, t; \text{ CSNZ } \text{lam}, y, t; \\ &\text{ADD } t, \text{lam}, 8; \text{ SRU } y, x, t; \text{ CSNZ } \text{lam}, y, t; \\ &\text{SRU } y, x, \text{lam}; \text{ LDB } t, \text{lamtab}, y; \text{ ADD } \text{lam}, \text{lam}, t; \end{aligned} \quad (56)$$

and the total time is  $\mu + 11v$ . In this case table *lamtab* has 256 entries, namely  $\lambda x$  for  $0 \leq x < 256$ . Notice that the “conditional set” (CS) and “zero or set” (ZS) instructions have been used here and in (50) instead of branch instructions.

de Bruijn cycles  
Martin  
L  uter  
Leiserson  
Prokop  
Randall  
Seal  
leftmost bits+  
 $\lambda x$ +  
 $\lfloor \lg x \rfloor$ +  
binary logarithm+  
leftmost  
floating point  
CSNZ  
ZSNZ  
MMIX  
conditional set  
zero or set  
branch instructions

There appears to be no simple way to extract the leftmost 1 bit that appears in a register, analogous to the trick by which we extracted the rightmost 1 in (37). For this purpose we could compute  $y \leftarrow \lambda x$  and then  $1 \ll y$ , if  $x \neq 0$ ; but a binary “smearing right” method is somewhat shorter and faster:

$$\begin{aligned} \text{Set } y &\leftarrow x, \text{ then } y \leftarrow y | (y \gg 2^k) \text{ for } 0 \leq k < d. \\ \text{The leftmost 1 bit of } x &\text{ is then } y - (y \gg 1). \end{aligned} \quad (57)$$

[These non-floating-point methods have been suggested by H. S. Warren, Jr.]

Other operations at the left of a register, like removing the leftmost run of 1s, are harder yet; see exercise 39. But there is a remarkably simple, machine-independent way to determine whether or not  $\lambda x = \lambda y$ , given unsigned integers  $x$  and  $y$ , in spite of the fact that we can’t compute  $\lambda x$  or  $\lambda y$  quickly:

$$\lambda x = \lambda y \quad \text{if and only if} \quad x \oplus y \leq x \& y. \quad (58)$$

[See exercise 40. This elegant relation was discovered by W. C. Lynch in 2006.] We will use (58) below, to devise another way to compute  $\lambda x$ .

**Sideways addition.** Binary  $n$ -bit numbers  $x = (x_{n-1} \dots x_1 x_0)_2$  are often used to represent subsets  $X$  of the  $n$ -element universe  $\{0, 1, \dots, n-1\}$ , with  $k \in X$  if and only if  $2^k \subseteq x$ . The functions  $\lambda x$  and  $\rho x$  then represent the largest and smallest elements of  $X$ . The function

$$\nu x = x_{n-1} + \dots + x_1 + x_0, \quad (59)$$

which is called the “sideways sum” or “population count” of  $x$ , also has obvious importance in this connection, because it represents the cardinality  $|X|$ , namely the number of elements in  $X$ . This function, which we considered in 4.6.3–(7), satisfies the recurrence

$$\nu 0 = 0; \quad \nu(2x) = \nu(x) \quad \text{and} \quad \nu(2x+1) = \nu(x) + 1, \quad \text{for } x \geq 0. \quad (60)$$

It also has an interesting connection with the ruler function (exercise 1.2.5–11),

$$\rho x = 1 + \nu(x-1) - \nu x; \quad \text{equivalently,} \quad \sum_{k=1}^n \rho k = n - \nu n. \quad (61)$$

The first textbook on programming, *The Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Gill, second edition (Reading, Mass.: Addison-Wesley, 1957), 155, 191–193, presented an interesting subroutine for sideways addition due to D. B. Gillies and J. C. P. Miller. Their method was devised for the 35-bit numbers of the EDSAC, but it is readily converted to the following 64-bit procedure for  $\nu x$  when  $x = (x_{63} \dots x_1 x_0)_2$ :

$$\begin{aligned} \text{Set } y &\leftarrow x - ((x \gg 1) \& \mu_0). \quad (\text{Now } y = (u_{31} \dots u_1 u_0)_4, \text{ where } u_j = x_{2j+1} + x_{2j}.) \\ \text{Set } y &\leftarrow (y \& \mu_1) + ((y \gg 2) \& \mu_1). \quad (\text{Now } y = (v_{15} \dots v_1 v_0)_{16}, \text{ } v_j = u_{2j+1} + u_{2j}.) \\ \text{Set } y &\leftarrow (y + (y \gg 4)) \& \mu_2. \quad (\text{Now } y = (w_7 \dots w_1 w_0)_{256}, \text{ } w_j = v_{2j+1} + v_{2j}.) \\ \text{Finally } \nu &\leftarrow ((a \cdot y) \bmod 2^{64}) \gg 56, \text{ where } a = (11111111)_{256}. \end{aligned} \quad (62)$$

The last step cleverly computes  $y \bmod 255 = w_7 + \dots + w_1 + w_0$  via multiplication, using the fact that the sum fits comfortably in eight bits. [David Muller had programmed a similar method for the ILLIAC I machine in 1954.]

smearing right  
Warren  
run of 1s  
Lynch  
sum of bits, see sideways sum  
ones counting, see sideways  
sideways addition+  
subsets  
largest  
smallest  
population count  
cardinality  
Wilkes  
Wheeler  
Gill  
Gillies  
Miller  
EDSAC  
remainder mod  $2^n - 1$   
Muller  
ILLIAC I



If  $x$  is expected to be “sparse,” having at most a few 1-bits, we can use a faster method [P. Wegner, *CACM* **3** (1960), 322]:

Set  $\nu \leftarrow 0$ ,  $y \leftarrow x$ . Then while  $y \neq 0$ , set  $\nu \leftarrow \nu + 1$ ,  $y \leftarrow y \& (y - 1)$ . (63)

A similar approach, using  $y \leftarrow y | (y + 1)$ , works when  $x$  is expected to be “dense.”

**Bit reversal.** For our next trick, let’s change  $x = (x_{63} \dots x_1 x_0)_2$  to its left-right mirror image,  $x^R = (x_0 x_1 \dots x_{63})_2$ . Anybody who has been following the developments so far, seeing methods like (50), (56), (57), and (62), will probably think, “Aha—once again we can divide by 2 and conquer! If we’ve already discovered how to reverse 32-bit numbers, we can reverse 64-bit numbers almost as fast, because  $(xy)^R = y^R x^R$ . All we have to do is apply the 32-bit method in parallel to both halves of the register, then swap the left half with the right half.”

Right. For example, we can reverse an 8-bit string in three easy steps:

$$\begin{array}{ll} \text{Given} & x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 \\ \text{Swap bits} & x_6 x_7 x_4 x_5 x_2 x_3 x_0 x_1 \\ \text{Swap nybs} & x_4 x_5 x_6 x_7 x_0 x_1 x_2 x_3 \\ \text{Swap nybbles} & x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 \end{array} \quad (64)$$

And six such easy steps will reverse 64 bits. Fortunately, each of the swapping operations turns out to be quite simple with the help of the magic masks  $\mu_k$ :

$$\begin{aligned} y &\leftarrow (x \gg 1) \& \mu_0, & z &\leftarrow (x \& \mu_0) \ll 1, & x &\leftarrow y | z; \\ y &\leftarrow (x \gg 2) \& \mu_1, & z &\leftarrow (x \& \mu_1) \ll 2, & x &\leftarrow y | z; \\ y &\leftarrow (x \gg 4) \& \mu_2, & z &\leftarrow (x \& \mu_2) \ll 4, & x &\leftarrow y | z; \\ y &\leftarrow (x \gg 8) \& \mu_3, & z &\leftarrow (x \& \mu_3) \ll 8, & x &\leftarrow y | z; \\ y &\leftarrow (x \gg 16) \& \mu_4, & z &\leftarrow (x \& \mu_4) \ll 16, & x &\leftarrow y | z; \\ x &\leftarrow (x \gg 32) | ((x \ll 32) \bmod 2^{64}). \end{aligned} \quad (65)$$

[Christopher Strachey foresaw some aspects of this construction in *CACM* **4** (1961), 146, and a similar *ternary* method was devised in 1973 by Bruce Baumgart (see exercise 49). The mature algorithm (65) was presented by Henry S. Warren, Jr., in *Hacker’s Delight* (Addison–Wesley, 2002), 102.]

But MMIX is once again able to trump this general-purpose technique with less traditional commands that do the job much faster. Consider

$$\text{rev GREG \#0102040810204080; MOR x,x,rev; MOR x,rev,x;} \quad (66)$$

the first MOR instruction reverses the bytes of  $x$  from big-endian to little-endian or vice versa, while the second reverses the bits within each byte.

**Bit swapping.** Suppose we only want to interchange two bits within a register,  $x_i \leftrightarrow x_j$ , where  $i > j$ . What would be a good way to proceed? (Dear reader, please pause for a moment and solve this problem in your head, or with pencil and paper—without looking at the answer below.)

Let  $\delta = i - j$ . Here is one solution (but don’t peek until you’re ready):

$$y \leftarrow (x \gg \delta) \& 2^j, \quad z \leftarrow (x \& 2^j) \ll \delta, \quad x \leftarrow (x \& m) | y | z, \quad \text{where } \overline{m} = 2^i | 2^j. \quad (67)$$

Wegner  
reversal of bits+  
divide by 2 and conquer  
magic masks  
Strachey  
Baumgart  
Warren  
big-endian  
little-endian  
MOR  
swapping bits+++

It uses two shifts and five bitwise Boolean operations, assuming that  $i$  and  $j$  are given constants. It is like each of the first lines of (65), except that a new mask  $m$  is needed because  $y$  and  $z$  don't account for all of the bits of  $x$ .

We can, however, do better, saving one operation and one constant:

$$y \leftarrow (x \oplus (x \gg \delta)) \& 2^j, \quad x \leftarrow x \oplus y \oplus (y \ll \delta). \quad (68)$$

The first assignment now puts  $x_i \oplus x_j$  into position  $j$ ; the second changes  $x_i$  to  $x_i \oplus (x_i \oplus x_j)$  and  $x_j$  to  $x_j \oplus (x_i \oplus x_j)$ , as desired. In general it's often wise to convert a problem of the form "change  $x$  to  $f(x)$ " into a problem of the form "change  $x$  to  $x \oplus g(x)$ ," since the bit-difference  $g(x)$  might be easy to calculate.

On the other hand, there's a sense in which (67) might be preferable to (68), because the assignments to  $y$  and  $z$  in (67) can sometimes be performed simultaneously. When expressed as a circuit, (67) has a depth of 4 while (68) has depth 5.

Operation (68) can of course be used to swap several pairs of bits simultaneously, when we use a mask  $\theta$  that's more general than  $2^j$ :

$$y \leftarrow (x \oplus (x \gg \delta)) \& \theta, \quad x \leftarrow x \oplus y \oplus (y \ll \delta). \quad (69)$$

Let us call this operation a " $\delta$ -swap," because it allows us to swap any non-overlapping pairs of bits that are  $\delta$  places apart. The mask  $\theta$  has a 1 in the rightmost position of each pair that's supposed to be swapped. For example, (69) will swap the leftmost 25 bits of a 64-bit word with the rightmost 25 bits, while leaving the 14 middle bits untouched, if we let  $\delta = 39$  and  $\theta = 2^{25} - 1 = \#1\text{fffff}$ .

Indeed, there's an astonishing way to reverse 64 bits using  $\delta$ -swaps, namely

$$\begin{aligned} y &\leftarrow (x \gg 1) \& \mu_0, \quad z \leftarrow (x \& \mu_0) \ll 1, \quad x \leftarrow y \mid z, \\ y &\leftarrow (x \oplus (x \gg 4)) \& \#0300\text{c}0303030\text{c}303, \quad x \leftarrow x \oplus y \oplus (y \ll 4), \\ y &\leftarrow (x \oplus (x \gg 8)) \& \#00\text{c}0300\text{c}03\text{f}0003\text{f}, \quad x \leftarrow x \oplus y \oplus (y \ll 8), \\ y &\leftarrow (x \oplus (x \gg 20)) \& \#00000\text{ff}\text{c}00003\text{fff}, \quad x \leftarrow x \oplus y \oplus (y \ll 20), \\ x &\leftarrow (x \gg 34) \mid ((x \ll 30) \bmod 2^{64}), \end{aligned} \quad (70)$$

saving two of the bitwise operations in (65) even though (65) looks "optimum."

**\*Bit permutation in general.** The methods we've just seen can be extended to obtain an *arbitrary* permutation of the bits in a register. In fact, there always exist masks  $\theta_0, \dots, \theta_5, \hat{\theta}_4, \dots, \hat{\theta}_0$  such that the following operations transform  $x = (x_{63} \dots x_1 x_0)_2$  into any desired rearrangement  $x^\pi = (x_{63\pi} \dots x_{1\pi} x_{0\pi})_2$  of its bits:

$$\begin{aligned} x &\leftarrow 2^k\text{-swap of } x \text{ with mask } \theta_k, \text{ for } k = 0, 1, 2, 3, 4, 5; \\ x &\leftarrow 2^k\text{-swap of } x \text{ with mask } \hat{\theta}_k, \text{ for } k = 4, 3, 2, 1, 0. \end{aligned} \quad (71)$$

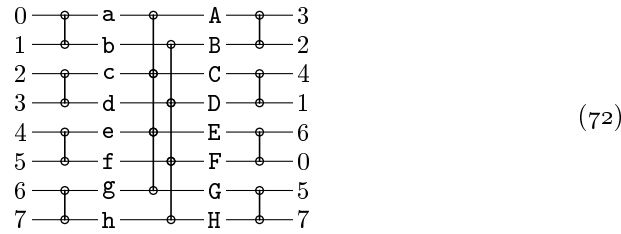
In general, a permutation of  $2^d$  bits can be achieved with  $2d - 1$  such steps, using appropriate masks  $\theta_k, \hat{\theta}_k$ , where the swap distances are respectively  $2^0, 2^1, \dots, 2^{d-1}, \dots, 2^1, 2^0$ .

To prove this fact, we can use a special case of the permutation networks discovered independently by A. M. Duguid and J. Le Corre in 1959, based on earlier work of D. Slepian [see V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic* (New York: Academic Press, 1965), Section 3.3].

depth  
 $\delta$ -swap  
 bit permutation++++  
 permutation networks  
 Duguid  
 Le Corre  
 Slepian  
 Beneš

Figure 12 shows a permutation network  $P(2n)$  for  $2n$  elements constructed from two permutation networks for  $n$  elements, when  $n = 4$ . Each ‘ $\begin{smallmatrix} \circ \\ | \\ \circ \end{smallmatrix}$ ’ connection between two lines represents a *crossbar module* that either leaves the line contents unaltered or interchanges them, as the data flows from left to right. Every setting of the individual crossbars therefore causes  $P(2n)$  to produce a permutation of its inputs; conversely, we wish to show that any permutation of the  $2n$  inputs can be achieved by some setting of the crossbars.

The construction of Fig. 12 is best understood by considering an example. Suppose we want to route the inputs  $(0, 1, 2, 3, 4, 5, 6, 7)$  to  $(3, 2, 4, 1, 6, 0, 5, 7)$ , respectively. The first job is to determine the contents of the lines just after the first column of crossbars and just before the last column, since we can then use a similar method to set the crossbars in the inner  $P(4)$ ’s. Thus, in the network



we want to find permutations  $abcdefgh$  and  $ABCDEFGH$  such that  $\{a, b\} = \{0, 1\}$ ,  $\{c, d\} = \{2, 3\}$ ,  $\dots$ ,  $\{g, h\} = \{6, 7\}$ ,  $\{a, c, e, g\} = \{A, C, E, G\}$ ,  $\{b, d, f, h\} = \{B, D, F, H\}$ ,  $\{A, B\} = \{3, 2\}$ ,  $\{C, D\} = \{4, 1\}$ ,  $\dots$ ,  $\{G, H\} = \{5, 7\}$ . Starting at the bottom, let us choose  $h = 7$ , because we don’t wish to disturb the contents of that line unless necessary. Then the following choices are *forced*:

$$H = 7; G = 5; e = 5; f = 4; D = 4; C = 1; a = 1; b = 0; F = 0; E = 6; g = 6. \quad (73)$$

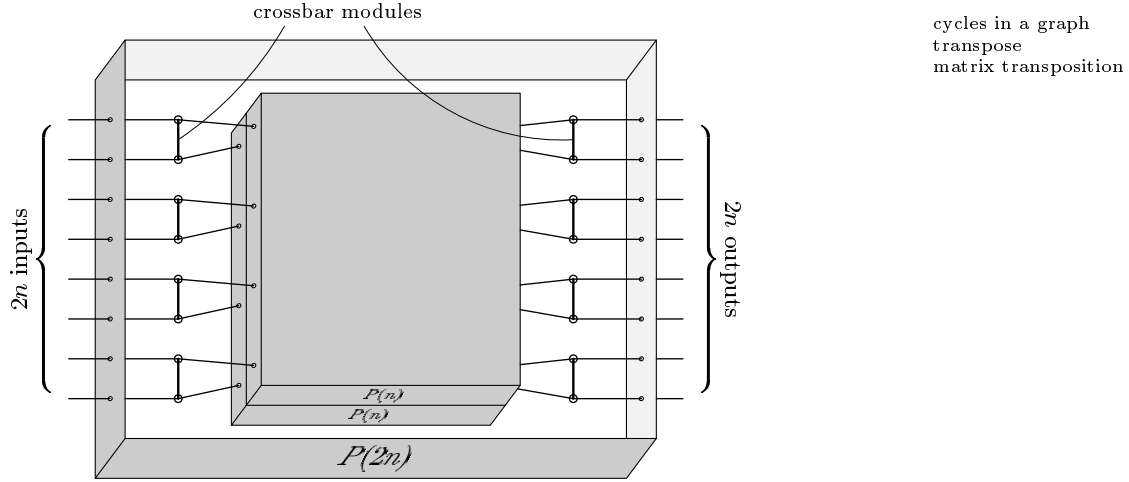
If we had chosen  $h = 6$ , the forcing pattern would have been similar but reversed,

$$F = 6; E = 0; a = 0; b = 1; D = 1; C = 4; e = 4; f = 5; H = 5; G = 7; g = 7. \quad (74)$$

Options (73) and (74) can both be completed by choosing either  $d = 3$  (hence  $B = 3, A = 2, c = 2$ ) or  $d = 2$  (hence  $B = 2, A = 3, c = 3$ ).

In general the forcing pattern will go in cycles, no matter what permutation we begin with. To see this, consider the graph on eight vertices  $\{ab, cd, ef, gh, AB, CD, EF, GH\}$  that has an edge from  $uv$  to  $UV$  whenever the pair of inputs connected to  $uv$  has an element in common with the pair of outputs connected to  $UV$ . Thus, in our example the edges are  $ab \text{ --- } EF$ ,  $ab \text{ --- } CD$ ,  $cd \text{ --- } AB$ ,  $cd \text{ --- } AB$ ,  $ef \text{ --- } CD$ ,  $ef \text{ --- } GH$ ,  $gh \text{ --- } EF$ ,  $gh \text{ --- } GH$ . We have a “double bond” between  $cd$  and  $AB$ , since the inputs connected to  $c$  and  $d$  are exactly the outputs connected to  $A$  and  $B$ ; subject to this slight bending of the strict definition of a graph, we see that each vertex is adjacent to exactly two other vertices, and lowercase vertices are always adjacent to uppercase ones. Therefore the graph

rearrangeable networks, see perm networks  
crossbar module  
graph  
bipartite graph



**Fig. 12.** The inside of a black box  $P(2n)$  that permutes  $2n$  elements in all possible ways, when  $n > 1$ . (Illustrated for  $n = 4$ .)

always consists of disjoint cycles of even length. In our example, the cycles are

$$\begin{array}{c} \text{ab} \begin{array}{l} \nearrow \text{EF} \text{---} \text{gh} \\ \searrow \text{CD} \text{---} \text{ef} \end{array} \text{GH} \end{array} \quad \text{cd} = \text{AB}, \quad (75)$$

where the longer cycle corresponds to (73) and (74). If there are  $k$  different cycles, there will be  $2^k$  different ways to specify the behavior of the first and last columns of crossbars.

To complete the network, we can process the inner 4-element permutations in the same way; and *any*  $2^d$ -element permutation is achievable in this same recursive fashion. The resulting crossbar settings determine the masks  $\theta_j$  and  $\hat{\theta}_j$  of (71). Some choices of crossbars may lead to a mask that is entirely zero; then we can eliminate the corresponding stage of the computation.

If the input and output are identical on the bottom lines of the network, our construction shows how to ensure that none of the crossbars touching those lines are active. For example, the 64-bit algorithm in (71) could be used also with a 60-bit register, without needing the four extra bits for any intermediate results.

Of course we can often beat the general procedure of (71) in special cases. For example, exercise 52 shows that method (71) needs nine swapping steps to transpose an  $8 \times 8$  matrix, but in fact three swaps suffice:

Given	7-swap	14-swap	28-swap
00 01 02 03 04 05 06 07	00 <b>10</b> 02 <b>12</b> 04 <b>14</b> 06 <b>16</b>	00 10 <b>20 30</b> 04 14 <b>24 34</b>	00 10 20 30 <b>40 50 60 70</b>
10 11 12 13 14 15 16 17	<b>01</b> 11 <b>03</b> 13 <b>05</b> 15 <b>07</b> 17	01 11 <b>21 31</b> 05 15 <b>25 35</b>	01 11 21 31 <b>41 51 61 71</b>
20 21 22 23 24 25 26 27	20 <b>30</b> 22 <b>32</b> 24 <b>34</b> 26 <b>36</b>	<b>02 12</b> 22 32 <b>06 16</b> 26 36	02 12 22 32 <b>42 52 62 72</b>
30 31 32 33 34 35 36 37	<b>21</b> 31 <b>23</b> 33 <b>25</b> 35 <b>27</b> 37	<b>03 13</b> 23 33 <b>07 17</b> 27 37	03 13 23 33 <b>43 53 63 73</b>
40 41 42 43 44 45 46 47	40 <b>50</b> 42 <b>52</b> 44 <b>54</b> 46 <b>56</b>	40 50 <b>60 70</b> 44 54 <b>64 74</b>	<b>04 14 24 34</b> 44 54 64 74
50 51 52 53 54 55 56 57	<b>41</b> 51 <b>43</b> 53 <b>45</b> 55 <b>47</b> 57	41 51 <b>61 71</b> 45 55 <b>65 75</b>	<b>05 15 25 35</b> 45 55 65 75
60 61 62 63 64 65 66 67	60 <b>70</b> 62 <b>72</b> 64 <b>74</b> 66 <b>76</b>	<b>42 52</b> 62 72 <b>46 56</b> 66 76	<b>06 16 26 36</b> 46 56 66 76
70 71 72 73 74 75 76 77	<b>61</b> 71 <b>63</b> 73 <b>65</b> 75 <b>67</b> 77	<b>43 53</b> 63 73 <b>47 57</b> 67 77	<b>07 17 27 37</b> 47 57 67 77

The “perfect shuffle” is another bit permutation that arises frequently in practice. If  $x = (\dots x_2 x_1 x_0)_2$  and  $y = (\dots y_2 y_1 y_0)_2$  are any 2-adic integers, we define  $x \ddagger y$  (“ $x$  zip  $y$ ,” the *zipper function* of  $x$  and  $y$ ) by interleaving their bits:

$$x \ddagger y = (\dots x_2 y_2 x_1 y_1 x_0 y_0)_2. \quad (76)$$

This operation has important applications to the representation of 2-dimensional data, because a small change in either  $x$  or  $y$  usually causes only a small change in  $x \ddagger y$  (see exercise 86). Notice also that the magic mask constants (47) satisfy

$$\mu_k \ddagger \mu_k = \mu_{k+1}. \quad (77)$$

If  $x$  appears in the left half of a register and  $y$  appears in the right half, a perfect shuffle is the permutation that changes the register contents to  $x \ddagger y$ .

A sequence of  $d-1$  swapping steps will perfectly shuffle a  $2^d$ -bit register; in fact, exercise 53 shows that there are several ways to achieve this. Once again, therefore, we are able to improve on the  $(2d-1)$ -step method of (71) and Fig. 12.

Conversely, suppose we’re given the shuffled value  $z = x \ddagger y$  in a  $2^d$ -bit register; is there an efficient way to extract the original value of  $y$ ? Sure: If the  $d-1$  swaps that do a perfect shuffle are performed in reverse order, they’ll undo the shuffle and recover both  $x$  and  $y$ . But if only  $y$  is wanted, we can save half of the work: Start with  $y \leftarrow z \& \mu_0$ ; then set  $y \leftarrow (y + (y \gg 2^{k-1})) \& \mu_k$  for  $k = 1, \dots, d-1$ . For example, when  $d = 3$  this procedure goes  $(0y_3 0y_2 0y_1 0y_0)_2 \mapsto (00y_3 y_2 00y_1 y_0)_2 \mapsto (0000y_3 y_2 y_1 y_0)_2$ . “Divide and conquer” conquers again.

Consider now a more general problem, where we want to extract and compress an *arbitrary* subset of a register’s bits. Suppose we’re given a  $2^d$ -bit word  $z = (z_{2^d-1} \dots z_1 z_0)_2$  and a mask  $\chi = (\chi_{2^d-1} \dots \chi_1 \chi_0)_2$  that has  $s$  1-bits; thus  $\nu\chi = s$ . The problem is to assemble the compact subword

$$y = (y_{s-1} \dots y_1 y_0)_2 = (z_{j_{s-1}} \dots z_{j_1} z_{j_0})_2, \quad (78)$$

where  $j_{s-1} > \dots > j_1 > j_0$  are the indices where  $\chi_j = 1$ . For example, if  $d = 3$  and  $\chi = (10110010)_2$ , we want to transform  $z = (y_3 x_3 y_2 y_1 x_2 x_1 y_0 x_0)_2$  into  $y = (y_3 y_2 y_1 y_0)_2$ . (The problem of going from  $x \ddagger y$  to  $y$ , considered above, is the special case  $\chi = \mu_0$ .) We know from (71) that  $y$  can be found by  $\delta$ -swapping, at most  $2d-1$  times; but in this problem the relevant data always moves to the right, so we can speed things up by doing *shifts* instead of swaps.

Let’s say that a  $\delta$ -shift of  $x$  with mask  $\theta$  is the operation

$$x \leftarrow x \oplus ((x \oplus (x \gg \delta)) \& \theta), \quad (79)$$

which changes bit  $x_j$  to  $x_{j+\delta}$  if  $\theta$  has 1 in position  $j$ , otherwise it leaves  $x_j$  unchanged. Guy Steele discovered that there always exist masks  $\theta_0, \theta_1, \dots, \theta_{d-1}$  so that the general extraction problem (78) can be solved with a few  $\delta$ -shifts:

$$\begin{aligned} &\text{Start with } x \leftarrow z; \text{ then do a } 2^k\text{-shift of } x \text{ with mask } \theta_k, \\ &\text{for } k = 0, 1, \dots, d-1; \text{ finally set } y \leftarrow x. \end{aligned} \quad (80)$$

In fact, the idea for finding appropriate masks is surprisingly simple. Every bit that wants to move a total of exactly  $l = (l_{d-1} \dots l_1 l_0)_2$  places to the right should be transported in the  $2^k$ -shifts for which  $l_k = 1$ .

perfect shuffle  
2-adic integers  
interleaving, see zipper function, perf shuffle  
2-dimensional data  
magic mask  
Divide and conquer  
extract and compress  
mask  
packing  
 $\delta$ -shift  
Steele

For example, suppose  $d = 3$  and  $\chi = (10110010)_2$ . (We must assume that  $\chi \neq 0$ .) Remembering that some 0s need to be shifted in from the left, we can set  $\theta_0 = (00011001)_2$ ,  $\theta_1 = (00000110)_2$ ,  $\theta_2 = (11111000)_2$ ; then (80) maps

$$(y_3 x_3 y_2 y_1 x_2 x_1 y_0 x_0)_2 \mapsto (y_3 x_3 y_2 y_2 y_1 x_1 y_0 y_0)_2 \mapsto (y_3 x_3 y_2 y_2 y_1 y_2 y_1 y_0)_2 \mapsto (0000 y_3 y_2 y_1 y_0)_2.$$

Exercise 69 proves that the bits being extracted will never interfere with each other during their journey. Furthermore, there's a slick way to compute the necessary masks  $\theta_k$  dynamically from  $\chi$ , in  $O(d^2)$  steps (see exercise 70).

A “sheep-and-goats” operation has been suggested for computer hardware, extending (78) to produce the general unshuffled word

$$z \cdot \chi = (x_{r-1} \dots x_1 x_0 y_{s-1} \dots y_1 y_0)_2 = (z_{i_{r-1}} \dots z_{i_1} z_{i_0} z_{j_{s-1}} \dots z_{j_1} z_{j_0})_2; \quad (81)$$

here  $i_{r-1} > \dots > i_1 > i_0$  are the indices where  $\chi_i = 0$ . Any permutation of  $2^d$  bits is achievable via at most  $d$  sheep-and-goats operations (see exercise 73).

Shifting also allows us to go beyond permutations, to arbitrary *mappings* of bits within a register. Suppose we want to transform

$$x = (x_{2^d-1} \dots x_1 x_0)_2 \mapsto x^\varphi = (x_{(2^d-1)\varphi} \dots x_{1\varphi} x_{0\varphi})_2, \quad (82)$$

where  $\varphi$  is any of the  $(2^d)^{2^d}$  functions from the set  $\{0, 1, \dots, 2^d - 1\}$  into itself. K. M. Chung and C. K. Wong [*IEEE Transactions* **C-29** (1980), 1029–1032] discovered an attractive way to do this in  $O(d)$  steps by using *cyclic*  $\delta$ -shifts, which are like (79) except that we set

$$x \leftarrow x \oplus ((x \oplus (x \gg \delta) \oplus (x \ll (2^d - \delta))) \& \theta). \quad (83)$$

Their idea is to let  $c_l$  be the number of indices  $j$  such that  $j\varphi = l$ , for  $0 \leq l < 2^d$ . Then they find masks  $\theta_0, \theta_1, \dots, \theta_{d-1}$  with the property that a cyclic  $2^k$ -shift of  $x$  with mask  $\theta_k$ , done successively for  $0 \leq k < d$ , will transform  $x$  into a number  $x'$  that contains exactly  $c_l$  copies of bit  $x_l$  for each  $l$ . Finally the general permutation procedure (71) can be used to change  $x' \mapsto x^\varphi$ .

For example, suppose  $d = 3$  and  $x^\varphi = (x_3 x_1 x_1 x_0 x_3 x_7 x_5 x_5)_2$ . Then we have  $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7) = (1, 2, 0, 2, 0, 2, 0, 1)$ . Using masks  $\theta_0 = (00011100)_2$ ,  $\theta_1 = (00001000)_2$ , and  $\theta_2 = (01100000)_2$ , three cyclic  $2^k$ -shifts now take  $x = (x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0)_2 \mapsto (x_7 x_6 x_5 x_5 x_4 x_3 x_1 x_0)_2 \mapsto (x_7 x_6 x_5 x_5 x_5 x_3 x_1 x_0)_2 \mapsto (x_7 x_3 x_1 x_5 x_5 x_3 x_1 x_0)_2 = x'$ . Then, some  $\delta$ -swaps:  $x' \mapsto (x_3 x_7 x_5 x_1 x_3 x_5 x_1 x_0)_2 \mapsto (x_3 x_1 x_5 x_7 x_3 x_5 x_1 x_0)_2 \mapsto (x_3 x_1 x_1 x_0 x_3 x_5 x_5 x_7)_2 \mapsto (x_3 x_1 x_1 x_0 x_3 x_7 x_5 x_5)_2 = x^\varphi$ ; we're done! Of course any 8-bit mapping can be achieved more quickly by brute force, one bit at a time; the method of Chung and Wong becomes much more impressive in a 256-bit register. Even with MMIX's 64-bit registers it's pretty good, needing at most 96 cycles in the worst case.

To find  $\theta_0$ , we use the fact that  $\sum c_l = 2^d$ , and we look at  $\Sigma_{\text{even}} = \sum c_{2l}$  and  $\Sigma_{\text{odd}} = \sum c_{2l+1}$ . If  $\Sigma_{\text{even}} = \Sigma_{\text{odd}} = 2^{d-1}$ , we can set  $\theta_0 = 0$  and omit the cyclic 1-shift. But if, say,  $\Sigma_{\text{even}} < \Sigma_{\text{odd}}$ , we find an even  $l$  with  $c_l = 0$ . Cyclically shifting into bits  $l, l+1, \dots, l+t$  (modulo  $2^d$ ) for some  $t$  will produce new counts  $(c'_0, \dots, c'_{2^d-1})$  for which  $\Sigma'_{\text{even}} = \Sigma'_{\text{odd}} = 2^{d-1}$ ; so  $\theta_0 = 2^l + \dots + 2^{(l+t) \bmod 2^d}$ .

sheep-and-goats  
notation  $z \cdot \chi$   
mappings  
Chung  
Wong  
cyclic  
masks  
pi, as “random” example

Then we can deal with the bits in even and odd positions separately, using the same method, until getting down to 1-bit subwords. Exercise 74 has the details.

**Working with fragmented fields.** Instead of extracting bits from various parts of a word and gathering them together, we can often manipulate those bits directly in their original positions.

For example, suppose we want to run through all subsets of a given set  $U$ , where (as usual) the set is specified by a mask  $\chi$  such that  $[k \in U] = (\chi \gg k) \& 1$ . If  $x \subseteq \chi$  and  $x \neq \chi$ , there's an easy way to calculate the next largest subset of  $U$  in lexicographic order, namely the smallest integer  $x' > x$  such that  $x' \subseteq \chi$ :

$$x' = (x - \chi) \& \chi. \quad (84)$$

In the special case when  $x = 0$  and  $\chi \neq 0$ , we've already seen in (37) that this formula produces the rightmost bit of  $\chi$ , which corresponds to the lexicographically smallest nonempty subset of  $U$ .

Why does formula (84) work? Imagine adding 1 to the number  $x \mid \bar{\chi}$ , which has 1s wherever  $\chi$  is 0. A carry will propagate through those 1s until it reaches the rightmost bit position where  $x$  has a 0 and  $\chi$  has a 1; furthermore all bits to the right of that position will become zero. Therefore  $x' = ((x \mid \bar{\chi}) + 1) \& \chi$ . But we have  $(x \mid \bar{\chi}) + 1 = (x + \bar{\chi}) + 1 = x + (\bar{\chi} + 1) = x - \chi$  when  $x \subseteq \chi$ . QED.

Notice further that  $x' = 0$  if and only if  $x = \chi$ . So we'll know when we've found the largest subset. Exercise 79 shows how to go back to  $x$ , given  $x'$ .

We might also want to run through all elements of a *subcube*—for example, to find all bit patterns that match a specification like  $*10*1*01$ , consisting of 0s, 1s, and  $*$ s (don't-cares). Such a specification can be represented by asterisk codes  $a = (a_{n-1} \dots a_0)_2$  and bit codes  $b = (b_{n-1} \dots b_0)_2$ , as in exercise 7.1.1–30; our example corresponds to  $a = (10010100)_2$ ,  $b = (01001001)_2$ . The problem of enumerating all subsets of a set is the special case where  $a = \chi$  and  $b = 0$ . In the more general subcube problem, the successor of a given bit pattern  $x$  is

$$x' = ((x - (a + b)) \& a) + b. \quad (85)$$

Suppose the bits of  $z = (z_{n-1} \dots z_0)_2$  have been stitched together from two subwords  $x = (x_{r-1} \dots x_0)_2$  and  $y = (y_{s-1} \dots y_0)_2$ , where  $r + s = n$ , using an arbitrary mask  $\chi$  for which  $\nu\chi = s$  to govern the stitching. For example,  $z = (y_2x_4x_3y_1x_2y_0x_1x_0)_2$  when  $n = 8$  and  $\chi = (10010100)_2$ . We can think of  $z$  as a “scattered accumulator,” in which alien bits  $x_i$  lurk among friendly bits  $y_j$ . From this viewpoint the problem of finding successive elements of a subcube is essentially the problem of computing  $y + 1$  inside a scattered accumulator  $z$ , without changing the value of  $x$ . The sheep-and-goats operation (81) would untangle  $x$  and  $y$ ; but it's expensive, and (85) shows that we can solve the problem without it. We can, in fact, compute  $y + y'$  when  $y' = (y'_{s-1} \dots y'_0)_2$  is *any* value inside a scattered accumulator  $z'$ , if  $y$  and  $y'$  both appear in the positions specified by  $\chi$ : Consider  $t = z \& \chi$  and  $t' = z' \& \chi$ . If we form the sum  $(t \mid \bar{\chi}) + t'$ , all carries that occur in a normal addition  $y + y'$  will propagate

recursively  
fragmented fields  
subsets  
mask  
lexicographic order  
subcube  
don't-cares  
asterisk codes  
bit codes  
scattered accumulator  
sheep-and-goats  
carries

through the blocks of 1s in  $\bar{\chi}$ , just as if the scattered bits were adjacent. Thus

$$((z \& \chi) + (z' \mid \bar{\chi})) \& \chi \quad (86)$$

is the sum of  $y$  and  $y'$ , modulo  $2^s$ , scattered according to the mask  $\chi$ .

**Tweaking several bytes at once.** Instead of concentrating on the data in one field within a word, we often want to deal simultaneously with two or more subwords, performing calculations on each of them in parallel. For example, many applications need to process long sequences of bytes, and we can gain speed by acting on eight bytes at a time; we might as well use all 64 bits that our machine provides. General multibyte techniques were introduced by Leslie Lamport in *CACM* **18** (1975), 471–475, and subsequently extended by many programmers.

Suppose first that we simply wish to take two sequences of bytes and find their sum, regarding them as coordinates of vectors, doing arithmetic modulo 256 in each byte. Algebraically speaking, we're given 8-byte vectors  $x = (x_7 \dots x_1 x_0)_{256}$  and  $y = (y_7 \dots y_1 y_0)_{256}$ ; we want to compute  $z = (z_7 \dots z_1 z_0)_{256}$ , where  $z_j = (x_j + y_j) \bmod 256$  for  $0 \leq j < 8$ . Ordinary addition of  $x$  to  $y$  doesn't quite work, because we need to prevent carries from propagating between bytes. So we extract the high-order bits and deal with them separately:

$$\begin{aligned} z &\leftarrow (x \oplus y) \& h, & \text{where } h = \#8080808080808080; \\ z &\leftarrow ((x \& \bar{h}) + (y \& \bar{h})) \oplus z. \end{aligned} \quad (87)$$

The total time for MMIX to do this is  $6v$ , plus  $3\mu + 3v$  if we also count the time to load  $x$ , load  $y$ , and store  $z$ . By contrast, eight one-byte additions (LDBU, LDBU, ADDU, and STBU, repeated eight times) would cost  $8 \times (3\mu + 4v) = 24\mu + 32v$ . Parallel *subtraction* of bytes is just as easy (see exercise 88).

We can also compute bitwise *averages*, with  $z_j = \lfloor (x_j + y_j)/2 \rfloor$  for each  $j$ :

$$\begin{aligned} z &\leftarrow ((x \oplus y) \& \bar{l}) \ggg 1, & \text{where } l = \#0101010101010101; \\ z &\leftarrow (x \& y) + z. \end{aligned} \quad (88)$$

This elegant trick, suggested by H. G. Dietz, is based on the well-known formula

$$x + y = (x \oplus y) + ((x \& y) \lll 1) \quad (89)$$

for radix-2 addition. (We can implement (88) with four MMIX instructions, not five, because a single MOR operation will change  $x \oplus y$  to  $((x \oplus y) \& \bar{l}) \ggg 1$ .)

Exercises 88–93 and 100–104 develop these ideas further, showing how to do mixed-radix arithmetic, as well as such things as the addition and subtraction of vectors whose components are treated modulo  $m$  when  $m$  needn't be a power of 2.

In essence, we can regard the bits, bytes, or other subfields of a register as if they were elements of an array of independent microprocessors, acting independently on their own subproblems yet tightly synchronized, and communicating with each other via shift instructions and carry bits. Computer designers have been interested for many years in the development of parallel processors with a so-called SIMD architecture, namely a “Single Instruction stream with Multiple Data streams”; see, for example, S. H. Unger, *Proc. IRE* **46** (1958), 1744–1750.

scattered sum  
packed data, operating on++  
Lamport  
parallel processing of subwords  
multibyte processing  
multibyte addition  
carries  
averages  
Dietz  
radix-2 addition  
MMIX  
MOR  
shift instructions  
carry bits  
parallel processors  
SIMD architecture  
Unger



The increased availability of 64-bit registers has meant that programmers of ordinary sequential computers are now able to get a taste of SIMD processing. Indeed, computations such as (87), (88), and (89) are called SWAR methods — “SIMD Within A Register,” a name coined by R. J. Fisher and H. G. Dietz [see *Lecture Notes in Computer Science* **1656** (1999), 290–305].

Of course bytes often contain alphabetic data as well as numbers, and one of the most common programming tasks is to search through a long string of characters in order to find the first appearance of some particular byte value. For example, strings are often represented as a sequence of nonzero bytes terminated by 0. In order to locate the end of a string quickly, we need a fast way to determine whether all eight bytes of a given word  $x$  are nonzero (because they usually are). Several fairly good solutions to this problem were found by Lamport and others; but Alan Mycroft discovered in 1987 that *three* instructions actually suffice:

$$t \leftarrow h \& (x - l) \& \bar{x}, \quad (90)$$

where  $h$  and  $l$  appear in (87) and (88). If each byte  $x_j$  is nonzero,  $t$  will be zero; for  $(x_j - 1) \& \bar{x}_j$  will be  $2^{\rho x_j} - 1$ , which is always less than  $\#80 = 2^7$ . But if  $x_j = 0$ , while its right neighbors  $x_{j-1}, \dots, x_0$  (if any) are all nonzero, the subtraction  $x - l$  will produce  $\#ff$  in byte  $j$ , and  $t$  will be nonzero. In fact,  $\rho t$  will be  $8j + 7$ .

*Caution:* Although the computation in (90) pinpoints the *rightmost* zero byte of  $x$ , we cannot deduce the position of the *leftmost* zero byte from the value of  $t$  alone. (See exercise 94.) In this respect the little-endian convention proves to be preferable to the corresponding big-endian behavior. An application that needs to locate the leftmost zero byte can use (90) to skip quickly over nonzeros, but then it must fall back on a slower method when the search has been narrowed down to eight finalists. The following 4-operation formula produces a completely precise test value  $t = (t_7 \dots t_1 t_0)_{256}$ , in which  $t_j = 128[x_j = 0]$  for each  $j$ :

$$t \leftarrow h \& \sim(x \mid ((x \mid h) - l)). \quad (91)$$

The leftmost zero byte of  $x$  is now  $x_j$ , where  $\lambda t = 8j + 7$ .

Incidentally, the single MMIX instruction ‘BDIF  $t, l, x$ ’ solves the zero-byte problem immediately by setting each byte  $t_j$  of  $t$  to  $[x_j = 0]$ , because  $1 \dot{-} x = [x = 0]$ . But we are primarily interested here in fairly universal techniques that don’t rely on exotic hardware; MMIX’s special features will be discussed later.

Now that we know a fast way to find the first 0, we can use the same ideas to search for *any* desired byte value. For example, to test if any byte of  $x$  is the newline character ( $\#a$ ), we simply look for a zero byte in  $x \oplus \#0a0a0a0a0a0a0a$ .

And these techniques also open up many other doors. Suppose, for instance, that we want to compute  $z = (z_7 \dots z_1 z_0)_{256}$  from  $x$  and  $y$ , where  $z_j = x_j$  when  $x_j = y_j$  but  $z_j = '*'$  when  $x_j \neq y_j$ . (Thus if  $x = \text{"beaching"}$  and  $y = \text{"belching"}$ , we’re supposed to set  $z \leftarrow \text{"be*ching"}$ .) It’s easy:

$$\begin{aligned} t &\leftarrow h \& ((x \oplus y) \mid (((x \oplus y) \mid h) - l)); \\ m &\leftarrow (t \leq 1) - (t \geq 7); \\ z &\leftarrow x \oplus ((x \oplus \text{"*****"}) \& m). \end{aligned} \quad (92)$$

bit slices  
SWAR  
Fisher  
Dietz  
alphabetic data  
strings  
Lamport  
Mycroft  
ruler function  
little-endian convention  
big-endian  
dotminus  
equality of bytes  
newline

The first step uses a variant of (91) to flag the high-order bits in each byte where  $x_j \neq y_j$ . The next step creates a mask to highlight those bytes: #00 if  $x_j = y_j$ , otherwise #ff. And the last step, which could also be written  $z \leftarrow (x \& \overline{m}) \mid ("*****" \& m)$ , sets  $z_j \leftarrow x_j$  or  $z_j \leftarrow 'f'$ , depending on the mask.

Operations (90) and (91) were originally designed as tests for bytes that are zero; but a closer look reveals that we can more wisely regard them as tests for bytes that are less than 1. Indeed, if we replace  $l$  by  $c \cdot l = (ccccccc)_{256}$  in either formula, where  $c$  is any positive constant  $\leq 128$ , we can use (90) or (91) to see if  $x$  contains any bytes that are less than  $c$ . Furthermore the comparison values  $c$  need not be the same in every byte position; and with a bit more work we can also do bitwise comparison in the cases where  $c > 128$ . Here's an 8-step formula that sets  $t_j \leftarrow 128[x_j < y_j]$  for each byte position  $j$  in the test word  $t$ :

$$t \leftarrow h \& \sim \langle x \bar{y} z \rangle, \quad \text{where } z = (x \mid h) - (y \& \bar{h}). \quad (93)$$

(See exercise 96.) The median operation in this general formula can often be simplified; for example, (93) reduces to (91) when  $y = l$ , because  $\langle x(-1)z \rangle = x \mid z$ .

Once we've found a nonzero  $t$  in (90) or (91) or (93), we might want to compute  $\rho t$  or  $\lambda t$  in order to discover the index  $j$  of the rightmost or leftmost byte that has been flagged. The problem of calculating  $\rho$  or  $\lambda$  is now simpler than before, since  $t$  can take on only 256 different values. Indeed, the operation

$$j \leftarrow \text{table}[(a \cdot t) \bmod 2^{64}] \gg 56], \quad \text{where } a = \frac{2^{56} - 1}{2^7 - 1}, \quad (94)$$

now suffices to compute  $j$ , given an appropriate 256-byte table. And the multiplication here can often be performed faster by doing three shift-and-add operations, " $t \leftarrow t + (t \ll 7)$ ,  $t \leftarrow t + (t \ll 14)$ ,  $t \leftarrow t + (t \ll 28)$ ," instead.

**Broadword computing.** We've now seen more than a dozen ways in which a computer's bitwise operations can produce astonishing results at high speed, and the exercises below contain many more such surprises.

Elwyn Berlekamp has remarked that computer chips containing  $N$  flip-flops continue to be built with ever larger values of  $N$ , yet in practice only  $O(\log N)$  of those components are flipping or flopping at any given moment. The surprising effectiveness of bitwise operations suggests that computers of the future might make use of this untapped potential by having enhanced memory units that are able to do efficient  $n$ -bit computations for fairly large values of  $n$ . To prepare for that day, we ought to have a good name for the concept of manipulating "wide words." Lyle Ramshaw has suggested the pleasant term *broadword*, so that we can speak of  $n$ -bit quantities as broadwords of width  $n$ .

Many of the methods we've discussed are *2-adic*, in the sense that they work correctly with binary numbers that have arbitrary (even infinite) precision. For example, the operation  $x \& -x$  always extracts  $2^{\rho x}$ , the least significant 1 bit of any nonzero 2-adic integer  $x$ . But other methods have an inherently broadword nature, such as the methods that use  $O(d)$  steps to perform sideways addition or bit permutation of  $2^d$ -bit words. Broadword computing is the art of dealing with  $n$ -bit words, when  $n$  is a parameter that is not extremely small.

flag: A 1-bit indicator  
mask  
comparison of bytes  
bytes, testing relative order of  
median operation  
 $\rho$   
 $\lambda$   
Berlekamp  
Ramshaw  
2-adic

Some broadword algorithms are of theoretical interest only, because they are efficient only in an asymptotic sense when  $n$  exceeds the size of the universe. But others are eminently practical even when  $n = 64$ . And in general, a broadword mindset often suggests good techniques.

One fascinating-but-impractical fact about broadword operations is the discovery by M. L. Fredman and D. E. Willard that  $O(1)$  broadword steps suffice to evaluate the function  $\lambda x = \lfloor \lg x \rfloor$  for any nonzero  $n$ -bit number  $x$ , no matter how big  $n$  is. Here is their remarkable scheme, when  $n = g^2$  and  $g$  is a power of 2:

$$\begin{aligned}
 t_1 &\leftarrow h \& (x \mid ((x \mid h) - l)), \quad \text{where } h = 2^{g-1}l \text{ and } l = (2^n - 1)/(2^g - 1); \\
 y &\leftarrow (((a \cdot t_1) \bmod 2^n) \gg (n - g)) \cdot l, \quad \text{where } a = (2^{n-g} - 1)/(2^{g-1} - 1); \\
 t_2 &\leftarrow h \& (y \mid ((y \mid h) - b)), \quad \text{where } b = (2^{n+g} - 1)/(2^{g+1} - 1); \\
 m &\leftarrow (t_2 \ll 1) - (t_2 \gg (g - 1)), \quad m \leftarrow m \oplus (m \gg g); \\
 z &\leftarrow (((l \cdot (x \& m)) \bmod 2^n) \gg (n - g)) \cdot l; \\
 t_3 &\leftarrow h \& (z \mid ((z \mid h) - b)); \\
 \lambda &\leftarrow ((l \cdot ((t_2 \gg (2g - \lg g - 1)) + (t_3 \gg (2g - 1)))) \bmod 2^n) \gg (n - g).
 \end{aligned} \tag{95}$$

(See exercise 106.) The method fails to be practical because five of these 29 steps are multiplications, so they aren't really "bitwise" operations. In fact, we'll prove later that multiplication by a constant requires at least  $\Omega(\log n)$  bitwise steps.

A multiplication-free way to find  $\lambda x$ , with only  $O(\log \log n)$  bitwise broadword operations, was discovered in 1997 by Gerth Brodal, whose method is even more remarkable than (95). It is based on a formula analogous to (49),

$$\lambda x = \lfloor \lambda x = \lambda(x \& \bar{\mu}_0) \rfloor + 2 \lfloor \lambda x = \lambda(x \& \bar{\mu}_1) \rfloor + 4 \lfloor \lambda x = \lambda(x \& \bar{\mu}_2) \rfloor + \cdots, \tag{96}$$

and the fact that the relation  $\lambda x = \lambda y$  is easily tested (see (58)):

**Algorithm B** (*Binary logarithm*). This algorithm uses  $n$ -bit operations to compute  $\lambda x = \lfloor \lg x \rfloor$ , assuming that  $0 < x < 2^n$  and  $n = d \cdot 2^d$ .

- B1.** [Scale down.] Set  $\lambda \leftarrow 0$ . Then set  $\lambda \leftarrow \lambda + 2^k$  and  $x \leftarrow x \gg 2^k$  if  $x \geq 2^{2^k}$ , for  $k = \lceil \lg n \rceil - 1, \lceil \lg n \rceil - 2, \dots, d$ .
- B2.** [Replicate.] (At this point  $0 < x < 2^{2^d}$ ; the remaining task is to increase  $\lambda$  by  $\lfloor \lg x \rfloor$ . We will replace  $x$  by  $d$  copies of itself, in  $2^d$ -bit fields.) Set  $x \leftarrow x \mid (x \ll 2^{d+k})$  for  $0 \leq k < \lceil \lg d \rceil$ .
- B3.** [Change leading bits.] Set  $y \leftarrow x \& \sim(\mu_{d,d-1} \dots \mu_{d,1} \mu_{d,0})_{2^{2^d}}$ . (See (48).)
- B4.** [Compare all fields.] Set  $t \leftarrow h \& (y \mid ((y \mid h) - (x \oplus y)))$ , where  $h = (2^{2^d-1} \dots 2^{2^{d-1}-1} 2^{2^{d-1}-1})_{2^{2^d}}$ .
- B5.** [Compress bits.] Set  $t \leftarrow (t + (t \ll (2^{d+k} - 2^k))) \bmod 2^n$  for  $0 \leq k < \lceil \lg d \rceil$ .
- B6.** [Finish.] Finally, set  $\lambda \leftarrow \lambda + (t \gg (n - d))$ . ■

This algorithm is almost competitive with (56) when  $n = 64$  (see exercise 107).

Another surprisingly efficient broadword algorithm was discovered in 2006 by M. S. Paterson and the author, who considered the problem of identifying all occurrences of the pattern  $01^r$  in a given  $n$ -bit binary string. This problem, which is related to storage allocation, is equivalent to computing

$$q = \bar{x} \& (x \ll 1) \& (x \ll 2) \& (x \ll 3) \& \cdots \& (x \ll r) \tag{97}$$

Fredman  
Willard  
Brodal  
Paterson  
Knuth, DE  
pattern  
storage allocation

when  $x = (x_{n-1} \dots x_1 x_0)_2$  is given. For example, when  $n = 16$ ,  $r = 3$ , and  $x = (1110111101100111)_2$ , we have  $q = (0001000000001000)_2$ . One might expect intuitively that  $\Omega(\log r)$  bitwise operations would be needed. But in fact the following 20-step computation does the job for all  $n > r > 0$ : Let  $s = \lceil r/2 \rceil$ ,  $l = \sum_{k \geq 0} 2^{ks} \bmod 2^n$ ,  $h = (2^{s-1}l) \bmod 2^n$ , and  $a = (\sum_{k \geq 0} (-1)^{k+1} 2^{2ks}) \bmod 2^n$ .

$$\begin{aligned}
 y &\leftarrow h \& x \& ((x \& \bar{h}) + l); \\
 t &\leftarrow (x + y) \& \bar{x} \& -2^r; \\
 u &\leftarrow t \& a, \quad v \leftarrow t \& \bar{a}; \\
 m &\leftarrow (u - (u \gg r)) \mid (v - (v \gg r)); \\
 q &\leftarrow t \& ((x \& m) + ((t \gg r) \& \sim(m \ll 1))).
 \end{aligned} \tag{98}$$

Exercise 111 explains why these machinations are valid. The method has little or no practical value; there's an easy way to evaluate (97) in  $2\lceil \lg r \rceil + 2$  steps, so (98) is not advantageous until  $r > 512$ . But (98) is another indication of the unexpected power of broadword methods.

**\*Lower bounds.** Indeed, the existence of so many tricks and techniques makes it natural to wonder whether we've only been scratching the surface. Are there many more incredibly fast methods, still waiting to be discovered? A few theoretical results are known by which we can derive certain limitations on what is possible, although such studies are still in their infancy.

Let's say that a *2-adic chain* is a sequence  $(x_0, x_1, \dots, x_r)$  of 2-adic integers in which each element  $x_i$  for  $i > 0$  is obtained from its predecessors via bitwise manipulation. More precisely, we want the steps of the chain to be defined by binary operations

$$x_i = x_{j(i)} \circ_i x_{k(i)} \quad \text{or} \quad c_i \circ_i x_{k(i)} \quad \text{or} \quad x_{j(i)} \circ_i c_i, \tag{99}$$

where each  $\circ_i$  is one of the operators  $\{+, -, \&, \mid, \oplus, \equiv, \subset, \supset, \overline{\subset}, \overline{\supset}, \overline{\&}, \overline{\mid}, \ll, \gg\}$  and each  $c_i$  is a constant. Furthermore, when the operator  $\circ_i$  is a left shift or right shift, the amount of shift must be a positive integer constant; operations such as  $x_{j(i)} \ll x_{k(i)}$  or  $c_i \gg x_{k(i)}$  are *not* permitted. (Without the latter restriction we couldn't derive meaningful lower bounds, because *every* 0–1 valued function of a nonnegative integer  $x$  would be computable in two steps as “ $(c \gg x) \& 1$ ” for some constant  $c$ .)

Similarly, a *broadword chain* of width  $n$ , also called an  $n$ -bit broadword chain, is a sequence  $(x_0, x_1, \dots, x_r)$  of  $n$ -bit numbers subject to essentially the same restrictions, where  $n$  is a parameter and all operations are performed modulo  $2^n$ . Broadword chains behave like 2-adic chains in many ways, but subtle differences can arise because of the information loss that occurs at the left of  $n$ -bit computations (see exercise 113).

Both types of chains compute a function  $f(x) = x_r$  when we start them out with a given value  $x = x_0$ . Exercise 114 shows that an  $mn$ -bit broadword chain is able to do  $m$  essentially simultaneous evaluations of any function that is computable with an  $n$ -bit chain. Our goal is to study the *shortest* chains that are able to evaluate a given function  $f$ .

2-adic chain++++  
broadword chain++++  
branchless+++  
table lookup by shifting

Any 2-adic or broadword chain  $(x_0, x_1, \dots, x_r)$  has a sequence of “shift sets”  $(S_0, S_1, \dots, S_r)$  and “bounds”  $(B_0, B_1, \dots, B_r)$ , defined as follows: Start with  $S_0 = \{0\}$  and  $B_0 = 1$ ; then for  $i \geq 1$ , let

$$S_i = \begin{cases} S_{j(i)} \cup S_{k(i)}, \\ S_{k(i)}, \\ S_{j(i)}, \\ S_{j(i)} + c_i, \\ S_{j(i)} - c_i, \end{cases} \quad \text{and} \quad B_i = \begin{cases} M_i B_{j(i)} B_{k(i)}, & \text{if } x_i = x_{j(i)} \circ_i x_{k(i)}, \\ M_i B_{k(i)}, & \text{if } x_i = c_i \circ_i x_{k(i)}, \\ M_i B_{j(i)}, & \text{if } x_i = x_{j(i)} \circ_i c_i, \\ B_{j(i)}, & \text{if } x_i = x_{j(i)} \gg c_i, \\ B_{j(i)}, & \text{if } x_i = x_{j(i)} \ll c_i, \end{cases} \quad (100)$$

where  $M_i = 2$  if  $\circ_i \in \{+, -\}$  and  $M_i = 1$  otherwise, and these formulas assume that  $\circ_i \notin \{\ll, \gg\}$ . For example, consider the following 7-step chain:

$x_i$	$S_i$	$B_i$	
$x_0 = x$	$\{0\}$	1	
$x_1 = x_0 \& -2$	$\{0\}$	1	
$x_2 = x_1 + 2$	$\{0\}$	2	
$x_3 = x_2 \gg 1$	$\{1\}$	2	(101)
$x_4 = x_2 + x_3$	$\{0, 1\}$	8	
$x_5 = x_4 \gg 4$	$\{4, 5\}$	8	
$x_6 = x_4 + x_5$	$\{0, 1, 4, 5\}$	128	
$x_7 = x_6 \gg 4$	$\{4, 5, 8, 9\}$	128	

(We encountered this chain in exercise 4.4–9, which proved that these operations will yield  $x_7 = \lfloor x/10 \rfloor$  for  $0 \leq x < 160$  when performed with 8-bit arithmetic.)

To begin a theory of lower bounds, let's notice first that the high-order bits of  $x = x_0$  cannot influence any low-order bits unless we shift them to the right.

**Lemma A.** *Given a 2-adic or broadword chain, let the binary representation of  $x_i$  be  $(\dots x_{i2}x_{i1}x_{i0})_2$ . Then bit  $x_{ip}$  can depend on bit  $x_{0q}$  only if  $q \leq p + \max S_i$ .*

*Proof.* By induction on  $i$  we can in fact show that, if  $B_i = 1$ , bit  $x_{ip}$  can depend on bit  $x_{0q}$  only if  $q - p \in S_i$ . Addition and subtraction, which force  $B_i > 1$ , allow any particular bit of their operands to affect all bits that lie to the left in the sum or difference, but not those that lie to the right. ■

**Corollary I.** *The function  $x \div 1$  cannot be computed by a 2-adic chain, nor can any function for which at least one bit of  $f(x)$  depends on an unbounded number of bits of  $x$ . ■*

**Corollary W.** *An  $n$ -bit function  $f(x)$  can be computed by an  $n$ -bit broadword chain without shifts if and only if  $x \equiv y \pmod{2^p}$  implies  $f(x) \equiv f(y) \pmod{2^p}$  for  $0 \leq p < n$ .*

*Proof.* If there are no shifts we have  $S_i = \{0\}$  for all  $i$ . Thus bit  $x_{rp}$  cannot depend on bit  $x_{0q}$  unless  $q \leq p$ . In other words we must have  $x_r \equiv y_r \pmod{2^p}$  whenever  $x_0 \equiv y_0 \pmod{2^p}$ .

Conversely, all such functions are achievable by a sufficiently long chain. Exercise 119 gives shift-free  $n$ -bit chains for the functions

$$f_{py}(x) = 2^p[x \bmod 2^{p+1} = y], \quad \text{when } 0 \leq p < n \text{ and } 0 \leq y < 2^{p+1}, \quad (102)$$

shift sets  
division, by 10  
monus

from which all the relevant functions arise by addition. [H. S. Warren, Jr., generalized this result to functions of  $m$  variables in *CACM* **20** (1977), 439–441.] ■

Shift sets  $S_i$  and bounds  $B_i$  are important chiefly because of a fundamental lemma that is our principal tool for proving lower bounds:

**Lemma B.** *Let  $X_{pqr} = \{x_r \& \lfloor 2^p - 2^q \rfloor \mid x_0 \in V_{pqr}\}$  in an  $n$ -bit broadword chain, where*

$$V_{pqr} = \{x \mid x \& \lfloor 2^{p+s} - 2^{q+s} \rfloor = 0 \text{ for all } s \in S_r\} \quad (103)$$

and  $p > q$ . Then  $|X_{pqr}| \leq B_r$ . (Here  $p$  and  $q$  are integers, possibly negative.)

This lemma states that at most  $B_r$  different bit patterns  $x_{r(p-1)} \dots x_{rq}$  can occur within  $f(x)$ , when certain intervals of bits in  $x$  are constrained to be zero.

*Proof.* The result certainly holds when  $r = 0$ . Otherwise if, for example,  $x_r = x_j + x_k$ , we know by induction that  $|X_{pqj}| \leq B_j$  and  $|X_{pqk}| \leq B_k$ . Furthermore  $V_{pqr} = V_{pqj} \cap V_{pqk}$ , since  $S_r = S_j \cup S_k$ . Thus at most  $B_j B_k$  possibilities for  $(x_j + x_k) \& \lfloor 2^p - 2^q \rfloor$  arise when there's no carry into position  $q$ , and at most  $B_j B_k$  when there is a carry, making a grand total of at most  $B_r = 2B_j B_k$  possibilities altogether. Exercise 122 considers the other cases. ■

We now can prove that the ruler function needs  $\Omega(\log \log n)$  steps.

**Theorem R.** *If  $n = d \cdot 2^d$ , every  $n$ -bit broadword chain that computes  $\rho x$  for  $0 < x < 2^n$  has more than  $\lg d$  steps that are not shifts.*

*Proof.* If there are  $l$  nonshift steps, we have  $|S_r| \leq 2^l$  and  $B_r \leq 2^{2^l-1}$ . Apply Lemma B with  $p = d$  and  $q = 0$ , and suppose  $|X_{d0r}| = 2^d - t$ . Then there are  $t$  values of  $k < 2^d$  such that

$$\{2^k, 2^{k+2^d}, 2^{k+2 \cdot 2^d}, \dots, 2^{k+(d-1)2^d}\} \notin V_{d0r}.$$

But  $V_{d0r}$  excludes at most  $2^l d$  of the  $n$  possible powers of 2; so  $t \leq 2^l$ .

If  $l \leq \lg d$ , Lemma B tells us that  $2^d - t \leq B_r \leq 2^{2^l-1}$ ; hence  $2^{d-1} \leq t \leq 2^l \leq d$ . But this is impossible unless  $d \leq 2$ , when the theorem clearly holds. ■

The same proof works also for the binary logarithm function:

**Corollary L.** *If  $n = d \cdot 2^d > 2$ , every  $n$ -bit broadword chain that computes  $\lambda x$  for  $0 < x < 2^n$  has more than  $\lg d$  steps that are not shifts.* ■

By using Lemma B with  $q > 0$  we can derive the stronger lower bound  $\Omega(\log n)$  for bit reversal, and hence for bit permutation in general.

**Theorem P.** *If  $2 \leq g \leq n$ , every  $n$ -bit broadword chain that computes the  $g$ -bit reversal  $x^R$  for  $0 \leq x < 2^g$  has at least  $\lfloor \frac{1}{3} \lg g \rfloor$  steps that are not shifts.*

*Proof.* Assume as above that there are  $l$  nonshifts. Let  $h = \lfloor \sqrt[3]{g} \rfloor$  and suppose that  $l < \lfloor \lg(h+1) \rfloor$ . Then  $S_r$  is a set of at most  $2^l \leq \frac{1}{2}(h+1)$  shift amounts  $s$ . We shall apply Lemma B with  $p = q + h$ , where  $p \leq g$  and  $q \geq 0$ , thus in  $g - h + 1$  cases altogether. The key observation is that  $x^R \& \lfloor 2^p - 2^q \rfloor$  is independent of  $x \& \lfloor 2^{p+s} - 2^{q+s} \rfloor$  whenever there are no indices  $j$  and  $k$  such that  $0 \leq j, k < h$  and  $g - 1 - q - j = q + s + k$ . The number of “bad” choices of  $q$  for which such

Warren  
carry  
ruler function  
binary logarithm  
reversal  
bit permutation

indices exist is at most  $\frac{1}{2}(h+1)h^2 \leq g-h$ ; therefore at least one “good” choice of  $q$  yields  $|X_{pqr}| = 2^h$ . But then Lemma B leads to a contradiction, because we obviously cannot have  $2^h \leq B_r \leq 2^{(h-1)/2}$ . ■

**Corollary M.** *Multiplication by certain constants, modulo  $2^n$ , requires  $\Omega(\log n)$  steps in an  $n$ -bit broadword chain.*

*Proof.* In Hack 167 of the classic memorandum HAKMEM (M.I.T. A.I. Laboratory, 1972), Richard Schroeppel observed that the operations

$$t \leftarrow ((ax) \bmod 2^n) \& b, \quad y \leftarrow ((ct) \bmod 2^n) \gg (n-g) \quad (104)$$

compute  $y = x^R$  whenever  $n = g^2$  and  $0 \leq x < 2^g$ , using the constants  $a = (2^{n+g} - 1)/(2^{g+1} - 1)$ ,  $b = 2^{g-1}(2^n - 1)/(2^g - 1)$ , and  $c = (2^{n-g} - 1)/(2^{g-1} - 1)$ . (See exercise 123.) ■

At this point the reader might well be thinking, “Okay, I agree that broadword chains sometimes have to be asymptotically long. But programmers needn’t be shackled by such chains; we can use other techniques, like conditional branches or references to precomputed tables, which go beyond those restrictions.”

Right. And we’re in luck, because broadword theory can also be extended to more general models of computation. Consider, for example, the following idealization of an abstract reduced-instruction-set computer, called a *basic RAM*: The machine has  $n$ -bit registers  $r_1, \dots, r_l$ , and  $n$ -bit memory words  $\{M[0], \dots, M[2^m - 1]\}$ . It can perform the instructions

$$\begin{aligned} r_i &\leftarrow r_j \pm r_k, & r_i &\leftarrow r_j \circ r_k, & r_i &\leftarrow r_j \gg r_k, & r_i &\leftarrow c, \\ r_i &\leftarrow M[r_j \bmod 2^m], & M[r_j \bmod 2^m] &\leftarrow r_i, \end{aligned} \quad (105)$$

where  $\circ$  is any bitwise Boolean operator, and where  $r_k$  in the shift instruction is treated as a signed integer in two’s complement notation. The machine is also able to branch if  $r_i \leq r_j$ , treating  $r_i$  and  $r_j$  as unsigned integers. Its *state* is the entire contents of all registers and memory, together with a “program counter” that points to the current instruction. Its program begins in a designated state, which may include precomputed tables in memory, and with an  $n$ -bit input value  $x$  in register  $r_1$ . This initial state is called  $Q(x, 0)$ , and  $Q(x, t)$  denotes the state after  $t$  instructions have been performed. When the machine stops,  $r_1$  will contain some  $n$ -bit value  $f(x)$ . Given a function  $f(x)$ , we want to find a lower bound on the least  $t$  such that  $r_1$  is equal to  $f(x)$  in state  $Q(x, t)$ , for  $0 \leq x < 2^n$ .

**Theorem R’.** *Let  $\epsilon = 2^{-e}$ . A basic  $n$ -bit RAM with memory parameter  $m \leq n^{1-\epsilon}$  requires at least  $\lg \lg n - e$  steps to evaluate the ruler function  $\rho x$ , as  $n \rightarrow \infty$ .*

*Proof.* Let  $n = 2^{2^{e+f}}$ , so that  $m \leq 2^{2^{e+f}-2^f}$ . Exercise 124 explains how an omniscient observer can construct a broadword chain from a certain class of inputs  $x$ , in such a way that each  $x$  causes the RAM to take the same branches, use the same shift amounts, and refer to the same memory locations. Our earlier methods can then be used to show that this chain has length  $\geq f$ . ■

A skeptical reader may still object that Theorem R’ has no practical value, because  $\lg \lg n$  never exceeds 6 in the real world. To this argument there is no rebuttal. But the following result is slightly more relevant:

HAKMEM  
Schroeppel  
abstract reduced-instruction-set computer  
basic RAM  
two’s complement notation  
program counter  
ruler function

**Theorem P'.** A basic  $n$ -bit RAM requires at least  $\frac{1}{3} \lg g$  steps to compute the  $g$ -bit reversal  $x^R$  for  $0 \leq x < 2^g$ , if  $g \leq n$  and

$$\max(m, 1 + \lg n) < \frac{h+1}{2 \lfloor \lg(h+1) \rfloor - 2}, \quad h = \lfloor \sqrt[3]{g} \rfloor. \quad (106)$$

*Proof.* An argument like the proof of Theorem R' appears in exercise 125. ■

Lemma B and Theorems R, P, R', P' and their corollaries are due to A. Brodnik, P. B. Miltersen, and J. I. Munro, *Lecture Notes in Comp. Sci.* **1272** (1997), 426–439, based on earlier work of Miltersen in *Lecture Notes in Comp. Sci.* **1099** (1996), 442–451.

Many unsolved questions remain (see exercises 126–130). For example, does sideways addition require  $\Omega(\log n)$  steps in an  $n$ -bit broadword chain? Can the parity function  $(\nu x) \bmod 2$ , or the majority function  $\lfloor \nu x > n/2 \rfloor$ , be computed substantially faster than  $\nu x$  itself, broadwordwise?

**An application to directed graphs.** Now let's use some of what we've learned, by implementing a simple algorithm. Given a digraph on a set of vertices  $V$ , we write  $u \rightarrow v$  when there's an arc from  $u$  to  $v$ . The *reachability problem* is to find all vertices that lie on oriented paths beginning in a specified set  $Q \subseteq V$ ; in other words, we seek the set

$$R = \{v \mid u \rightarrow^* v \text{ for some } u \in Q\}, \quad (107)$$

where  $u \rightarrow^* v$  means that there is a sequence of  $t$  arcs

$$u = u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_t = v, \quad \text{for some } t \geq 0. \quad (108)$$

This problem arises frequently in practice. For example, we encountered it in Section 2.3.5 when marking all elements of Lists that are not “garbage.”

If the number of vertices is small, say  $|V| \leq 64$ , we may want to approach the reachability problem in quite a different way than we did before, by working directly with subsets of vertices. Let

$$S[u] = \{v \mid u \rightarrow v\} \quad (109)$$

be the set of successors of vertex  $u$ , for all  $u \in V$ . Then the following algorithm is almost completely different from Algorithm 2.3.5E, yet it solves the same abstract problem:

**Algorithm R (Reachability).** Given a simple directed graph, represented by the successor sets  $S[u]$  in (109), this algorithm computes the elements  $R$  that are reachable from a given set  $Q$ .

**R1.** [Initialize.] Set  $R \leftarrow Q$  and  $X \leftarrow \emptyset$ . (In the following steps,  $X$  is the subset of vertices  $u \in R$  for which we've looked at  $S[u]$ .)

**R2.** [Done?] If  $X = R$ , the algorithm terminates.

**R3.** [Examine another vertex.] Let  $u$  be an element of  $R \setminus X$ . Set  $X \leftarrow X \cup \{u\}$ ,  $R \leftarrow R \cup S[u]$ , and return to step R2. ■

The algorithm is correct because (i) every element placed into  $R$  is reachable; (ii) every reachable element  $u_j$  in (108) is present in  $R$ , by induction on  $j$ ; and (iii) termination eventually occurs, because step R3 always increases  $|X|$ .

Brodnik  
Miltersen  
Munro  
sideways addition  
parity function  
majority function  
graph algorithms+  
reachability problem  
oriented paths  
transitive closure  
notation  $u \rightarrow^* v$   
garbage



To implement Algorithm R we will assume that  $V = \{0, 1, \dots, n-1\}$ , with  $n \leq 64$ . The set  $X$  is conveniently represented by the integer  $\sigma(X) = \sum \{2^u \mid u \in X\}$ , and the same convention works nicely for the other sets  $Q$ ,  $R$ , and  $S[u]$ . Notice that the bits of  $S[0], S[1], \dots, S[n-1]$  are essentially the *adjacency matrix* of the given digraph, as explained in Section 7, but in little-endian order: The “diagonal” elements, which tell us whether or not  $u \in S[u]$ , go from right to left. For example, if  $n = 3$  and the arcs are  $\{0 \rightarrow 0, 0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 0\}$ , we have  $S[0] = (011)_2$  and  $S[1] = S[2] = (001)_2$ , while the adjacency matrix is  $\begin{pmatrix} 110 \\ 100 \\ 100 \end{pmatrix}$ .

Step R3 allows us to choose any element of  $R \setminus X$ , so we use the ruler function  $u \leftarrow \rho(\sigma(R) - \sigma(X))$  to choose the smallest. The bitwise operations require no further trickery when we adapt the algorithm to MMIX:

**Program R** (*Reachability*). The input set  $Q$  is given in register **q**, and each successor set  $S[u]$  appears in octabyte  $M_8[\text{succ} + 8u]$ . The output set  $R$  will appear in register **r**; other registers **s**, **t**, **tt**, **u**, and **x** hold intermediate results.

01	1H	SET	<b>r</b> , <b>q</b>	1	<u>R1. Initialize.</u> $\mathbf{r} \leftarrow \sigma(Q)$ .
02		SET	<b>x</b> ,0	1	$\mathbf{x} \leftarrow \sigma(\emptyset)$ .
03		JMP	2F	1	To R2.
04	3H	SUBU	<b>tt</b> , <b>t</b> ,1	R	<u>R3. Examine another vertex.</u> $\mathbf{tt} \leftarrow \mathbf{t} - 1$ .
05		SADD	<b>u</b> , <b>tt</b> , <b>t</b>	R	$\mathbf{u} \leftarrow \rho(\mathbf{t})$ [see (46)].
06		SLU	<b>s</b> , <b>u</b> ,3	R	$\mathbf{s} \leftarrow 8\mathbf{u}$ .
07		LDOU	<b>s</b> , <b>succ</b> , <b>s</b>	R	$\mathbf{s} \leftarrow \sigma(S[\mathbf{u}])$ .
08		ANDN	<b>tt</b> , <b>t</b> , <b>tt</b>	R	$\mathbf{tt} \leftarrow \mathbf{t} \& \sim \mathbf{tt} = 2^u$ .
09		OR	<b>x</b> , <b>x</b> , <b>tt</b>	R	$\mathbf{X} \leftarrow \mathbf{X} \cup \{u\}$ ; that is, $\mathbf{x} \leftarrow \mathbf{x} \mid 2^u$ , since $\mathbf{x} = \sigma(X)$ .
10		OR	<b>r</b> , <b>r</b> , <b>s</b>	R	$\mathbf{R} \leftarrow \mathbf{R} \cup S[\mathbf{u}]$ ; that is, $\mathbf{r} \leftarrow \mathbf{r} \mid \mathbf{s}$ , since $\mathbf{r} = \sigma(R)$ .
11	2H	SUBU	<b>t</b> , <b>r</b> , <b>x</b>	R  + 1	<u>R2. Done?</u> $\mathbf{t} \leftarrow \mathbf{r} - \mathbf{x} = \sigma(R \setminus X)$ , since $X \subseteq R$ .
12		PBNZ	<b>t</b> ,3B	R  + 1	To R3 if $R \neq X$ . ■

The total running time is  $(\mu + 9v)|R| + 7v$ . By contrast, exercise 131 implements Algorithm R with linked lists; the overall execution time then grows to  $(3S + 4|R| - 2|Q| + 1)\mu + (5S + 12|R| - 5|Q| + 4)v$ , where  $S = \sum_{u \in R} |S[u]|$ . (But of course that program is also able to handle graphs with millions of vertices.)

Exercise 132 presents another instructive algorithm where bitwise operations work nicely on not-too-large graphs.

**Application to data representation.** Computers are binary, but (alas?) the world isn't. We often must find a way to encode nonbinary data into 0s and 1s. One of the most common problems of this sort is to choose an efficient representation for items that can be in exactly three different states.

Suppose we know that  $x \in \{a, b, c\}$ , and we want to represent  $x$  by two bits  $x_l x_r$ . We could, for example, map  $a \mapsto 00$ ,  $b \mapsto 01$ , and  $c \mapsto 10$ . But there are many other possibilities—in fact, 4 choices for  $a$ , then 3 choices for  $b$ , and 2 for  $c$ , making 24 altogether. Some of these mappings might be much easier to deal with than others, depending on what we want to do with  $x$ .

Given two elements  $x, y \in \{a, b, c\}$ , we typically want to compute  $z = x \circ y$ , for some binary operation  $\circ$ . If  $x = x_l x_r$  and  $y = y_l y_r$  then  $z = z_l z_r$ , where

$$z_l = f_l(x_l, x_r, y_l, y_r) \quad \text{and} \quad z_r = f_r(x_l, x_r, y_l, y_r); \quad (110)$$

adjacency matrix  
little-endian order  
ruler function  
encoding of ternary data  
representing three states with two bits+++  
mapping three items into two-bit codes+++

these Boolean functions  $f_l$  and  $f_r$  of four variables depend on  $\circ$  and the chosen representation. We seek a representation that makes  $f_l$  and  $f_r$  easy to compute.

Suppose, for example, that  $\{a, b, c\} = \{-1, 0, +1\}$  and that  $\circ$  is multiplication. If we decide to use the natural mapping  $x \mapsto x \bmod 3$ , namely

$$0 \mapsto 00, \quad +1 \mapsto 01, \quad -1 \mapsto 10, \quad (111)$$

so that  $x = x_r - x_l$ , then the truth tables for  $f_l$  and  $f_r$  are respectively

$$f_l \leftrightarrow 000*001*010***** \quad \text{and} \quad f_r \leftrightarrow 000*010*001*****. \quad (112)$$

(There are seven “don’t-cares,” for cases where  $x_l x_r = 11$  and/or  $y_l y_r = 11$ .) The methods of Section 7.1.2 tell us how to compute  $z_l$  and  $z_r$  optimally, namely

$$z_l = (x_l \oplus y_l) \wedge (x_r \oplus y_r), \quad z_r = (x_l \oplus y_r) \wedge (x_r \oplus y_l); \quad (113)$$

unfortunately the functions  $f_l$  and  $f_r$  in (112) are independent, in the sense that they cannot both be evaluated in fewer than  $C(f_l) + C(f_r) = 6$  steps.

On the other hand the somewhat less natural mapping scheme

$$+1 \mapsto 00, \quad 0 \mapsto 01, \quad -1 \mapsto 10 \quad (114)$$

leads to the transformation functions

$$f_l \leftrightarrow 001*000*100***** \quad \text{and} \quad f_r \leftrightarrow 010*111*010*****, \quad (115)$$

and three operations now suffice to do the desired evaluation:

$$z_r = x_r \vee y_r, \quad z_l = (x_l \oplus y_l) \wedge \bar{z}_r. \quad (116)$$

Is there an easy way to discover such improvements? Fortunately we don’t need to try all 24 possibilities, because many of them are basically alike. For example, the mapping  $x \mapsto x_r x_l$  is equivalent to  $x \mapsto x_l x_r$ , because the new representation  $x'_l x'_r = x_r x_l$  obtained by swapping coordinates makes

$$f'_l(x'_l, x'_r, y'_l, y'_r) = z'_l = z_r = f_r(x_l, x_r, y_l, y_r);$$

the new transformation functions  $f'_l$  and  $f'_r$  defined by

$$f'_l(x_l, x_r, y_l, y_r) = f_r(x_r, x_l, y_r, y_l), \quad f'_r(x_l, x_r, y_l, y_r) = f_l(x_r, x_l, y_r, y_l) \quad (117)$$

have the same complexity as  $f_l$  and  $f_r$ . Similarly we can complement a coordinate, letting  $x'_l x'_r = \bar{x}_l x_r$ ; then the transformation functions turn out to be

$$f'_l(x_l, x_r, y_l, y_r) = \bar{f}_l(\bar{x}_l, x_r, \bar{y}_l, y_r), \quad f'_r(x_l, x_r, y_l, y_r) = f_r(\bar{x}_l, x_r, \bar{y}_l, y_r), \quad (118)$$

and again the complexity is essentially unchanged.

Repeated use of swapping and/or complementation leads to eight mappings that are equivalent to any given one. So the 24 possibilities reduce to only three, which we shall call classes I, II, and III:

	Class I	Class II	Class III
$a \mapsto$	00 01 10 11 00 10 01 11 00 01 10 11 00 10 01 11 00 01 10 11 00 10 01 11;		
$b \mapsto$	01 00 11 10 10 00 11 01 01 00 11 10 10 00 11 01 11 10 01 00 11 01 10 00;		
$c \mapsto$	10 11 00 01 01 11 00 10 11 10 01 00 11 01 10 00 01 00 11 10 10 00 11 01.		

(119)

multiplication of signed bits+  
signed bits, representation of  
don't-cares  
2-cube equivalence

To choose a representation we need consider only one representative of each class. For example, if  $a = +1$ ,  $b = 0$ , and  $c = -1$ , representation (111) belongs to class II, and (114) belongs to class I. Class III turns out to have cost 3, like class I. So it appears that representation (114) is as good as any, with  $z$  computed by (116), for the 3-element multiplication problem we've been studying.

one-to-many mapping  
don't-cares  
2-cube equivalence  
don't-cares

Appearances can, however, be deceiving, because we need not map  $\{a, b, c\}$  into *unique* two-bit codes. Consider the one-to-many mapping

$$+1 \mapsto 00, \quad 0 \mapsto 01 \text{ or } 11, \quad -1 \mapsto 10, \quad (120)$$

where both 01 and 11 are allowed as representations of zero. The truth tables for  $f_l$  and  $f_r$  are now quite different from (112) and (115), because all inputs are legal but some outputs can be arbitrary:

$$f_l \leftrightarrow 0*1*****1*0***** \quad \text{and} \quad f_r \leftrightarrow 0101111101011111. \quad (121)$$

And in fact, this approach needs just two operations, instead of the three in (116):

$$z_l = x_l \oplus y_l, \quad z_r = x_r \vee y_r. \quad (122)$$

A moment's thought shows that indeed, these operations obviously yield the product  $z = x \cdot y$  when the three elements  $\{+1, 0, -1\}$  are represented as in (120).

Such nonunique mappings add 36 more possibilities to the 24 that we had before. But again, they reduce under "2-cube equivalence" to a small number of equivalence classes. First there are three classes that we call  $IV_a$ ,  $IV_b$ , and  $IV_c$ , depending on which element has an ambiguous representation:

$$\begin{array}{ccc} \text{Class } IV_a & \text{Class } IV_b & \text{Class } IV_c \\ a \mapsto 0*0*1*1*0*0*1*11100100110110001011000101110010; \\ b \mapsto 10110001011100100*0*1*1*0*0*1*1110010011011000; \\ c \mapsto 111001001101100010110001011100100*0*1*1*0*0*1*1. \end{array} \quad (123)$$

(Representation (120) belongs to class  $IV_b$ . Classes  $IV_a$  and  $IV_c$  don't work well for  $z = x \cdot y$ .) Then there are three further classes with only four mappings each:

$$\begin{array}{ccc} \text{Class } V_a & \text{Class } V_b & \text{Class } V_c \\ a \mapsto tt & t\bar{t} & t\bar{t} & tt & 10 & 11 & 00 & 01 & 01 & 00 & 11 & 10; \\ b \mapsto 01 & 00 & 11 & 10 & t\bar{t} & t\bar{t} & t\bar{t} & tt & 10 & 11 & 00 & 01; \\ c \mapsto 10 & 11 & 00 & 01 & 01 & 00 & 11 & 10 & t\bar{t} & t\bar{t} & t\bar{t} & tt. \end{array} \quad (124)$$

These classes are a bit of a nuisance, because the indeterminacy in their truth tables cannot be expressed simply in terms of don't-cares as we did in (121). For example, if we try

$$+1 \mapsto 00 \text{ or } 11, \quad 0 \mapsto 01, \quad -1 \mapsto 10, \quad (125)$$

which is the first mapping in class  $V_a$ , there are binary variables  $pqrst$  such that

$$f_l \leftrightarrow p01q000010r1s01t \quad \text{and} \quad f_r \leftrightarrow p10q111101r0s10t. \quad (126)$$

Furthermore, mappings of classes  $V_a$ ,  $V_b$ , and  $V_c$  almost never turn out to be better than the mappings of the other six classes (see exercise 138). Still, representatives of all nine classes must be examined before we can be sure that an optimal mapping has been found.

In practice we often want to perform several different operations on ternary-valued variables, not just a single operation like multiplication. For example, we might want to compute  $\max(x, y)$  as well as  $x \cdot y$ . With representation (120), the best we can do is  $z_l = x_l \wedge y_l$ ,  $z_r = (x_l \wedge y_r) \vee (x_r \wedge (y_l \vee y_r))$ ; but the “natural” mapping (111) now shines, with  $z_l = x_l \wedge y_l$ ,  $z_r = x_r \vee y_r$ . Class III turns out to have cost 4; other classes are inferior. To choose between classes II, III, and  $IV_b$  in this case, we need to know the relative frequencies of  $x \cdot y$  and  $\max(x, y)$ . And if we add  $\min(x, y)$  to the mix, classes II, III, and  $IV_b$  compute it with the respective costs 2, 5, 5; hence (111) looks better yet.

The ternary max and min operations arise also in other contexts, such as the three-valued logic developed by Jan Łukasiewicz in 1917. [See his *Selected Works*, edited by L. Borkowski (1970), 84–88, 153–178.] Consider the logical values “true,” “false,” and “maybe,” denoted respectively by 1, 0, and \*. Łukasiewicz defined the three basic operations of conjunction, disjunction, and implication on these values by specifying the tables

$$\begin{array}{ccc}
 \begin{array}{c} y \\ \hline 0 \quad * \quad 1 \\ \hline x \left\{ \begin{array}{l} 0 \\ * \\ 1 \end{array} \right. \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & * & * \\ \hline 0 & * & 1 \\ \hline \end{array} \end{array} & , & \begin{array}{c} y \\ \hline 0 \quad * \quad 1 \\ \hline x \left\{ \begin{array}{l} 0 \\ * \\ 1 \end{array} \right. \begin{array}{|c|c|c|} \hline 0 & * & 1 \\ \hline 0 & * & 1 \\ \hline * & * & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \end{array} & , & \begin{array}{c} y \\ \hline 0 \quad * \quad 1 \\ \hline x \left\{ \begin{array}{l} 0 \\ * \\ 1 \end{array} \right. \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline * & 1 & 1 \\ \hline 0 & * & 1 \\ \hline \end{array} \end{array} . \quad (127)
 \end{array}$$

$x \wedge y$                        $x \vee y$                        $x \Rightarrow y$

For these operations the methods above show that the binary representation

$$0 \mapsto 00, \quad * \mapsto 01, \quad 1 \mapsto 11 \quad (128)$$

works well, because we can compute the logical operations thus:

$$\begin{aligned}
 x_l x_r \wedge y_l y_r &= (x_l \wedge y_l)(x_r \wedge y_r), & x_l x_r \vee y_l y_r &= (x_l \vee y_l)(x_r \vee y_r), \\
 x_l x_r \Rightarrow y_l y_r &= ((x_r \oplus y_r) \wedge \neg(\bar{x}_l \wedge y_r))(x_l \wedge \bar{y}_r).
 \end{aligned} \quad (129)$$

Of course  $x$  need not be an isolated ternary value in this discussion; we often want to deal with ternary *vectors*  $x = x_1 x_2 \dots x_n$ , where each  $x_j$  is either  $a$ ,  $b$ , or  $c$ . Such ternary vectors are conveniently represented by two binary vectors

$$x_l = x_{1l} x_{2l} \dots x_{nl} \quad \text{and} \quad x_r = x_{1r} x_{2r} \dots x_{nr}, \quad (130)$$

where  $x_j \mapsto x_{jl} x_{jr}$  as above. We could also pack the ternary values into two-bit fields of a single vector,

$$x = x_{1l} x_{1r} x_{2l} x_{2r} \dots x_{nl} x_{nr}; \quad (131)$$

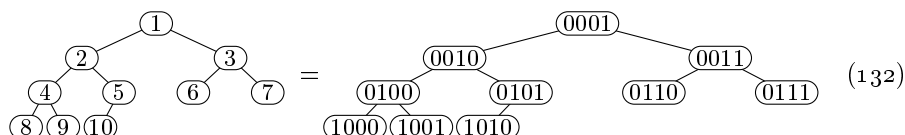
that would work fine if, say, we’re doing Łukasiewicz logic with the operations  $\wedge$  and  $\vee$  but not  $\Rightarrow$ . Usually, however, the two-vector approach of (130) is better, because it lets us do bitwise calculations without shifting and masking.

max  
min  
three-valued logic  
Łukasiewicz  
Borkowski  
modal logic  
maybe  
conjunction  
disjunction  
implication  
groupoids, multiplication tables  
ternary vectors  
pack  
masking: ANDing with a mask

**Applications to data structures.** Bitwise operations offer many efficient ways to represent elements of data and the relationships between them. For example, chess-playing programs often use a “bit board” to represent the positions of pieces (see exercise 143).

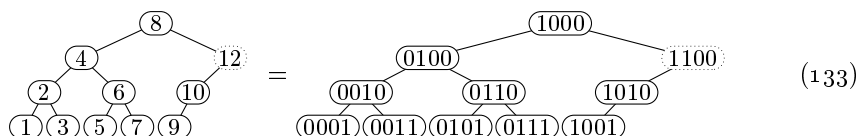
In Chapter 8 we shall discuss an important data structure developed by Peter van Emde Boas for representing a dynamically changing subset of integers between 0 and  $N$ . Insertions, deletions, and other operations such as “find the largest element less than  $x$ ” can be done in  $O(\log \log N)$  steps with his methods; the general idea is to organize the full structure recursively as  $\sqrt{N}$  substructures for subsets of intervals of size  $\sqrt{N}$ , together with an auxiliary structure that tells which of those intervals are occupied. [See *Information Processing Letters* **6** (1977), 80–82; also P. van Emde Boas, R. Kaas, and E. Zijlstra, *Math. Systems Theory* **10** (1977), 99–127.] Bitwise operations make those computations fast.

Hierarchical data can sometimes be arranged so that the links between elements are implicit rather than explicit. For example, we studied “heaps” in Section 5.2.3, where  $n$  elements of a sequential array implicitly have a binary tree structure like



when, say,  $n = 10$ . (Node numbers are shown here both in decimal and binary notation.) There is no need to store pointers in memory to relate node  $j$  of a heap to its parent (which is node  $j \gg 1$  if  $j \neq 1$ ), or to its sibling (which is node  $j \oplus 1$  if  $j \neq 1$ ), or to its children (which are nodes  $j \ll 1$  and  $(j \ll 1) + 1$  if those numbers don't exceed  $n$ ), because a simple calculation leads directly from  $j$  to any desired neighbor.

Similarly, a *sideways heap* provides implicit links for another useful family of  $n$ -node binary tree structures, typified by



when  $n = 10$ . (We sometimes need to go beyond  $n$  when moving from a node to its parent, as in the path from 10 to 12 to 8 shown here.) Heaps and sideways heaps can both be regarded as nodes 1 to  $n$  of *infinite* binary tree structures: The heap with  $n = \infty$  is rooted at node 1 and has no leaves; by contrast, the sideways heap with  $n = \infty$  has infinitely many leaves 1, 3, 5, ..., but no root(!).

The leaves of a sideways heap are the odd numbers, and their parents are the odd multiples of 2. The grandparents of leaves, similarly, are the odd multiples of 4; and so on. Thus the ruler function  $\rho j$  tells how high node  $j$  is above leaf level.

The parent of node  $j$  in the infinite sideways heap is easily seen to be node

$$(j - k) \mid (k \ll 1), \quad \text{where } k = j \& -j; \quad (134)$$

chess  
bit board  
Emde Boas  
van Emde Boas  
Kaas  
Zijlstra  
implicit data structures—  
heaps  
sibling  
sideways heap  
binary tree structures  
ruler function

this formula rounds  $j$  to the nearest odd multiple of  $2^{1+\rho j}$ . And the children are

$$j - (k \gg 1) \quad \text{and} \quad j + (k \gg 1) \quad (135)$$

when  $j$  is even. In general the descendants of node  $j$  form a closed interval

$$[j - 2^{\rho j} + 1 \dots j + 2^{\rho j} - 1], \quad (136)$$

arranged as a complete binary tree of  $2^{1+\rho j} - 1$  nodes. The ancestor of node  $j$  at height  $h$  is node

$$(j \mid (1 \ll h)) \& -(1 \ll h) = ((j \gg h) \mid 1) \ll h \quad (137)$$

when  $h \geq \rho j$ . Notice that the symmetric order of the nodes, also called inorder, is just the natural order 1, 2, 3, . . . .

Dov Harel noted these properties in his Ph.D. thesis (U. of California, Irvine, 1980), and observed that the *nearest* common ancestor of any two nodes of a sideways heap can also be easily calculated. Indeed, if node  $l$  is the nearest common ancestor of nodes  $i$  and  $j$ , where  $i \leq j$ , there is a remarkable identity

$$\rho l = \max\{\rho x \mid i \leq x \leq j\} = \lambda(j \& -i), \quad (138)$$

which relates the  $\rho$  and  $\lambda$  functions. (See exercise 146.) We can therefore use formula (137) with  $h = \lambda(j \& -i)$  to calculate  $l$ .

Subtle extensions of this approach lead to an asymptotically efficient algorithm that finds nearest common ancestors in *any* oriented forest whose arcs grow dynamically [D. Harel and R. E. Tarjan, *SICOMP* **13** (1984), 338–355]. Baruch Schieber and Uzi Vishkin [*SICOMP* **17** (1988), 1253–1262] subsequently discovered a much simpler way to compute nearest common ancestors in an arbitrary (but fixed) oriented forest, using an attractive and instructive blend of bitwise and algorithmic techniques that we shall consider next.

Recall that an oriented forest with  $m$  trees and  $n$  vertices is an acyclic digraph with  $n - m$  arcs. There is at most one arc from each vertex; the vertices with out-degree zero are the roots of the trees. We say that  $v$  is the *parent* of  $u$  when  $u \rightarrow v$ , and  $v$  is an *ancestor* of  $u$  when  $u \rightarrow^* v$ . Two vertices have a common ancestor if and only if they belong to the same tree. Vertex  $w$  is called the nearest common ancestor of  $u$  and  $v$  when we have

$$u \rightarrow^* z \text{ and } v \rightarrow^* z \quad \text{if and only if} \quad w \rightarrow^* z. \quad (139)$$

Schieber and Vishkin preprocess the given forest, mapping its vertices into a sideways heap  $S$  of size  $n$  by computing three quantities for each vertex  $v$ :

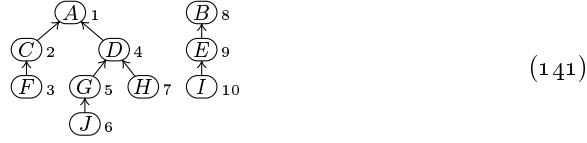
- $\pi v$ , the rank of  $v$  in preorder ( $1 \leq \pi v \leq n$ );
- $\beta v$ , a node of the sideways heap  $S$  ( $1 \leq \beta v \leq n$ );
- $\alpha v$ , a  $(1 + \lambda n)$ -bit routing code ( $1 \leq \alpha v < 2^{1+\lambda n}$ ).

If  $u \rightarrow v$  we have  $\pi u > \pi v$  by the definition of preorder. Node  $\beta v$  is defined to be the nearest common ancestor of all sideways-heap nodes  $\pi u$  such that  $v$  is an ancestor of vertex  $u$ . And we define

$$\alpha v = \sum \{2^{\rho \beta w} \mid v \rightarrow^* w\}. \quad (140)$$

rounds  
complete binary tree  
symmetric order  
inorder  
Harel  
lowest common ancestor, see Nearest common ancestor  
 $\lambda$   
Harel  
Tarjan  
Schieber  
Vishkin  
oriented forest  
acyclic digraph  
ancestor  
reachability  
transitive closure  
nearest common ancestor  
preorder++

For example, here's an oriented forest with ten vertices and two trees:



Each node has been labeled with its preorder rank, from which we can compute the  $\beta$  and  $\alpha$  codes:

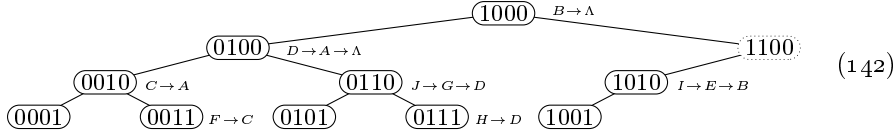
$$\begin{array}{rcccccccccc}
 v & = & A & B & C & D & E & F & G & H & I & J \\
 \pi v & = & 0001 & 1000 & 0010 & 0100 & 1001 & 0011 & 0101 & 0111 & 1010 & 0110 \\
 \beta v & = & 0100 & 1000 & 0010 & 0100 & 1010 & 0011 & 0110 & 0111 & 1010 & 0110 \\
 \alpha v & = & 0100 & 1000 & 0110 & 0100 & 1010 & 0111 & 0110 & 0101 & 1010 & 0110
 \end{array}$$

Notice that, for instance,  $\beta A = 4 = 0100$  because the preorder ranks of the descendants of  $A$  are  $\{1, 2, 3, 4, 5, 6, 7\}$ . And  $\alpha H = 0101$  because the ancestors of  $H$  have  $\beta$  codes  $\{\beta H, \beta D, \beta A\} = \{0111, 0100\}$ . One can prove without difficulty that the mapping  $v \mapsto \beta v$  satisfies the following key properties:

- i) If  $u \rightarrow v$  in the forest, then  $\beta u$  is a descendant of  $\beta v$  in  $S$ .
- ii) If several vertices have the same value of  $\beta v$ , they form a path in the forest.

Property (ii) holds because exactly one child  $u$  of  $v$  has  $\beta u = \beta v$  when  $\beta v \neq \pi v$ .

Now let's imagine placing every vertex  $v$  of the forest into node  $\beta v$  of  $S$ :



If  $k$  vertices map into node  $j$ , we can arrange them into a path

$$v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_{k-1} \rightarrow v_k, \quad \text{where } \beta v_0 = \beta v_1 = \dots = \beta v_{k-1} = j. \quad (143)$$

These paths are illustrated in (142); for example,  $J \rightarrow G \rightarrow D$  is a path in (141), and ' $J \rightarrow G \rightarrow D$ ' appears with node  $0110 = \beta J = \beta G$ .

The preprocessing algorithm also computes a table  $\tau j$  for all nodes  $j$  of  $S$ , containing pointers to the vertices  $v_k$  at the tail ends of (143):

$$\begin{array}{rcccccccccc}
 j & = & 0001 & 0010 & 0011 & 0100 & 0101 & 0110 & 0111 & 1000 & 1001 & 1010 \\
 \tau j & = & \Lambda & A & C & \Lambda & \Lambda & D & D & \Lambda & \Lambda & B
 \end{array}$$

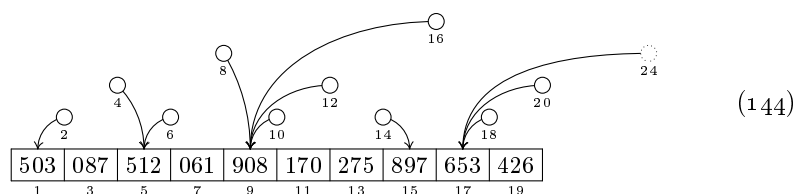
Exercise 149 shows that all four tables  $\pi v$ ,  $\beta v$ ,  $\alpha v$ , and  $\tau j$  can be prepared in  $O(n)$  steps. And once those tables are ready, they contain just enough information to identify the nearest common ancestor of any two given vertices quickly:

**Algorithm V** (*Nearest common ancestors*). Suppose  $\pi v$ ,  $\beta v$ ,  $\alpha v$ , and  $\tau j$  are known for all  $n$  vertices  $v$  of an oriented forest, and for  $1 \leq j \leq n$ . A dummy vertex  $\Lambda$  is also assumed to be present, with  $\pi \Lambda = \beta \Lambda = \alpha \Lambda = 0$ . This algorithm computes the nearest common ancestor  $z$  of any given vertices  $x$  and  $y$ , returning  $z = \Lambda$  if  $x$  and  $y$  belong to different trees. We assume that the values  $\lambda j = \lfloor \lg j \rfloor$  have been precomputed for  $1 \leq j \leq n$ , and that  $\lambda 0 = \lambda n$ .

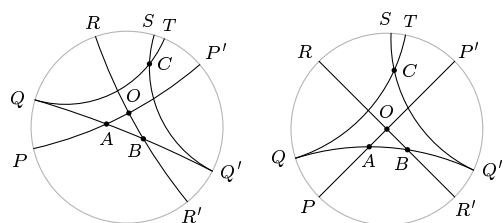
- V1.** [Find common height.] If  $\beta x \leq \beta y$ , set  $h \leftarrow \lambda(\beta y \& -\beta x)$ ; otherwise set  $h \leftarrow \lambda(\beta x \& -\beta y)$ . (See (138).)
- V2.** [Find true height.] Set  $k \leftarrow \alpha x \& \alpha y \& -(1 \ll h)$ , then  $h \leftarrow \lambda(k \& -k)$ .
- V3.** [Find  $\beta z$ .] Set  $j \leftarrow ((\beta x \gg h) \mid 1) \ll h$ . (Now  $j = \beta z$ , if  $z \neq \Lambda$ .)
- V4.** [Find  $\hat{x}$  and  $\hat{y}$ .] (We now seek the lowest ancestors of  $x$  and  $y$  in node  $j$ .)  
If  $j = \beta x$ , set  $\hat{x} \leftarrow x$ ; otherwise set  $l \leftarrow \lambda(\alpha x \& ((1 \ll h) - 1))$  and  $\hat{x} \leftarrow \tau(((\beta x \gg l) \mid 1) \ll l)$ . Similarly, if  $j = \beta y$ , set  $\hat{y} \leftarrow y$ ; otherwise set  $l \leftarrow \lambda(\alpha y \& ((1 \ll h) - 1))$  and  $\hat{y} \leftarrow \tau(((\beta y \gg l) \mid 1) \ll l)$ .
- V5.** [Find  $z$ .] Set  $z \leftarrow \hat{x}$  if  $\pi \hat{x} \leq \pi \hat{y}$ , otherwise  $z \leftarrow \hat{y}$ . ■

These artful dodges obviously exploit (137); exercise 152 explains why they work.

Sideways heaps can also be used to implement an interesting type of priority queue that J. Katajainen and F. Vitale call a “navigation pile,” illustrated here for  $n = 10$ :



Data elements go into the leaf positions  $1, 3, \dots, 2n-1$  of the sideways heap; they can be many bits wide, and they can appear in any order. By contrast, each branch position  $2, 4, 6, \dots$  contains a pointer to its largest descendant. And the novel point is that these pointers take up almost no extra space—fewer than two bits per item of data, on average—because only one bit is needed for pointers  $2, 6, 10, \dots$ , only two bits for pointers  $4, 12, 20, \dots$ , and only  $\rho j$  for pointer  $j$  in general. (See exercise 153.) Thus the navigation pile requires very little memory, and it behaves nicely with respect to cache performance on a typical computer.



**Fig. 13.** Two views of five lines in the hyperbolic plane.

**\*Cells in the hyperbolic plane.** Hyperbolic geometry suggests an instructive implicit data structure that has a rather different flavor. The *hyperbolic plane* is a fascinating example of non-Euclidean geometry that is conveniently viewed by projecting its points into the interior of a circle. Its straight lines then become circular arcs, which meet the rim at right angles. For example, the lines  $PP'$ ,  $QQ'$ , and  $RR'$  in Fig. 13 intersect at points  $O$ ,  $A$ ,  $B$ , and those points form a triangle. Lines  $SQ'$  and  $QQ'$  are *parallel*: They never touch, but their points get closer and closer together. Line  $QT$  is also parallel to  $QQ'$ .

priority queue  
Katajainen  
Vitale  
navigation pile  
cache  
hyperbolic plane—  
non-Euclidean geometry



We get different views by focusing on different center points. For example, the second view in Fig. 13 puts  $O$  smack in the center. Notice that if a line passes through the very center, it remains straight after being projected; such diameter-spanning chords are the special case of a “circular arc” whose radius is infinite.

Most of Euclid’s axioms for plane geometry remain valid in the hyperbolic plane. For example, exactly one line passes through any two distinct points; and if point  $A$  lies on line  $PP'$  there’s exactly one line  $QQ'$  such that angle  $PAQ$  has any given value  $\theta$ , for  $0 < \theta < 180^\circ$ . But Euclid’s famous fifth postulate does *not* hold: If point  $C$  is *not* on line  $QQ'$ , there always are exactly *two* lines through  $C$  that are parallel to  $QQ'$ . Furthermore there are many pairs of lines, like  $RR'$  and  $SQ'$  in Fig. 13, that are totally disjoint or *ultraparallel*, in the sense that their points never become arbitrarily close. [These properties of the hyperbolic plane were discovered by G. Saccheri in the early 1700s, and made rigorous by N. I. Lobachevsky, J. Bolyai, and C. F. Gauss a century later.]

Quantitatively speaking, when points are projected onto the unit disk  $|z| < 1$ , the arc that meets the circle at  $e^{i\theta}$  and  $e^{-i\theta}$  has center at  $\sec \theta$  and radius  $\tan \theta$ . The actual distance between two points whose projections are  $z$  and  $z'$  is  $d(z, z') = \ln(|1 - \bar{z}z'| + |z - z'|) - \ln(|1 - \bar{z}z'| - |z - z'|)$ . Thus objects far from the center appear dramatically shrunken when we see them near the circle’s rim.

The sum of the angles of a hyperbolic triangle is always *less* than  $180^\circ$ . For example, the angles at  $O$ ,  $A$ , and  $B$  in Fig. 13 are respectively  $90^\circ$ ,  $45^\circ$ , and  $36^\circ$ . Ten such  $36^\circ$ - $45^\circ$ - $90^\circ$  triangles can be placed together to make a regular pentagon with  $90^\circ$  angles at each corner. And four such pentagons fit snugly together at their corners, allowing us to tile the entire hyperbolic plane with right regular pentagons (see Fig. 14). The edges of these pentagons form an interesting family of lines, every two of which are either ultraparallel or perpendicular; so we have a grid structure analogous to the unit squares of the ordinary plane. We call it the *pentagrid*, because each cell now has five neighbors instead of four.

There’s a nice way to navigate in the pentagrid using Fibonacci numbers, based on ideas of Maurice Margenstern [see F. Herrmann and M. Margenstern, *Theoretical Comp. Sci.* **296** (2003), 345–351]. Instead of the ordinary Fibonacci sequence  $\langle F_n \rangle$ , however, we shall use the *negaFibonacci* numbers  $\langle F_{-n} \rangle$ , namely

$$F_{-1} = 1, F_{-2} = -1, F_{-3} = 2, F_{-4} = -3, \dots, F_{-n} = (-1)^{n-1} F_n. \quad (145)$$

Exercise 1.2.8–34 introduced the Fibonacci number system, in which every non-negative integer  $x$  can be written uniquely in the form

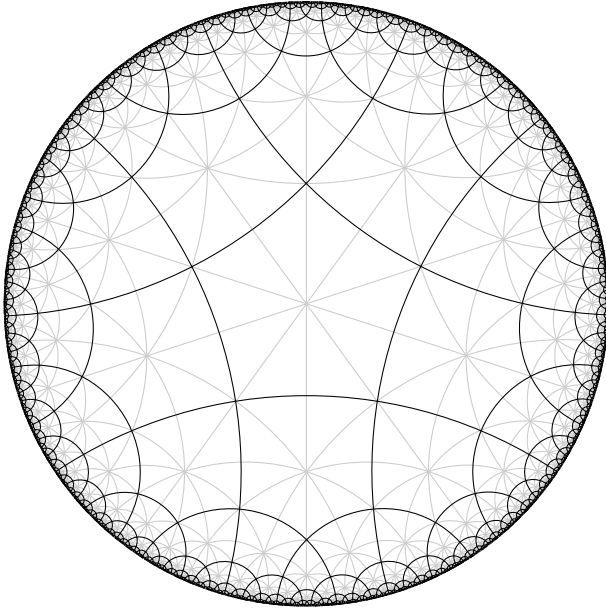
$$x = F_{k_1} + F_{k_2} + \dots + F_{k_r}, \quad \text{where } k_1 \succcurlyeq k_2 \succcurlyeq \dots \succcurlyeq k_r \succcurlyeq 0; \quad (146)$$

here ‘ $j \succcurlyeq k$ ’ means ‘ $j \geq k+2$ ’. But there’s also a *negaFibonacci number system*, which suits our purposes better: *Every integer  $x$ , whether positive, negative, or zero, can be written uniquely in the form*

$$x = F_{k_1} + F_{k_2} + \dots + F_{k_r}, \quad \text{where } k_1 \prec k_2 \prec \dots \prec k_r \prec 1. \quad (147)$$

For example,  $4 = 5 - 1 = F_{-5} + F_{-2}$  and  $-2 = -3 + 1 = F_{-4} + F_{-1}$ . This representation can conveniently be expressed as a binary code  $\alpha = \dots a_3 a_2 a_1$ ,

Euclid  
ultraparallel  
Saccheri  
Lobachevsky  
Bolyai  
Gauss  
tile  
ultraparallel  
perpendicular  
grid structure  
tessellation  
pentagrid  
Fibonacci numbers  
Margenstern  
Herrmann  
negaFibonacci  
Fibonacci number system  
negaFibonacci number system



negadecimal system  
2-adic  
magic mask

**Fig. 14.** The pentagrid,  
in which identical pentagons  
tile the hyperbolic plane.

*A circular regular tiling, confined on all sides  
by infinitely small shapes, is really wonderful.*

— M. C. ESCHER, letter to George Escher (9 November 1958)

standing for  $N(\alpha) = \sum_k a_k F_{-k}$ , with no two 1s in a row. For example, here are the negaFibonacci representation codes of all integers between  $-14$  and  $+15$ :

$-14 = 10010100$	$-8 = 100000$	$-2 = 1001$	$4 = 10010$	$10 = 1001000$
$-13 = 10010101$	$-7 = 100001$	$-1 = 10$	$5 = 10000$	$11 = 1001001$
$-12 = 101010$	$-6 = 100100$	$0 = 0$	$6 = 10001$	$12 = 1000010$
$-11 = 101000$	$-5 = 100101$	$1 = 1$	$7 = 10100$	$13 = 1000000$
$-10 = 101001$	$-4 = 1010$	$2 = 100$	$8 = 10101$	$14 = 1000001$
$-9 = 100010$	$-3 = 1000$	$3 = 101$	$9 = 1001010$	$15 = 1000100$

As in the negadecimal system (see 4.1–(6) and (7)), we can tell whether  $x$  is negative or not by seeing if its representation has an even or odd number of digits.

The predecessor  $\alpha-$  and successor  $\alpha+$  of any negaFibonacci binary code  $\alpha$  can be computed recursively by using the rules

$$\begin{aligned} (\alpha 01)- &= \alpha 00, & (\alpha 000)- &= \alpha 010, & (\alpha 100)- &= \alpha 001, & (\alpha 10)- &= (\alpha- )01, \\ & & (\alpha 10)+ &= \alpha 00, & (\alpha 00)+ &= \alpha 01, & (\alpha 1)+ &= (\alpha- )0. \end{aligned} \quad (148)$$

(See exercise 157.) But ten elegant 2-adic steps do the calculation directly:

$$\begin{aligned} y &\leftarrow x \oplus \bar{\mu}_0, \quad z \leftarrow y \oplus (y \pm 1), \quad \text{where } x = (\alpha)_2; \\ z &\leftarrow z \mid (x \& (z \ll 1)); \\ w &\leftarrow x \oplus z \oplus ((z+1) \gg 2); \quad \text{then } w = (\alpha \pm)_2. \end{aligned} \quad (149)$$

We just use  $y-1$  in the top line to get the predecessor,  $y+1$  to get the successor.

And now here's the point: A negaFibonacci code can be assigned to each cell of the pentagrid in such a way that the codes of its five neighbors are easy to compute. Let's call the neighbors  $n$ ,  $s$ ,  $e$ ,  $w$ , and  $o$ , for "north," "south," "east," "west," and "other." If  $\alpha$  is the code assigned to a given cell, we define

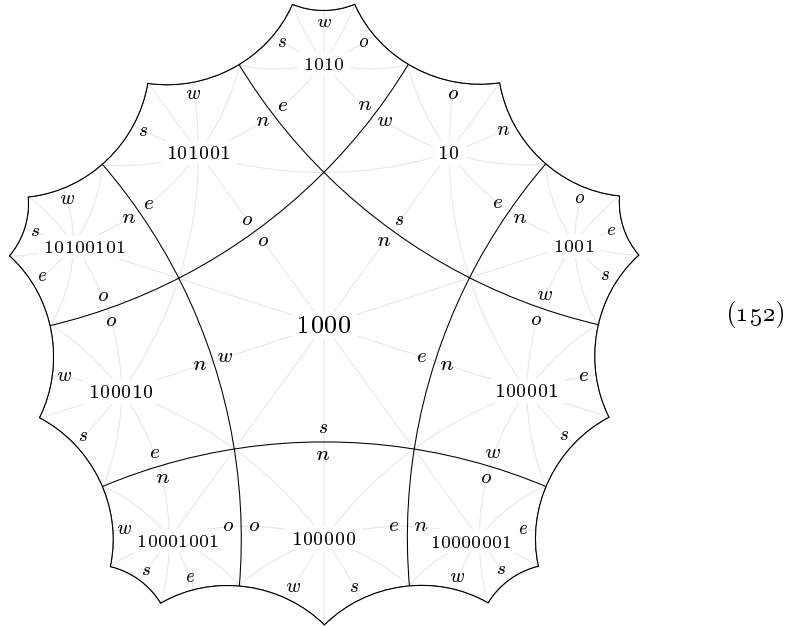
$$\alpha_n = \alpha \gg 2, \quad \alpha_s = \alpha \ll 2, \quad \alpha_e = \alpha_s +, \quad \alpha_w = \alpha_s -; \quad (150)$$

thus  $\alpha_{sn} = \alpha$ , and also  $\alpha_{en} = (\alpha 01)_n = \alpha$ . The "other" direction is trickier:

$$\alpha_o = \begin{cases} \alpha_n +, & \text{if } \alpha \& 1 = 1; \\ \alpha_w -, & \text{if } \alpha \& 1 = 0. \end{cases} \quad (151)$$

For example,  $1000_o = 101001$  and  $101001_o = 1000$ . This mysterious interloper lies between north and east when  $\alpha$  ends with 1, but between north and west when  $\alpha$  ends with 0.

If we choose any cell and label it with code 0, and if we also choose an orientation so that its neighbors are  $n$ ,  $e$ ,  $s$ ,  $w$ , and  $o$  in clockwise order, rules (150) and (151) will assign consistent labels to every cell of the pentagrid. (See exercise 160.) For example, the vicinity of a cell labeled 1000 will look like this:

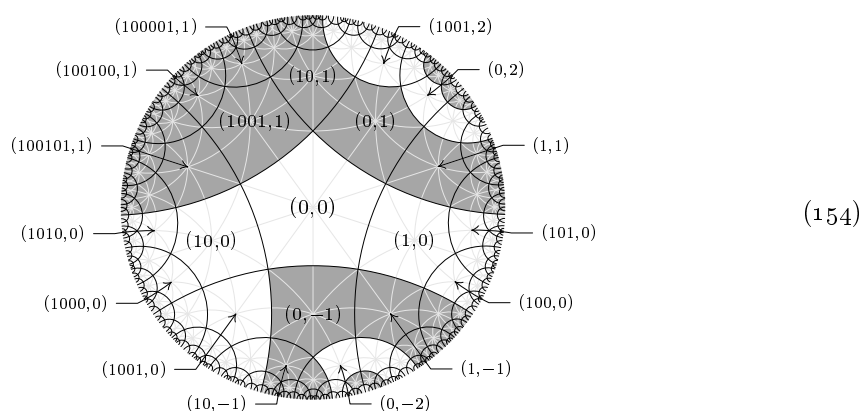


The code labels do not, however, identify cells uniquely, because infinitely many cells receive the same label. (Indeed, we clearly have  $0_n = 0_s = 0$  and  $1_w = 1_o = 1$ .) To get a unique identifier, we attach a second coordinate so that each cell's full name has the form  $(\alpha, y)$ , where  $y$  is an integer. When  $y$  is constant and  $\alpha$  ranges over all negaFibonacci codes, the cells  $(\alpha, y)$  form a more-or-less hook-shaped strip whose edges take a  $90^\circ$  turn next to cell  $(0, y)$ . In general, the five neighbors of  $(\alpha, y)$  are  $(\alpha, y)_n = (\alpha_n, y + \delta_n(\alpha))$ ,  $(\alpha, y)_s = (\alpha_s, y + \delta_s(\alpha))$ ,

$(\alpha, y)_e = (\alpha_e, y + \delta_e(\alpha))$ ,  $(\alpha, y)_w = (\alpha_w, y + \delta_w(\alpha))$ , and  $(\alpha, y)_o = (\alpha_o, y + \delta_o(\alpha))$ , where

$$\begin{aligned} \delta_n(\alpha) &= [\alpha = 0], \quad \delta_s(\alpha) = -[\alpha = 0], \quad \delta_e(\alpha) = 0, \quad \delta_w(\alpha) = -[\alpha = 1]; \\ \delta_o(\alpha) &= \begin{cases} \text{sign}(\alpha_o - \alpha_n)[\alpha_o \& \alpha_n = 0], & \text{if } \alpha \& 1 = 1; \\ \text{sign}(\alpha_o - \alpha_w)[\alpha_o \& \alpha_w = 0], & \text{if } \alpha \& 1 = 0. \end{cases} \end{aligned} \quad (153)$$

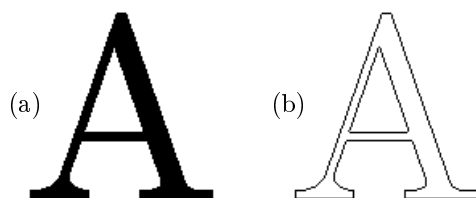
(See the illustration below.) Bitwise operations now allow us to surf the entire hyperbolic plane with ease. On the other hand, we could also ignore the  $y$  coordinates as we move, thereby wrapping around a “hyperbolic cylinder” of pentagons; the  $\alpha$  coordinates define an interesting multigraph on the set of all negaFibonacci codes, in which every vertex has degree 5.



**Bitmap graphics.** It’s fun to write programs that deal with pictures and shapes, because they involve our left and right brains simultaneously. When image data is involved, the results can be engrossing even if there are bugs in our code.

The book you are now reading was typeset by software that treated each page as a gigantic matrix of 0s and 1s, called a “raster” or “bitmap,” containing millions of square picture elements called “pixels.” The rasters were transmitted to printing machines, causing tiny dots of ink to be placed wherever a 1 appeared in the matrix. Physical properties of ink and paper caused those small clusters of dots to look like smooth curves; but each pixel’s basic squareness becomes evident if we enlarge the images tenfold, as in the letter ‘A’ shown in Fig. 15(a).

With bitwise operations we can achieve special effects like “custering,” in which the black pixels disappear when they are surrounded on all sides:



**Fig. 15.** The letter A, before and after clustering.

cylinder  
–implicit data structures  
bitmap graphics–  
typeset  
raster  
pixels  
printing  
clustering

This operation, introduced by R. A. Kirsch, L. Cahn, C. Ray, and G. H. Urban [*Proc. Eastern Joint Computer Conf.* **12** (1957), 221–229], can be expressed as

$$\text{custer}(X) = X \& \sim((X \vee\!\!\vee 1) \& (X \gg 1) \& (X \ll 1) \& (X \wedge\!\!\wedge 1)), \quad (155)$$

where ‘ $X \vee\!\!\vee 1$ ’ and ‘ $X \wedge\!\!\wedge 1$ ’ stand respectively for the result of shifting the bitmap  $X$  down or up by one row. Let us write

$$X_N = X \vee\!\!\vee 1, \quad X_W = X \gg 1, \quad X_E = X \ll 1, \quad X_S = X \wedge\!\!\wedge 1 \quad (156)$$

for the 1-pixel shifts of a bitmap  $X$ . Then, for example, the symbolic expression ‘ $X_N \& (X_S \mid \overline{X_E})$ ’ evaluates to 1 in those pixel positions whose northern neighbor is black, and which also have either a black neighbor on the south side or a white neighbor to the east. With these abbreviations, (155) takes the form

$$\text{custer}(X) = X \& \sim(X_N \& X_W \& X_E \& X_S), \quad (157)$$

which can also be expressed as  $X \& (\overline{X_N} \mid \overline{X_W} \mid \overline{X_E} \mid \overline{X_S})$ .

Every pixel has four “rook-neighbors,” with which it shares an edge at the top, left, right, or bottom. It also has eight “king-neighbors,” with which it shares at least one corner point. For example, the king-neighbors that lie to the northeast of all pixels in a bitmap  $X$  can be denoted by  $X_{NE}$ , which is equivalent to  $(X_N)_E$  in pixel algebra. Notice that we also have  $X_{NE} = (X_E)_N$ .

A  $3 \times 3$  *cellular automaton* is an array of pixels that changes dynamically via a sequence of local transformations, all performed simultaneously: The state of each pixel at time  $t + 1$  depends entirely on its state at time  $t$  and the states of its king-neighbors at that time. Thus the automaton defines a sequence of bitmaps  $X^{(0)}, X^{(1)}, X^{(2)}, \dots$  that lead from any given initial state  $X^{(0)}$ , where

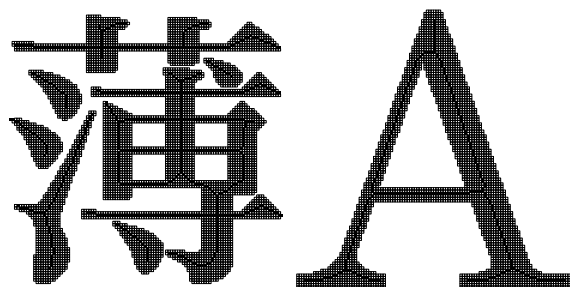
$$X^{(t+1)} = f(X_{NW}^{(t)}, X_N^{(t)}, X_{NE}^{(t)}, X_W^{(t)}, X^{(t)}, X_E^{(t)}, X_{SW}^{(t)}, X_S^{(t)}, X_{SE}^{(t)}) \quad (158)$$

and  $f$  is any bitwise Boolean function of nine variables. Fascinating patterns often emerge in this way. For example, after Martin Gardner introduced John Conway’s game of Life to the world in 1970, more computer time was probably devoted to studying its implications than to any other computational task during the next several years—although the people paying the computer bills were rarely told! (See exercise 167.)

There are  $2^{512}$  Boolean functions of nine variables, so there are  $2^{512}$  different  $3 \times 3$  cellular automata. Many of them are trivial, but most of them probably have such complicated behavior that they are humanly impossible to understand. Fortunately there also are many cases that do turn out to be useful in practice—and much easier to justify on economic grounds than the simulation of a game.


For example, algorithms for recognizing alphabetic characters, fingerprints, or similar patterns often make use of a “thinning” process, which removes excess black pixels and reduces each component of the image to an underlying skeleton that is comparatively simple to analyze. Several authors have proposed cellular automata for this problem, beginning with D. Rutovitz [*J. Royal Stat. Society* **A129** (1966), 512–513] who suggested a  $4 \times 4$  scheme. But parallel algorithms are notoriously subtle, and flaws tended to turn up after various methods had

Kirsch  
Cahn  
Ray  
Urban  
black  
white  
rook-neighbors  
king-neighbors  
8-neighbors, see king-neighbors  
4-neighbors, see rook-neighbors  
pixel algebra  
cellular automaton  
Gardner  
Conway  
Life  
game  
pattern recognition  
optical character recognition  
fingerprints  
thinning  
Rutovitz



**Fig. 16.** Example results of Guo and Hall's  $3 \times 3$  automaton for thinning the components of a bitmap. ("Hollow" pixels were originally black.)

Guo  
Hall  
connectivity structure  
kingwise connected  
rookwise connected  
Rosenfeld

been published. For example, at least two of the black pixels in a component like  should be removed, yet a symmetrical scheme will erroneously erase all four.

A satisfactory solution to the thinning problem was finally found by Z. Guo and R. W. Hall [*CACM* **32** (1989), 359–373, 759], using a  $3 \times 3$  automaton that invokes alternate rules on odd and even steps. Consider the function

$$f(x_{NW}, x_N, x_{NE}, x_W, x, x_E, x_{SW}, x_S, x_{SE}) = x \wedge \neg g(x_{NW}, \dots, x_W, x_E, \dots, x_{SE}), \quad (159)$$

where  $g = 1$  only in the following 37 configurations surrounding a black pixel:



Then we use (158), but with  $f(x_{NW}, x_N, x_{NE}, x_W, x, x_E, x_{SW}, x_S, x_{SE})$  replaced by its  $180^\circ$  rotation  $f(x_{SE}, x_S, x_{SW}, x_E, x, x_W, x_{NE}, x_N, x_{NW})$  on even-numbered steps. The process stops when two consecutive cycles make no change.

With this rule Guo and Hall proved that the  $3 \times 3$  automaton will preserve the connectivity structure of the image, in a strong sense that we will discuss below. Furthermore their algorithm obviously leaves an image intact if it is already so thin that it contains no three pixels that are king-neighbors of each other. On the other hand it usually succeeds in "removing the meat off the bones" of each black component, as shown in Fig. 16. Slightly thinner thinning is obtained in certain cases if we add four additional configurations

$$\begin{array}{cccc} \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{array} \quad (160)$$

to the 37 listed above. In either case the function  $g$  can be evaluated with a Boolean chain of length 25. (See exercises 170–172.)

In general, the black pixels of an image can be grouped into segments or components that are *kingwise connected*, in the sense that any black pixel can be reached from any other pixel of its component by a sequence of king moves through black pixels. The white pixels also form components, which are *rookwise connected*: Any two white cells of a component are mutually reachable via rook moves that touch nothing black. It's best to use different kinds of connectedness for white and black, in order to preserve the topological concepts of "inside" and "outside" that are familiar from continuous geometry [see A. Rosenfeld, *JACM* **17** (1970), 146–160]. If we imagine that the corner points of a raster are black, an infinitely thin black curve can cross between pixels at a corner, but a white curve cannot. (We could also imagine white corner points, which would lead to rookwise connectivity for black and kingwise connectivity for white.)

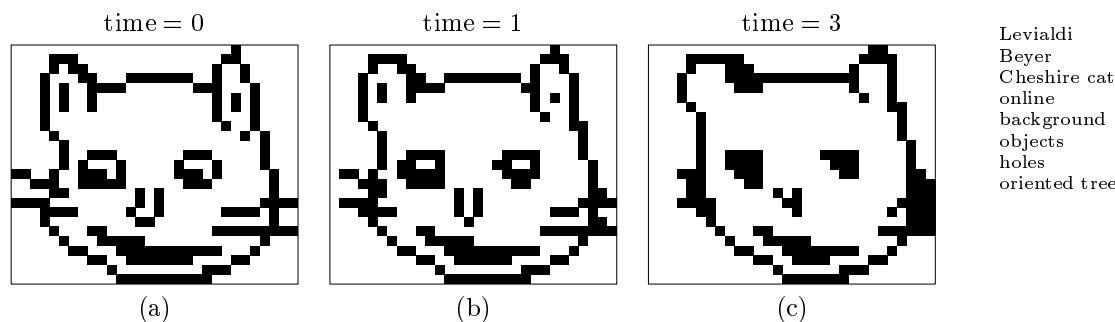


Fig. 17. The shrinking of a Cheshire cat

An amusing algorithm for shrinking a picture while preserving its connectivity, except that isolated black or white pixels disappear, was presented by S. Levialdi in *CACM* **15** (1972), 7–10; an equivalent algorithm, but with black and white reversed, had also appeared in T. Beyer’s Ph.D. thesis (M.I.T., 1969). The idea is to use a cellular automaton with the simple transition function

$$f(x_{NW}, x_N, x_{NE}, x_W, x, x_E, x_{SW}, x_S, x_{SE}) = (x \wedge (x_W \vee x_{SW} \vee x_S)) \vee (x_W \wedge x_S) \quad (161)$$

at each step. This formula is actually a  $2 \times 2$  rule, but we still need a  $3 \times 3$  window if we want to keep track of the cases when a one-pixel component goes away.

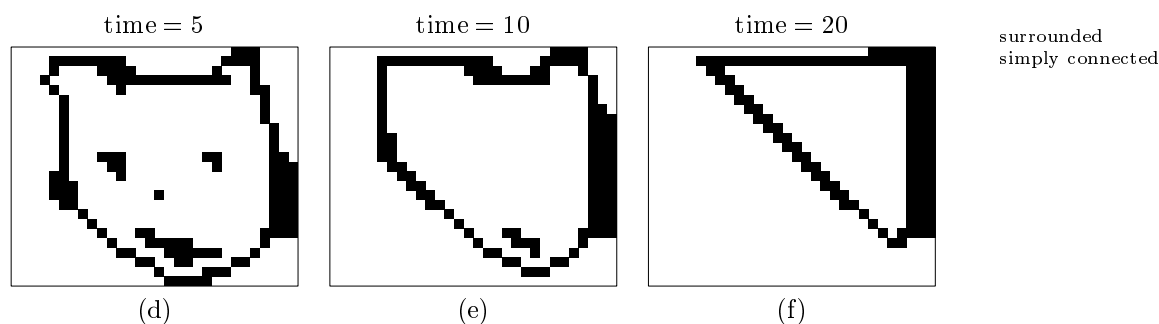
For example, the  $25 \times 30$  picture of a Cheshire cat in Fig. 17(a) has seven kingwise black components: the outline of its head, the two earholes, the two eyes, the nose, and the smile. The result after one application of (161) is shown in Fig. 17(b): Seven components remain, but there’s an isolated point in one ear, and the other earhole will become isolated after the next step. Hence Fig. 17(c) has only five components. After six steps the cat loses its nose, and even the smile will be gone at time 14. Sadly, the last bit of cat will vanish during step 46.

At most  $M + N - 1$  transitions will wipe out any  $M \times N$  picture, because the lowest visible northwest-to-southeast diagonal line moves relentlessly upward each time. Exercises 176 and 177 prove that different components will never merge together and interfere with each other.

Of course this cubic-time cellular method isn’t the fastest way to count or identify the components of a picture. We can actually do that job “online,” while looking at a large image one row at a time, not bothering to keep all of the previously seen rows in memory if we don’t wish to look at them again.

While we’re analyzing the components we might as well also record the relationships between them. Let’s assume that only finitely many black pixels are present. Then there’s an infinite component of white pixels called the *background*. Black components adjacent to the background constitute the main *objects* of the image. And these objects may in turn have *holes*, which may serve as a background for another level of objects, and so on. Thus the connected components of any finite picture form a hierarchy—an oriented tree, rooted at the background. Black components appear at the odd-numbered levels of this tree, and white components at the even-numbered levels, alternating between

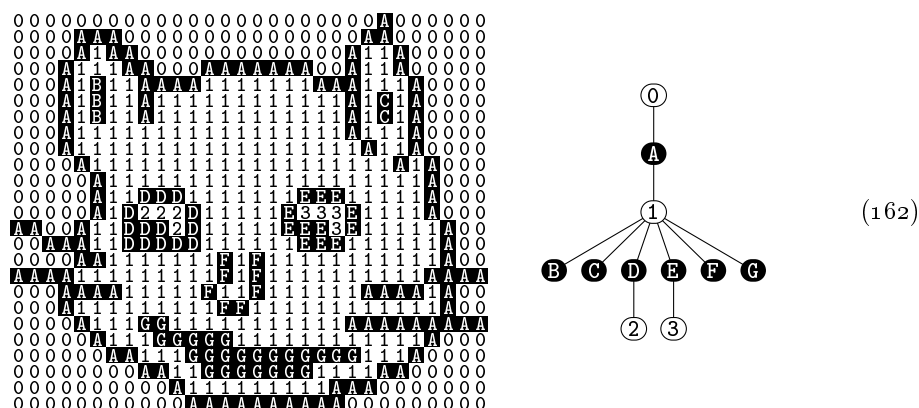
Levialdi  
Beyer  
Cheshire cat  
online  
background  
objects  
holes  
oriented tree



by repeated application of Levaldi's transformation.

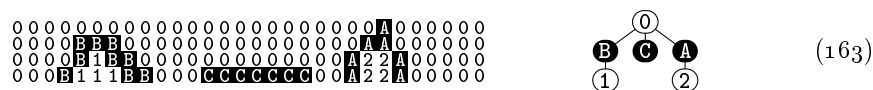
kingwise and rookwise connectedness. Each component except the background is *surrounded* by its parent. Childless components are said to be *simply connected*.

For example, here are the Cheshire cat's components, labeled with digits for white pixels and letters for the black ones, and the corresponding oriented tree:



During the shrinking process of Fig. 17, components disappear in the order **C**, {**B**, **2**, **3**} (all at time 3), **F**, **E**, **D**, **G**, **1**, **A**.

Suppose we want to analyze the components of such a picture by reading one row at a time. After we've seen four rows the result-so-far will be



and we'll be ready to scan row five. A comparison of rows four and five will then show that **B** and **C** should merge into **A**, but that new components **B** and **3** should also be launched. Exercise 179 contains full details about an instructive algorithm that properly updates the current tree as new rows are input. Additional information can also be computed on the fly: For example, we could determine the area of each component, the locations of its first and last pixels, the smallest enclosing rectangle, and/or its center of gravity.



**\*Filling.** Let's complete our quick tour of raster graphics by considering how to fill regions that are bounded by straight lines and/or simple curves. Particularly efficient algorithms are available when the curves are built up from "conic sections" — circles, ellipses, parabolas, or hyperbolas, as in classical geometry.

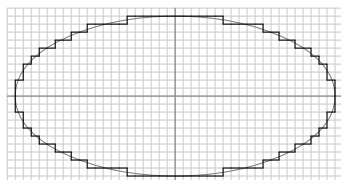
In keeping with geometric tradition, we shall adopt Cartesian coordinates  $(x, y)$  in the following discussion, instead of speaking about rows or columns of pixels: An increase of  $x$  will signify a move to the right, while an increase of  $y$  will move upward. More significantly, we will focus on the *edges* between square pixels, instead of on the pixels themselves. Edges run between integer points  $(x, y)$  and  $(x', y')$  of the plane when  $|x - x'| + |y - y'| = 1$ . Each pixel is bounded by the four edges  $(x, y) - (x-1, y) - (x-1, y-1) - (x, y-1) - (x, y)$ . Experience has shown that algorithms for filling contours become simpler and faster when we concentrate on the edge transitions between white and black, instead of on the black pixels of a custerized boundary. (See, for example, the discussion by B. D. Ackland and N. Weste in *IEEE Trans.* **C-30** (1981), 41–47.)

Consider a continuous curve  $z(t) = (x(t), y(t))$  that is traced out as  $t$  varies from 0 to 1. We assume that the curve doesn't intersect itself for  $0 \leq t < 1$ , and that  $z(0) = z(1)$ . The famous Jordan curve theorem [C. Jordan, *Cours d'analyse* **3** (1887), 587–594; O. Veblen, *Trans. Amer. Math. Soc.* **6** (1905), 83–98] states that every such curve divides the plane into two regions, called the inside and the outside. We can "digitize"  $z(t)$  by forcing it to travel along edges between pixels; then we obtain an approximation in which the inside pixels are black and the outside pixels are white. This digitization process essentially replaces the original curve by the sequence of integer points

$$\text{round}(z(t)) = (\lfloor x(t) + \tfrac{1}{2} \rfloor, \lfloor y(t) + \tfrac{1}{2} \rfloor), \quad \text{for } 0 \leq t \leq 1. \quad (164)$$

The curve can be perturbed slightly, if necessary, so that  $z(t)$  never passes exactly through the center of a pixel. Then the digitized curve takes discrete steps along pixel edges as  $t$  grows; and a pixel lies inside the digitization if and only if its center lies inside the original continuous curve  $\{z(t) \mid 0 \leq t \leq 1\}$ .

For example, the equations  $x(t) = 20 \cos 2\pi t$  and  $y(t) = 10 \sin 2\pi t$  define an ellipse. Its digitization,  $\text{round}(z(t))$ , starts at  $(20, 0)$  when  $t = 0$ , then jumps to  $(20, 1)$  when  $t \approx .008$  and  $10 \sin 2\pi t = 0.5$ . Then it proceeds to the points  $(20, 2)$ ,  $(19, 2)$ ,  $(19, 3)$ ,  $(19, 4)$ ,  $(18, 4)$ ,  $\dots$ ,  $(20, -1)$ ,  $(20, 0)$ , as  $t$  increases through the values  $.024, .036, .040, .057, .062, \dots, .976, .992$ :



(165)

The horizontal edges of such a boundary are conveniently represented by bit vectors  $H(y)$  for each  $y$ ; for example,  $H(10) = \dots 0000001111111111000000\dots$  and  $H(9) = \dots 01111100000000000011110\dots$  in (165). If the ellipse is filled

raster graphics  
conic sections  
circles  
ellipses  
parabolas  
hyperbolas  
Cartesian coordinates  
boundary curves++++  
edge transitions++++  
custerized  
Ackland  
Weste  
Jordan curve theorem  
Veblen  
inside  
outside  
digitization

with black to obtain a bitmap  $B$ , the  $H$  vectors mark transitions between black and white, so we have the symbolic relation

$$H = B \oplus (B \frown 1). \quad (166)$$

Conversely, it's easy to obtain  $B$  when the  $H$  vectors are given:

$$\begin{aligned} B(y) &= H(y_{\max}) \oplus H(y_{\max-1}) \oplus \cdots \oplus H(y+1) \\ &= H(y_{\min}) \oplus H(y_{\min+1}) \oplus \cdots \oplus H(y). \end{aligned} \quad (167)$$

Notice that  $H(y_{\min}) \oplus H(y_{\min+1}) \oplus \cdots \oplus H(y_{\max})$  is the zero vector, because each bitmap is white at both top and bottom. Notice further that the analogous *vertical* edge vectors  $V(x)$  are redundant: They satisfy the formulas  $V = B \oplus (B \ll 1)$  and  $B = V^\oplus$  (see exercise 36), but we need not bother to keep track of them.

Conic sections are easier to deal with than most other curves, because we can readily eliminate the parameter  $t$ . For example, the ellipse that led to (165) can be defined by the equation  $(x/20)^2 + (y/10)^2 = 1$ , instead of using sines and cosines. Therefore pixel  $(x, y)$  should be black if and only if its center point  $(x - \frac{1}{2}, y - \frac{1}{2})$  lies inside the ellipse, if and only if  $(x - \frac{1}{2})^2/400 + (y - \frac{1}{2})^2/100 - 1 < 0$ .

In general, every conic section is the set of points for which  $F(x, y) = 0$ , when  $F$  is an appropriate quadratic form. Therefore there's a quadratic form

$$Q(x, y) = F(x - \tfrac{1}{2}, y - \tfrac{1}{2}) = ax^2 + bxy + cy^2 + dx + ey + f \quad (168)$$

that is negative at the integer point  $(x, y)$  if and only if pixel  $(x, y)$  lies on a given side of the digitized curve.

For practical purposes we may assume that the coefficients  $(a, b, \dots, f)$  of  $Q$  are not-too-large integers. Then we're in luck, because the exact value of  $Q(x, y)$  is easy to compute. In fact, as pointed out by M. L. V. Pitteway [*Comp. J.* **10** (1967), 282–289], there's a nice “three-register algorithm” by which we can quickly track the boundary points: Let  $x$  and  $y$  be integers, and suppose we've got the values of  $Q(x, y)$ ,  $Q_x(x, y)$ , and  $Q_y(x, y)$  in three registers  $(Q, Q_x, Q_y)$ , where

$$Q_x(x, y) = 2ax + by + d \quad \text{and} \quad Q_y(x, y) = bx + 2cy + e \quad (169)$$

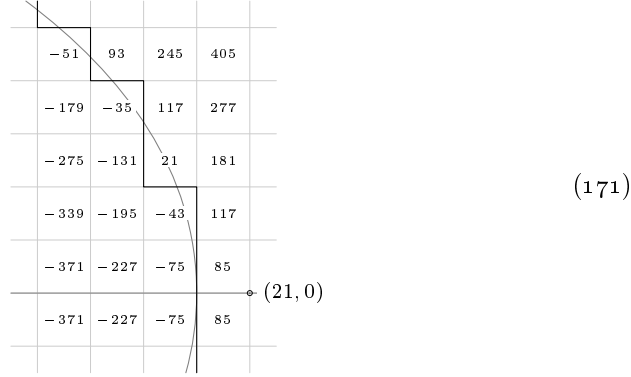
are  $\frac{\partial}{\partial x}Q$  and  $\frac{\partial}{\partial y}Q$ . We can then move to any adjacent integer point, because

$$\begin{aligned} Q(x \pm 1, y) &= Q(x, y) \pm Q_x(x, y) + a, & Q(x, y \pm 1) &= Q(x, y) \pm Q_y(x, y) + c, \\ Q_x(x \pm 1, y) &= Q_x(x, y) \pm 2a, & Q_x(x, y \pm 1) &= Q_x(x, y) \pm b, \\ Q_y(x \pm 1, y) &= Q_y(x, y) \pm b; & Q_y(x, y \pm 1) &= Q_y(x, y) \pm 2c. \end{aligned} \quad (170)$$

Furthermore we can divide the contour into separate pieces, in each of which  $x(t)$  and  $y(t)$  are both monotonic. For example, when the ellipse (165) travels from  $(20, 0)$  to  $(0, 10)$ , the value of  $x$  decreases while  $y$  increases; thus we need only move from  $(x, y)$  to  $(x-1, y)$  or to  $(x, y+1)$ . If registers  $(Q, R, S)$  respectively hold  $(Q, Q_x - a, Q_y + c)$ , a move to  $(x-1, y)$  simply sets  $Q \leftarrow Q - R$ ,  $R \leftarrow R - 2a$ , and  $S \leftarrow S - b$ ; a move to  $(x, y+1)$  is just as quick. With care, this idea leads to a blindingly fast way to discover the correctly digitized edges of almost any conic curve.

quadratic form  
Pitteway  
three-register algorithm+++

For example, the quadratic form  $Q(x, y)$  for ellipse (165) is  $4x^2 + 16y^2 - (4x + 16y + 1595)$ , when we integerize its coefficients. We have  $Q(20, 0) = F(19.5, -0.5) = -75$  and  $Q(21, 0) = +85$ ; therefore pixel  $(20, 0)$ , whose center is  $(19.5, -0.5)$ , is inside the ellipse, but pixel  $(21, 0)$  isn't. Let's zoom in closer:



The boundary can be deduced without examining  $Q$  at very many points. In fact, we don't need to look at  $Q(21, 0)$ , because we know that all edges between  $(20, 0)$  and  $(0, 10)$  must go either upwards or to the left. First we test  $Q(20, 1)$  and find it negative  $(-75)$ ; so we move up. Also  $Q(20, 2)$  is negative  $(-43)$ , so we go up again. Then we test  $Q(20, 3)$ , and find it positive  $(21)$ ; so we move left. And so on. Only the  $Q$  values  $-75, -43, 21, -131, -35, 93, -51, \dots$  actually need to be examined, if we've set the three-register method up properly.

**Algorithm T** (*Three-register algorithm for conics*). Given two integer points  $(x, y)$  and  $(x', y')$ , and an integer quadratic form  $Q$  as in (168), this algorithm decides how to digitize a portion of the conic section defined by  $F(x, y) = 0$ , where  $F(x, y) = Q(x + \frac{1}{2}, y + \frac{1}{2})$ . It creates  $|x' - x|$  horizontal edges and  $|y' - y|$  vertical edges, which form a path from  $(x, y)$  to  $(x', y')$ . We assume that

- i) Real-valued points  $(\xi, \eta)$  and  $(\xi', \eta')$  exist such that  $F(\xi, \eta) = F(\xi', \eta') = 0$ .
- ii) The curve travels from  $(\xi, \eta)$  to  $(\xi', \eta')$  monotonically in both coordinates.
- iii)  $x = \lfloor \xi + \frac{1}{2} \rfloor$ ,  $y = \lfloor \eta + \frac{1}{2} \rfloor$ ,  $x' = \lfloor \xi' + \frac{1}{2} \rfloor$ , and  $y' = \lfloor \eta' + \frac{1}{2} \rfloor$ .
- iv) If we traverse the curve from  $(\xi, \eta)$  to  $(\xi', \eta')$ , we see  $F < 0$  on our left.
- v) No edge of the integer grid contains two roots of  $Q$  (see exercise 183).

**T1.** [Initialize.] If  $x = x'$ , go to T11; if  $y = y'$ , go to T10. If  $x < x'$  and  $y < y'$ , set  $Q \leftarrow Q(x+1, y+1)$ ,  $R \leftarrow Q_x(x+1, y+1) + a$ ,  $S \leftarrow Q_y(x+1, y+1) + c$ , and go to T2. If  $x < x'$  and  $y > y'$ , set  $Q \leftarrow Q(x+1, y)$ ,  $R \leftarrow Q_x(x+1, y) + a$ ,  $S \leftarrow Q_y(x+1, y) - c$ , and go to T3. If  $x > x'$  and  $y < y'$ , set  $Q \leftarrow Q(x, y+1)$ ,  $R \leftarrow Q_x(x, y+1) - a$ ,  $S \leftarrow Q_y(x, y+1) + c$ , and go to T4. If  $x > x'$  and  $y > y'$ , set  $Q \leftarrow Q(x, y)$ ,  $R \leftarrow Q_x(x, y) - a$ ,  $S \leftarrow Q_y(x, y) - c$ , and go to T5.

**T2.** [Right or up.] If  $Q < 0$ , do T9; otherwise do T6. Repeat until interrupted.

**T3.** [Down or right.] If  $Q < 0$ , do T7; otherwise do T9. Repeat until interrupted.

- T4.** [Up or left.] If  $Q < 0$ , do T6; otherwise do T8. Repeat until interrupted.
- T5.** [Left or down.] If  $Q < 0$ , do T8; otherwise do T7. Repeat until interrupted.
- T6.** [Move up.] Create the edge  $(x, y) \text{ --- } (x, y+1)$ , then set  $y \leftarrow y+1$ . Interrupt to T10 if  $y = y'$ ; otherwise set  $Q \leftarrow Q + S$ ,  $R \leftarrow R + b$ ,  $S \leftarrow S + 2c$ .
- T7.** [Move down.] Create the edge  $(x, y) \text{ --- } (x, y-1)$ , then set  $y \leftarrow y - 1$ . Interrupt to T10 if  $y = y'$ ; otherwise set  $Q \leftarrow Q - S$ ,  $R \leftarrow R - b$ ,  $S \leftarrow S - 2c$ .
- T8.** [Move left.] Create the edge  $(x, y) \text{ --- } (x-1, y)$ , then set  $x \leftarrow x - 1$ . Interrupt to T11 if  $x = x'$ ; otherwise set  $Q \leftarrow Q - R$ ,  $R \leftarrow R - 2a$ ,  $S \leftarrow S - b$ .
- T9.** [Move right.] Create the edge  $(x, y) \text{ --- } (x+1, y)$ , then set  $x \leftarrow x + 1$ . Interrupt to T11 if  $x = x'$ ; otherwise set  $Q \leftarrow Q + R$ ,  $R \leftarrow R + 2a$ ,  $S \leftarrow S + b$ .
- T10.** [Finish horizontally.] While  $x < x'$ , create the edge  $(x, y) \text{ --- } (x+1, y)$  and set  $x \leftarrow x + 1$ . While  $x > x'$ , create the edge  $(x, y) \text{ --- } (x-1, y)$  and set  $x \leftarrow x - 1$ . Terminate the algorithm.
- T11.** [Finish vertically.] While  $y < y'$ , create the edge  $(x, y) \text{ --- } (x, y+1)$  and set  $y \leftarrow y + 1$ . While  $y > y'$ , create the edge  $(x, y) \text{ --- } (x, y-1)$  and set  $y \leftarrow y - 1$ . Terminate the algorithm. ■

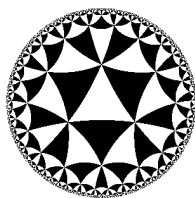
For example, when this algorithm is invoked with  $(x, y) = (20, 0)$ ,  $(x', y') = (0, 10)$ , and  $Q(x, y) = 4x^2 + 16y^2 - 4x - 16y - 1595$ , it will create the edges  $(20, 0) \text{ --- } (20, 1) \text{ --- } (20, 2) \text{ --- } (19, 2) \text{ --- } (19, 3) \text{ --- } (19, 4) \text{ --- } (18, 4) \text{ --- } (18, 5) \text{ --- } (17, 5) \text{ --- } (17, 6) \text{ --- } \dots \text{ --- } (5, 9) \text{ --- } (5, 10)$ , then make a beeline for  $(0, 10)$ . (See (165) and (171).) Exercise 182 explains why it works.

Movement to the right in step T9 is conveniently implemented by setting  $H(y) \leftarrow H(y) \oplus (1 \ll (x_{\max} - x))$ , using the  $H$  vectors of (166) and (167). Movement to the left is similar, but we set  $x \leftarrow x - 1$  first. Step T10 could set

$$H(y) \leftarrow H(y) \oplus ((1 \ll (x_{\max} + 1 - \min(x, x'))) - (1 \ll (x_{\max} - \max(x, x')))); \quad (172)$$

but one move at a time might be just as good, because  $|x' - x|$  is often small. Movement up or down needs no action, because vertical edges are redundant.

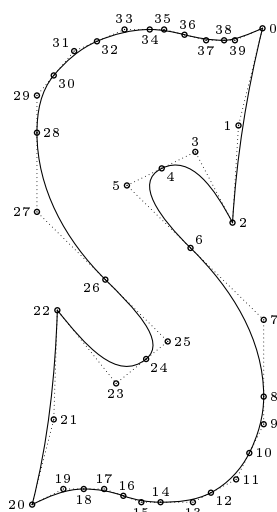
Notice that the algorithm runs somewhat faster in the special case when  $b = 0$ ; circles always belong to this case. The even more special case of straight lines, when  $a = b = c = 0$ , is of course faster yet; then we have a simple *one-register* algorithm (see exercise 185).



**Fig. 18.** Pixels change from white to black and back again, at the edges of digitized circles.

When many contours are filled in the same image, using  $H$  vectors, the pixel values change between black and white whenever we cross an odd number of edges. Figure 18 illustrates a tiling of the hyperbolic plane by equilateral  $45^\circ$ - $45^\circ$ - $45^\circ$  triangles, obtained by superimposing the results of several hundred applications of Algorithm T.

circles  
tiling  
hyperbolic plane  
eofill



**Fig. 19.** Squines that define the outline contour of an ‘S’.

Bézier splines  
squines  
control points  
S, the letter  
drawing on a bitmap  
–filling a contour in a bitmap  
Hobby  
–bitmap graphics  
conditional  
branch  
CSNZ  
pipelined machine

Algorithm T applies only to conic curves. But that’s not really a limitation in practice, because just about every shape we ever need to draw can be well approximated by “piecewise conics” called quadratic Bézier splines or *squines*. For example, Fig. 19 shows a typical squine curve with 40 points  $(z_0, z_1, \dots, z_{39}, z_{40})$ , where  $z_{40} = z_0$ . The even-numbered points  $(z_0, z_2, \dots, z_{40})$  lie on the curve; the others,  $(z_1, z_3, \dots, z_{39})$ , are called “control points,” because they regulate local bending and flexing. Each section  $S(z_{2j}, z_{2j+1}, z_{2j+2})$  begins at point  $z_{2j}$ , traveling in direction  $z_{2j+1} - z_{2j}$ . It ends at point  $z_{2j+2}$ , traveling in direction  $z_{2j+2} - z_{2j+1}$ . Thus if  $z_{2j}$  lies on the straight line from  $z_{2j-1}$  to  $z_{2j+1}$ , the squine passes smoothly through point  $z_{2j}$  without changing direction.

Exercise 186 defines  $S(z_{2j}, z_{2j+1}, z_{2j+2})$  precisely, and exercise 187 explains how to digitize any squine curve using Algorithm T. The region inside the digitized edges can then be filled with black pixels.

Incidentally, the task of *drawing* lines and curves on a bitmap turns out to be much more difficult than the task of *filling* a digitized contour, because we want diagonal strokes to have the same apparent thickness as vertical and horizontal strokes do. An excellent solution to the line-drawing problem was found by John D. Hobby, *JACM* **36** (1989), 209–229.

**\*Branchless computation.** Modern computers tend to slow down when a program contains conditional branch instructions, because an uncertain flow of control can interfere with predictive lookahead circuitry. Therefore we’ve used MMIX’s conditional-set instructions like CSNZ in programs like (56). Indeed, four instructions such as ‘ADD  $z, y, 1$ ; SR  $t, u, 2$ ; CSNZ  $x, q, z$ ; CSNZ  $v, q, t$ ’ are probably faster than their three-instruction counterpart

$$\text{BZ } q, @+12; \text{ ADD } x, y, 1; \text{ SR } v, u, 2 \quad (173)$$

when the actual running time is measured on a highly pipelined machine, even though the rule-of-thumb cost of (173) is only  $3v$  according to Table 1.3.1’–1.

Bitwise operations can help diminish the need for costly branching. For example, if MMIX didn't have a CSNZ instruction we could write

```

    NEGU m,q;  OR m,m,q;  SR m,m,63;
    ADD t,y,1;  XOR t,t,x;  AND t,t,m;  XOR x,x,t;
    SR t,u,2;  XOR t,t,v;  AND t,t,m;  XOR v,v,t;

```

(174)

here the first line creates the mask  $m = -[q \neq 0]$ . On some computers these eleven branchless instructions would still run faster than the three instructions in (173).

The inner loop of a merge sort algorithm provides an instructive example. Suppose we want to do the following operations repeatedly:

If  $x_i < y_j$ , set  $z_k \leftarrow x_i$ ,  $i \leftarrow i + 1$ , and go to  $x\_done$  if  $i = i_{\max}$ .  
 Otherwise set  $z_k \leftarrow y_j$ ,  $j \leftarrow j + 1$ , and go to  $y\_done$  if  $j = j_{\max}$ .  
 Then set  $k \leftarrow k + 1$  and go to  $z\_done$  if  $k = k_{\max}$ .

If we implement them in the “obvious” way, four conditional branches are involved, three of which are active on each path through the loop:

<pre> 1H CMP  t,xi,yj; BNN t,2F    STO  xi,zbase,kk    ADD  ii,ii,8    BZ   ii,X_Done    LDO  xi,xbase,ii    JMP  3F 2H STO  yj,zbase,kk    ADD  jj,jj,8    BZ   jj,Y_Done    LDO  yj,ybase,jj 3H ADD  kk,kk,8    PBNZ kk,1B    JMP  Z_Done </pre>	<pre> Branch if <math>x_i \geq y_j</math>. <math>z_k \leftarrow x_i</math>. <math>i \leftarrow i + 1</math>. To <math>x\_done</math> if <math>i = i_{\max}</math>. Load <math>x_i</math> into register <math>xi</math>. Join the other branch. <math>z_k \leftarrow y_j</math>. <math>j \leftarrow j + 1</math>. To <math>y\_done</math> if <math>j = j_{\max}</math>. Load <math>y_j</math> into register <math>yj</math>. <math>k \leftarrow k + 1</math>. Repeat if <math>k \neq k_{\max}</math>. To <math>z\_done</math>. ■ </pre>
--	--

(Here  $ii = 8(i - i_{\max})$ ,  $jj = 8(j - j_{\max})$ , and  $kk = 8(k - k_{\max})$ ; the factor of 8 is needed because  $x_i$ ,  $y_j$ , and  $z_k$  are octabytes.) Those four branches can be reduced to just one:

<pre> 1H CMP  t,xi,yj    CSN  yj,t,xi    STO  yj,zbase,kk    AND  t,t,8    ADD  ii,ii,t    LDO  xi,xbase,ii    XOR  t,t,8    ADD  jj,jj,t    LDO  yj,ybase,jj    ADD  kk,kk,8    AND  u,ii,jj; AND u,u,kk    PBN  u,1B </pre>	<pre> <math>t \leftarrow \text{sign}(x_i - y_j)</math>. <math>yj \leftarrow \min(x_i, y_j)</math>. <math>z_k \leftarrow yj</math>. <math>t \leftarrow 8[x_i &lt; y_j]</math>. <math>i \leftarrow i + [x_i &lt; y_j]</math>. Load <math>x_i</math> into register <math>xi</math>. <math>t \leftarrow t \oplus 8</math>. <math>j \leftarrow j + [x_i \geq y_j]</math>. Load <math>y_j</math> into register <math>yj</math>. <math>k \leftarrow k + 1</math>. <math>u \leftarrow ii \&amp; jj \&amp; kk</math>. Repeat if <math>i &lt; i_{\max}</math>, <math>j &lt; j_{\max}</math>, and <math>k &lt; k_{\max}</math>. ■ </pre>
---	---

When the loop stops in this version, we can readily decide whether to continue at  $x\_done$ ,  $y\_done$ , or  $z\_done$ . These instructions load both  $x_i$  and  $y_j$  from memory each time, but the redundant value will already be present in the cache.

mask  
signed shift right  
NEG  
merge sort  
cache

**\*More applications of MOR and MXOR.** Let's finish off our study of bitwise manipulation by taking a look at two operations that are specifically designed for 64-bit work. MMIX's instructions **MOR** and **MXOR**, which essentially carry out matrix multiplication on  $8 \times 8$  Boolean matrices, turn out to be extremely flexible and powerful, both by themselves and in combination with other bitwise operations.

If  $x = (x_7 \dots x_1 x_0)_{256}$  is an octabyte and  $a = (a_7 \dots a_1 a_0)_2$  is a single byte, the instruction **MOR t, x, a** sets  $t \leftarrow a_7 x_7 \mid \dots \mid a_1 x_1 \mid a_0 x_0$ , while **MXOR t, x, a** sets  $t \leftarrow a_7 x_7 \oplus \dots \oplus a_1 x_1 \oplus a_0 x_0$ . For example, **MOR t, x, 2** and **MXOR t, x, 2** both set  $t \leftarrow x_1$ ; **MOR t, x, 3** sets  $t \leftarrow x_1 \mid x_0$ ; and **MXOR t, x, 3** sets  $t \leftarrow x_1 \oplus x_0$ .

In general, of course, **MOR** and **MXOR** are functions of octabytes. When  $y = (y_7 \dots y_1 y_0)_{256}$  is a general octabyte, the instruction **MOR t, x, y** produces the octabyte  $t$  whose  $j$ th byte  $t_j$  is the result of **MOR** applied to  $x$  and  $y_j$ .

Suppose  $x = -1 = \# \text{ffffffffffffffff}$ . Then **MOR t, x, y** computes the mask  $t$  in which byte  $t_j$  is  $\# \text{ff}$  whenever  $y_j \neq 0$ , while  $t_j$  is zero when  $y_j = 0$ . This simple special case is quite useful, because it accomplishes in just one instruction what we previously needed seven operations to achieve in situations like (92).

We observed in (66) that two **MOR**s will suffice to reverse the bits of any 64-bit word, and many other important bit permutations also become easy when **MOR** is in a computer's repertoire. Suppose  $\pi$  is a permutation of  $\{0, 1, \dots, 7\}$  that takes  $0 \mapsto 0\pi$ ,  $1 \mapsto 1\pi$ ,  $\dots$ ,  $7 \mapsto 7\pi$ . Then the octabyte  $p = (2^{7\pi} \dots 2^{1\pi} 2^{0\pi})_{256}$  corresponds to a permutation matrix that makes **MOR** do nice tricks: **MOR t, x, p** will *permute the bytes* of  $x$ , setting  $t_j \leftarrow x_{j\pi}$ . Furthermore, **MOR u, p, y** will *permute the bits* of each byte of  $y$ , according to the *inverse* permutation; it sets  $u_j \leftarrow (a_7 \dots a_1 a_0)_2$  when  $y_j = (a_{7\pi} \dots a_{1\pi} a_{0\pi})_2$ .

With a little more skulduggery we can also expedite further permutations such as the perfect shuffle (76), which transforms a given octabyte  $z = 2^{32}x + y = (x_{31} \dots x_1 x_0 y_{31} \dots y_1 y_0)_2$  into the "zippered" octabyte

$$w = x \ddagger y = (x_{31} y_{31} \dots x_1 y_1 x_0 y_0)_2. \quad (175)$$

With appropriate permutation matrices  $p$ ,  $q$ , and  $r$ , the intermediate results

$$t = (x_{31} x_{27} x_{30} x_{26} x_{29} x_{25} x_{28} x_{24} y_{31} y_{27} y_{30} y_{26} y_{29} y_{25} y_{28} y_{24} \dots \\ x_7 x_3 x_6 x_2 x_5 x_1 x_4 x_0 y_7 y_3 y_6 y_2 y_5 y_1 y_4 y_0)_2, \quad (176)$$

$$u = (y_{27} y_{31} y_{26} y_{30} y_{25} y_{29} y_{24} y_{28} x_{27} x_{31} x_{26} x_{30} x_{25} x_{29} x_{24} x_{28} \dots \\ y_3 y_7 y_2 y_6 y_1 y_5 y_0 y_4 x_3 x_7 x_2 x_6 x_1 x_5 x_0 x_4)_2 \quad (177)$$

can be computed quickly via the four instructions

$$\text{MOR t, z, p; MOR t, q, t; MOR u, t, r; MOR u, r, u;} \quad (178)$$

see exercise 204. So there's a mask  $m$  for which 'PUT rM, m; MUX w, t, u' completes the perfect shuffle in just six cycles altogether. By contrast, the traditional method in exercise 53 requires 30 cycles (five  $\delta$ -swaps).

The analogous instruction **MXOR** is especially useful when binary linear algebra is involved. For example, exercise 1.3.1'–37 shows that **XOR** and **MXOR** directly implement addition and multiplication in a finite field of  $2^k$  elements, for  $k \leq 8$ .

MOR++  
MXOR++  
matrix multiplication  
mask  
bit permutations  
byte permutations  
permutation matrix  
*inverse* permutation  
perfect shuffle  
zippered  
MUX  
 $\delta$ -swaps  
finite field

The problem of *cyclic redundancy checking* provides an instructive example of another case where **MXOR** shines. Streams of data are often accompanied by “CRC bytes” in order to detect common types of transmission errors [see W. W. Peterson and D. T. Brown, *Proc. IRE* **49** (1961), 228–235]. One popular method, used for example in MP3 audio files, is to regard each byte  $\alpha = (a_7 \dots a_1 a_0)_2$  as if it were the polynomial

$$\alpha(x) = (a_7 \dots a_1 a_0)_x = a_7 x^7 + \dots + a_1 x + a_0. \quad (179)$$

When transmitting  $n$  bytes  $\alpha_{n-1} \dots \alpha_1 \alpha_0$ , we then compute the remainder

$$\beta = (\alpha_{n-1}(x)x^{8(n-1)} + \dots + \alpha_1(x)x^8 + \alpha_0(x))x^{16} \bmod p(x), \quad (180)$$

where  $p(x) = x^{16} + x^{15} + x^2 + 1$ , using polynomial arithmetic mod 2, and append the coefficients of  $\beta$  as a 16-bit redundancy check.

The usual way to compute  $\beta$  is to process one byte at a time, according to classical methods like Algorithm 4.6.1D. The basic idea is to define the partial result  $\beta_m = (\alpha_{n-1}(x)x^{8(n-1)} + \dots + \alpha_m(x)x^{8m})x^{16} \bmod p(x)$  so that  $\beta_n = 0$ , and then to use the recursion

$$\beta_m = ((\beta_{m+1} \ll 8) \& \#ff00) \oplus \text{crc\_table}[(\beta_{m+1} \gg 8) \oplus \alpha_m] \quad (181)$$

to decrease  $m$  by 1 until  $m = 0$ . Here  $\text{crc\_table}[\alpha]$  is a 16-bit table entry that holds the remainder of  $\alpha(x)x^{16}$ , modulo  $p(x)$  and mod 2, for  $0 \leq \alpha < 256$ . [See A. Perez, *IEEE Micro* **3**, 3 (June 1983), 40–50.]

But of course we’d prefer to process 64 bits at once instead of 8. The solution is to find  $8 \times 8$  matrices  $A$  and  $B$  such that

$$\alpha(x)x^{64} \equiv (\alpha A)(x) + (\alpha B)(x)x^{-8} \quad (\text{modulo } p(x) \text{ and } 2), \quad (182)$$

for arbitrary bytes  $\alpha$ , considering  $\alpha$  to be a  $1 \times 8$  vector of bits. Then we can pad the given data bytes  $\alpha_{n-1} \dots \alpha_1 \alpha_0$  with leading zeros so that  $n$  is a multiple of 8, and use the following efficient reduction method:

$$\begin{aligned} &\text{Begin with } c \leftarrow 0, n \leftarrow n - 8, \text{ and } t \leftarrow (\alpha_{n+7} \dots \alpha_n)_{256}. \\ &\text{While } n > 0, \text{ set } u \leftarrow t \cdot A, v \leftarrow t \cdot B, n \leftarrow n - 8, \\ &\quad t \leftarrow (\alpha_{n+7} \dots \alpha_n)_{256} \oplus u \oplus (v \gg 8) \oplus (c \ll 56), \text{ and } c \leftarrow v \& \#ff. \end{aligned} \quad (183)$$

Here  $t \cdot A$  and  $t \cdot B$  denote matrix multiplication via **MXOR**. The desired CRC bytes,  $(tx^{16} + cx^8) \bmod p(x)$ , are then readily obtained from the 64-bit quantity  $t$  and the 8-bit quantity  $c$ . Exercise 213 contains full details; the total running time for  $n$  bytes comes to only  $(\mu + 10v)n/8 + O(1)$ .

The exercises below contain many more instances where **MOR** and **MXOR** lead to substantial economies. New tricks undoubtedly remain to be discovered.

**For further reading.** The book *Hacker’s Delight* by Henry S. Warren, Jr. (Addison–Wesley, 2002) discusses bitwise operations in depth, emphasizing the great variety of options that are available on real-world computers that are not as ideal as **MMIX**.

cyclic redundancy checking  
CRC  
Peterson  
Brown  
MP3 (MPEG-1 Audio Layer III)  
Perez  
Warren



## EXERCISES

- 1. [15] What is the net effect of setting  $x \leftarrow x \oplus y$ ,  $y \leftarrow y \oplus (x \& m)$ ,  $x \leftarrow x \oplus y$ ?
2. [16] (H. S. Warren, Jr.) Are any of the following relations valid for all integers  $x$  and  $y$ ? (i)  $x \oplus y \leq x \mid y$ ; (ii)  $x \& y \leq x \mid y$ ; (iii)  $|x - y| \leq x \oplus y$ .
3. [M20] If  $x = (x_{n-1} \dots x_1 x_0)_2$  with  $x_{n-1} = 1$ , let  $x^M = (\bar{x}_{n-1} \dots \bar{x}_1 \bar{x}_0)_2$ . Thus we have  $0^M, 1^M, 2^M, 3^M, \dots = -1, 0, 1, 0, 3, 2, 1, 0, 7, 6, \dots$ , if we let  $0^M = -1$ . Prove that  $(x \oplus y)^M < |x - y| \leq x \oplus y$  for all  $x, y \geq 0$ .
- 4. [M16] Let  $x^C = \bar{x}$ ,  $x^N = -x$ ,  $x^S = x + 1$ , and  $x^P = x - 1$  denote the complement, the negative, the successor, and the predecessor of an infinite-precision integer  $x$ . Then we have  $x^{CC} = x^{NN} = x^{SP} = x^{PS} = x$ . What are  $x^{CN}$  and  $x^{NC}$ ?
5. [M21] Prove or disprove the following conjectured laws concerning binary shifts:
- $(x \ll j) \ll k = x \ll (j + k)$ ;
  - $(x \gg j) \& (y \ll k) = ((x \gg (j + k)) \& y) \ll k = (x \& (y \ll (j + k))) \gg j$ .
6. [M22] Find all integers  $x$  and  $y$  such that (a)  $x \gg y = y \gg x$ ; (b)  $x \ll y = y \ll x$ .
7. [M22] (R. Schroeppel, 1972.) Find a fast way to convert the binary number  $x = (\dots x_2 x_1 x_0)_2$  to its negabinary counterpart  $x = (\dots x'_2 x'_1 x'_0)_{-2}$ , and vice versa. *Hint*: Only two bitwise operations are needed!

- 8. [M22] Given a finite set  $S$  of nonnegative integers, the “minimal excludant” of  $S$  is defined to be

$$\text{mex}(S) = \min\{k \mid k \geq 0 \text{ and } k \notin S\}.$$

Let  $x \oplus S$  denote the set  $\{x \oplus y \mid y \in S\}$ , and let  $S \oplus y$  denote  $\{x \oplus y \mid x \in S\}$ . Prove that if  $x = \text{mex}(S)$  and  $y = \text{mex}(T)$  then  $x \oplus y = \text{mex}((S \oplus y) \cup (x \oplus T))$ .

9. [M26] (*Nim*.) Two people play a game with  $k$  piles of sticks, where there are  $a_j$  sticks in pile  $j$ . If  $a_1 = \dots = a_k = 0$  when it is a player's turn to move, that player loses; otherwise the player reduces one of the piles by any desired amount, throwing away the removed sticks, and it is the other player's turn. Prove that the player to move can force a victory if and only if  $a_1 \oplus \dots \oplus a_k \neq 0$ .

10. [HM40] (*Conway's field*.) Continuing exercise 8, define the operation  $x \otimes y$  of “nim multiplication” recursively by the formula

$$x \otimes y = \text{mex}\{(x \otimes j) \oplus (i \otimes y) \oplus (i \otimes j) \mid 0 \leq i < x, 0 \leq j < y\}.$$

Prove that  $\oplus$  and  $\otimes$  define a *field* over the set of all nonnegative integers. Prove also that if  $0 \leq x, y < 2^{2^n}$  then  $x \otimes y < 2^{2^n}$ , and  $2^{2^n} \otimes y = 2^{2^n} y$ . (In particular, this field contains subfields of size  $2^{2^n}$  for all  $n \geq 0$ .) Explain how to compute  $x \otimes y$  efficiently.

- 11. [M26] (H. W. Lenstra, 1978.) Find a simple way to characterize all pairs of positive integers  $(m, n)$  for which  $m \otimes n = mn$  in Conway's field.
12. [M26] Devise an algorithm for *division* in Conway's field. *Hint*: If  $x < 2^{2^{n+1}}$  then we have  $x \otimes (x \oplus (x \gg 2^n)) < 2^{2^n}$ .
13. [M32] (*Second-order nim*.) Extend the game of exercise 9 by allowing two kinds of moves: Either  $a_j$  is reduced for some  $j$ , as before; or  $a_j$  is reduced and  $a_i$  is replaced by an arbitrary nonnegative integer, for some  $i < j$ . Prove that the player to move can now force a victory if and only if the pile sizes satisfy either  $a_2 \neq a_3 \oplus \dots \oplus a_k$  or  $a_1 \neq a_3 \oplus (2 \otimes a_4) \oplus \dots \oplus ((k-2) \otimes a_k)$ . For example, when  $k = 4$  and  $(a_1, a_2, a_3, a_4) = (7, 5, 0, 5)$ , the only winning move is to  $(7, 5, 6, 3)$ .

Warren  
subtraction  
complement  
negative  
infinite-precision  
Schroeppel  
negabinary  
radix  $-2$   
minimal excludant  
mex  
Nim  
game  
Conway's field  
nim multiplication  
recursively  
field  
Lenstra  
nim division  
Nim, second-order

**14.** [M30] Suppose each node of a complete, infinite binary tree has been labeled with 0 or 1. Such a labeling is conveniently represented as a sequence  $T = (t, t_0, t_1, t_{00}, t_{01}, t_{10}, t_{11}, t_{000}, \dots)$ , with one bit  $t_\alpha$  for every binary string  $\alpha$ ; the root is labeled  $t$ , the left subtree labels are  $T_0 = (t_0, t_{00}, t_{01}, t_{000}, \dots)$ , and the right subtree labels are  $T_1 = (t_1, t_{10}, t_{11}, t_{100}, \dots)$ . Any such labeling can be used to transform a 2-adic integer  $x = (\dots x_2 x_1 x_0)_2$  into the 2-adic integer  $y = (\dots y_2 y_1 y_0)_2 = T(x)$  by setting  $y_0 = t$ ,  $y_1 = t_{x_0}$ ,  $y_2 = t_{x_0 x_1}$ , etc., so that  $T(x) = 2T_{x_0}(\lfloor x/2 \rfloor) + t$ . (In other words,  $x$  defines an infinite path in the binary tree, and  $y$  corresponds to the labels on that path, from right to left in the bit strings as we proceed from top to bottom of the tree.)

A *branching function* is the mapping  $x^T = x \oplus T(x)$  defined by such a labeling. For example, if  $t_{01} = 1$  and all of the other  $t_\alpha$  are 0, we have  $x^T = x \oplus 4[x \bmod 4 = 2]$ .

- Prove that every branching function is a permutation of the 2-adic integers.
- For which integers  $k$  is  $x \oplus (x \ll k)$  a branching function?
- Let  $x \mapsto x^T$  be a mapping from 2-adic integers into 2-adic integers. Prove that  $x^T$  is a branching function if and only if  $\rho(x \oplus y) = \rho(x^T \oplus y^T)$  for all 2-adic  $x$  and  $y$ .
- Prove that compositions and inverses of branching functions are branching functions. (Thus the set  $\mathcal{B}$  of all branching functions is a permutation group.)
- A branching function is *balanced* if the labels satisfy  $t_\alpha = t_{\alpha 0} \oplus t_{\alpha 1}$  for all  $\alpha$ . Show that the set of all balanced branching functions is a subgroup of  $\mathcal{B}$ .

- **15.** [M26] J. H. Quick noticed that  $((x+2) \oplus 3) - 2 = ((x-2) \oplus 3) + 2$  for all  $x$ . Find all constants  $a$  and  $b$  such that  $((x+a) \oplus b) - a = ((x-a) \oplus b) + a$  is an identity.

**16.** [M31] A function of  $x$  is called *animating* if it can be written in the form

$$((\dots(((x + a_1) \oplus b_1) + a_2) \oplus b_2) + \dots) + a_m) \oplus b_m$$

for some integer constants  $a_1, b_1, a_2, b_2, \dots, a_m, b_m$ , with  $m > 0$ .

- Prove that every animating function is a branching function (see exercise 14).
- Furthermore, prove that it is balanced if and only if  $b_1 \oplus b_2 \oplus \dots \oplus b_m = 0$ . *Hint:* What binary tree labeling corresponds to the animating function  $((x \oplus c) - 1) \oplus c$ ?
- Let  $\lfloor x \rfloor = x \oplus (x - 1) = 2^{\rho(x)+1} - 1$ . Show that every balanced animating function can be written in the form

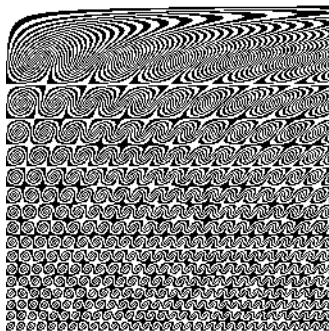
$$x \oplus \lfloor x \oplus p_1 \rfloor \oplus \lfloor x \oplus p_2 \rfloor \oplus \dots \oplus \lfloor x \oplus p_l \rfloor, \quad p_1 < p_2 < \dots < p_l,$$

for some integers  $\{p_1, p_2, \dots, p_l\}$ , where  $l \geq 0$ , and this representation is unique.

- Conversely, show that every such expression defines a balanced animating function.

**17.** [HM36] The results of exercise 16 make it possible to decide whether or not any two given animating functions are equal. Is there an algorithm that decides whether *any* given expression is identically zero, when that expression is constructed from a finite number of integer variables and constants using only the binary operations  $+$  and  $\oplus$ ? What if we also allow  $\&$ ?

**18.** [M25] The curious pixel pattern shown here has  $(x^2 y \gg 11) \& 1$  in row  $x$  and column  $y$ , for  $1 \leq x, y \leq 256$ . Is there any simple way to explain some of its major characteristics mathematically?



complete, infinite binary tree  
2-adic integer  
branching function  
permutation  
ruler function rho  
group  
composition of permutations  
balanced  
Quick  
XOR identities  
animating  
pixel pattern

- 19. [M37] (*Paley's rearrangement theorem.*) Given three vectors  $A = (a_0, \dots, a_{2^n-1})$ ,  $B = (b_0, \dots, b_{2^n-1})$ , and  $C = (c_0, \dots, c_{2^n-1})$  of nonnegative numbers, let

$$f(A, B, C) = \sum_{j \oplus k \oplus l = 0} a_j b_k c_l.$$

For example, if  $n = 2$  we have  $f(A, B, C) = a_0 b_0 c_0 + a_0 b_1 c_1 + a_0 b_2 c_2 + a_0 b_3 c_3 + a_1 b_0 c_1 + a_1 b_1 c_3 + a_1 b_2 c_0 + a_1 b_3 c_2 + a_2 b_0 c_2 + a_2 b_1 c_0 + a_2 b_2 c_3 + a_2 b_3 c_1 + a_3 b_0 c_3 + a_3 b_1 c_2 + a_3 b_2 c_1 + a_3 b_3 c_0$ ; in general there are  $2^{2n}$  terms, one for each choice of  $j$  and  $k$ . Our goal is to prove that  $f(A, B, C) \leq f(A^*, B^*, C^*)$ , where  $A^*$  denotes the vector  $A$  sorted into nonincreasing order:  $a_0^* \geq a_1^* \geq \dots \geq a_{2^n-1}^*$ .

- a) Prove the result when all elements of  $A$ ,  $B$ , and  $C$  are 0s and 1s.  
 b) Show that it is therefore true in general.  
 c) Similarly,  $f(A, B, C, D) = \sum_{j \oplus k \oplus l \oplus m = 0} a_j b_k c_l d_m \leq f(A^*, B^*, C^*, D^*)$ .
- 20. [21] (*Gosper's hack.*) The following seven operations produce a useful function  $y$  of  $x$ , when  $x$  is a positive integer. Explain what this function is and why it is useful.

$$u \leftarrow x \& -x; \quad v \leftarrow x + u; \quad y \leftarrow v + (((v \oplus x)/u) \gg 2).$$

21. [22] Construct the *reverse* of Gosper's hack: Show how to compute  $x$  from  $y$ .
22. [21] Implement Gosper's hack efficiently with MMIX code, assuming that  $x < 2^{64}$ , without using division.
- 23. [27] A sequence of nested parentheses can be represented as a binary number by putting a 1 in the position of each right parenthesis. For example,  $'(())()'$  corresponds in this way to  $(001101)_2$ , the number 13. Call such a number a *parenthesis trace*.
- a) What are the smallest and largest parenthesis traces that have exactly  $m$  1s?  
 b) Suppose  $x$  is a parenthesis trace and  $y$  is the next larger parenthesis trace with the same number of 1s. Show that  $y$  can be computed from  $x$  with a short chain of operations analogous to Gosper's hack.  
 c) Implement your method on MMIX, assuming that  $\nu x \leq 32$ .
- 24. [M30] Program 1.3.2'P instructed MMIX to produce a table of the first five hundred prime numbers, using trial division to establish primality. Write an MMIX program that uses the "sieve of Eratosthenes" (exercise 4.5.4–8) to build a table of all odd primes that are less than  $N$ , packed into octabytes  $Q_0, Q_1, \dots, Q_{N/128-1}$  as in (27). Assume that  $N \leq 2^{32}$ , and that it's a multiple of 128. What is the running time when  $N = 3584$ ?
- 25. [15] Four volumes sit side by side on a bookshelf. Each of them contains exactly 500 pages, printed on 250 sheets of paper 0.1 mm thick; each book also has a front and back cover whose thicknesses are 1 mm each. A bookworm gnaws its way from page 1 of Volume 1 to page 500 of Volume 4. How far does it travel while doing so?
26. [22] Suppose we want random access to a table of 12 million items of 5-bit data. We could pack 12 such items into one 64-bit word, thereby fitting the table into 8 megabytes of memory. But random access then seems to require division by 12, which is rather slow; we might therefore prefer to let each item occupy a full byte, thus using 12 megabytes altogether.
- Show, however, that there's a memory-efficient approach that avoids division.
27. [21] In the notation of Eqs. (32)–(43), how would you compute (a)  $(\alpha 10^a 01^b)_2$ ? (b)  $(\alpha 10^a 11^b)_2$ ? (c)  $(\alpha 00^a 01^b)_2$ ? (d)  $(0^\infty 11^a 00^b)_2$ ? (e)  $(0^\infty 01^a 00^b)_2$ ? (f)  $(0^\infty 11^a 11^b)_2$ ?
28. [16] What does the operation  $(x+1) \& \bar{x}$  produce?
29. [20] (V. R. Pratt.) Express the magic mask  $\mu_k$  of (47) in terms of  $\mu_{k+1}$ .

Paley  
 sorted  
 zero-one principle  
 0–1 principle  
 Gosper's hack  
 nested parentheses  
 parenthesis trace  
 Gosper's hack  
 MMIX  
 prime numbers  
 sieve  
 Eratosthenes  
 bookworm  
 pack  
 allocation of memory  
 storage allocation  
 division, avoiding  
 Pratt  
 magic mask

30. [20] If  $x = 0$ , the MMIX instructions (46) will set  $\rho \leftarrow 64$  (which is a close enough approximation to  $\infty$ ). What changes to (50) and (51) will produce the same result?

- 31. [20] A mathematician named Dr. L. I. Presume decided to calculate the ruler function with a simple loop as follows: “Set  $\rho \leftarrow 0$ ; then while  $x \& 1 = 0$ , set  $\rho \leftarrow \rho + 1$  and  $x \leftarrow x \gg 1$ .” He reasoned that, when  $x$  is a random integer, the average number of right shifts is the average value of  $\rho$ , which is 1; and the standard deviation is only  $\sqrt{2}$ , so the loop almost always terminates quickly. Criticize his decision.

32. [20] What is the execution time for  $\rho x$  when (52) is programmed for MMIX?

- 33. [26] (Leiserson, Prokop, and Randall, 1998.) Show that if ‘58’ is replaced by ‘49’ in (52), we can use that method to identify *both* bits of the number  $y = 2^j + 2^k$  quickly, when  $64 > j > k \geq 0$ . (Altogether  $\binom{64}{2} = 2016$  cases need to be distinguished.)

34. [M23] Let  $x$  and  $y$  be 2-adic integers. True or false: (a)  $\rho(x \& y) = \max(\rho x, \rho y)$ ; (b)  $\rho(x \mid y) = \min(\rho x, \rho y)$ ; (c)  $\rho x = \rho y$  if and only if  $x \oplus y = (x - 1) \oplus (y - 1)$ .

- 35. [M26] According to Reitwiesner’s theorem, exercise 4.1–34, every integer  $n$  has a unique representation  $n = n^+ - n^-$  such that  $\nu(n^+) + \nu(n^-)$  is minimized. Show that  $n^+$  and  $n^-$  can be calculated quickly with bitwise operations. *Hint*: Prove the identity  $(x \oplus 3x) \& ((x \oplus 3x) \gg 1) = 0$ .

36. [20] Given  $x = (x_{63} \dots x_1 x_0)_2$ , suggest efficient ways to calculate the quantities

- i)  $x^\oplus = (x_{63}^\oplus \dots x_1^\oplus x_0^\oplus)_2$ , where  $x_k^\oplus = x_k \oplus \dots \oplus x_1 \oplus x_0$  for  $0 \leq k < 64$ ;
- ii)  $x^\& = (x_{63}^\& \dots x_1^\& x_0^\&)_2$ , where  $x_k^\& = x_k \& \dots \& x_1 \& x_0$  for  $0 \leq k < 64$ .

37. [16] What changes to (55) and (56) will make  $\lambda 0$  come out  $-1$ ?

38. [17] How long does the leftmost-bit-extraction procedure (57) take when implemented on MMIX?

- 39. [20] Formula (43) shows how to remove the rightmost run of 1 bits from a given number  $x$ . How would you remove the *leftmost* run of 1 bits?

- 40. [21] Prove (58), and find a simple way to decide if  $\lambda x < \lambda y$ , given  $x$  and  $y \geq 0$ .

41. [M22] What are the generating functions of the integer sequences (a)  $\rho n$ , (b)  $\lambda n$ , and (c)  $\nu n$ ?

42. [M21] If  $n = 2^{e_1} + \dots + 2^{e_r}$ , with  $e_1 > \dots > e_r \geq 0$ , express the sum  $\sum_{k=0}^{n-1} \nu k$  in terms of the exponents  $e_1, \dots, e_r$ .

- 43. [20] How sparse should  $x$  be, to make (63) faster than (62) on MMIX?

- 44. [23] (E. Freed, 1983.) What’s a fast way to evaluate the *weighted* bit sum  $\sum j x_j$ ?

- 45. [20] (T. Rokicki, 1999.) Explain how to test if  $x^R < y^R$ , without reversing  $x$  and  $y$ .

46. [22] Method (68) uses six operations to interchange two bits  $x_i \leftrightarrow x_j$  of a register. Show that this interchange can actually be done with only *three* MMIX instructions.

47. [10] Can the general  $\delta$ -swap (69) also be done with a method like (67)?

48. [M21] How many different  $\delta$ -swaps are possible in an  $n$ -bit register? (When  $n = 4$ , a  $\delta$ -swap can transform 1234 into 1234, 1243, 1324, 1432, 2134, 2143, 3214, 3412, 4231.)

- 49. [M30] Let  $s(n)$  denote the fewest  $\delta$ -swaps that suffice to reverse an  $n$ -bit number.

- a) Prove that  $s(n) \geq \lceil \log_3 n \rceil$  when  $n$  is odd,  $s(n) \geq \lceil \log_3 3n/2 \rceil$  when  $n$  is even.
- b) Evaluate  $s(n)$  when  $n = 3^m$ ,  $2 \cdot 3^m$ ,  $(3^m + 1)/2$ , and  $(3^m - 1)/2$ .
- c) What are  $s(32)$  and  $s(64)$ ? *Hint*: Show that  $s(5n + 2) \leq s(n) + 2$ .

50. [M37] Continuing exercise 49, prove that  $s(n) = \log_3 n + O(\log \log n)$ .

$\infty$   
Presume  
ruler function  
analysis of algorithms  
Leiserson  
Prokop  
Randall  
2-adic integers  
XOR identities  
ruler function  
Reitwiesner  
notation  $x^\oplus$   
suffix parity  
prefix problem, see suffix parity  
leftmost  
run of 1 bits  
[lg x]  
generating functions  
 $\rho n$   
 $\lambda n$   
 $\nu n$   
binary recurrence  
nu(k) summed  
Freed  
weighted  
sum of bits, weighted  
Rokicki  
reversing  
interchange two bits

**51.** [23] Let  $c$  be a constant,  $0 \leq c < 2^d$ . Find all sequences of masks  $(\theta_0, \theta_1, \dots, \theta_{d-1}, \hat{\theta}_{d-2}, \dots, \hat{\theta}_1, \hat{\theta}_0)$  such that the general permutation scheme (71) takes  $x \mapsto x^\pi$ , where the bit permutation  $\pi$  is defined by either (a)  $j\pi = j \oplus c$ ; or (b)  $j\pi = (j + c) \bmod 2^d$ . [The masks should satisfy  $\theta_k \subseteq \mu_{d,k}$  and  $\hat{\theta}_k \subseteq \mu_{d,k}$ , so that (71) corresponds to Fig. 12; see (48). Notice that reversal,  $x^\pi = x^R$ , is the special case  $c = 2^d - 1$  of part (a), while part (b) corresponds to the cyclic right shift  $x^\pi = (x \gg c) + (x \ll (2^d - c))$ .]

**52.** [22] Find hexadecimal constants  $(\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \hat{\theta}_4, \hat{\theta}_3, \hat{\theta}_2, \hat{\theta}_1, \hat{\theta}_0)$  that cause (71) to produce the following important 64-bit permutations, based on the binary representation  $j = (j_5 j_4 j_3 j_2 j_1 j_0)_2$ : (a)  $j\pi = (j_0 j_5 j_4 j_3 j_2 j_1)_2$ ; (b)  $j\pi = (j_2 j_1 j_0 j_5 j_4 j_3)_2$ ; (c)  $j\pi = (j_1 j_0 j_5 j_4 j_3 j_2)_2$ ; (d)  $j\pi = (j_0 j_1 j_2 j_3 j_4 j_5)_2$ . [Case (a) is called a “perfect shuffle” because it takes  $(x_{63} \dots x_{33} x_{32} x_{31} \dots x_1 x_0)_2$  into  $(x_{63} x_{31} \dots x_{33} x_1 x_{32} x_0)_2$ ; case (b) transposes an  $8 \times 8$  matrix of bits; case (c), similarly, transposes a  $4 \times 16$  matrix; and case (d) arises in connection with “fast Fourier transforms,” see exercise 4.6.4–14.]

► **53.** [M25] The permutations in exercise 52 are said to be “induced by a permutation of index digits,” because we obtain  $j\pi$  by permuting the binary digits of  $j$ . Suppose  $j\pi = (j_{(d-1)\psi} \dots j_{1\psi} j_{0\psi})_2$ , where  $\psi$  is a permutation of  $\{0, 1, \dots, d-1\}$ . Prove that if  $\psi$  has  $t$  cycles, the  $2^d$ -bit permutation  $x \mapsto x^\pi$  can be obtained with only  $d - t$  swaps. In particular, show that this observation speeds up all four cases of exercise 52.

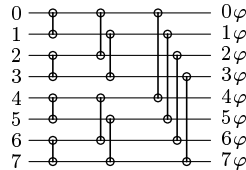
**54.** [22] (R. W. Gosper, 1985.) If an  $m \times m$  bit matrix is stored in the rightmost  $m^2$  bits of a register, show that it can be transposed by doing  $(2^k(m-1))$ -swaps for  $0 \leq k < \lceil \lg m \rceil$ . Write out the method in detail when  $m = 7$ .

► **55.** [26] Suppose an  $n \times n$  bit matrix is stored in the rightmost  $n^2$  bits of an  $n^3$ -bit register. Prove that  $18d + 2$  bitwise operations suffice to multiply two such matrices, when  $n = 2^d$ ; the matrix multiplication can be either Boolean (like MOR) or mod 2 (like MXOR).

**56.** [24] Suggest a way to transpose a  $7 \times 9$  bit matrix in a 64-bit register.

**57.** [22] The network  $P(2^d)$  of Fig. 12 has a total of  $(2d-1)2^{d-1}$  crossbars. Prove that any permutation of  $2^d$  elements can be realized by some setting in which at most  $d2^{d-1}$  of them are active.

► **58.** [M32] The first  $d$  columns of crossbar modules in the permutation network  $P(2^d)$  perform a 1-swap, then a 2-swap,  $\dots$ , and finally a  $2^{d-1}$ -swap, when the wires of the network are stretched into horizontal lines as shown here for  $d = 3$ . Let  $N = 2^d$ . These  $N$  lines, together with the  $Nd/2$  crossbars, form a so-called “Omega router.” The purpose of this exercise is to study the set  $\Omega$  of all permutations  $\varphi$  such that we can obtain  $(0\varphi, 1\varphi, \dots, (N-1)\varphi)$  as outputs on the right of an Omega router when the inputs at the left are  $(0, 1, \dots, N-1)$ .



a) Prove that  $|\Omega| = 2^{Nd/2}$ . (Thus  $\lg |\Omega| = Nd/2 \sim \frac{1}{2} \lg N!$ .)

b) Prove that a permutation  $\varphi$  of  $\{0, 1, \dots, N-1\}$  belongs to  $\Omega$  if and only if

$$i \bmod 2^k = j \bmod 2^k \quad \text{and} \quad i\varphi \gg k = j\varphi \gg k \quad \text{implies} \quad i\varphi = j\varphi \quad (*)$$

for all  $0 \leq i, j < N$  and all  $0 \leq k \leq d$ .

c) Simplify condition (\*) to the following, for all  $0 \leq i, j < N$ :

$$\lambda(i\varphi \oplus j\varphi) < \rho(i \oplus j) \quad \text{implies} \quad i = j.$$

d) Let  $T$  be the set of all permutations  $\tau$  of  $\{0, 1, \dots, N-1\}$  such that  $\rho(i \oplus j) = \rho(i\tau \oplus j\tau)$  for all  $i$  and  $j$ . (This is the set of branching functions considered in exer-

reversal  
cyclic right shift  
perfect shuffle  
outshuffle  
transposes  
fast Fourier transforms  
permutation of index digits  
Gosper  
transposed  
matrix multiplication  
Boolean matrix multiplication  
MOR  
MXOR  
swap  
Omega network for routing  
butterfly network  
shuffle network for routing  
branching functions

cise 14, modulo  $2^d$ ; so it has  $2^{N-1}$  members,  $2^{N/2+d-1}$  of which are the animating functions modulo  $2^d$ .) Prove that  $\varphi \in \Omega$  if and only if  $\tau\varphi \in \Omega$  for all  $\tau \in T$ .

- e) Suppose  $\varphi$  and  $\psi$  are permutations of  $\Omega$  that operate on different elements; that is,  $j\varphi \neq j$  implies  $j\psi = j$ , for  $0 \leq j < N$ . Prove that  $\varphi\psi \in \Omega$ .

**59.** [M30] Given  $0 \leq a < b < N = 2^d$ , how many Omega-routable permutations operate only on the interval  $[a..b]$ ? (Thus we want to count the number of  $\varphi \in \Omega$  such that  $j\varphi \neq j$  implies  $a \leq j \leq b$ . Exercise 58(a) is the special case  $a = 0$ ,  $b = N - 1$ .)

**60.** [HM28] Given a random permutation of  $\{0, 1, \dots, 2n-1\}$ , let  $p_{nk}$  be the probability that there are  $2^k$  ways to set the crossbars in the first and last columns of the permutation network  $P(2n)$  when realizing this permutation. In other words,  $p_{nk}$  is the probability that the associated graph has  $k$  cycles (see (75)). What is the generating function  $\sum_{k \geq 0} p_{nk} z^k$ ? What are the mean and variance of  $2^k$ ?

**61.** [46] Is it NP-hard to decide whether a given permutation is realizable with at least one mask  $\theta_j = 0$ , using the recursive method of Fig. 12 as implemented in (71)?

- **62.** [22] Let  $N = 2^d$ . We can obviously represent a permutation  $\pi$  of  $\{0, 1, \dots, N-1\}$  by storing a table of  $N$  numbers,  $d$  bits each. With this representation we have instant access to  $y = x\pi$ , given  $x$ ; but it takes  $\Omega(N)$  steps to find  $x = y\pi^{-1}$  when  $y$  is given.

Show that, with the same amount of memory, we can represent an arbitrary permutation in such a way that  $x\pi$  and  $y\pi^{-1}$  are both computable in  $O(d)$  steps.

**63.** [19] For what integers  $w, x, y$ , and  $z$  does the zipper function satisfy (i)  $x \ddagger y = y \ddagger x$ ? (ii)  $(x \ddagger y) \gg z = (x \gg \lfloor z/2 \rfloor) \ddagger (y \gg \lfloor z/2 \rfloor)$ ? (iii)  $(w \ddagger x) \& (y \ddagger z) = (w \& y) \ddagger (x \& z)$ ?

**64.** [22] Find a “simple” expression for the zipper-of-sums  $(x + x') \ddagger (y + y')$ , as a function of  $z = x \ddagger y$  and  $z' = x' \ddagger y'$ .

**65.** [M16] The binary polynomial  $u(x) = u_0 + u_1x + \dots + u_{n-1}x^{n-1} \pmod{2}$  can be represented by the integer  $u = (u_{n-1} \dots u_1 u_0)_2$ . If  $u(x)$  and  $v(x)$  correspond to integers  $u$  and  $v$  in this way, what polynomial corresponds to  $u \ddagger v$ ?

- **66.** [M26] Suppose the polynomial  $u(x)$  has been represented as an  $n$ -bit integer  $u$  as in exercise 65, and let  $v = u \oplus (u \ll \delta) \oplus (u \ll 2\delta) \oplus (u \ll 3\delta) \oplus \dots$  for some integer  $\delta$ .

- What’s a simple way to describe the polynomial  $v(x)$ ?
- Suppose  $n$  is large, and the bits of  $u$  have been packed into 64-bit words. How would you compute  $v$  when  $\delta = 1$ , using bitwise operations in 64-bit registers?
- Consider the same question as (b), but when  $\delta = 64$ .
- Consider the same question as (b), but when  $\delta = 3$ .
- Consider the same question as (b), but when  $\delta = 67$ .

**67.** [M31] If  $u(x)$  is a polynomial of degree  $< n$ , represented as in exercise 65, discuss the computation of  $v(x) = u(x)^2 \pmod{(x^n + x^m + 1)}$ , when  $0 < m < n$  and both  $m$  and  $n$  are odd. *Hint:* This problem has an interesting connection with perfect shuffling.

**68.** [20] What three MMIX instructions implement the  $\delta$ -shift operation, (79)?

**69.** [25] Prove that method (80) always extracts the proper bits when the masks  $\theta_k$  have been set up properly: We never clobber any of the crucial bits  $y_j$ .

- **70.** [31] (Guy L. Steele Jr., 1994.) What’s a good way to compute the masks  $\theta_0, \theta_1, \dots, \theta_{d-1}$  that are needed in the general compression procedure (80), given  $\chi \neq 0$ ?

**71.** [17] Explain how to *reverse* the procedure of (80), going from the compact value  $y = (y_{r-1} \dots y_1 y_0)_2$  to a number  $z = (z_{63} \dots z_1 z_0)_2$  that has  $z_{j_i} = y_i$  for  $0 \leq i < r$ .

**72.** [10] Simplify the expression  $(x \ddagger y) \cdot \mu_0$ , when  $x, y < 2^{2^{d-1}}$ . (See Eqs. (76) and (81).)

animating functions  
permutation network  
generating function  
variance  
NP-hard  
represent an arbitrary permutation  
zipper function  
polynomial  
polynomial remainder mod 2  
trinomial  
squaring a polynomial  
perfect shuffling  
**MMIX**  
Steele  
compression  
unpacking  
uncompressing

73. [22] Prove that  $d$  sheep-and-goats steps will implement any  $2^d$ -bit permutation.

74. [22] Given counts  $(c_0, c_1, \dots, c_{2^d-1})$  for the Chung–Wong procedure, explain why an appropriate cyclic 1-shift can always produce new counts  $(c'_0, c'_1, \dots, c'_{2^d-1})$  for which  $\sum c'_{2l} = \sum c'_{2l+1}$ , thus allowing the recursion to proceed.

- 75. [32] The method of Chung and Wong replicates bit  $l$  of a register exactly  $c_l$  times, but it produces results in scrambled order. For example, the case  $(c_0, \dots, c_7) = (1, 2, 0, 2, 0, 2, 0, 1)$  illustrated in the text produces  $(x_7 x_3 x_1 x_5 x_5 x_3 x_1 x_0)_2$ . In some applications this can be a disadvantage; we might prefer to have the bits retain their original order, namely  $(x_7 x_5 x_5 x_3 x_3 x_1 x_1 x_0)_2$  in that example.

Prove that the permutation network  $P(2^d)$  of Fig. 12 can be modified to achieve this goal, given any sequence of counts  $(c_0, c_1, \dots, c_{2^d-1})$ , if we replace the  $d \cdot 2^{d-1}$  crossbar modules in the right-hand half by general  $2 \times 2$  mapping modules. (A crossbar module with inputs  $(a, b)$  produces either  $(a, b)$  or  $(b, a)$  as output; a mapping module can also produce  $(a, a)$  or  $(b, b)$ .)

76. [47] A mapping network is analogous to a sorting network or a permutation network, but it uses  $2 \times 2$  mapping modules instead of comparators or crossbars, and it is supposed to be able to output all  $n^n$  possible mappings of its  $n$  inputs. Exercise 75, in conjunction with Fig. 12, shows that a mapping network for  $n = 2^d$  exists with only  $4d - 2$  levels of delay, and with  $n/2$  modules on each level; furthermore, this construction needs general  $2 \times 2$  mapping modules (instead of simple crossbars) in only  $d$  of those levels.

To within  $O(n)$ , what is the smallest number  $G(n)$  of modules that are sufficient to implement a general  $n$ -element mapping network?

77. [26] (R. W. Floyd and V. R. Pratt.) Design an algorithm that tests whether or not a given standard  $n$ -network is a sorting network, as defined in the exercises of Section 5.3.4. When the given network has  $r$  comparator modules, your algorithm should use  $O(r)$  bitwise operations on words of length  $2^n$ .

78. [M27] (Testing disjointness.) Suppose the binary numbers  $x_1, x_2, \dots, x_m$  each represent sets in a universe of  $n - k$  elements, so that each  $x_j$  is less than  $2^{n-k}$ . J. H. Quick (a student) decided to test whether the sets are disjoint by testing the condition

$$x_1 \mid x_2 \mid \dots \mid x_m = (x_1 + x_2 + \dots + x_m) \bmod 2^n.$$

Prove or disprove: Quick's test is valid if and only if  $k \geq \lg(m - 1)$ .

- 79. [20] If  $x \neq 0$  and  $x \subseteq \chi$ , what is an easy way to determine the largest integer  $x_i < x$  such that  $x_i \subseteq \chi$ ? (Thus  $(x_i)' = (x')_i = x$ , in connection with (84).)
80. [20] Suggest a fast way to find all maximal proper subsets of a set. More precisely, given  $\chi$  with  $\nu\chi = m$ , we want to find all  $x \subseteq \chi$  such that  $\nu x = m - 1$ .
81. [21] Find a formula for “scattered difference,” to go with the “scattered sum” (86).
82. [21] Is it easy to shift a scattered accumulator to the left by 1, for example to change  $(y_2 x_4 x_3 y_1 x_2 y_0 x_1 x_0)_2$  to  $(y_1 x_4 x_3 y_0 x_2 0 x_1 x_0)_2$ ?
- 83. [33] Continuing exercise 82, find a way to shift a scattered  $2^d$ -bit accumulator to the right by 1, given  $z$  and  $\chi$ , in  $O(d)$  steps.
84. [25] Given  $n$ -bit numbers  $z = (z_{n-1} \dots z_1 z_0)_2$  and  $\chi = (\chi_{n-1} \dots \chi_1 \chi_0)_2$ , explain how to calculate the “stretched” quantities  $z \leftarrow \chi = (z_{(n-1) \leftarrow \chi} \dots z_{1 \leftarrow \chi} z_{0 \leftarrow \chi})_2$  and  $z \rightarrow \chi = (z_{(n-1) \rightarrow \chi} \dots z_{1 \rightarrow \chi} z_{0 \rightarrow \chi})_2$ , where

$$j \leftarrow \chi = \max\{k \mid k \leq j \text{ and } \chi_k = 1\}, \quad j \rightarrow \chi = \min\{k \mid k \geq j \text{ and } \chi_k = 1\};$$

sheep-and-goats  
Chung  
Wong  
replicates  
mapping modules  
crossbar module  
mapping network  
sorting network  
distribution network, see mapping network  
permutation network  
Floyd  
Pratt  
sorting network  
disjointness  
represent sets  
Quick  
maximal proper subsets  
scattered difference  
scattered accumulator  
scattered shifting  
stretched  
segmented broadcasting, see stretching

we let  $z_{j \leftarrow \chi} = 0$  if  $\chi_k = 0$  for  $0 \leq k \leq j$ , and  $z_{j \rightarrow \chi} = 0$  if  $\chi_k = 0$  for  $n > k \geq j$ . For example, if  $n = 11$  and  $\chi = (01101110010)_2$ , then  $z \leftarrow \chi = (z_9 z_8 z_8 z_6 z_5 z_4 z_4 z_1 z_1 0)_2$  and  $z \rightarrow \chi = (0 z_9 z_8 z_8 z_6 z_5 z_4 z_4 z_1 z_1)_2$ .

**85.** [22] (K. D. Tocher, 1954.) Imagine that you have a vintage 1950s computer with a drum memory for storing data, and that you need to do some computations with a  $32 \times 32 \times 32$  array  $a[i, j, k]$ , whose subscripts are 5-bit integers in the range  $0 \leq i, j, k < 32$ . Unfortunately your machine has only a very small high-speed memory: You can access only 128 consecutive elements of the array in fast memory at any time. Since your application usually moves from  $a[i, j, k]$  to a neighboring position  $a[i', j', k']$ , where  $|i - i'| + |j - j'| + |k - k'| = 1$ , you have decided to allocate the array so that, if  $i = (i_4 i_3 i_2 i_1 i_0)_2$ ,  $j = (j_4 j_3 j_2 j_1 j_0)_2$ , and  $k = (k_4 k_3 k_2 k_1 k_0)_2$ , the array entry  $a[i, j, k]$  is stored in drum location  $(k_4 j_4 i_4 k_3 j_3 i_3 k_2 j_2 i_2 k_1 j_1 i_1 k_0 j_0 i_0)_2$ . By interleaving the bits in this way, a small change to  $i$ ,  $j$ , or  $k$  will cause only a small change in the address.

Discuss the implementation of this addressing function: (a) How does it change when  $i$ ,  $j$ , or  $k$  changes by  $\pm 1$ ? (b) How would you handle a random access to  $a[i, j, k]$ , given  $i$ ,  $j$ , and  $k$ ? (c) How would you detect a “page fault” (namely, the condition that a new segment of 128 elements must be swapped into fast memory from the drum)?

**86.** [M27] An array of  $2^p \times 2^q \times 2^r$  elements is to be allocated by putting  $a[i, j, k]$  into a location whose bits are the  $p + q + r$  bits of  $(i, j, k)$ , permuted in some fashion. Furthermore, this array is to be stored in an external memory using pages of size  $2^s$ . (Exercise 85 considers the case  $p = q = r = 5$  and  $s = 7$ .) What allocation strategy of this kind minimizes the number of times that  $a[i, j, k]$  is on a different page from  $a[i', j', k']$ , summed over all  $i, j, k, i', j'$ , and  $k'$  such that  $|i - i'| + |j - j'| + |k - k'| = 1$ ?

- **87.** [20] Suppose each byte of a 64-bit word  $x$  contains an ASCII code that represents either a letter, a digit, or a space. What three bitwise operations will convert all the lowercase letters to uppercase?

**88.** [20] Given  $x = (x_7 \dots x_0)_{256}$  and  $y = (y_7 \dots y_0)_{256}$ , compute  $z = (z_7 \dots z_0)_{256}$ , where  $z_j = (x_j - y_j) \bmod 256$  for  $0 \leq j < 8$ . (See the addition operation in (87).)

**89.** [23] Given  $x = (x_{31} \dots x_1 x_0)_4$  and  $y = (y_{31} \dots y_1 y_0)_4$ , compute  $z = (z_{31} \dots z_1 z_0)_4$ , where  $z_j = \lfloor x_j / y_j \rfloor$  for  $0 \leq j < 32$ , assuming that no  $y_j$  is zero.

**90.** [20] The bitwise averaging rule (88) always rounds downward when  $x_j + y_j$  is odd. Make it less biased by rounding to the nearest odd integer in such cases.

- **91.** [26] (*Alpha channels*.) Recipe (88) is a good way to compute bitwise averages, but applications to computer graphics often require a more general blending of 8-bit values. Given three octabytes  $x = (x_7 \dots x_0)_{256}$ ,  $y = (y_7 \dots y_0)_{256}$ ,  $\alpha = (a_7 \dots a_0)_{256}$ , show that bitwise operations allow us to compute  $z = (z_7 \dots z_0)_{256}$ , where each byte  $z_j$  is a good approximation to  $((255 - a_j)x_j + a_j y_j) / 255$ , *without* doing any multiplication. Implement your method with MMIX instructions.
- **92.** [21] What happens if the second line of (88) is changed to ‘ $z \leftarrow (x \mid y) - z$ ’?
- 93.** [18] What basic formula for subtraction is analogous to formula (89) for addition?
- 94.** [21] Let  $x = (x_7 \dots x_1 x_0)_{256}$  and  $t = (t_7 \dots t_1 t_0)_{256}$  in (90). Can  $t_j$  be nonzero when  $x_j$  is nonzero? Can  $t_j$  be zero when  $x_j$  is zero?
- 95.** [22] What’s a bitwise way to tell if all bytes of  $x = (x_7 \dots x_1 x_0)_{256}$  are distinct?
- 96.** [21] Explain (93), and find a similar formula that sets test flags  $t_j \leftarrow 128[x_j \leq y_j]$ .
- 97.** [23] Leslie Lamport’s paper in 1975 presented the following “problem taken from an actual compiler optimization algorithm”: Given octabytes  $x = (x_7 \dots x_0)_{256}$  and  $y =$

Tocher  
allocate  
storage allocation  
interleaving the bits  
page fault  
ASCII code  
lowercase letters  
uppercase  
multibyte subtraction  
packed data+  
division, 2-bit  
averaging  
rounding to the nearest odd  
unbiased rounding  
Alpha channels  
subtraction  
distinct  
flags  
Lamport



$(y_7 \dots y_0)_{256}$ , compute  $t = (t_7 \dots t_0)_{256}$  and  $z = (z_7 \dots z_0)_{256}$  so that  $t_j \neq 0$  if and only if  $x_j \neq 0$ ,  $x_j \neq '*'$ , and  $x_j \neq y_j$ ; and  $z_j = (x_j = 0? y_j: (x_j \neq '*' \wedge x_j \neq y_j? '*' : x_j))$ .

98. [20] Given  $x = (x_7 \dots x_0)_{256}$  and  $y = (y_7 \dots y_0)_{256}$ , compute  $z = (z_7 \dots z_0)_{256}$  and  $w = (w_7 \dots w_0)_{256}$ , where  $z_j = \max(x_j, y_j)$  and  $w_j = \min(x_j, y_j)$  for  $0 \leq j < 8$ .

- 99. [28] Find hexadecimal constants  $a, b, c, d, e$  such that the six bitwise operations

$$y \leftarrow x \oplus a, \quad t \leftarrow (((y \& b) + c) | y) \oplus d \& e$$

will compute the flags  $t = (f_7 \dots f_1 f_0)_{256} \ll 7$  from any bytes  $x = (x_7 \dots x_1 x_0)_{256}$ , where

$$f_0 = [x_0 = '!'], \quad f_1 = [x_1 \neq '*'], \quad f_2 = [x_2 < 'A'], \quad f_3 = [x_3 > 'Z'], \quad f_4 = [x_4 \geq 'a'], \\ f_5 = [x_5 \in \{'0', '1', \dots, '9'\}], \quad f_6 = [x_6 \leq 168], \quad f_7 = [x_7 \in \{'<', '=, '>', '?'\}].$$

100. [25] Suppose  $x = (x_{15} \dots x_1 x_0)_{16}$  and  $y = (y_{15} \dots y_1 y_0)_{16}$  are *binary-coded decimal* numbers, where  $0 \leq x_j, y_j < 10$  for each  $j$ . Explain how to compute their sum  $u = (u_{15} \dots u_1 u_0)_{16}$  and difference  $v = (v_{15} \dots v_1 v_0)_{16}$ , where  $0 \leq u_j, v_j < 10$  and

$$(u_{15} \dots u_1 u_0)_{10} = ((x_{15} \dots x_1 x_0)_{10} + (y_{15} \dots y_1 y_0)_{10}) \bmod 10^{16}, \\ (v_{15} \dots v_1 v_0)_{10} = ((x_{15} \dots x_1 x_0)_{10} - (y_{15} \dots y_1 y_0)_{10}) \bmod 10^{16},$$

without bothering to do any radix conversion.

- 101. [22] Two octabytes  $x$  and  $y$  contain amounts of time, represented in five fields that respectively signify days (3 bytes), hours (1 byte), minutes (1 byte), seconds (1 byte), and milliseconds (2 bytes). Can you add and subtract them quickly, without converting from this mixed-radix representation to binary and back again?
102. [25] Discuss routines for the addition and subtraction of polynomials modulo 5, when (a) 16 4-bit coefficients or (b) 21 3-bit coefficients are packed into a 64-bit word.
- 103. [22] Sometimes it's convenient to represent small numbers in *unary* notation, so that  $0, 1, 2, 3, \dots, k$  appear respectively as  $(0)_2, (1)_2, (11)_2, (111)_2, \dots, 2^k - 1$  inside the computer. Then max and min are easily implemented as  $|$  and  $\&$ .

Suppose the bytes of  $x = (x_7 \dots x_0)_{256}$  are such unary numbers, while the bytes of  $y = (y_7 \dots y_0)_{256}$  are all either 0 or 1. Explain how to “add”  $y$  to  $x$  or “subtract”  $y$  from  $x$ , giving  $u = (u_7 \dots u_0)_{256}$  and  $v = (v_7 \dots v_0)_{256}$  where

$$u_j = 2^{\min(8, \lg(x_j+1)+y_j)} - 1 \quad \text{and} \quad v_j = 2^{\max(0, \lg(x_j+1)-y_j)} - 1.$$

104. [22] Use bitwise operations to check the validity of a date represented in “year-month-day” fields  $(y, m, d)$  as in (22). You should compute a value  $t$  that is zero if and only if  $1900 < y < 2100$ ,  $1 \leq m \leq 12$ , and  $1 \leq d \leq \max\_day(m)$ , where month  $m$  has at most  $\max\_day(m)$  days. Can it be done in fewer than 20 operations?
105. [30] Given  $x = (x_7 \dots x_0)_{256}$  and  $y = (y_7 \dots y_0)_{256}$ , discuss bitwise operations that will *sort* the bytes into order, so that  $x_0 \leq y_0 \leq \dots \leq x_7 \leq y_7$  afterwards.
106. [27] Explain the Fredman–Willard procedure (95). Also show that a simple modification of their method will compute  $2^{\lambda x}$  without doing any left shifts.
- 107. [22] Implement Algorithm B on **MMIX** when  $d = 4$ , and compare it with (56).
108. [26] Adapt Algorithm B to cases where  $n$  does not have the form  $d \cdot 2^d$ .
109. [20] Evaluate  $\rho x$  for  $n$ -bit numbers  $x$  in  $O(\log \log n)$  broadword steps.

multibyte max and min  
comparison of bytes  
bytes, testing relative order of  
flags  
binary-coded decimal  
radix conversion  
time  
mixed-radix representation  
polynomials modulo 5  
unary  
max  
min  
date  
range checking  
sort  
Fredman  
Willard  
 $2^{\lambda x} +$   
binary log+  
 $\lambda x +$   
extract the most significant bit  
hyperfloor  
**MMIX**  
broadword  
ruler function

- **110.** [30] Suppose  $n = 2^{2^e}$  and  $0 \leq x < n$ . Show how to compute  $1 \ll x$  in  $O(e)$  broadword steps, using only shift commands that shift by a constant amount. (Together with Algorithm B we can therefore extract the most significant bit of an  $n$ -bit number in  $O(\log \log n)$  such steps.)

**111.** [23] Explain the  $01^r$  pattern recognizer, (98).

**112.** [46] Can all occurrences of the pattern  $1^r 0$  be identified in  $O(1)$  broadword steps?

**113.** [23] A *strong broadword chain* is a broadword chain of a specified width  $n$  that is also a 2-adic chain, for all  $n$ -bit choices of  $x_0$ . For example, the 2-bit broadword chain  $(x_0, x_1)$  with  $x_1 = x_0 + 1$  is not strong because  $x_0 = (11)_2$  makes  $x_1 = (00)_2$ . But  $(x_0, x_1, \dots, x_4)$  is a strong broadword chain that computes  $(x_0 + 1) \bmod 4$  for all  $0 \leq x_0 < 4$  if we set  $x_1 = x_0 \oplus 1$ ,  $x_2 = x_0 \& 1$ ,  $x_3 = x_2 \ll 1$ , and  $x_4 = x_1 \oplus x_3$ .

Given a broadword chain  $(x_0, x_1, \dots, x_r)$  of width  $n$ , construct a strong broadword chain  $(x'_0, x'_1, \dots, x'_{r'})$  of the same width, such that  $r' = O(r)$  and  $(x_0, x_1, \dots, x_r)$  is a subsequence of  $(x'_0, x'_1, \dots, x'_{r'})$ .

**114.** [16] Suppose  $(x_0, x_1, \dots, x_r)$  is a strong broadword chain of width  $n$  that computes the value  $f(x) = x_r$  whenever an  $n$ -bit number  $x = x_0$  is given. Construct a broadword chain  $(X_0, X_1, \dots, X_r)$  of width  $mn$  that computes  $X_r = (f(\xi_1) \dots f(\xi_m))_{2^n}$  for any given  $mn$ -bit value  $X_0 = (\xi_1 \dots \xi_m)_{2^n}$ , where  $0 \leq \xi_1, \dots, \xi_m < 2^n$ .

- **115.** [24] Given a 2-adic integer  $x = (\dots x_2 x_1 x_0)_2$ , we might want to compute  $y = (\dots y_2 y_1 y_0)_2 = f(x)$  from  $x$  by zeroing out all blocks of consecutive 1s that (a) are not immediately followed by two 0s; or (b) are followed by an odd number of 0s before the next block of 1s begins; or (c) contain an odd number of 1s. For example, if  $x$  is  $(\dots 01110111001101000110)_2$  then  $y$  is (a)  $(\dots 00000111000001000110)_2$ ; (b)  $(\dots 00000111000000000110)_2$ ; (c)  $(\dots 00000000001100000110)_2$ . (Infinitely many 0s are assumed to appear at the right of  $x_0$ . Thus, in case (a) we have

$$y_j = x_j \wedge ((\bar{x}_{j-1} \wedge \bar{x}_{j-2}) \vee (x_{j-1} \wedge \bar{x}_{j-2} \wedge \bar{x}_{j-3}) \vee (x_{j-1} \wedge x_{j-2} \wedge \bar{x}_{j-3} \wedge \bar{x}_{j-4}) \vee \dots)$$

for all  $j$ , where  $x_k = 0$  for  $k < 0$ .) Find 2-adic chains for  $y$  in each case.

**116.** [HM30] Suppose  $x = (\dots x_2 x_1 x_0)_2$  and  $y = (\dots y_2 y_1 y_0)_2 = f(x)$ , where  $y$  is computable by a 2-adic chain having no shift operations. Let  $L$  be the set of all binary strings such that  $y_j = [x_j \dots x_1 x_0 \in L]$ , and assume that all constants used in the chain are rational 2-adic numbers. Prove that  $L$  is a regular language. What languages  $L$  correspond to the functions in exercise 115(a) and 115(b)?

**117.** [HM46] Continuing exercise 116, is there any simple way to characterize the regular languages  $L$  that arise in shift-free 2-adic chains? (The language  $L = 0^*(10^*10^*)^*$  does not seem to correspond to any such chain.)

**118.** [30] According to Lemma A, we cannot compute the function  $x \gg 1$  for all  $n$ -bit numbers  $x$  by using only additions, subtractions, and bitwise Boolean operations (no shifts or branches). Show, however, that  $O(n)$  such operations are necessary and sufficient if we include also the “monus” operator  $y \dot{-} z$  in our repertoire.

**119.** [20] Evaluate the function  $f_{py}(x)$  in (102) with four broadword steps.

- **120.** [M25] There are  $2^{n^{2^m}}$  functions that take  $n$ -bit numbers  $(x_1, \dots, x_m)$  into an  $n$ -bit number  $f(x_1, \dots, x_m)$ . How many of them can be implemented with addition, subtraction, multiplication, and nonshift bitwise Boolean operations (modulo  $2^n$ )?
- **121.** [M25] By exercise 3.1–6, a function from  $[0 \dots 2^n)$  into itself is eventually periodic.

extract the most significant bit  
pattern  
strong broadword chain  
broadword chain, strong  
2-adic chains  
rational 2-adic numbers  
regular language  
shifts  
branches  
monus

- a) Prove that if  $f$  is any  $n$ -bit broadword function that can be implemented without shift instructions, the lengths of its periods are always powers of 2.
- b) However, for every  $p$  between 1 and  $n$ , there's an  $n$ -bit broadword chain of length 3 that has a period of length  $p$ .

**122.** [M22] Complete the proof of Lemma B.

**123.** [M23] Let  $a_q$  be the constant  $1 + 2^q + 2^{2q} + \dots + 2^{(q-1)q} = (2^{q^2} - 1)/(2^q - 1)$ . Using (104), show that there are infinitely many  $q$  such that the operation of multiplying by  $a_q$ , modulo  $2^{q^2}$ , requires  $\Omega(\log q)$  steps in any  $n$ -bit broadword chain with  $n \geq q^2$ .

**124.** [M38] Complete the proof of Theorem R' by defining an  $n$ -bit broadword chain  $(x_0, x_1, \dots, x_f)$  and sets  $(U_0, U_1, \dots, U_f)$  such that, for  $0 \leq t \leq f$ , all inputs  $x \in U_t$  lead to an essentially similar state  $Q(x, t)$ , in the following sense: (i) The current instruction in  $Q(x, t)$  does not depend on  $x$ . (ii) If register  $r_j$  has a known value in  $Q(x, t)$ , it holds  $x_{j'}$  for some definite index  $j' \leq t$ . (iii) If memory location  $M[z]$  has been changed, it holds  $x_{z''}$  for some definite index  $z'' \leq t$ . (The values of  $j'$  and  $z''$  depend on  $j$ ,  $z$ , and  $t$ , but not on  $x$ .) Furthermore  $|U_t| \geq n/2^{2^t-1}$ , and the program cannot guarantee that  $r_1 = \rho x$  when  $t < f$ . *Hint:* Lemma B implies that a limited number of shift amounts and memory addresses need to be considered when  $t$  is small.

**125.** [M33] Prove Theorem P'. *Hint:* Lemma B remains true if we replace '=' 0' by '='  $\alpha_s$ ' in (103), for any values  $\alpha_s$ .

**126.** [M46] Does the operation of extracting the most significant bit,  $2^{\lambda x}$ , require  $\Omega(\log \log n)$  steps in an  $n$ -bit basic RAM? (See exercise 110.)

**127.** [HM40] Prove that at least  $\Omega(\log n / \log \log n)$  broadword steps are needed to compute the parity function,  $(\nu x) \bmod 2$ , using the theory of circuit complexity. [*Hint:* Every boardword operation is in complexity class  $AC_0$ .]

**128.** [M46] Can  $(\nu x) \bmod 2$  be computed in  $O(\log n / \log \log n)$  broadword steps?

**129.** [M46] Does sideways addition require  $\Omega(\log n)$  broadword steps?

**130.** [M46] Is there an  $n$ -bit constant  $a$  such that the function  $(a \ll x) \bmod 2^n$  requires  $\Omega(\log n)$   $n$ -bit broadword steps?

- **131.** [23] Write an MMIX program for Algorithm R when the graph is represented by arc lists. Vertex nodes have at least two fields, called LINK and ARCS, and arc nodes have TIP and NEXT fields, as explained in Section 7. Initially all LINK fields are zero, except in the given set of vertices  $Q$ , which is represented as a circular list. Your program should change that circular list so that it represents the set  $R$  of all reachable vertices.
- **132.** [M27] A *clique* in a graph is a set of mutually adjacent vertices; a clique is *maximal* if it's not contained in any other. The purpose of this exercise is to discuss an algorithm due to J. K. M. Moody and J. Hollis, which provides a convenient way to find every maximal clique of a not-too-large graph, using bitwise operations.

Suppose  $G$  is a graph with  $n$  vertices  $V = \{0, 1, \dots, n-1\}$ . Let  $\rho_v = \sum \{2^u \mid u \text{ --- } v \text{ or } u = v\}$  be row  $v$  of  $G$ 's reflexive adjacency matrix, and let  $\delta_v = \sum \{2^u \mid u \neq v\} = 2^n - 1 - 2^v$ . Every subset  $U \subseteq V$  is representable as an  $n$ -bit integer  $\sigma(U) = \sum_{u \in U} 2^u$ ; for example,  $\delta_v = \sigma(V \setminus v)$ . We also define the bitwise intersection

$$\tau(U) = \big\&_{0 \leq u < n} (u \in U? \rho_u : \delta_u).$$

For example, if  $n = 5$  we have  $\tau(\{0, 2\}) = \rho_0 \& \delta_1 \& \rho_2 \& \delta_3 \& \delta_4$ .

- a) Prove that  $U$  is a clique if and only if  $\tau(U) = \sigma(U)$ .

period length  
multiplying  
broadword chain  
extracting the most significant bit  
basic RAM  
circuit complexity  
 $AC_0$   
parity function  
sideways addition  
circular list  
graph algorithms+  
cliques+  
graph representation  
set representation  
maximal  
Moody  
Hollis  
adjacency matrix

- b) Show that if  $\tau(U) = \sigma(T)$  then  $T$  is a clique.  
 c) For  $1 \leq k \leq n$ , consider the  $2^k$  bitwise intersections

$$C_k = \left\{ \bigwedge_{0 \leq u < k} (u \in U? \rho_u : \delta_u) \mid U \subseteq \{0, 1, \dots, k-1\} \right\},$$

and let  $C_k^+$  be the maximal elements of  $C_k$ . Prove that  $U$  is a maximal clique if and only if  $\sigma(U) \in C_n^+$ .

- d) Explain how to compute  $C_k^+$  from  $C_{k-1}^+$ , starting with  $C_0^+ = \{2^n - 1\}$ .

- **133.** [20] Given a graph  $G$ , how can the algorithm of exercise 132 be used to find (a) all maximal independent sets of vertices? (b) all minimal vertex covers (sets that hit every edge)?

**134.** [15] Nine classes of mappings for ternary values appear in (119), (123), and (124). To which class does the representation (128) belong, if  $a = 0$ ,  $b = *$ ,  $c = 1$ ?

**135.** [22] Lukasiewicz included a few operations besides (127) in his three-valued logic:  $\neg x$  (negation) interchanges 0 with 1 but leaves  $*$  unchanged;  $\diamond x$  (possibility) is defined as  $\neg x \Rightarrow x$ ;  $\Box x$  (necessity) is defined as  $\neg \diamond \neg x$ ; and  $x \Leftrightarrow y$  (equivalence) is defined as  $(x \Rightarrow y) \wedge (y \Rightarrow x)$ . Explain how to perform these operations using representation (128).

**136.** [29] Suggest two-bit encodings for binary operations on the set  $\{a, b, c\}$  that are defined by the following “multiplication tables”:

$$(a) \begin{pmatrix} a & b & c \\ b & c & c \\ c & c & c \end{pmatrix}; \quad (b) \begin{pmatrix} a & c & b \\ c & b & a \\ b & a & c \end{pmatrix}; \quad (c) \begin{pmatrix} a & b & a \\ a & a & c \\ a & b & c \end{pmatrix}.$$

**137.** [21] Show that the operation in exercise 136(c) is simpler with packed vectors like (131) than with the unpacked form (130).

**138.** [24] Find an example of three-state-to-two-bit encoding where class  $V_a$  is best.

**139.** [25] If  $x$  and  $y$  are signed bits 0, +1, or -1, what 2-bit encoding is good for calculating their sum  $(z_1 z_2)_3 = x + y$ , where  $z_1$  and  $z_2$  are also required to be signed bits? (This is a “half adder” for balanced ternary numbers.)

**140.** [27] Design an economical *full adder* for balanced ternary numbers: Show how to compute signed bits  $u$  and  $v$  such that  $3u + v = x + y + z$  when  $x, y, z \in \{0, +1, -1\}$ .

- **141.** [30] The *Ulam numbers*  $\langle U_1, U_2, \dots \rangle = \langle 1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, \dots \rangle$  are defined for  $n \geq 3$  by letting  $U_n$  be the smallest integer  $> U_{n-1}$  that has a *unique* representation  $U_n = U_j + U_k$  for  $0 < j < k < n$ . Show that a million Ulam numbers can be computed rapidly with the help of bitwise techniques.

- **142.** [33] A subcube such as  $*10*1*01$  can be represented by asterisk codes 10010100 and bit codes 01001001, as in (85); but many other encodings are also possible. What representation scheme for subcubes works best, for finding prime implicants by the consensus-based algorithm of exercise 7.1.1–31?

**143.** [20] Let  $x$  be a 64-bit number that represents an  $8 \times 8$  chessboard, with a 1 bit in every position where a knight is present. Find a formula for the 64-bit number  $f(x)$  that has a 1 in every position reachable in one move by a knight of  $x$ . For example, the white knights at the start of a game correspond to  $x = \#42$ ; then  $f(x) = \#a51800$ .

**144.** [16] What node is the sibling of node  $j$  in a sideways heap? (See (134).)

**145.** [17] Interpret (137) when  $h$  is *less* than the height of  $j$ .

- **146.** [M20] Prove Eq. (138), which relates the  $\rho$  and  $\lambda$  functions.

independent sets  
 vertex covers  
 mappings for ternary values  
 Lukasiewicz  
 three-valued logic  
 negation  
 possibility  
 necessity  
 equivalence  
 multiplication tables for groupoids  
 groupoids, mult tables for  
 packed  
 2-bit encoding  
 half adder  
 balanced ternary numbers  
 full adder  
 Ulam numbers  
 subcube  
 asterisk codes  
 bit codes  
 prime implicants  
 consensus  
 chessboard  
 knight  
 bit board  
 sibling  
 sideways heap+  
 $\rho$   
 $\lambda$



**165.** [21] (R. A. Kirsch.) Discuss the computation of the  $3 \times 3$  cellular automaton with

$$X^{(t+1)} = \text{custer}(\overline{X}^{(t)}) = \sim X^{(t)} \& (X_N^{(t)} \mid X_W^{(t)} \mid X_E^{(t)} \mid X_S^{(t)}).$$

**166.** [M23] Let  $f(M, N)$  be the maximum number of black pixels in an  $M \times N$  bitmap  $X$  for which  $X = \text{custer}(X)$ . Prove that  $f(M, N) = \frac{4}{5}MN + O(M + N)$ .

**167.** [24] (*Life*.) If the bitmap  $X$  represents an array of cells that are either dead (0) or alive (1), the Boolean function

$$f(x_{NW}, \dots, x, \dots, x_{SE}) = [2 < x_{NW} + x_N + x_{NE} + x_W + \frac{1}{2}x + x_E + x_{SW} + x_S + x_{SE} < 4]$$

can lead to astonishing life histories when it governs a cellular automaton as in (158).

a) Find a way to evaluate  $f$  with a Boolean chain of 26 steps or less.

b) Let  $X_j^{(t)}$  denote row  $j$  of  $X$  at time  $t$ . Show that  $X_j^{(t+1)}$  can be evaluated in at most 23 broadword steps, as a function of the three rows  $X_{j-1}^{(t)}$ ,  $X_j^{(t)}$ , and  $X_{j+1}^{(t)}$ .

- **168.** [23] To keep an image finite, we might insist that a  $3 \times 3$  cellular automaton treats a  $M \times N$  bitmap as a *torus*, wrapping around seamlessly between top and bottom and between left and right. The task of simulating its actions efficiently with bitwise operations is somewhat tricky: We want to minimize references to memory, yet each new pixel value depends on old values that lie on all sides. Furthermore the shifting of bits between neighboring words tends to be awkward, taxing the capacity of a register.

Show that such difficulties can be surmounted by maintaining an array of  $n$ -bit words  $A_{jk}$  for  $0 \leq j \leq M$  and  $0 \leq k \leq N' = \lceil N/(n-2) \rceil$ . If  $j \neq M$  and  $k \neq 0$ , word  $A_{jk}$  should contain the pixels of row  $j$  and columns  $(k-1)(n-2)$  through  $k(n-2)+1$ , inclusive; the other words  $A_{Mk}$  and  $A_{j0}$  provide auxiliary buffer space. (Notice that some bits of the raster appear twice.)

**169.** [22] Continuing the previous two exercises, what happens to the Cheshire cat of Fig. 17(a) when it is subjected to the vicissitudes of Life, in a  $26 \times 31$  torus?

- **170.** [21] What result does the Guo–Hall thinning automaton produce when given a solid black rectangle of  $M$  rows and  $N$  columns? How long does it take?

**171.** [24] Find a Boolean chain of length  $\leq 25$  to evaluate the local thinning function  $g(x_{NW}, x_N, x_{NE}, x_W, x_E, x_{SW}, x_S, x_{SE})$  of (159), with or without the extra cases in (160).

**172.** [M29] Prove or disprove: If a pattern contains three black pixels that are king-neighbors of each other, the Guo–Hall procedure extended by (160) will reduce it, unless none of those pixels can be removed without destroying the connectivity.

- **173.** [M30] Raster images often need to be cleaned up if they contain noisy data. For example, accidental specks of black or white may well spoil the results when a thinning algorithm is used for optical character recognition.

Say that a bitmap  $X$  is *closed* if every white pixel is part of a  $2 \times 2$  square of white pixels, and *open* if every black pixel is part of a  $2 \times 2$  square of black pixels. Let

$$X^D = \& \{Y \mid Y \supseteq X \text{ and } Y \text{ is closed}\}; \quad X^L = \mid \{Y \mid Y \subseteq X \text{ and } Y \text{ is open}\}.$$

A bitmap is called *clean* if it equals  $X^{DL}$  for some  $X$ . We might, for example, have

$$X = \begin{array}{|c|c|c|} \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \end{array}; \quad X^D = \begin{array}{|c|c|c|} \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \end{array}; \quad X^{DL} = \begin{array}{|c|c|c|} \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \blacksquare & \blacksquare & \blacksquare \\ \hline \end{array}.$$

In general  $X^D$  is “darker” than  $X$ , while  $X^L$  is “lighter”:  $X^D \supseteq X \supseteq X^L$ .

- a) Prove that  $(X^{DL})^{DL} = X^{DL}$ . *Hint:*  $X \subseteq Y$  implies  $X^D \subseteq Y^D$  and  $X^L \subseteq Y^L$ .  
b) Show that  $X^D$  can be computed with one step of a  $3 \times 3$  cellular automaton.

Kirsch  
cellular automaton  
Life  
broadword  
torus  
Cheshire cat  
Life  
Guo  
Hall  
thinning  
noisy data  
thinning  
optical character recognition  
closed  
open  
clean

**174.** [M46] (M. Minsky and S. Papert.) Is there a three-dimensional shrinking algorithm that preserves connectivity, analogous to (161)?

**175.** [15] How many *rookwise* connected black components does the Cheshire cat have?

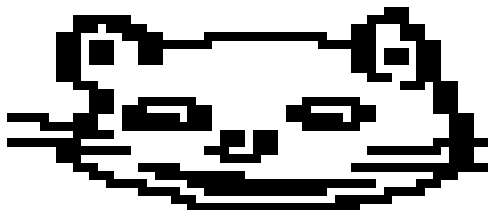
**176.** [M24] Let  $G$  be the graph whose vertices are the black pixels of a given bitmap  $X$ , with  $u \sim v$  when  $u$  and  $v$  are a king move apart. Let  $G'$  be the corresponding graph after the shrinking transformation (161) has been applied. The purpose of this exercise is to show that the number of connected components of  $G'$  is the number of components of  $G$  minus the number of isolated vertices of  $G$ .

Let  $N_{(i,j)} = \{(i,j), (i-1,j), (i-1,j+1), (i,j+1)\}$  be pixel  $(i,j)$  together with its north and/or east neighbors. For each  $v \in G$  let  $S(v) = \{v' \in G' \mid v' \in N_v\}$ .

- Prove that  $S(v)$  is empty if and only if  $v$  is isolated in  $G$ .
- If  $u \sim v$  in  $G$ ,  $u' \in S(u)$ , and  $v' \in S(v)$ , prove that  $u' \sim^* v'$  in  $G'$ .
- For each  $v' \in G'$  let  $S'(v') = \{v \in G \mid v' \in N_v\}$ . Is  $S'(v')$  always nonempty?
- If  $u' \sim v'$  in  $G'$ ,  $u \in S'(u')$ , and  $v \in S'(v')$ , prove that  $u \sim^* v$  in  $G$ .
- Hence there's a one-to-one correspondence between the nontrivial components of  $G$  and the components of  $G'$ .

**177.** [M22] Continuing exercise 176, prove an analogous result for the white pixels.

**178.** [20] If  $X$  is an  $M \times N$  bitmap, let  $X^*$  be the  $M \times (2N+1)$  bitmap  $X \ddagger (X \mid (X \ll 1))$ . Show that the kingwise connected components of  $X^*$  are also rookwise connected, and that bitmap  $X^*$  has the same “surroundedness tree” (162) as  $X$ .



- **179.** [34] Design an algorithm that constructs the surroundedness tree of a given  $M \times N$  bitmap, scanning the image one row at a time as discussed in the text. (See (162) and (163).)
- **180.** [M24] Digitize the hyperbola  $y^2 = x^2 + 13$  by hand, for  $0 < y \leq 7$ .
- 181.** [HM20] Explain how to subdivide a general conic (168) with rational coefficients into monotonic parts so that Algorithm T applies.
- 182.** [M31] Why does the three-register method (Algorithm T) digitize correctly?
- **183.** [M29] (G. Rote.) Explain why Algorithm T might fail if condition (v) is false.
- **184.** [M22] Find a quadratic form  $Q'(x,y)$  so that, when Algorithm T is applied to  $(x',y')$ ,  $(x,y)$ , and  $Q'$ , it produces exactly the same edges as it does from  $(x,y)$ ,  $(x',y')$ , and  $Q$ , but in the reverse order.
- **185.** [22] Design an algorithm that properly digitizes a straight line from  $(\xi,\eta)$  to  $(\xi',\eta')$ , when  $\xi$ ,  $\eta$ ,  $\xi'$ , and  $\eta'$  are rational numbers, by simplifying Algorithm T.
- 186.** [HM22] Given three complex numbers  $(z_0, z_1, z_2)$ , consider the curve traced out by

$$B(t) = (1-t)^2 z_0 + 2(1-t)t z_1 + t^2 z_2, \quad \text{for } 0 \leq t \leq 1.$$

- What is the approximate behavior of  $B(t)$  when  $t$  is near 0 or 1?
- Let  $S(z_0, z_1, z_2) = \{B(t) \mid 0 \leq t \leq 1\}$ . Prove that all points of  $S(z_0, z_1, z_2)$  lie on or inside the triangle whose vertices are  $z_0$ ,  $z_1$ , and  $z_2$ .
- True or false?  $S(w + \zeta z_0, w + \zeta z_1, w + \zeta z_2) = w + \zeta S(z_0, z_1, z_2)$ .
- Prove that  $S(z_0, z_1, z_2)$  is part of a straight line if and only if  $z_0$ ,  $z_1$ , and  $z_2$  are collinear; otherwise it is part of a parabola.

Minsky  
Papert  
shrinking  
Cheshire cat  
zipper function  
kingwise connected  
rookwise connected  
surroundedness tree  
hyperbola  
conic  
monotonic parts  
three-register method  
Rote  
straight line  
Bézier splines+  
squines  
parabola

- e) Prove that if  $0 \leq \theta \leq 1$ , we have the recurrence

$$S(z_0, z_1, z_2) = S(z_0, (1-\theta)z_0 + \theta z_1, B(\theta)) \cup S(B(\theta), (1-\theta)z_1 + \theta z_2, z_2).$$

**187.** [M29] Continuing exercise 186, show how to digitize  $S(z_0, z_1, z_2)$  using the three-register method (Algorithm T). For best results, the digitizations of  $S(z_2, z_1, z_0)$  and  $S(z_0, z_1, z_2)$  should produce the same edges, but in reverse order.

**188.** [25] Given a  $64 \times 64$  bitmap, what's a good way (a) to transpose it, or (b) to rotate it by  $90^\circ$ , using operations on 64-bit numbers?

- **189.** [25] Bitmap images can often be viewed conveniently using pixels that are *shades of gray* instead of just black or white. Such gray levels typically are 8-bit values that range from 0 (black) to 255 (white); notice that the black/white convention is traditionally *reversed* with respect to the 1-bit case. An  $m \times n$  bitmap whose resolution is 600 dots per inch corresponds nicely to the  $(m/8) \times (n/8)$  grayscale image with 75 pixels per inch that is obtained by mapping each  $8 \times 8$  subarray of 1-bit pixels into the gray level  $\lfloor 255(1 - k/64)^{1/\gamma} + \frac{1}{2} \rfloor$ , where  $\gamma = 1.3$  and  $k$  is the number of 1s in the subarray.

Write an MMIX routine that converts a given  $m \times n$  array **BITMAP** into the corresponding  $(m/8) \times (n/8)$  image **GRAYMAP**, assuming that  $m = 8m'$  and  $n = 64n'$ .

**190.** [23] A *parity pattern* of length  $m$  and width  $n$  is an  $m \times n$  matrix of 0s and 1s with the property that each element is the sum of its neighbors, mod 2. For example,

$$\begin{array}{ccccc} & & & 100 & & 01110 \\ & & 0011 & & 110 & 10101 \\ 11 & & 0100 & 01010 & & 11011 \\ 00, & 1101, & 11011, & 101, & \text{and} & 10101 \\ 11 & 0101 & 01010 & 011 & & 01110 \\ & & & 001 & & \end{array}$$

are parity patterns of sizes  $3 \times 2$ ,  $4 \times 4$ ,  $3 \times 5$ ,  $5 \times 3$ , and  $5 \times 5$ .

- If the binary vectors  $\alpha_1, \alpha_2, \dots, \alpha_m$  are the rows of a parity pattern, show that  $\alpha_2, \dots, \alpha_m$  can all be computed from the top row  $\alpha_1$  by using bitwise operations. Thus at most one  $m \times n$  parity pattern can begin with any given bit vector.
  - True or false: The sum (mod 2) of two  $m \times n$  parity patterns is a parity pattern.
  - A parity pattern is called *perfect* if it contains no all-zero row or column. For example, three of the matrices above are perfect, but the  $3 \times 2$  and  $3 \times 5$  examples are not. Show that every  $m \times n$  parity pattern contains a perfect parity pattern as a submatrix. Furthermore, all such submatrices have the same size,  $m' \times n'$ , where  $m' + 1$  is a divisor of  $m + 1$  and  $n' + 1$  is a divisor  $n + 1$ .
  - There's a perfect parity pattern whose first row is 0011, but there is no such pattern beginning with 01010. Is there a simple way to decide whether a given binary vector is the top row of a perfect parity pattern?
  - Prove that there's a unique perfect parity pattern that begins with  $1 \overbrace{0 \dots 0}^{n-1}$ .
- 191.** [M30] A *wraparound parity pattern* is analogous to the parity patterns of exercise 190, except that the leftmost and rightmost elements of each row are also neighbors.
- Find a simple relation between the parity pattern of width  $n$  that begins with  $\alpha$  and the wraparound parity pattern of width  $2n + 2$  that begins with  $0\alpha 0\alpha^R$ .
  - The Fibonacci polynomials  $F_j(x)$  are defined by the recurrence

$$F_0(x) = 0, \quad F_1(x) = 1, \quad \text{and} \quad F_{j+1}(x) = xF_j(x) + F_{j-1}(x) \quad \text{for } j \geq 1.$$

Show that there's a simple relation between the wraparound parity patterns that begin with  $10 \dots 0$  ( $N-1$  zeros) and the Fibonacci polynomials modulo  $x^N + 1$ . *Hint:* Consider  $F_j(x^{-1} + 1 + x)$ , and do arithmetic mod 2 as well as mod  $x^N + 1$ .

transpose  
rotate  
pixels  
shades of gray  
gray levels  
black  
white  
parity pattern+  
perfect parity pattern+  
0-1 matrix, see also bitmap  
Boolean matrix, see also bitmap  
wraparound parity pattern  
recurrence  
Fibonacci polynomials+  
polynomial remainder mod 2



- c) If  $\alpha$  is the binary string  $a_1 \dots a_n$ , let  $f_\alpha(x) = a_1x + \dots + a_nx^n$ . Show that

$$f_{(\alpha_j 0 \alpha_j^R)}(x) = (f_\alpha(x) + f_\alpha(x^{-1}))F_j(x^{-1}+1+x) \bmod (x^N + 1) \text{ and } \bmod 2,$$

when  $N = 2n + 2$  and  $\alpha_j$  is row  $j$  of a width- $n$  parity pattern that begins with  $\alpha$ .

- d) Consequently we can compute  $\alpha_j$  from  $\alpha$  in only  $O(n^2 \log j)$  steps. *Hints:* See exercise 4.6.3–26; and use the identity  $F_{m+n}(x) = F_m(x)F_{n+1}(x) + F_{m-1}(x)F_n(x)$ , which generalizes Eq. 1.2.8–(6).

**192.** [HM38] The shortest parity pattern that begins with a given string can be quite long; for example, it turns out that the perfect pattern of width 120 whose first row is  $10\dots 0$  has length 36,028,797,018,963,966(!). The purpose of this exercise is to consider how to calculate the interesting function

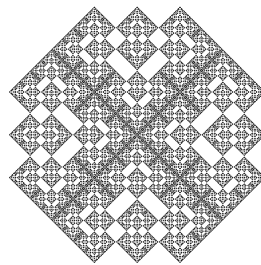
$$c(q) = 1 + \max\{m \mid \text{there exists a perfect parity pattern of length } m \text{ and width } q-1\},$$

whose initial values  $(1, 3, 4, 6, 5, 24, 9, 12, 28)$  for  $1 \leq q \leq 9$  are easy to compute by hand.

- Characterize  $c(q)$  algebraically, using the Fibonacci polynomials of exercise 191.
- Explain how to calculate  $c(q)$  if we know a number  $M$  such that  $c(q)$  divides  $M$ , and if we also know the prime factors of  $M$ .
- Prove that  $c(2^e) = 3 \cdot 2^{e-1}$  when  $e > 0$ . *Hint:*  $F_{2^e}(y)$  has a simple form, mod 2.
- Prove that when  $q$  is odd and not a multiple of 3,  $c(q)$  is a divisor of  $2^{2^e} - 1$ , where  $e$  is the order of 2 modulo  $q$ . *Hint:*  $F_{2^e-1}(y)$  has a simple form, mod 2.
- What happens when  $q$  is an odd multiple of 3?
- Finally, explain how to handle the case when  $q$  is even.

- **193.** [M21] If a perfect  $m \times n$  parity pattern exists, when  $m$  and  $n$  are odd, show that there's also a perfect  $(2m+1) \times (2n+1)$  parity pattern. (Intricate fractals arise when this observation is applied repeatedly; for example, the  $5 \times 5$  pattern in exercise 190 leads to Fig. 20.)

**194.** [M24] Find all  $n \leq 383$  for which there exists a perfect  $n \times n$  parity pattern with 8-fold symmetry, such as the example in Fig. 20. *Hint:* The diagonal elements of all such patterns must be zero.



**Fig. 20.** A perfect  $383 \times 383$  parity pattern.

- **195.** [HM25] Let  $A$  be a binary matrix having rows  $\alpha_1, \dots, \alpha_m$  of length  $n$ . Explain how to use bitwise operations to compute the rank  $m - r$  of  $A$  over the binary field  $\{0, 1\}$ , and to find linearly independent binary vectors  $\theta_1, \dots, \theta_r$  of length  $m$  such that  $\theta_j A = 0 \dots 0$  for  $1 \leq j \leq r$ . *Hint:* See the “triangularization” algorithm for null spaces, Algorithm 4.6.2N.

**196.** [21] (K. Thompson, 1992.) Integers in the range  $0 \leq x < 2^{31}$  can be encoded as a string of up to six bytes  $\alpha(x) = \alpha_1 \dots \alpha_l$  in the following way: If  $x < 2^7$ , set  $l \leftarrow 1$  and  $\alpha_1 \leftarrow x$ . Otherwise let  $x = (x_5 \dots x_1 x_0)_{64}$ ; set  $l \leftarrow \lceil (\lambda x)/5 \rceil$ ,  $\alpha_1 \leftarrow 2^8 - 2^{8-l} + x_{l-1}$ , and  $\alpha_j = 2^7 + x_{l-j}$  for  $2 \leq j \leq l$ . Notice that  $\alpha(x)$  contains a zero byte if and only if  $x = 0$ .

- What are the encodings of #a, #3a3, #7b97, and #1d141?
- If  $x \leq x'$ , prove that  $\alpha(x) \leq \alpha(x')$  in lexicographic order.
- Suppose a sequence of values  $x^{(1)}x^{(2)}\dots x^{(n)}$  has been encoded as a byte string  $\alpha(x^{(1)})\alpha(x^{(2)})\dots\alpha(x^{(n)})$ , and let  $\alpha_k$  be the  $k$ th byte in that string. Show that it's easy to determine the value  $x^{(i)}$  from which  $\alpha_k$  came, by looking at a few of the neighboring bytes if necessary.

fractals  
0–1 matrix  
rank  
triangularization  
null spaces  
Thompson  
multibyte encoding+  
lexicographic order

**197.** [22] The Universal Character Set (UCS), also known as Unicode, is a standard mapping of characters to integer codepoints  $x$  in the range  $0 \leq x < 2^{20} + 2^{16}$ . An encoding called UTF-16 represents such integers as one or two wydes  $\beta(x) = \beta_1$  or  $\beta(x) = \beta_1\beta_2$ , in the following way: If  $x < 2^{16}$  then  $\beta(x) = x$ ; otherwise

$$\beta_1 = \#d800 + \lfloor y/2^{10} \rfloor \text{ and } \beta_2 = \#dc00 + (y \bmod 2^{10}), \text{ where } y = x - 2^{16}.$$

Answer questions (a), (b), and (c) of exercise 196 for this encoding.

- **198.** [21] Unicode characters are often represented as strings of bytes using a scheme called UTF-8, which is the encoding of exercise 196 restricted to integers in the range  $0 \leq x < 2^{20} + 2^{16}$ . Notice that UTF-8 efficiently preserves the standard ASCII character set (the codepoints with  $x < 2^7$ ), and that it is quite different from UTF-16.

Let  $\alpha_1$  be the first byte of a UTF-8 string  $\alpha(x)$ . Show that there are reasonably small integer constants  $a$ ,  $b$ , and  $c$  such that only four bitwise operations

$$(a \gg ((\alpha_1 \gg b) \& c)) \& 3$$

suffice to determine the number  $l - 1$  of bytes between  $\alpha_1$  and the end of  $\alpha(x)$ .

- **199.** [23] A person might try to encode `#a` as `#c08a` or `#e0808a` or `#f080808a` in UTF-8, because the obvious decoding algorithm produces the same result in each case. But such unnecessarily long forms are illegal, because they could lead to security holes.

Suppose  $\alpha_1$  and  $\alpha_2$  are bytes such that  $\alpha_1 \geq \#80$  and  $\#80 \leq \alpha_2 < \#c0$ . Find a branchless way to decide whether  $\alpha_1$  and  $\alpha_2$  are the first two bytes of at least one legitimate UTF-8 string  $\alpha(x)$ .

**200.** [20] Interpret the contents of register `$3` after the following three MMIX instructions have been executed: `MOR $1,$0,#94`; `MXOR $2,$0,#94`; `SUBU $3,$1,$2`.

**201.** [20] Suppose  $x = (x_{15} \dots x_1 x_0)_{16}$  has sixteen hexadecimal digits. What one MMIX instruction will change each nonzero digit to `f`, while leaving zeros untouched?

**202.** [20] What two instructions will change an octabyte's nonzero wydes to `#ffff`?

**203.** [22] Suppose we want to convert a tetrabyte  $x = (x_7 \dots x_1 x_0)_{16}$  to the octabyte  $y = (y_7 \dots y_1 y_0)_{256}$ , where  $y_j$  is the ASCII code for the hexadecimal digit  $x_j$ . For example, if  $x = \#1234abcd$ ,  $y$  should represent the 8-character string "1234abcd". What clever choices of five constants `a`, `b`, `c`, `d`, and `e` will make the following MMIX instructions do the job?

```
MOR t,x,a; SLU s,t,4; XOR t,s,t; AND t,t,b;
ADD t,t,c; MOR s,d,t; ADD t,t,e; ADD y,t,s.
```

- **204.** [22] What are the amazing constants  $p$ ,  $q$ ,  $r$ ,  $m$  that achieve a perfect shuffle with just six MMIX commands? (See (175)–(178).)
- **205.** [22] How would you perfectly *unshuffle* on MMIX, going from  $w$  in (175) back to  $z$ ?
- 206.** [20] The perfect shuffle (175) is sometimes called an “outshuffle,” by comparison with the “inshuffle” that takes  $z \mapsto y \ddagger x = (y_{31}x_{31} \dots y_1x_1y_0x_0)_2$ ; the outshuffle preserves the leftmost and rightmost bits of  $z$ , but the inshuffle has no fixed points. Can an inshuffle be performed as efficiently as an outshuffle?
- 207.** [22] Use `MOR` to perform a 3-way perfect shuffle or “triple zip,” taking  $(x_{63} \dots x_0)_2$  to  $(x_{21}x_{42}x_{63}x_{20} \dots x_{23}x_{44}x_{122}x_{43}x_0)_2$ , as well as the inverse of this shuffle.
- **208.** [23] What's a fast way for MMIX to transpose an  $8 \times 8$  Boolean matrix?
- **209.** [21] Is the suffix parity operation  $x^\oplus$  of exercise 36 easy to compute with `MXOR`?

Universal Character Set  
UCS  
Unicode  
UTF-16: 16-bit UCS Transformation Format  
UTF-8  
ASCII  
packing  
fractional precision  
table lookup by shifting  
wyde: a 16-bit quantity  
byte: an 8-bit quantity  
nybble: a 4-bit quantity  
nyp: a 2-bit quantity  
tetrabyte or tetra: a 32-bit quantity  
octabyte or octa: a 64-bit quantity  
security  
branchless  
MOR+  
MXOR+  
hexadecimal digits  
masks  
ASCII  
hexadecimal digit  
perfect shuffle  
outshuffle  
inshuffle  
3-way perfect shuffle  
triple zip  
transpose  
Boolean matrix  
suffix parity

**210.** [22] A puzzle: Register **x** contains a number  $8j+k$ , where  $0 \leq j, k < 8$ . Registers **a** and **b** contain arbitrary octabytes  $(a_7 \dots a_1 a_0)_{256}$  and  $(b_7 \dots b_1 b_0)_{256}$ . Find a sequence of four **MMIX** instructions that will put  $a_j$  &  $b_k$  into register **x**.

- **211.** [M25] The truth table of a Boolean function  $f(x_1, \dots, x_6)$  is essentially a 64-bit number  $f = (f(0, 0, 0, 0, 0, 0) \dots f(1, 1, 1, 1, 1, 0) f(1, 1, 1, 1, 1, 1))_2$ . Show that two **MOR** instructions will convert  $f$  to the truth table of the least monotone Boolean function,  $\hat{f}$ , that is greater than or equal to  $f$  at each point.

**212.** [M32] Suppose  $a = (a_{63} \dots a_1 a_0)_2$  represents the polynomial

$$a(x) = (a_{63} \dots a_1 a_0)_x = a_{63}x^{63} + \dots + a_1x + a_0.$$

Discuss using **MXOR** to compute the product  $c(x) = a(x)b(x)$ , modulo  $x^{64}$  and mod 2.

- **213.** [HM26] Implement the CRC procedure (183) on **MMIX**.
- **214.** [HM28] (R. W. Gosper.) Find a short, branchless **MMIX** computation that computes the inverse of any given  $8 \times 8$  matrix  $X$  of 0s and 1s, modulo 2, if  $\det X$  is odd.
- **215.** [21] What's a quick way for **MMIX** to test if a 64-bit number is a multiple of 3?
- **216.** [M26] Given  $n$ -bit integers  $x_1, \dots, x_m \geq 0$ ,  $n \geq \lambda m$ , compute in  $O(m)$  steps the least  $y > 0$  such that  $y \notin \{a_1x_1 + \dots + a_mx_m \mid a_1, \dots, a_m \in \{0, 1\}\}$ , if  $\lambda x$  takes unit time.
- 217.** [40] Explore the processing of long strings of text by packing them in a “transposed” or “sliced” manner: Represent 64 consecutive characters as a sequence of eight octabytes  $w_0 \dots w_7$ , where  $w_k$  contains all 64 of their  $k$ th bits.

truth table  
monotone Boolean function  
polynomial multiplication  
**MXOR**  
**CRC**  
Gosper  
branchless  
inverse  
matrix  $X$  of 0s and 1s  
Divisibility by 3  
 $\lambda x$   
missing subset sum  
subset sum, first missing  
transposed  
sliced

## SECTION 7.1.3

1. These operations interchange the bits of  $x$  and  $y$  in positions where  $m$  is 1. (In particular, if  $m = -1$ , the step ' $y \leftarrow y \oplus (x \& m)$ ' becomes just ' $y \leftarrow y \oplus x$ ', and the three assignments will swap  $x \leftrightarrow y$  without needing an auxiliary register. H. S. Warren, Jr., has located this trick in vintage-1961 IBM programming course notes.)

2. All three hold when  $x$  and  $y$  are nonnegative, or if we regard  $x$  and  $y$  as "unsigned 2-adic integers" in which  $0 < 1 < 2 < \dots < -3 < -2 < -1$ . But if negative integers are less than nonnegative integers, (i) fails if and only if  $x < 0$  and  $y < 0$ ; (ii) and (iii) fail if and only if  $x \oplus y < 0$ , namely, if and only if  $x < 0$  and  $y \geq 0$  or  $x \geq 0$  and  $y < 0$ .

3. Note that  $x - y = (x \oplus y) - 2(\bar{x} \& y)$  (see exercise 93). By removing bits common to  $x$  and  $y$  at the left, we may assume that  $x_{n-1} = 1$  and  $y_{n-1} = 0$ . Then  $2(\bar{x} \& y) \leq 2((x \oplus y) - 2^{n-1}) = (x \oplus y) - (x \oplus y)^M - 1$ .

4.  $x^{CN} = x + 1 = x^S$ , by (16). Hence  $x^{NC} = x^{NCSP} = x^{NCCNP} = x^{NNP} = x^P$ .

5. (a) Disproof: Let  $x = (\dots x_2 x_1 x_0)_2$ . Then digit  $l$  of  $x \ll k$  is  $x_{l-k}[l \geq k]$ . So digit  $l$  of the left-hand side is  $x_{l-k-j}[l \geq k][l - k \geq j]$ , while digit  $l$  of the right-hand side is  $x_{l-j-k}[l \geq j + k]$ . These expressions agree if  $j \geq 0$  or  $k \leq 0$ . But if  $j < 0 < k$ , they differ when  $l = \max(0, j + k)$  and  $x_{l-j-k} = 1$ .

(We do, however, have  $(x \ll j) \ll k \subseteq x \ll (j + k)$  in all cases.)

(b) Proof: Digit  $l$  in all three formulas is  $x_{l+j}[l \geq -j] \wedge y_{l-k}[l \geq k]$ .

6. Since  $x \ll y \geq 0$  if and only if  $x \geq 0$ , we must have  $x \geq 0$  if and only if  $y \geq 0$ . Obviously  $x = y$  is always a solution. The solutions with  $x > y$  are (a)  $x = -1$  and  $y = -2$ , or  $2^y > x > y > 0$ ; (b)  $x = 2$  and  $y = 1$ , or  $2^{-x} \geq -y > -x > 0$ .

7. Set  $x' \leftarrow (x + \bar{\mu}_0) \oplus \bar{\mu}_0$ , where  $\mu_0$  is the constant in (47). Then  $x' = (\dots x'_2 x'_1 x'_0)_2$ , since  $(x' \oplus \bar{\mu}_0) - \bar{\mu}_0 = (\dots \bar{x}'_3 \bar{x}'_2 \bar{x}'_1 \bar{x}'_0)_2 - (\dots 1010)_2 = (\dots 0x'_2 0x'_0)_2 - (\dots x'_3 0x'_1 0)_2 = x$ .

[This is Hack 128 in HAKMEM; see answer 20 below. An alternative formula,  $x' \leftarrow (\mu_0 - x) \oplus \mu_0$ , has also been suggested by D. P. Agrawal, *IEEE Trans. C-29* (1980), 1032–1035. The results are correct modulo  $2^n$  for all  $n$ , but overflow or underflow can occur. For example, two's complement binary numbers in an  $n$ -bit register range from  $-2^{n-1}$  to  $2^{n-1} - 1$ , inclusive, but negabinary numbers range from  $-\frac{2}{3}(2^n - 1)$  to  $\frac{1}{3}(2^n - 1)$  when  $n$  is even. In general the formula  $x' \leftarrow (x + \mu) \oplus \mu$  converts from binary notation to the general number system with binary basis  $\langle 2^n(-1)^{m_n} \rangle$  discussed in exercise 4.1–30(c), when  $\mu = (\dots m_2 m_1 m_0)_2$ .]

8. First,  $x \oplus y \notin (S \oplus y) \cup (x \oplus T)$ . Second, suppose that  $0 \leq k < x \oplus y$ , and let  $x \oplus y = (\alpha 1 \alpha')_2$ ,  $k = (\alpha 0 \alpha'')_2$ , where  $\alpha$ ,  $\alpha'$ , and  $\alpha''$  are strings of 0s and 1s with  $|\alpha'| = |\alpha''|$ . Assume by symmetry that  $x = (\beta 1 \beta')_2$  and  $y = (\gamma 0 \gamma')_2$ , where  $|\alpha| = |\beta| = |\gamma|$ . Then  $k \oplus y = (\beta 0 \gamma'')_2$  is less than  $x$ . Hence  $k \oplus y \in S$ , and  $k = (k \oplus y) \oplus y \in S \oplus y$ . [See R. P. Sprague, *Tôhoku Math. J.* **41** (1936), 438–444; P. M. Grundy, *Eureka* **2** (1939), 6–8.]

9. The Sprague–Grundy theorem in the previous exercise shows that two piles of  $x$  and  $y$  sticks are equivalent in play to a single pile of  $x \oplus y$  sticks. (There is a nonnegative integer  $k < x \oplus y$  if and only if there either is a nonnegative  $i < x$  with  $i \oplus y < x \oplus y$  or a nonnegative  $j < y$  with  $x \oplus j < x \oplus y$ .) So the  $k$  piles are equivalent to a single pile of size  $a_1 \oplus \dots \oplus a_k$ . [See C. L. Bouton, *Annals of Math. (2)* **3** (1901–1902), 35–39.]

10. For clarity and brevity we shall write simply  $xy$  for  $x \otimes y$  and  $x + y$  for  $x \oplus y$ , in parts (i) through (iv) of this answer only.

(i) Clearly  $0y = 0$  and  $x + y = y + x$  and  $xy = yx$ . Also  $1y = y$ , by induction on  $y$ .

(ii) If  $x \neq x'$  and  $y \neq y'$  then  $xy + xy' + x'y + x'y' \neq 0$ , because the definition of  $xy$  says that  $xy' + x'y + x'y' < xy$  when  $x' < x$  and  $y' < y$ . In particular, if  $x \neq 0$  and

interchange  
swap  
Warren  
unsigned 2-adic integers  
magic  
HAKMEM  
Agrawal  
two's complement  
binary basis  
Sprague  
Grundy  
Bouton  
commutative laws

$y \neq 0$  then  $xy \neq 0$ . Another consequence is that, if  $x = \text{mex}(S)$  and  $y = \text{mex}(T)$  for arbitrary finite sets  $S$  and  $T$ , we have  $xy = \text{mex}\{xj + iy + ij \mid i \in S, j \in T\}$ .

(iii) Consequently, by induction on the (ordinary) sum of  $x$ ,  $y$ , and  $z$ ,  $(x + y)z$  is

$$\text{mex}\{(x + y)z' + (x' + y)z + (x' + y)z', (x + y)z' + (x + y')z + (x + y')z' \\ \mid 0 \leq x' < x, 0 \leq y' < y, 0 \leq z' < z\},$$

which is  $\text{mex}\{xz' + x'z + x'z' + yz, xz + yz' + y'z + y'z'\} = xz + yz$ . In particular, there's a cancellation law: If  $xz = yz$  then  $(x + y)z = 0$ , so  $x = y$  or  $z = 0$ .

(iv) By a similar induction,  $(xy)z = \text{mex}\{(xy)z' + (xy' + x'y + x'y')(z + z')\} = \text{mex}\{(xy)z' + (xy')z + (x'y')z' + \dots\} = \text{mex}\{x(yz') + x(y'z) + x(y'z') + \dots\} = \text{mex}\{(x + x')(yz' + y'z + y'z') + x'(yz)\} = x(yz)$ .

(v) If  $0 \leq x, y < 2^{2^n}$  we shall prove that  $x \otimes y < 2^{2^n}$ ,  $2^{2^n} \otimes y = 2^{2^n}y$ , and  $2^{2^n} \otimes 2^{2^n} = \frac{3}{2}2^{2^n}$ . By the distributive law (iii) it suffices to consider the case  $x = 2^a$  and  $y = 2^b$  for  $0 \leq a, b < 2^n$ . Let  $a = 2^p + a'$  and  $b = 2^q + b'$ , where  $0 \leq a' < 2^p$  and  $0 \leq b' < 2^q$ ; then  $x = 2^{2^p} \otimes 2^{a'}$  and  $y = 2^{2^q} \otimes 2^{b'}$ , by induction on  $n$ .

If  $p < n-1$  and  $q < n-1$  we've already proved that  $x \otimes y < 2^{2^{n-1}}$ . If  $p = q = n-1$ , then  $x \otimes 2^{2^b} < 2^{2^q}$ , hence  $x \otimes y < 2^{2^n}$ . And if  $p = q = n-1$ , we have  $x \otimes y = 2^{2^p} \otimes 2^{2^p} \otimes 2^{a'} \otimes 2^{b'} = (\frac{3}{2}2^{2^p}) \otimes z$ , where  $z < 2^{2^p}$ . Thus  $x \otimes y < 2^{2^n}$  in all cases.

By the cancellation law, the nonnegative integers less than  $2^{2^n}$  form a subfield. Hence in the formula

$$2^{2^n} \otimes y = \text{mex}\{2^{2^n}y' \oplus x'(y \oplus y') \mid 0 \leq x' < 2^{2^n}, 0 \leq y' < y\}$$

we can choose  $x'$  for each  $y'$  to exclude all numbers between  $2^{2^n}y'$  and  $2^{2^n}(y' + 1) - 1$ ; but  $2^{2^n}y$  is never excluded.

Finally in  $2^{2^n} \otimes 2^{2^n} = \text{mex}\{2^{2^n}(x' \oplus y') \oplus (x' \otimes y') \mid 0 \leq x', y' < 2^{2^n}\}$ , choosing  $x' = y'$  will exclude all numbers up to and including  $2^{2^n} - 1$ , since  $x \otimes x = y \otimes y$  implies that  $(x \oplus y) \otimes (x \oplus y) = 0$ , hence  $x = y$ . Choosing  $x' = y' \oplus 1$  excludes numbers from  $2^{2^n}$  to  $\frac{3}{2}2^{2^n} - 1$ , since  $(x \otimes x) \oplus x = (y \otimes y) \oplus y$  implies that  $x = y$  or  $x = y \oplus 1$ , and since the most significant bit of  $x \otimes x$  is the same as that of  $x$ . This same observation shows that  $\frac{3}{2}2^{2^n}$  is *not* excluded. QED.

Consider, for example, the subfield  $\{0, 1, \dots, 15\}$ . By the distributive law we can reduce  $x \otimes y$  to a sum of  $x \otimes 1$ ,  $x \otimes 2$ ,  $x \otimes 4$ , and/or  $x \otimes 8$ . We have  $2 \otimes 2 = 3$ ,  $2 \otimes 4 = 8$ ,  $4 \otimes 4 = 6$ ; and multiplication by 8 can be done by multiplying first by 2 and then by 4 or vice versa, because  $8 = 2 \otimes 4$ . Thus  $2 \otimes 8 = 12$ ,  $4 \otimes 8 = 11$ ,  $8 \otimes 8 = 13$ .

In general, for  $n > 0$ , let  $n = 2^m + r$  where  $0 \leq r < 2^m$ . There is a  $2^{m+1} \times 2^{m+1}$  matrix  $Q_n$  such that multiplication by  $2^n$  is equivalent to applying  $Q_n$  to blocks of  $2^{m+1}$  bits and working mod 2. For example,  $Q_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ , and  $(\dots x_4 x_3 x_2 x_1 x_0)_2 \otimes 2^1 = (\dots y_4 y_3 y_2 y_1 y_0)_2$ , where  $y_0 = x_1$ ,  $y_1 = x_1 \oplus x_0$ ,  $y_2 = x_3$ ,  $y_3 = x_3 \oplus x_2$ ,  $y_4 = x_5$ , etc. The matrices are formed recursively as follows: Let  $Q_0 = R_0 = (1)$  and

$$Q_{2^m+r} = \begin{pmatrix} I & R_m \\ I & 0 \end{pmatrix} \begin{pmatrix} Q_r & 0 \\ 0 & Q_r \end{pmatrix}, \quad R_{m+1} = \begin{pmatrix} R_m & R_m^2 \\ R_m & 0 \end{pmatrix} = Q_{2^{m+1}-1},$$

where  $Q_r$  is replicated enough times to make  $2^{m+1}$  rows and columns. For example,

$$Q_2 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}; \quad Q_3 = Q_2 \begin{pmatrix} Q_1 & 0 \\ 0 & Q_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = R_2.$$

cancellation law  
distributive law  
cancellation law  
distributive law  
recursively

If register  $x$  holds any 64-bit number, and if  $1 \leq j \leq 7$ , the MMIX instruction `MXOR y,qj,x` will compute  $y = x \otimes 2^j$ , given the hexadecimal matrix constants

$$\begin{aligned} q_1 &= \text{c08030200c080302}, & q_4 &= \text{8d4b2c1880402010}, & q_6 &= \text{b9678d4bb0608040}, \\ q_2 &= \text{b06080400b060804}, & q_5 &= \text{c68d342cc0803020}, & q_7 &= \text{deb9c68dd0b0c080}, \\ q_3 &= \text{d0b0c0800d0b0c08}, \end{aligned}$$

[J. H. Conway, *On Numbers and Games* (1976), Chapter 6, shows that these definitions actually yield an algebraically closed field over the ordinal numbers.]

**11.** Let  $m = 2^{a_s} + \cdots + 2^{a_1}$  with  $a_s > \cdots > a_1 \geq 0$  and  $n = 2^{b_t} + \cdots + 2^{b_1}$  with  $b_t > \cdots > b_1 \geq 0$ . Then  $m \otimes n = mn$  if and only if  $(a_s \mid \cdots \mid a_1) \& (b_t \mid \cdots \mid b_1) = 0$ .

**12.** If  $x = 2^{2^n}a + b$  where  $0 \leq a, b < 2^{2^n}$ , let  $x' = x \otimes (x \oplus a)$ . Then

$$x' = ((2^{2^n} \otimes a) \oplus b) \otimes ((2^{2^n} \otimes a) \oplus a \oplus b) = (2^{2^n-1} \otimes a \otimes a) \oplus (b \otimes (a \oplus b)) < 2^{2^n}.$$

To nim-divide by  $x$  we can therefore nim-divide by  $x'$  and multiply by  $x \oplus a$ . [This algorithm is due to H. W. Lenstra, Jr.; see *Séminaire de Théorie des Nombres* (Université de Bordeaux, 1977–1978), exposé 11, exercise 5.]

**13.** If  $a_2 \oplus \cdots \oplus a_k = a_1 \oplus a_3 \oplus \cdots \oplus ((k-2) \otimes a_k) = 0$ , every move breaks this condition; we can't have  $(a \otimes x) \oplus (b \otimes y) = (a \otimes x') \oplus (b \otimes y')$  when  $a \neq b$  unless  $(x, y) = (x', y')$ .

Conversely, if  $a_2 \oplus \cdots \oplus a_k \neq 0$  we can reduce some  $a_j$  with  $j \geq 2$  to make this sum zero; then  $a_1$  can be set to  $a_3 \oplus \cdots \oplus ((k-2) \otimes a_k)$ . If  $a_2 \oplus \cdots \oplus a_k = 0$  and  $a_1 \neq a_3 \oplus \cdots \oplus ((k-2) \otimes a_k)$ , we simply reduce  $a_1$  if it is too large. Otherwise there's a  $j \geq 3$  such that equality will occur if  $(j-2) \otimes a_j$  is replaced by an appropriate smaller value  $((j-2) \otimes a'_j) \oplus ((i-2) \otimes (a_j \oplus a'_j))$ , for some  $2 \leq i < j$  and  $0 \leq a'_j < a_j$ , because of the definition of nim multiplication; hence both of the desired equalities are achieved by setting  $a_j \leftarrow a'_j$  and  $a_i \leftarrow a_i \oplus a_j \oplus a'_j$ . [This game was introduced in *Winning Ways* by Berlekamp, Conway, and Guy, at the end of Chapter 14.]

**14.** (a) Each  $y = (\dots y_2 y_1 y_0)_2 = x^T$  determines  $x = (\dots x_2 x_1 x_0)_2$  uniquely, since  $x_0 = y_0 \oplus t$  and  $\lfloor y/2 \rfloor = \lfloor x/2 \rfloor^{T_{x_0}}$ .

(b) When  $k > 0$ , it is a branching function with labels  $t_{\alpha\alpha\beta} = a$  for  $|\beta| = k-1$ , and  $t_\alpha = 0$  for  $|\alpha| < k$ . But when  $k \leq 0$ , the mapping is not a permutation; in fact, it sends  $2^{-k}$  different 2-adic integers into 0, when  $k < 0$ .

[The case  $k = 1$  is particularly interesting: Then  $x^T$  takes nonnegative integers into nonnegative integers of even parity, negative integers into nonnegative integers of odd parity, and  $-1/3 \mapsto -1$ . Furthermore  $\lfloor x^T/2 \rfloor$  is “Gray binary code,” 7.2.1.1–(g).]

(c) If  $\rho(x \oplus y) = k$  we have  $T(x) \equiv T(y)$  and  $x \equiv y + 2^k$  (modulo  $2^{k+1}$ ). Hence  $\rho(x^T \oplus y^T) = \rho(x \oplus y \oplus T(x) \oplus T(y)) = k$ . Conversely, if  $\rho(x^T \oplus y^T) = k$  whenever  $y = x + 2^k$ , we obtain a suitable bit labeling by letting  $t_\alpha = (x^T \gg |\alpha|) \bmod 2$  when  $x = (\alpha^R)_2$ .

(d) This statement follows immediately from (a) and (c). For if we always have  $\rho(x \oplus y) = \rho(x^U \oplus y^U) = \rho(x^V \oplus y^V)$ , then  $\rho(x \oplus y) = \rho(x^U \oplus y^U) = \rho(x^{UV} \oplus y^{UV})$ . And if  $x^{TU} = x$  for all  $x$ ,  $\rho(x^U \oplus y^U) = \rho(x \oplus y)$  is equivalent to  $\rho(x \oplus y) = \rho(x^T \oplus y^T)$ .

We can also construct the labelings explicitly: If  $W = UV$ , note that when  $a, b, c \in \{0, 1\}$  we have  $W_a = U_a V_{a'}$ ,  $W_{ab} = U_{ab} V_{a'b'}$ , and  $W_{abc} = U_{abc} V_{a'b'c'}$ , where  $a' = a \oplus u$ ,  $b' = b \oplus u_a$ ,  $c' = c \oplus u_{ab}$ , and so on; hence  $w = u \oplus v$ ,  $w_a = u_a \oplus v_{a'}$ ,  $w_{ab} = u_{ab} \oplus v_{a'b'}$ , etc. The labeling  $T$  inverse to  $U$  is obtained by swapping left and right subtrees of all nodes labeled 1; thus  $t = u$ ,  $t_{a'} = u_a$ ,  $t_{a'b'} = u_{ab}$ , etc.

(e) The explicit constructions in (d) demonstrate that the balance condition is preserved by compositions and inverses, because  $\{0', 1'\} = \{0, 1\}$  at each level.

*Notes:* Hendrik Lenstra observes that branching functions can profitably be viewed as the *isometries* (distance-preserving permutations) of the 2-adic integers, when we

MMIX  
MXOR  
Conway  
ordinal numbers  
Lenstra  
nim multiplication  
Berlekamp  
Conway  
Guy  
parity  
Gray binary code  
Lenstra  
isometries

use the formula  $1/2^{\rho(x \oplus y)}$  to define the “distance” between 2-adic integers  $x$  and  $y$ . Moreover, the branching functions mod  $2^d$  turn out to be the Sylow 2-subgroup of the group of all permutations of  $\{0, 1, \dots, 2^d - 1\}$ , namely the unique (up to isomorphism) subgroup that has maximum power-of-2 order among all subgroups of that group. They also are equivalent to the automorphisms of the complete binary tree with  $2^d$  leaves.

**15.** Equivalently,  $(x + 2a) \oplus b = (x \oplus b) + 2a$ ; so we might as well find all  $b$  and  $c$  such that  $(x \oplus b) + c = (x + c) \oplus b$ . Setting  $x = 0$  and  $x = -c$  implies that  $b + c = b \oplus c$  and  $b - c = b \oplus (-c)$ ; hence  $b \& c = b \& (-c) = 0$  by (8g), and we have  $b < 2^{\rho c}$ . This condition is also sufficient. Thus  $0 \leq b < 2^{\rho a + 1}$  is necessary and sufficient for the original problem.

**16.** (a) If  $\rho(x \oplus y) = k$  we have  $x \equiv y + 2^k$  (modulo  $2^{k+1}$ ); hence  $x + a \equiv y + a + 2^k$  and  $\rho((x + a) \oplus (y + a)) = k$ . And  $\rho((x \oplus b) \oplus (y \oplus b))$  is obviously  $k$ .

(b) The hinted labeling, call it  $P(c)$ , has 1s on the path corresponding to  $c$ , and 0s elsewhere; thus it is balanced. The general animating function can be written

$$x^{P(c_0) - a_1 P(c_1) - a_2 \dots P(c_{m-1}) - a_m} \oplus c_m, \quad \text{where } c_j = b_1 \oplus \dots \oplus b_j;$$

so it is balanced if and only if  $c_m = 0$ .

[Incidentally, the set  $S = \{P(0)\} \cup \{P(k) \oplus P(k + 2^e) \mid k \geq 0 \text{ and } 2^e > k\}$  provides an interesting *basis* for all possible balanced labelings: A labeling is balanced if and only if it is  $\bigoplus \{q \mid q \in Q\}$  for some  $Q \subseteq S$ . This exclusive-or operation is well defined even though  $Q$  might be infinite, because only finitely many 1s appear at each node.]

(c) The function  $P(c)$  in (b) has this form, because  $x^{P(c)} = x \oplus \lfloor x \oplus c \rfloor$ . Its inverse,  $x^{S(c)} = ((x \oplus c) + 1) \oplus c$ , is  $x \oplus \lfloor x \oplus \bar{c} \rfloor = x^{P(\bar{c})}$ . Furthermore we have  $x^{P(c)P(d)} = x^{P(c)} \oplus \lfloor x^{P(c)} \oplus d \rfloor = x \oplus \lfloor x \oplus c \rfloor \oplus \lfloor x \oplus d^{S(c)} \rfloor$ , because  $\lfloor x \oplus y \rfloor = \lfloor x^T \oplus y^T \rfloor$  for any branching function  $x^T$ . Similarly  $x^{P(c)P(d)P(e)} = x \oplus \lfloor x \oplus c \rfloor \oplus \lfloor x \oplus d^{S(c)} \rfloor \oplus \lfloor x \oplus e^{S(d)S(c)} \rfloor$ , etc. After discarding equal terms we obtain the desired form. The resulting numbers  $p_j$  are unique because they are the only values of  $x$  at which the function changes sign.

(d) We have, for example,  $x \oplus \lfloor x \oplus a \rfloor \oplus \lfloor x \oplus b \rfloor \oplus \lfloor x \oplus c \rfloor = x^{P(a')P(b')P(c')}$  where  $a' = a$ ,  $b' = b^{P(a')}$ , and  $c' = c^{P(a')P(b')}$ .

[The theory of animating functions was developed by J. H. Conway in Chapter 13 of his book *On Numbers and Games* (1976), inspired by previous work of C. P. Welter in *Indagationes Math.* **14** (1952), 304–314; **16** (1954), 194–200.]

**17.** (Solution by M. Slanina.) Such equations are decidable even if we also allow operations such as  $x \& y$ ,  $\bar{x}$ ,  $x \ll 1$ ,  $x \gg 1$ ,  $2^{\rho x}$ , and  $2^{\lambda x}$ , and even if we allow Boolean combinations of statements and quantifications over integer variables, by translating them into formulas of second-order monadic logic with one successor (S1S). Each 2-adic variable  $x = (\dots x_2 x_1 x_0)_2$  corresponds to an S1S set variable  $X$ , where  $j \in X$  means  $x_j = 1$ :

$$\begin{aligned} z = \bar{x} & \text{ becomes } \forall t(t \in Z \Leftrightarrow t \notin X); \\ z = x \& y & \text{ becomes } \forall t(t \in Z \Leftrightarrow (t \in X \wedge t \in Y)); \\ z = 2^{\rho x} & \text{ becomes } \forall t(t \in Z \Leftrightarrow (t \in X \wedge \forall s(s < t \Rightarrow s \notin X))); \\ z = x + y & \text{ becomes } \exists C \forall t(0 \notin C \wedge (t \in Z \Leftrightarrow (t \in X) \oplus (t \in Y) \oplus (t \in C))) \\ & \quad \wedge (t+1 \in C \Leftrightarrow ((t \in X)(t \in Y)(t \in C)))). \end{aligned}$$

An identity such as  $x \& (-x) = 2^{\rho x}$  is equivalent to the translation of

$$\forall X \forall Y \forall Z ((\text{integer}(X) \wedge 0 = x + y \wedge z = x \& y) \Rightarrow z = 2^{\rho x}),$$

where  $\text{integer}(X)$  stands for  $\exists t \forall s(s > t \Rightarrow (s \in X \Leftrightarrow t \in X))$ . We can also include 2-adic constants if they are, say, ratios of integers; for example,  $z = \mu_0$  is equivalent to

distance  
2-adic integers as a metric space  
Sylow 2-subgroup  
symmetric group  
complete binary tree  
infinite exclusive-or operation  
Conway  
Welter  
Slanina  
quantifications  
second-order monadic logic with one successor  
S1S  
magic

the formula  $0 \in Z \wedge \forall t(t \in Z \Leftrightarrow t+1 \notin Z)$ . But of course we cannot include arbitrary (uncomputable) constants.

J. R. Büchi proved that all formulas of S1S are decidable, in *Logic, Methodology, and Philosophy of Science: Proceedings* (Stanford, 1960), 1–11. If we restrict attention to equations, one can show in fact that exponential time suffices.

On the other hand M. Hamburg has shown that the problem would be unsolvable if  $\rho x$ ,  $\lambda x$ , or  $1 \ll x$  were added to the repertoire; multiplication could then be encoded.

Incidentally, many nontrivial identities exist, even if we use only the operations  $x \oplus y$  and  $x + 1$ . For example, C. P. Welter noticed in 1952 that

$$((x \oplus (y + 1)) + 1) \oplus (x + 1) = (((x + 1) \oplus y) + 1) \oplus x + 1.$$

**18.** Of course row  $x$  is entirely blank when  $x$  is a multiple of 64. The fine details of this image are apparently “chaotic” and complex, but there is a fairly easy way to understand what happens near the points where the straight lines  $x = 64\sqrt{j}$  intersect the hyperbolas  $xy = 2^{11}k$ , for integers  $j, k \geq 1$  that aren’t too large.

Indeed, when  $x$  and  $y$  are integers, the value of  $x^2y \gg 11$  is odd if and only if  $x^2y/2^{12} \bmod 1 \geq \frac{1}{2}$ . Thus, if  $x = 64\sqrt{j} + \delta$  and  $xy = 2^{11}(k + \epsilon)$  we have

$$\frac{x^2y}{2^{12}} \bmod 1 = \left(\frac{128\sqrt{j}\delta + \delta^2}{4096}\right)y \bmod 1 = \left(\frac{2\delta x - \delta^2}{4096}\right)y \bmod 1 = \left((k + \epsilon)\delta - \frac{\delta^2y}{4096}\right) \bmod 1,$$

and this quantity has a known relation to  $\frac{1}{2}$  when, say,  $\delta$  is close to a small integer. [See C. A. Pickover and A. Lakhtakia, *J. Recreational Math.* **21** (1989), 166–169.]

**19.** (a) When  $n = 1$ ,  $f(A, B, C)$  has the same value under all arrangements except when  $a_0 \neq a_1$ ,  $b_0 \neq b_1$ , and  $c_0 \neq c_1$ ; and then it cannot exceed 1. For larger values of  $n$  we argue by induction, assuming that  $n = 3$  in order to avoid cumbersome notation. Let  $A_0 = (a_0, a_1, a_2, a_3)$ ,  $A_1 = (a_4, a_5, a_6, a_7)$ ,  $\dots$ ,  $C_1 = (c_4, c_5, c_6, c_7)$ . Then  $f(A, B, C) = \sum_{j \oplus k \oplus l = 0} f(A_j, B_k, C_l) \leq \sum_{j \oplus k \oplus l = 0} f(A_j^*, B_k^*, C_l^*)$  by induction. Thus we can assume that  $a_0 \geq a_1 \geq a_2 \geq a_3$ ,  $a_4 \geq a_5 \geq a_6 \geq a_7$ ,  $\dots$ ,  $c_4 \geq c_5 \geq c_6 \geq c_7$ . We can also sort the subvectors  $A'_0 = (a_0, a_1, a_4, a_5)$ ,  $A'_1 = (a_2, a_3, a_6, a_7)$ ,  $\dots$ ,  $C'_1 = (c_2, c_3, c_6, c_7)$  in a similar way. Finally, we can sort  $A''_0 = (a_0, a_1, a_6, a_7)$ ,  $A''_1 = (a_2, a_3, a_4, a_5)$ ,  $\dots$ ,  $C''_1 = (c_2, c_3, c_4, c_5)$ , because in each term  $a_j b_k c_l$  the number of subscripts  $\{j, k, l\}$  with leading bits 01, 10, and 11 must satisfy  $s_{01} \equiv s_{10} \equiv s_{11} \pmod{2}$ . And these three sorting operations leave  $A, B, C$  fully sorted, by exercise 5.3.4–48. (Exactly three sorts on subvectors of length  $2^{n-1}$  are needed, for all  $n \geq 2$ .)

(b) Suppose  $A = A^*$ ,  $B = B^*$ , and  $C = C^*$ . Then we have  $a_j = \sum_{t=0}^{2^n-1} \alpha_t[j \leq t]$ , where  $\alpha_j = a_j - a_{j+1} \geq 0$  and we set  $a_{2^n} = 0$ ; similar formulas hold for  $b_k$  and  $c_l$ . Let  $A_{(p)}$  denote the vector  $(a_{p(0)}, \dots, a_{p(2^n-1)})$  when  $p$  is a permutation of  $\{0, 1, \dots, 2^n-1\}$ . Then by part (a) we have

$$\begin{aligned} f(A_{(p)}, B_{(q)}, C_{(r)}) &= \sum_{j \oplus k \oplus l = 0} \sum_{t, u, v} \alpha_t \beta_u \gamma_v [p(j) \leq t][q(k) \leq u][r(l) \leq v] \\ &\leq \sum_{j \oplus k \oplus l = 0} \sum_{t, u, v} \alpha_t \beta_u \gamma_v [j \leq t][k \leq u][l \leq v] = f(A, B, C). \end{aligned}$$

[This proof is due to Hardy, Littlewood, and Pólya, *Inequalities* (1934), §10.3.]

(c) The same proof technique extends to any number of vectors. [R. E. A. C. Paley, *Proc. London Math. Soc.* (2) **34** (1932), 263–279, Theorem 15.]

**20.** The given steps compute the least integer  $y$  greater than  $x$  such that  $\nu y = \nu x$ . They’re useful for generating all combinations of  $n$  objects, taken  $m$  at a time (that is, all  $m$ -element subsets of an  $n$ -element set, with elements represented by 1 bits).

[This tidbit is Hack 175 in HAKMEM, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo No. 239 (29 February 1972).]

Büchi  
Hamburg  
unsolvable  
Welter  
hyperbolas  
Pickover  
Lakhtakia  
sorting  
Hardy  
Littlewood  
Pólya  
Paley  
HAKMEM  
combinations+  
set repr



21. Set  $t \leftarrow y + 1$ ,  $u \leftarrow t \oplus y$ ,  $v \leftarrow t \& y$ ,  $x \leftarrow v - (v \& -v)/(u + 1)$ . If  $y = 2^m - 1$  is the *first*  $m$ -combination, these eight operations set  $x$  to zero. (The fact that  $x = \overline{f(\overline{y})}$  does not seem to yield any shorter scheme.)

22. Sideways addition avoids the division: SUBU t,x,1; ANDN u,x,t; SADD k,t,x; ADDU v,x,u; XOR t,v,x; ADDU k,k,2; SRU t,t,k; ADDU y,v,t. But we can actually save a step by judiciously using the constant `mone` = -1: SUBU t,x,1; XOR u,t,x; ADDU y,x,u; SADD k,t,y; ANDN y,y,u; SLU t,mone,k; ORN y,y,t.

23. (a)  $(0 \dots 01 \dots 1)_2 = 2^m - 1$  and  $(0101 \dots 01)_2 = (2^{2m} - 1)/3$ .

(b) This solution uses the 2-adic constant  $\mu_0 = (\dots 010101)_2 = -1/3$ :

$$t \leftarrow x \oplus \mu_0, \quad u \leftarrow (t-1) \oplus t, \quad v \leftarrow x \mid u, \quad w \leftarrow v + 1, \quad y \leftarrow w + \left\lfloor \frac{v \& \overline{w}}{\sqrt{u+1}} \right\rfloor.$$

If  $x = (2^{2m} - 1)/3$ , the operations produce a strange result because  $u = 2^{2m+1} - 1$ .

(c) XOR t,x,m0; SUBU u,t,1; XOR u,t,u; OR v,x,u; SADD y,u,m0; ADDU w,v,1; ANDN t,v,w; SRU y,t,y; ADDU y,w,y. [This exercise was inspired by Jörg Arndt.]

24. It's expedient to "prime the pump" by initializing the array to the state that it should have after all multiples of 3, 5, 7, and 11 have been sieved out. We can combine 3 with 11 and 5 with 7, as suggested by E. Wada:

```

LOC Data_Segment
qbase GREG @ ;N IS 3584 ;n GREG N ;one GREG 1
Q      OCTA #816d129a64b4cb6e          Q0 (little-endian)
      LOC Q+N/16

qtop GREG @                          End of the Q table
Init  OCTA #9249249249249249|#4008010020040080 Multiples of 3 or 11 in [129..255]
      OCTA #8421084210842108|#0408102040810204 Multiples of 5 or 7
t IS $255 ;x33 IS $0 ;x35 IS $1 ;j IS $4
      LOC #100
Main  LDOU x33,Init; LDOU x35,Init+8
      LDA j,qbase,8; SUB j,j,qtop          Prepare to set Q1.
1H    NOR t,x33,x33; ANDN t,t,x35; STOU t,qtop,j Initialize 64 sieve bits.
      SLU t,x33,2; SRU x33,x33,31; OR x33,x33,t Prepare for the next 64 values.
      SLU t,x35,6; SRU x35,x35,29; OR x35,x35,t
      ADD j,j,8; PBN j,1B                  Repeat until reaching qtop. ■

```

Then we cast out nonprimes  $p^2$ ,  $p^2 + 2p$ , ..., for  $p = 13, 17, \dots$ , until  $p^2 > N$ :

```

p IS $0 ;pp IS $1 ;m IS $2 ;mm IS $3 ;q IS $4 ;s IS $5
      LDOU q,qbase,0; LDA pp,qbase,8
      SET p,13; NEG m,13*13,n; SRU q,q,6      Begin with p = 13.
1H    SR m,m,1                               m ← ⌊(p2 - N)/2⌋.
2H    SR mm,m,3; LDOU s,qtop,mm; AND t,m,#3f;
      SLU t,one,t; ANDN s,s,t; STOU s,qtop,mm Zero out a bit.
      ADD m,m,p; PBN m,2B                     Advance by p bits.
      SRU q,q,1; PBNZ q,3F                     Move to next potential prime.
2H    LDOU q,pp,0; INCL pp,8                   Read in another batch
      OR p,p,#7f; PBNZ q,3F                     of potential primes.
      ADD p,p,2; JMP 2B                         Skip past 128 nonprimes.
2H    SRU q,q,1
3H    ADD p,p,2; PBEV q,2B                     Set p ← p + 2 until p is prime.
      MUL m,p,p; SUB m,m,n; PBN m,1B          Repeat until p2 > N. ■

```

mone  
magic mask  
SADD  
Arndt  
Wada  
NEG  
little-endian

The running time,  $1172\mu + 5166\nu$ , is of course much less than the time needed for steps P1–P8 of Program 1.3.2'P, namely  $10037\mu + 641543\nu$  (improved to  $10096\mu + 215351\nu$  in exercise 1.3.2'–14). [See P. Pritchard, *Science of Computer Programming* **9** (1987), 17–35, for several instructive variations. In practice, a program like this one tends to slow down dramatically when the sieve is too big for the computer's cache. Better results are obtained by working with a segmented sieve, which contains bits for numbers between  $N_0 + k\delta$  and  $N_0 + (k+1)\delta$ , as suggested by L. J. Lander and T. R. Parkin, *Math. Comp.* **21** (1967), 483–488; C. Bays and R. H. Hudson, *BIT* **17** (1977), 121–127. Here  $N_0$  can be quite large, but  $\delta$  is limited by the cache size; calculations are done separately for  $k = 0, 1, \dots$ . Segmented sieves have become highly developed; see, for example, T. R. Nicely, *Math. Comp.* **68** (1999), 1311–1315, and the references cited there. The author used such a program in 2006 to discover an unusually large gap of length 1370 between 418032645936712127 and the next larger prime.]

**25.**  $(1 + 1 + 25 + 1 + 1 + 25 + 1 + 1 = 56)$  mm; the worm never sees pages 2–500 of Volume 1 or 1–499 of Volume 4. (Unless the books have been placed in little-endian fashion on the bookshelf; then the answer would be 106 mm.) This classic brain-teaser can be found in Sam Loyd's *Cyclopedia* (New York: 1914), pages 327 and 383.

**26.** We could multiply by `#aa...ab` instead of dividing by 12 (see exercise 1.3.1'–17); but multiplication is slow too. Or we could deal with a “flat” sequence of  $12000000 \times 5$  consecutive bits (= 7.5 megabytes), ignoring the boundaries between words. Another possibility is to use a scheme that is neither big-endian nor little-endian but *transposed*: Put item  $k$  into octabyte  $8(k \bmod 2^{20})$ , where it is shifted left by  $5\lfloor k/2^{20} \rfloor$ . Since  $k < 12000000$ , the amount of shift is always less than 60. The MMIX code to put item  $k$  into register \$1 is `AND $0,k,[#fffff]; SLU $0,$0,3; LDOU $1,base,$0; SRU $0,k,20; 4ADDU $0,$0,$0; SRU $1,$1,$0; AND $1,$1,#1f`.

[This solution uses 8 large megabytes ( $2^{23}$  bytes). Any convenient scheme for converting item numbers to octabyte addresses and shift amounts will work, as long as the same method is used consistently. Of course, just `LDBU $1,base,k` would be faster.]

**27.** (a)  $((x-1) \oplus x) + x$ . [This exercise is based on an idea of Luther Woodrum, who noticed that  $((x-1) \mid x) + 1 = (x \& -x) + x$ .]

(b)  $(y + x) \mid y$ , where  $y = (x-1) \oplus x$ .

(c,d,e)  $((z \oplus x) + x) \& z$ ,  $((z \oplus x) + x) \oplus z$ , and  $\overline{((z \oplus x) + x)} \& z$ , where  $z = x-1$ .

(f)  $x \oplus (a)$ ; alternatively,  $t \oplus (t+1)$ , where  $t = x \mid (x-1)$ . [The number  $(0^\infty 01^a 11^b)_2$  looks simpler, but it apparently requires *five* operations:  $((t+1) \& \bar{t}) - 1$ .]

These constructions all give sensible results in the exceptional cases when  $x = -2^b$ .

**28.** A 1 bit indicates  $x$ 's rightmost 0 (for example,  $(101011)_2 \mapsto (000100)_2$ );  $-1 \mapsto 0$ .

**29.**  $\mu_k = \mu_{k+1} \oplus (\mu_{k+1} \ll 2^k)$  [see *STOC* **6** (1974), 125]. This relation holds also for the constants  $\mu_{d,k}$  of (48), when  $0 \leq k < d$ , if we start with  $\mu_{d,d} = 2^{2^d} - 1$ . (There is, however, no easy way to go from  $\mu_k$  to  $\mu_{k+1}$ , unless we use the “zip” operation; see (77).)

**30.** Append `'CSZ rho,x,64'` to (50), thereby adding  $1\nu$  to its execution time; or replace the last two lines by `SRU t,y,rho; SLU t,t,2; SRU t,[#300020104],t; AND t,t,#f; ADD rho,rho,t`, saving  $1\nu$ . For (51), we simply need to make sure that `rhatab[0] = 8`.

**31.** In the first place, his code loops forever when  $x = 0$ . But even after that bug is patched, his assumption that  $x$  is a random integer is highly questionable. In many applications when we want to compute  $\rho x$  for a nonzero 64-bit number  $x$ , a more reasonable assumption would be that each of the outcomes  $\{0, 1, \dots, 63\}$  is equally likely. The average and standard deviation then become 31.5 and  $\approx 18.5$ .

Pritchard  
cache  
segmented sieve  
Lander  
Parkin  
Bays  
Hudson  
Nicely  
gap  
Knut  
little-endian  
Loyd  
big-endian  
little-endian  
transposed  
large megabytes  
Woodrum  
zip  
CSZ  
table lookup via shifting

**32.** ‘`NEGU y,x; AND y,x,y; MULU y,debruijn,y; SRU y,y,58; LDB rho,decode,y`’ has estimated cost  $\mu + 14\nu$ , although multiplication by a power of 2 might well be faster than a typical multiplication. Add  $1\nu$  for the correction in answer 30.

**33.** In fact, an exhaustive calculation shows that exactly 94727 suitable constants  $a$  yield a “perfect hash function” for this problem, 90970 of which also identify the power-of-two cases  $y = 2^j$ ; 90918 of those also distinguish the case  $y = 0$ . The multiplier #208b2430c8c82129 is uniquely best, in the sense that it doesn’t need to refer to table entries above *decode*[32400] when  $y$  is known to be a valid input.

**34.** Identity (a) fails when  $x = 5$ ,  $y = 6$ ; but (b) is true, also when  $xy = 0$ . Proof of (c): If  $x \neq y$  and  $\rho x = \rho y = k$  we have  $x = \alpha 10^k$  and  $y = \beta 10^k$ ; hence  $x \oplus y = (\alpha \oplus \beta) 00^k = (x-1) \oplus (y-1)$ . If  $\rho x > \rho y = k$  we have  $(x \oplus y) \bmod 2^{k+2} \neq ((x-1) \oplus (y-1)) \bmod 2^{k+2}$ .

**35.** Let  $f(x) = x \oplus 3x$ . Clearly  $f(2x) = 2f(x)$ , and  $f(4x+1) = 4f(x) + 2$ . We also have  $f(4x-1) = 4f(x) + 2$ , by exercise 34(c). The hinted identity follows.

Given  $n$ , set  $u \leftarrow n \gg 1$ ,  $v \leftarrow u + n$ ,  $t \leftarrow u \oplus v$ ,  $n^+ \leftarrow v \& t$ , and  $n^- \leftarrow u \& t$ . Clearly  $u = \lfloor n/2 \rfloor$  and  $v = \lfloor 3n/2 \rfloor$ , so  $n^+ - n^- = v - u = n$ . And this is Reitwiesner’s representation, because  $n^+ | n^-$  has no consecutive 1s. [H. Prodinger, *Integers* 0 (2000), paper a8, 14 pp. Incidentally we also have  $f(-x) = f(x)$ .]

**36.** (i) The commands  $x \leftarrow x \oplus (x \ll 1)$ ,  $x \leftarrow x \oplus (x \ll 2)$ ,  $x \leftarrow x \oplus (x \ll 4)$ ,  $x \leftarrow x \oplus (x \ll 8)$ ,  $x \leftarrow x \oplus (x \ll 16)$ ,  $x \leftarrow x \oplus (x \ll 32)$  change  $x$  to  $x^\oplus$ . (ii)  $x^\& = x \& \sim(x+1)$ .

(See exercises 66, 70, and 117 for applications of  $x^\oplus$ ; see also exercise 209.)

**37.** Insert ‘`CSZ y,x,half`’ after the `FLOTU` in (55), where `half` = #3fe0000000000000; note that (55) says ‘`SR`’ (not ‘`SRU`’). No change is needed to (56), if *lamtab*[0] = -1.

**38.** ‘`SRU t,x,1; OR y,x,t; SRU t,y,2; OR y,y,t; SRU t,y,4; OR y,y,t; ...; SRU t,y,32; OR y,y,t; SRU y,y,1; CMPU t,x,0; ADDU y,y,t`’ takes  $15\nu$ .

**39.** (Solution by H. S. Warren, Jr.) Let  $\sigma(x)$  denote the result of smearing  $x$  to the right, as in the first line of (57). Compute  $x \& \sigma((x \gg 1) \& \bar{x})$ .

**40.** Suppose  $\lambda x = \lambda y = k$ . If  $x = y = 0$ , (58) certainly holds, regardless of how we define  $\lambda 0$ . Otherwise  $x = (1\alpha)_2$  and  $y = (1\beta)_2$ , for some binary strings  $\alpha$  and  $\beta$  with  $|\alpha| = |\beta| = k$ ; and  $x \oplus y < 2^k \leq x \& y$ . On the other hand if  $\lambda x < \lambda y = k$ , we have  $x \oplus y \geq 2^k > x \& y$ . And H. S. Warren, Jr., notes that  $\lambda x < \lambda y$  if and only if  $x < y \& \bar{x}$ .

**41.** (a)  $\sum_{n=1}^{\infty} (\rho n) z^n = \sum_{k=1}^{\infty} z^{2^k} / (1 - z^{2^k}) = z / (1 - z) - \sum_{k=0}^{\infty} z^{2^k} / (1 + z^{2^k})$ . The Dirichlet generating function is simpler:  $\sum_{n=1}^{\infty} (\rho n) / n^z = \zeta(z) / (2^z - 1)$ .

(b)  $\sum_{n=1}^{\infty} (\lambda n) z^n = \sum_{k=1}^{\infty} z^{2^k} / (1 - z)$ .

(c)  $\sum_{n=1}^{\infty} (\nu n) z^n = \sum_{k=0}^{\infty} z^{2^k} / ((1 - z)(1 + z^{2^k})) = \sum_{k=0}^{\infty} z^{2^k} \mu_k(z)$ , where  $\mu_k(z) = (1 + z + \cdots + z^{2^k-1}) / (1 - z^{2^{k+1}})$ . (The “magic masks” of (47) correspond to  $\mu_k(2)$ .)

[See *Automatic Sequences* by J.-P. Allouche and J. Shallit (2003), Chapter 3, for further information about the functions  $\rho$  and  $\nu$ , which they denote by  $\nu_2$  and  $s_2$ .]

**42.**  $e_1 2^{e_1-1} + (e_2 + 2) 2^{e_2-1} + \cdots + (e_r + 2r - 2) 2^{e_r-1}$ , by induction on  $r$ . [D. E. Knuth, *Proc. IFIP Congress* (1971), 1, 19–27. The fractal aspects of this sum are illustrated in Figs. 3.1 and 3.2 of the book by Allouche and Shallit.]

**43.** The straightforward implementation of (63), ‘`SET nu,0; SET y,x; BZ y,Done; 1H ADD nu,nu,1; SUBU t,y,1; AND y,y,t; PBNZ y,1B`’ costs  $(5 + 4\nu x)\nu$ ; it beats the implementation of (62) when  $\nu x < 4$ , ties when  $\nu x = 4$ , and loses when  $\nu x > 4$ .

But we can save  $4\nu$  from the implementation of (62) if we replace the final multiplication-and-shift by ‘ $y \leftarrow y + (y \gg 8)$ ,  $y \leftarrow y + (y \gg 16)$ ,  $y \leftarrow y + (y \gg 32)$ ,  $\nu \leftarrow y \& \#ff$ ’. [Of course, *MMIX*’s single instruction ‘`SADD nu,x,0`’ is much better.]

perfect hash function  
Prodinger  
signed right shift  
CSZ  
Warren  
smearing  
Warren  
Dirichlet generating function  
zeta function  
magic masks  
Allouche  
Shallit  
 $\rho$   
 $\nu$   
Knuth  
fractal

44. Let this sum be  $\nu^{(2)}x$ . If we can solve the problem for  $2^d$ -bit numbers, we can solve it for  $2^{d+1}$ -bit numbers, because  $\nu^{(2)}(2^{2^d}x + x') = \nu^{(2)}x + \nu^{(2)}x' + 2^d\nu x$ . Therefore a solution analogous to (62) suggests itself, on a 64-bit machine:

Set  $z \leftarrow (x \gg 1) \& \mu_0$  and  $y \leftarrow x - z$ .  
 Set  $z \leftarrow ((z + (z \gg 2)) \& \mu_1) + ((y \& \bar{\mu}_1) \gg 1)$  and  $y \leftarrow (y \& \mu_1) + ((y \gg 2) \& \mu_1)$ .  
 Set  $z \leftarrow ((z + (z \gg 4)) \& \mu_2) + ((y \& \bar{\mu}_2) \gg 2)$  and  $y \leftarrow (y + (y \gg 4)) \& \mu_2$ .  
 Finally  $\nu^{(2)} \leftarrow (((Az) \bmod 2^{64}) \gg 56) + (((By) \bmod 2^{64}) \gg 56) \ll 3$ ,  
 where  $A = (11111111)_{256}$  and  $B = (01234567)_{256}$ .

But another approach is better on MMIX, which has sideways addition built in:

```
SADD nu2,x,m0      SADD t,x,m2      8ADDU nu2,t,nu2      SADD t,x,m5
SADD t,x,m1        4ADDU nu2,t,nu2    SADD t,x,m4        SLU t,t,5
2ADDU nu2,t,nu2     SADD t,x,m3       16ADDU nu2,t,nu2     ADD nu2,nu2,t █
```

[In general,  $\nu^{(2)}x = \sum_k 2^k \nu(x \& \bar{\mu}_k)$ . See *Dr. Dobbs's Journal* 8,4 (April 1983), 24–37.]

45. Let  $d = (x - y) \& (y - x)$ ; test if  $d \& y \neq 0$ . [Rokicki found that this idea, which is called *colex ordering*, can be used with node addresses to near-randomize binary search trees or Cartesian trees as if they were treaps, without needing an additional random “priority key” in each node. See *U.S. Patent 6347318* (12 February 2002).]

46. `SADD t,x,m; NXOR y,x,m; CSOD x,t,y`; the mask  $m$  is  $\sim(1 \ll i \mid 1 \ll j)$ . (In general, these instructions complement the bits specified by  $\bar{m}$  if those bits have odd parity.)

47.  $y \leftarrow (x \gg \delta) \& \theta$ ,  $z \leftarrow (x \& \theta) \ll \delta$ ,  $x \leftarrow (x \& m) \mid y \mid z$ , where  $\bar{m} = \theta \mid (\theta \ll \delta)$ .

48. Given  $\delta$ , there are  $s_\delta = \prod_{j=0}^{\delta-1} F_{\lfloor (n+j)/\delta \rfloor + 1}$  different  $\delta$ -swaps, including the identity permutation. (See exercise 4.5.3–32.) Summing over  $\delta$  gives  $1 + \sum_{\delta=1}^{n-1} (s_\delta - 1)$  altogether.

49. (a) The set  $S = \{a_1\delta_1 + \dots + a_m\delta_m \mid \{a_1, \dots, a_m\} \subseteq \{-1, 0, +1\}\}$  for displacements  $\delta_1, \dots, \delta_m$  must contain  $\{n-1, n-3, \dots, 1-n\}$ , because the  $k$ th bit must be exchanged with the  $(n+1-k)$ th bit for  $1 \leq k \leq n$ . Hence  $|S| \geq n$ . And  $S$  contains at most  $3^m$  numbers, at most  $2 \cdot 3^{m-1}$  of which are odd.

(b) Clearly  $s(mn) \leq s(m) + s(n)$ , because we can reverse  $m$  fields of  $n$  bits each. Thus  $s(3^m) \leq m$  and  $s(2 \cdot 3^m) \leq m + 1$ . Furthermore the reversal of  $3^m$  bits uses only  $\delta$ -swaps with even values of  $\delta$ ; the corresponding  $(\delta/2)$ -swaps prove that we have  $s((3^m \pm 1)/2) \leq m$ . These upper bounds match the lower bounds of (a) when  $m > 1$ .

(c) The string  $\alpha\alpha\beta\theta\psi z\omega$  with  $|\alpha| = |\beta| = |\theta| = |\psi| = |\omega| = n$  can be changed to  $\omega z\psi\theta\beta\alpha\alpha$  with a  $(3n+1)$ -swap followed by an  $(n+1)$ -swap. Then  $s(n)$  further swaps reverse all. Hence  $s(32) \leq s(6) + 2 = 4$ , and  $s(64) \leq 5$ . Again, equality holds by (a).

Incidentally,  $s(63) = 4$  because  $s(7) = s(9) = 2$ . The lower bound in (a) turns out to be the exact value of  $s(n)$  for  $1 \leq n \leq 22$ , except that  $s(16) = 4$ .

50. Express  $n = (t_m \dots t_1 t_0)_3$  in balanced ternary notation. Let  $n_j = (t_m \dots t_j)_3$  and  $\delta_j = 2n_j + t_{j-1}$ , so that  $n_{j-1} - \delta_j = n_j$  and  $2\delta_j - n_{j-1} = n_j + t_{j-1}$  for  $1 \leq j \leq m$ . Let  $E_0 = \{0\}$  and  $E_{j+1} = E_j \cup \{t_j - x \mid x \in E_j\}$  for  $0 \leq j < m$ . (Thus, for example,  $E_1 = \{0, t_0\}$  and  $E_2 = \{0, t_0, t_1, t_1 - t_0\}$ .) Notice that  $\varepsilon \in E_j$  implies  $|\varepsilon| \leq j$ .

Assume by induction on  $j$  that  $\delta$ -swaps for  $\delta = \delta_1, \dots, \delta_j$  have changed the  $n$ -bit word  $\alpha_1 \dots \alpha_{3j}$  to  $\alpha_{3j} \dots \alpha_1$ , where each subword  $\alpha_k$  has length  $n_j + \varepsilon_k$  for some  $\varepsilon_k \in E_j$ . If  $n_{j+1} > j$ , a  $\delta_{j+1}$ -swap within each subword will preserve this assumption. Otherwise each subword  $\alpha_k$  has  $|\alpha_k| \leq n_j + j \leq 3n_{j+1} + 1 + j \leq 4j + 1 < 4m$ . Therefore  $2^k$ -swaps for  $\lfloor \lg 4m \rfloor \geq k \geq 0$  will reverse them all. (Note that a  $2^k$ -swap on a subword of size  $t$ , where  $2^k < t \leq 2^{k+1}$ , reduces it to three subwords of sizes  $t - 2^k$ ,  $2^{k+1} - t$ ,  $t - 2^k$ .)

MMIX  
 sideways addition  
 SADD  
 2ADDU  
 4ADDU  
 8ADDU  
 16ADDU  
 Rokicki  
 colex ordering  
 randomized data structures  
 binary search trees  
 Cartesian trees  
 treaps  
 Patent  
 parity  
 SADD  
 NXOR  
 CSOD  
 balanced ternary notation

**51.** (a) If  $c = (c_{d-1} \dots c_0)_2$ , we must have  $\theta_{d-1} = c_{d-1}\mu_{d,d-1}$ . But for  $0 \leq k < d-1$  we can take  $\theta_k = c_k\mu_{d,k} \oplus \hat{\theta}_k$ , where  $\hat{\theta}_k$  is any mask  $\subseteq \mu_{d,k}$ .

(b) Let  $\Theta(d, c)$  be the set of all such mask sequences. Clearly  $\Theta(1, c) = \{c\}$ . When  $d > 1$  we will have, recursively,

$$\Theta(d, c) = \{(\theta_0, \dots, \theta_{d-2}, \theta_{d-1}, \hat{\theta}_{d-2}, \dots, \hat{\theta}_0) \mid \theta_k = \theta'_{k-1} \dagger \theta''_{k-1}, \hat{\theta}_k = \hat{\theta}'_{k-1} \dagger \hat{\theta}''_{k-1}\},$$

by “zipping together” two sequences  $(\theta'_0, \dots, \theta'_{d-3}, \theta'_{d-2}, \hat{\theta}'_{d-3}, \dots, \hat{\theta}'_0) \in \Theta(d-1, c')$  and  $(\theta''_0, \dots, \theta''_{d-3}, \theta''_{d-2}, \hat{\theta}''_{d-3}, \dots, \hat{\theta}''_0) \in \Theta(d-1, c'')$  for some appropriate  $\theta_0, \hat{\theta}_0, c'$ , and  $c''$ .

When  $c$  is odd, the bigraph corresponding to (75) has only one cycle; so  $(\theta_0, \hat{\theta}_0, c', c'')$  is either  $(\mu_{d,0}, 0, \lceil c/2 \rceil, \lfloor c/2 \rfloor)$  or  $(0, \mu_{d,0}, \lfloor c/2 \rfloor, \lceil c/2 \rceil)$ . But when  $c$  is even, the bigraph has  $2^{d-1}$  double bonds; so  $\theta_0 = \hat{\theta}_0$  is any mask  $\subseteq \mu_{d,0}$ , and  $c' = c'' = c/2$ . [Incidentally,  $\lg |\Theta(d, c)| = 2^{d-1}(d-1) - \sum_{k=1}^{d-1} (2^{d-k} - 1)(2^{k-1} - |2^{k-1} - c \bmod 2^k|)$ .]

In both cases we can therefore let  $\hat{\theta}_{d-2} = \dots = \hat{\theta}_0 = 0$  and omit the second half of (71) entirely. Of course in case (b) we would do the cyclic shift directly, instead of using (71) at all. But exercise 58 proves that many other useful permutations, such as selective reversal followed by cyclic shift, can also be handled by (71) with  $\hat{\theta}_k = 0$  for all  $k$ . The *inverses* of those permutations can be handled with  $\theta_k = 0$  for  $0 \leq k < d-1$ .

**52.** The following solutions make  $\hat{\theta}_j = 0$  whenever possible. We shall express the  $\theta$  masks in terms of the  $\mu$ 's, for example by writing  $\mu_{6,5}$  &  $\mu_0$  instead of stating the requested hexadecimal form #55555555; the  $\mu$  form is shorter and more instructive.

(a)  $\theta_k = \mu_{6,k}$  &  $\mu_5$  and  $\hat{\theta}_k = \mu_{6,k}$  &  $(\mu_{k+1} \oplus \mu_{k-1})$  for  $0 \leq k < 5$ ;  $\theta_5 = \theta_4$ . (Here  $\mu_{-1} = 0$ . To get the “other” perfect shuffle,  $(x_{31}x_{63} \dots x_{1x_{33}x_0x_{32}})_2$ , let  $\hat{\theta}_0 = \mu_{6,0}$  &  $\bar{\mu}_1$ .)

(b)  $\theta_0 = \theta_3 = \hat{\theta}_0 = \mu_{6,0}$  &  $\mu_3$ ;  $\theta_1 = \theta_4 = \hat{\theta}_1 = \mu_{6,1}$  &  $\mu_4$ ;  $\theta_2 = \theta_5 = \hat{\theta}_2 = \mu_{6,2}$  &  $\mu_5$ ;  $\hat{\theta}_3 = \hat{\theta}_4 = 0$ . [See J. Lenfant, *IEEE Trans. C-27* (1978), 637–647, for a general theory.]

(c)  $\theta_0 = \mu_{6,0}$  &  $\mu_4$ ;  $\theta_1 = \mu_{6,1}$  &  $\mu_5$ ;  $\theta_2 = \theta_4 = \mu_{6,2}$  &  $\mu_4$ ;  $\theta_3 = \theta_5 = \mu_{6,3}$  &  $\mu_5$ ;  $\hat{\theta}_0 = \mu_{6,0}$  &  $\mu_2$ ;  $\hat{\theta}_1 = \mu_{6,1}$  &  $\mu_3$ ;  $\hat{\theta}_2 = \hat{\theta}_0 \oplus \theta_2$ ;  $\hat{\theta}_3 = \hat{\theta}_1 \oplus \theta_3$ ;  $\hat{\theta}_4 = 0$ .

(d)  $\theta_k = \mu_{6,k}$  &  $\mu_{5-k}$  for  $0 \leq k \leq 5$ ;  $\hat{\theta}_k = \theta_k$  for  $0 \leq k \leq 2$ ;  $\hat{\theta}_3 = \hat{\theta}_4 = 0$ .

**53.** We can write  $\psi$  as a product of  $d-t$  transpositions,  $(u_1v_1) \dots (u_{d-t}v_{d-t})$  (see exercise 5.2.2-2). The permutation induced by a single transposition  $(uv)$  on the index digits, when  $u < v$ , corresponds to a  $(2^v - 2^u)$ -swap with mask  $\mu_{d,v}$  &  $\bar{\mu}_u$ . We should do such a swap for  $(u_1v_1)$  first,  $\dots$ ,  $(u_{d-1}v_{d-1})$  last.

In particular, the perfect shuffle in a  $2^d$ -bit register corresponds to the case where  $\psi = (01 \dots (d-1))$  is a one-cycle; so it can be achieved by doing such  $(2^v - 2^u)$ -swaps for  $(u, v) = (0, 1), \dots, (0, d-1)$ . For example, when  $d = 3$  the two-step procedure is  $12345678 \mapsto 13245768 \mapsto 15263748$ . [Guy Steele suggests an alternative  $(d-1)$ -step procedure: We can do a  $2^k$ -swap with mask  $\mu_{d,k+1}$  &  $\bar{\mu}_k$  for  $d-1 > k \geq 0$ . When  $d = 3$  his method takes  $12345678 \mapsto 12563478 \mapsto 15263748$ .]

The matrix transposition in exercise 52(b) corresponds to  $d = 6$  and  $(u, v) = (0, 3), (1, 4), (2, 5)$ . These operations are the 7-swap, 14-swap, and 28-swap steps for  $8 \times 8$  matrix transposition illustrated in the text; they can be done in any order.

For exercise 52(c), use  $d = 6$  and  $(u, v) = (0, 2), (1, 3), (0, 4), (1, 5)$ . Exercise 52(d) is as easy as 52(b), with  $(u, v) = (0, 5), (1, 4), (2, 3)$ .

**54.** Transposition amounts to reversing the bits of the minor diagonals. Successive elements of those diagonals are  $m-1$  apart in the register. Simultaneous reversal of all diagonals corresponds to simultaneous reversal of subwords of sizes  $1, \dots, m$ , which can be done with  $2^k$ -swaps for  $0 \leq k < \lceil \lg m \rceil$  (because such transposition is easy

zipping  
magic masks  
inshuffle  
Lenfant  
transpositions  
perfect shuffle  
Steele  
matrix transposition

when  $m$  is a power of 2, as illustrated in the text). Here's the procedure for  $m = 7$ :

Given	6-swap	12-swap	24-swap
00 01 02 03 04 05 06	00 <b>10</b> 02 <b>12</b> 04 <b>14</b> 06	00 10 <b>20 30</b> 04 14 <b>24</b>	00 10 20 30 <b>40 50 60</b>
10 11 12 13 14 15 16	<b>01</b> 11 <b>03</b> 13 <b>05</b> 15 <b>25</b>	01 11 <b>21 31</b> 05 15 25	01 11 21 31 <b>41 51 61</b>
20 21 22 23 24 25 26	20 <b>30</b> 22 <b>32</b> 24 <b>16</b> 26	<b>02</b> 12 22 32 <b>06</b> 16 26	02 12 22 32 <b>42 52 62</b>
30 31 32 33 34 35 36	<b>21</b> 31 <b>23</b> 33 <b>43</b> 35 <b>45</b>	<b>03</b> 13 <b>23</b> 33 43 <b>53 63</b>	03 13 23 33 43 53 63
40 41 42 43 44 45 46	40 <b>50</b> 42 <b>34</b> 44 <b>36</b> 46	40 50 <b>60</b> 34 44 <b>54 64</b>	<b>04</b> 14 <b>24</b> 34 44 54 64
50 51 52 53 54 55 56	<b>41</b> 51 <b>61</b> 53 <b>63</b> 55 <b>65</b>	41 51 61 <b>35</b> 45 55 65	<b>05</b> 15 <b>25</b> 35 45 55 65
60 61 62 63 64 65 66	60 <b>52</b> 62 <b>54</b> 64 <b>56</b> 66	<b>42</b> 52 62 <b>36</b> 46 56 66	<b>06</b> 16 <b>26</b> 36 46 56 66

Pratt  
Stockmeyer  
permutation networks  
banyan  
Lawrie  
mapping  
don't-care  
notation

**55.** Given  $x$  and  $y$ , first set  $x \leftarrow x \mid (x \ll 2^k)$  and  $y \leftarrow y \mid (y \ll 2^k)$  for  $2d \leq k < 3d$ . Then set  $x \leftarrow (2^{2d+k} - 2^k)$ -swap of  $x$  with mask  $\mu_{2d+k} \& \bar{\mu}_k$  and  $y \leftarrow (2^{2d+k} - 2^{d+k})$ -swap of  $y$  with mask  $\mu_{2d+k} \& \bar{\mu}_{d+k}$  for  $0 \leq k < d$ . Finally set  $z \leftarrow x \& y$ , then either  $z \leftarrow z \mid (z \gg 2^k)$  or  $z \leftarrow z \oplus (z \gg 2^k)$  for  $2d \leq k < 3d$ , and  $z \leftarrow z \& (2^{n^2} - 1)$ . [The idea is to form two  $n \times n \times n$  arrays  $x = (x_{000} \dots x_{(n-1)(n-1)(n-1)})_2$  and  $y = (y_{000} \dots y_{(n-1)(n-1)(n-1)})_2$  with  $x_{ijk} = a_{jk}$  and  $y_{ijk} = b_{jk}$ , then transpose coordinates so that  $x_{ijk} = a_{ji}$  and  $y_{ijk} = b_{ik}$ ; now  $x \& y$  does all  $n^3$  bitwise multiplications at once. This method is due to V. R. Pratt and L. J. Stockmeyer, *J. Computer and System Sci.* **12** (1976), 210–213.]

**56.** Use (71) with  $\theta_0 = \hat{\theta}_0 = 0$ ,  $\theta_1 = \#0010201122113231$ ,  $\theta_2 = \#00080e0400080c06$ ,  $\theta_3 = \#00000092008100a2$ ,  $\theta_4 = \#0000000000000f16$ ,  $\theta_5 = \#0000000003199c26$ ,  $\hat{\theta}_4 = \#00000c9f0000901a$ ,  $\hat{\theta}_3 = \#003a00b50015002b$ ,  $\hat{\theta}_2 = \#000103080c0d0f0c$ , and  $\hat{\theta}_1 = \#0020032033233333$ .

**57.** The two choices for each cycle when  $d > 1$  have complementary settings. So we can choose a setting in which at least half of the crossbars are inactive, except in the middle column. (See exercise 5.3.4–55 for more about permutation networks.)

**58.** (a) Every different setting of the crossbars gives a different permutation, because there is exactly one path from input line  $i$  to output line  $j$  for all  $0 \leq i, j < N$ . (A network with that property is called a “banyan.”) The unique such path carries input  $i$  on line  $l(i, j, k) = ((i \gg k) \ll k) + (j \bmod 2^k)$  after  $k$  swapping steps have been made.

(b) We have  $l(i\varphi, i, k) = l(j\varphi, j, k)$  if and only if  $i \bmod 2^k = j \bmod 2^k$  and  $i\varphi \gg k = j\varphi \gg k$ ; so (\*) is necessary. And it is also sufficient, because a mapping  $\varphi$  that satisfies (\*) can always be routed in such a way that  $j\varphi$  appears on line  $l = l(j\varphi, j, k)$  after  $k$  steps: If  $k > 1$ ,  $j\varphi$  will appear on line  $l(j\varphi, j, k-1)$ , which is one of the inputs to  $l$ . Condition (\*) says that we can route it to  $l$  without conflict, even if  $l$  is  $l(i\varphi, i, k)$ .

[In *IEEE Transactions C-24* (1975), 1145–1155, Duncan Lawrie proved that condition (\*) is necessary and sufficient for an arbitrary mapping  $\varphi$  of the set  $\{0, 1, \dots, N-1\}$  into itself, when the crossbar modules are allowed to be general  $2 \times 2$  mapping modules as in exercise 75. Furthermore the mapping  $\varphi$  might be only partially specified, with  $j\varphi = *$  (“wild card” or “don’t-care”) for some values of  $j$ . The proof that appears in the previous paragraph actually demonstrates Lawrie’s more general theorem.]

(c)  $i \bmod 2^k = j \bmod 2^k$  if and only if  $k \leq \rho(i \oplus j)$ ;  $i \gg k = j \gg k$  if and only if  $k > \lambda(i \oplus j)$ ; and  $i\varphi = j\varphi$  if and only if  $i = j$ , when  $\varphi$  is a permutation.

(d)  $\lambda(i\varphi \oplus j\varphi) \geq \rho(i \oplus j)$  for all  $i \neq j$  if and only if  $\lambda(i\tau\varphi \oplus j\tau\varphi) \geq \rho(i\tau \oplus j\tau) = \rho(i \oplus j)$  for all  $i \neq j$ , because  $\tau$  is a permutation. [Note that the notation can be confusing: Bit  $j\tau\phi$  appears in bit position  $j$  if permutation  $\phi$  is applied first, then  $\tau$ .]

(e) Since  $l(j, j, k) = j$  for  $0 \leq k \leq d$ , a permutation of  $\Omega$  fixes  $j$  if and only if each of its swaps fixes  $j$ . Thus the swaps performed by  $\varphi$  and by  $\psi$  operate on disjoint elements. The union of these swaps gives  $\varphi\psi$ .

**59.** It is  $2^{M_d(a,b)}$ , where  $M_d(a,b)$  is the number of crossbars that have both endpoints in  $[a \dots b]$ . To count them, let  $k = \lambda(a \oplus b)$ ,  $a' = a \bmod 2^k$ , and  $b' = b \bmod 2^k$ ; notice that  $b - a = 2^k + b' - a'$ , and  $M_d(a,b) = M_{k+1}(a', 2^k + b')$ . Counting the crossbars in the top half and bottom half, plus those that jump between halves, gives  $M_{k+1}(a', 2^k + b') = M_k(a', 2^k - 1) + M_k(0, b') + ((b' + 1) \dot{-} a')$ . Finally, we have  $M_k(0, b') = S(b' + 1)$ ; and  $M_k(a', 2^k - 1) = M_k(0, 2^k - 1 - a') = S(2^k - a') = k2^{k-1} - ka' + S(a')$ , where  $S(n)$  is evaluated in exercise 42.

monus  
nu(k) summed  
Heckel  
Schroeppel  
magic mask

**60.** A cycle of length  $2l$  corresponds to a pattern  $u_0 \leftarrow v_0 \leftrightarrow v_1 \rightarrow u_1 \leftrightarrow u_2 \leftarrow v_2 \leftrightarrow \dots \leftrightarrow v_{2l-1} \rightarrow u_{2l-1} \leftrightarrow u_{2l}$ , where  $u_{2l} = u_0$  and ' $u \leftarrow v$ ' or ' $v \rightarrow u$ ' means that the permutation sends  $u$  to  $v$ , ' $x \leftrightarrow y$ ' means that  $x = y \oplus 1$ .

We can generate a random permutation as follows: Given  $u_0$ , there are  $2n$  choices for  $v_0$ , then  $2n - 1$  choices for  $u_1$  only one of which causes  $u_2 = u_0$ , then  $2n - 2$  choices for  $v_2$ , then  $2n - 3$  choices for  $u_3$  only one of which closes a cycle, etc.

Consequently the generating function is  $G(z) = \prod_{j=1}^n \frac{2n-2j+z}{2n-2j+1}$ . The expected number of cycles,  $k$ , is  $G'(1) = H_{2n} - \frac{1}{2}H_n = \frac{1}{2} \ln n + \ln 2 + \frac{1}{2}\gamma + O(n^{-1})$ . The mean of  $2^k$  is  $G(2) = (2^n n!)^2 / (2n)! = \sqrt{\pi n} + O(n^{-1/2})$ ; and the variance is  $G(4) - G(2)^2 = (n+1 - G(2))G(2) = \sqrt{\pi n}^{3/2} + O(n)$ .

**62.** The crossbar settings in  $P(2^d)$  can be stored in  $(2d-1)2^{d-1} = Nd - \frac{1}{2}N$  bits. To get the inverse permutation proceed from right to left. [See P. Heckel and R. Schroeppel, *Electronic Design* **28**, 8 (12 April 1980), 148–152. Note that *any* way to represent an arbitrary permutation requires at least  $\lg N! > Nd - N/\ln 2$  bits of memory; so this representation is nearly optimum, spacewise.]

**63.** (i)  $x = y$ . (ii) Either  $z$  is even or  $x \oplus y < 2^{\max(0, (z-1)/2)}$ . (When  $z$  is odd we have  $(x \ddagger y) \gg z = (y \gg \lceil z/2 \rceil) \ddagger (x \gg \lfloor z/2 \rfloor)$ , even when  $z < 0$ .) (iii) This identity holds for all  $w, x, y$ , and  $z$  (and also with any other bitwise Boolean operator in place of  $\&$ ).

**64.**  $((z \& \mu_0) + (z' \mid \bar{\mu}_0)) \& \mu_0 \mid (((z \& \bar{\mu}_0) + (z' \mid \mu_0)) \& \bar{\mu}_0)$ . (See (86).)

**65.**  $xu(x^2) + v(x^2) = xu(x)^2 + v(x)^2$ .

**66.** (a)  $v(x) = (u(x)/(1+x^\delta)) \bmod x^n$ ; it's the unique polynomial of degree less than  $n$  such that  $(1+x^\delta)v(x) \equiv u(x) \pmod{x^n}$ . (Equivalently,  $v$  is the unique  $n$ -bit integer such that  $(v \oplus (v \ll \delta)) \bmod 2^n = u$ .)

(b) We may as well assume that  $n = 64m$ , and that  $u = (u_{m-1} \dots u_1 u_0)_{2^{64}}$ ,  $v = (v_{m-1} \dots v_1 v_0)_{2^{64}}$ . Set  $c \leftarrow 0$ ; then, using exercise 36, set  $v_j \leftarrow u_j^\oplus \oplus (-c)$  and  $c \leftarrow v_j \gg 63$  for  $j = 0, 1, \dots, m-1$ .

(c) Set  $c \leftarrow v_0 \leftarrow u_0$ ; then  $v_j \leftarrow u_j \oplus c$  and  $c \leftarrow v_j$ , for  $j = 1, 2, \dots, m-1$ .

(d) Start with  $c \leftarrow 0$  and do the following for  $j = 0, 1, \dots, m-1$ : Set  $t \leftarrow u_j$ ,  $t \leftarrow t \oplus (t \ll 3)$ ,  $t \leftarrow t \oplus (t \ll 6)$ ,  $t \leftarrow t \oplus (t \ll 12)$ ,  $t \leftarrow t \oplus (t \ll 24)$ ,  $t \leftarrow t \oplus (t \ll 48)$ ,  $v_j \leftarrow t \oplus c$ ,  $c \leftarrow (t \gg 61) \times \#9249249249249249$ .

(e) Start with  $v \leftarrow u$ . Then, for  $j = 1, 2, \dots, m-1$ , set  $v_j \leftarrow v_j \oplus (v_{j-1} \ll 3)$  and (if  $j < m-1$ )  $v_{j+1} \leftarrow v_{j+1} \oplus (v_{j-1} \gg 61)$ .

**67.** Let  $n = 2l - 1$  and  $m = n - 2d$ . If  $\frac{1}{2}n < k < n$  we have  $x^{2k} \equiv x^{m+t} + x^t \pmod{x^n + x^m + 1}$ , where  $t = 2k - n$  is odd. Consequently, if  $v = (v_{n-1} \dots v_1 v_0)_2$ , the number

$$w = u \oplus (((u \gg d) \oplus (u \gg 2d) \oplus (u \gg 3d) \oplus \dots) \& -2^{l-d})$$

turns out to equal  $(v_{n-2} \dots v_3 v_1 v_{n-1} \dots v_2 v_0)_2$ . For example, when  $l = 4$  and  $d = 2$ , the square of  $u_6 x^6 + \dots + u_1 x + u_0$  modulo  $(x^7 + x^3 + 1)$  is  $u_6 x^5 + u_5 x^3 + (u_6 \oplus u_4) x^1 + (u_5 \oplus u_3) x^6 + (u_6 \oplus u_4 \oplus u_2) x^4 + u_1 x^2 + u_0$ . To compute  $v$ , we therefore do a perfect

shuffle,  $v = \lfloor w/2^l \rfloor \ddagger (w \bmod 2^l)$ . The number  $w$  can be calculated by methods like those of the previous exercise. [See R. P. Brent, S. Larvala, and P. Zimmermann, *Math. Comp.* **72** (2003), 1443–1452; **74** (2005), 1001–1002.]

68. SRU t,x,delta; PUT rM,theta; MUX x,t,x.

69. Notice that the procedure might fail if we attempt to do the  $2^{d-1}$ -shift first instead of last. The key to proving that a small-shift-first strategy works correctly is to watch the spaces *between* selected bits; we will prove that the lengths of these spaces are multiples of  $2^{k+1}$  after the  $2^k$ -shift.

Consider the infinite string  $\chi_k = \dots 1^{t_4} 0^{2^k} 1^{t_3} 0^{2^k} 1^{t_2} 0^{2^k} 1^{t_1} 0^{2^k} 1^{t_0}$ , which represents the situation where  $t_l \geq 0$  items need to move  $2^k l$  places to the right. A  $2^k$ -shift with any mask of the form  $\theta_k = \dots 0^{t_4} *^{2^{k+1}} 1^{t_3} 0^{t_2} *^{2^{k+1}} 1^{t_1} 0^{t_0}$  leaves us with the situation represented by the string  $\chi_{k+1} = \dots 1^{T_2} 0^{2^{k+1}} 1^{T_1} 0^{2^{k+1}} 1^{T_0}$ , where exactly  $T_l = t_{2l} + t_{2l+1}$  items need to move right  $2^{k+1} l$  places. So the claim holds by induction on  $k$ .

70. Let  $\psi_k = \theta_k \oplus (\theta_k \ll 1)$ , so that  $\theta_k = \psi_k^\oplus$  in the notation of exercise 36. If we take  $*^{2^{k+1}} = 0^{2^k} 1^{2^k}$  in the previous answer, we have  $\psi_0 = \bar{\chi}$  and  $\psi_{k+1} = (\psi_k \& \bar{\theta}_k) \gg 2^k$ . Therefore we can proceed as follows:

Set  $\psi \leftarrow \bar{\chi}$ ,  $k \leftarrow 0$ , and repeat the following steps while  $\psi \neq 0$ : Set  $x \leftarrow \psi$ , then  $x \leftarrow x \oplus (x \ll 2^l)$  for  $0 \leq l < d$ , then  $\theta_k \leftarrow x$ ,  $\psi \leftarrow (\psi \& \bar{x}) \gg 2^k$ , and  $k \leftarrow k + 1$ .

The computation ends with  $k = \lambda \nu \bar{\chi} + 1$ ; the remaining masks  $\theta_k, \dots, \theta_{d-1}$ , if any, are zero and those steps can be omitted from (80). Sometimes this procedure gives nonzero masks  $\theta_k$  that actually do nothing useful, because  $t_1 = t_3 = \dots = 0$ . To avoid such redundancy, change ' $\theta_k \leftarrow x$ ' to ' $\theta_k \leftarrow x \& (x + (x \& \psi \& (\psi \gg 2^k)))$ '.

[See *compress* in H. S. Warren, Jr., *Hacker's Delight* (Addison-Wesley, 2002), §7–4; also G. L. Steele Jr., *U.S. Patent 6715066* (30 March 2004).]

71. Start with  $x \leftarrow y$ . Do a  $(-2^k)$ -shift of  $x$  with mask  $\theta_k$ , for  $k = d-1, \dots, 1, 0$ , using the masks of exercise 70. Finally set  $z \leftarrow x$  (or  $z \leftarrow x \& \chi$ , if you want a “clean” result).

72.  $2^{2^{d-1}} x + y$ .

73. Equivalently,  $d$  sheep-and-goats operations must be able to transform the word  $x^\pi = (x_{(2^d-1)\pi} \dots x_{1\pi} x_{0\pi})_2$  into  $(x_{2^d-1} \dots x_1 x_0)_2$ , for any permutation  $\pi$  of  $\{0, 1, \dots, 2^d-1\}$ . And this can be done by radix-2 sorting (Algorithm 5.2.5R): First bring the odd numbered bits to the left, then bring the bits  $j$  for odd  $\lfloor j/2 \rfloor$  left, and so on. For example, when  $d = 3$  and  $x^\pi = (x_3 x_1 x_0 x_7 x_5 x_2 x_6 x_4)_2$ , the three operations yield successively  $(x_3 x_1 x_7 x_5 x_0 x_2 x_6 x_4)_2$ ,  $(x_3 x_7 x_2 x_6 x_1 x_5 x_0 x_4)_2$ ,  $(x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0)_2$ . [See Z. Shi and R. Lee, *Proc. IEEE Conf. ASAP'00* (IEEE CS Press, 2000), 138–148.]

*Historical note:* The BESM-6 computer, designed in 1965, implemented half of the sheep-and-goats operation: Its «сборка» (“gather” or “pack”) command produced  $(z \& \chi) \cdot \bar{\chi}$ , and its «разборка» command (“scatter” or “unpack”) went the other way.

74. If  $|\sum c_{2l} - \sum c_{2l+1}| = 2\Delta > 0$ , we must rob  $\Delta$  from the rich half and give it to the poor. There's a position  $l$  in the poor half with  $c_l = 0$ ; otherwise that half would sum to at least  $2^{d-1}$ . A cyclic 1-shift that modifies positions  $l$  through  $(l+t) \bmod 2^d$  makes  $c'_{l+k} = c_{l+k+1}$  for  $0 \leq k < t$ ,  $c'_{l+t} = c_{l+t+1} - \delta$ ,  $c'_{l+t+1} = \delta$ , and  $c'_{l+k} = c_{l+k}$  for all other  $k$ ; here  $\delta$  can be any desired value in the range  $0 \leq \delta \leq c_{l+t+1}$ . (We've treated all subscripts modulo  $2^d$  in these formulas.) So we can use the smallest even  $t$  such that  $c_{l+1} + c_{l+3} + \dots + c_{l+t+1} = c_l + c_{l+2} + \dots + c_{l+t} + \Delta + \delta$  for some  $\delta \geq 0$ .

(The 1-shift need not be cyclic, if we allow ourselves to shift left instead of right. But the cyclic property may be needed in subsequent steps.)

zipper  
Brent  
Larvala  
Zimmermann  
MUX  
Warren  
compress  
Steele Jr.  
Patent  
Shi  
Lee  
BESM-6 computer  
gather  
pack  
compress  
scatter  
unpack



**75.** Equivalently, given indices  $0 \leq i_0 < i_1 < \dots < i_{s-1} < i_s = 2^d$  and  $0 = j_0 < j_1 < \dots < j_{s-1} < j_s = 2^d$ , we want to map  $(x_{2^d-1} \dots x_1 x_0)_2 \mapsto (x_{(2^d-1)\varphi} \dots x_{1\varphi} x_{0\varphi})_2$ , where  $j\varphi = i_r$  for  $j_r \leq j < j_{r+1}$  and  $0 \leq r < s$ . If  $d = 1$ , a mapping module does this.

When  $d > 1$ , we can set the left-hand crossbars so that they route input  $i_r$  to line  $i_r \oplus ((i_r + r) \bmod 2)$ . If  $s$  is even, we recursively ask one of the networks  $P(2^{d-1})$  inside  $P(2^d)$  to solve the problem for indices  $\lfloor \{i_0, i_2, \dots, i_s\}/2 \rfloor$  and  $\lfloor \{j_0, j_2, \dots, j_s\}/2 \rfloor$ , while the other solves it for  $\lfloor \{i_1, i_3, \dots, i_{s-1}\}/2 \rfloor$  and  $\lceil \{j_1, j_3, \dots, j_{s-1}\}/2 \rceil$ . At the right of  $P(2^d)$ , one can now check that when  $j_r \leq j < j_{r+1}$ , the mapping module for lines  $j$  and  $j \oplus 1$  has input  $i_r$  on line  $j$  if  $j \equiv r \pmod{2}$ , otherwise  $i_r$  is on line  $j \oplus 1$ . A similar proof works when  $s$  is odd.

*Notes:* This network is a slight improvement over a construction by Yu. P. Ofman, *Trudy Mosk. Mat. Obshchestva* **14** (1965), 186–199. We can implement the corresponding network by substituting a “ $\delta$ -map” for a  $\delta$ -swap; instead of (6g), we use two masks and do seven operations instead of six:  $y \leftarrow x \oplus (x \gg \delta)$ ,  $x \leftarrow x \oplus (y \& \theta) \oplus ((y \& \theta') \ll \delta)$ . This extension of (71) therefore takes only  $d$  additional units of time.

**76.** When a mapping network realizes a permutation, all of its modules must act as crossbars; hence  $G(n) \geq \lg n!$ . Ofman proved that  $G(n) \leq 2.5n \lg n$ , and remarked in a footnote that the constant 2.5 could be improved (without giving any details). We have seen that in fact  $G(n) \leq 2n \lg n$ . Note that  $G(3) = 3$ .

**77.** Represent an  $n$ -network by  $(x_{2^n-1} \dots x_1 x_0)_2$ , where  $x_k =$  [the binary representation of  $k$  is a possible configuration of 0s and 1s when the network has been applied to all  $2^n$  sequences of 0s and 1s], for  $0 \leq k < 2^n$ . Thus the empty network is represented by  $2^{2^n} - 1$ , and a sorting network for  $n = 3$  is represented by  $(10001011)_2$ . In general,  $x$  represents a sorting network for  $n$  elements if and only if it represents an  $n$ -network and  $\nu x = n + 1$ , if and only if  $x = 2^0 + 2^1 + 2^3 + 2^7 + \dots + 2^{2^n-1}$ .

If  $x$  represents  $\alpha$  according to these conventions, the representation of  $\alpha[i:j]$  is  $(x \oplus y) \mid (y \gg (2^{n-i} - 2^{n-j}))$ , where  $y = x \& \bar{\mu}_{n-i} \& \mu_{n-j}$ .

[See V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer, *STOC* **6** (1974), 122–126.]

**78.** If  $k \geq \lg(m-1)$  the test is valid, because we always have  $x_1 + x_2 + \dots + x_m \geq x_1 \mid x_2 \mid \dots \mid x_m$ , with equality if and only if the sets are disjoint. Moreover, we have  $(x_1 + \dots + x_m) - (x_1 \mid \dots \mid x_m) \leq (m-1)(2^{n-k-1} + \dots + 1) < (m-1)2^{n-k} \leq 2^n$ .

Conversely, if  $m \geq 2^k + 2$  and  $n > 2k$ , the test is invalid. We might have, for example,  $x_1 + \dots + x_m = (2^k + 1)(2^{n-k} - 2^{n-2k-1}) + 2^{n-k-1} = 2^n + (2^{n-k} - 2^{n-2k-1})$ .

But if  $n \leq 2k$  the test is still valid when  $m = 2^k + 2$ , because our proof shows that  $x_1 + \dots + x_m - (x_1 \mid \dots \mid x_m) \leq (2^k + 1)(2^{n-k} - 1) < 2^n$  in that case.

**79.**  $x_i = (x-1) \& \chi$ . (And the formula  $x_i = ((x-b-1) \& a) + b$  corresponds to (85).) These recipes for  $x'$  and  $x_i$  are part of Jörg Arndt’s “bit wizardry” routines (2001); their origin is unknown.

**80.** Perhaps the nicest way is to start with  $x \leftarrow \chi - 1$  as a signed number; then while  $x \geq 0$ , set  $x \leftarrow x \& \chi$ , visit  $x$ , and set  $x \leftarrow 2x - \chi$ . (The operation  $2x - \chi$  can in fact be performed with a single MMIX instruction, ‘2ADDU x,x,minuschi’.)

But that trick fails if  $\chi$  is so large as to be *already* “negative.” A slightly slower but more general method starts with  $x \leftarrow \chi$  and does the following while  $x \neq 0$ : Set  $t \leftarrow x \& -x$ , visit  $\chi - t$ , and set  $x \leftarrow x - t$ .

**81.**  $((z \& \chi) - (z' \& \chi)) \& \chi$ . (One way to verify this formula is to use (18).)

**82.** Yes, by letting  $z = z'$  in (86):  $w \mid (z \& \bar{\chi})$ , where  $w = ((z \& \chi) + (z \mid \bar{\chi})) \& \chi$ .

recursively  
Ofman  
 $\delta$ -map  
Ofman  
magic masks  
Pratt  
Rabin  
Stockmeyer  
Arndt  
MMIX  
2ADDU

**83.** (The following iteration propagates bits of  $y$  to the right, in the gaps of a scattered accumulator  $t$ . Auxiliary variables  $u$  and  $v$  respectively mark the left and right of each gap; they double in size until being wiped out by  $w$ .) Set  $t \leftarrow z \& \chi$ ,  $u' \leftarrow (\chi \gg 1) \& \bar{\chi}$ ,  $v \leftarrow ((\chi \ll 1) + 1) \& \bar{\chi}$ ,  $w \leftarrow 3(u' \& v)$ ,  $u \leftarrow 3u'$ ,  $v \leftarrow 3v$ , and  $k \leftarrow 1$ . Then, while  $u \neq 0$ , do the following steps:  $t \leftarrow t \mid ((t \gg k) \& u')$ ,  $k \leftarrow k \ll 1$ ,  $u \leftarrow u \& \bar{w}$ ,  $v \leftarrow v \& \bar{w}$ ,  $w \leftarrow ((v \& (u \gg 1) \& \bar{u}) \ll (k + 1)) - ((u \& (v \ll 1) \& \bar{v}) \gg k)$ ,  $u' \leftarrow (u \& \bar{v}) \gg k$ ,  $v \leftarrow v + ((v \& \bar{u}) \ll k)$ ,  $u \leftarrow u + u'$ . Finally return the answer  $((t \gg 1) \& \chi) \mid (z \& \bar{\chi})$ .

**84.**  $z \leftarrow \chi = w - (z \& \chi)$ , where  $w = (((z \& \chi) \ll 1) + \bar{\chi}) \& \chi$  appears in answer 82;  $z \rightarrow \chi$  is the quantity  $t$  computed (with more difficulty) in the answer to exercise 83.

**85.** (a) If  $x = \text{LOC}(a[i, j, k])$  is the drum location corresponding to interleaved bits as stated, then  $\text{LOC}(a[i + 1, j, k]) = x \oplus ((x \oplus ((x \& \chi) - \chi)) \& \chi)$  and  $\text{LOC}(a[i - 1, j, k]) = x \oplus ((x \oplus ((x \& \chi) - 1)) \& \chi)$ , where  $\chi = (11111)_8$ , by (84) and answer 79. The formulas for  $\text{LOC}(a[i, j \pm 1, k])$  and  $\text{LOC}(a[i, j, k \pm 1])$  are similar, with masks  $2\chi$  and  $4\chi$ .

(b) For random access, let's hope there is room for a table of length 32 giving  $f[(i_4 i_3 i_2 i_1 i_0)_2] = (i_4 i_3 i_2 i_1 i_0)_8$ . Then  $\text{LOC}(a[i, j, k]) = (((f[k] \ll 1) + f[j]) \ll 1) + f[i]$ . (On a vintage machine, bitwise computation of  $f$  would be much worse than table lookup, because register operations used to be as slow as fetches from memory.)

(c) Let  $p$  be the location of the page currently in fast memory, and let  $z = -128$ . When accessing location  $x$ , if  $x \& z \neq p$  it is necessary to read 128 words from drum location  $x \& z$  (after saving the current data to drum location  $p$  if it has changed); then set  $p \leftarrow x \& z$ . [See *J. Royal Stat. Soc. B-16* (1954), 53–55. This scheme of array allocation for external storage was devised independently by E. W. Dijkstra, circa 1960, who called it the “zip-fastener” method. It has often been rediscovered, for example in 1966 by G. M. Morton and later by developers of quadrees; see Hanan Samet, *Applications of Spatial Data Structures* (Addison-Wesley, 1990). See also R. Raman and D. S. Wise, *IEEE Trans. C57* (2008), to appear, for a contemporary perspective. Georg Cantor had considered interleaving the digits of decimal fractions in *Crelle* **84** (1878), 242–258, §7; but he observed that this idea does *not* lead to an easy one-to-one correspondence between the unit interval  $[0..1]$  and the unit square  $[0..1] \times [0..1]$ .]

**86.** If  $(p', q', r')$  rightmost bits and  $(p'', q'', r'')$  other bits of  $(i, j, k)$  are in the part of the address that does not affect the page number, the total number of page faults is  $2((2^{p-p'} - 1)2^{q+r} + (2^{q-q'} - 1)2^{p+r} + (2^{r-r'} - 1)2^{p+q})$ . Hence we want to minimize  $2^{-p'} + 2^{-q'} + 2^{-r'}$  over nonnegative integers  $(p', q', r', p'', q'', r'')$  with  $p' + p'' \leq p$ ,  $q' + q'' \leq q$ ,  $r' + r'' \leq r$ ,  $p' + q' + r' + p'' + q'' + r'' = s$ . Since  $2^a + 2^b > 2^{a-1} + 2^{b+1}$  when  $a$  and  $b$  are integers with  $a > b + 1$ , the minimum (for all  $s$ ) occurs when we select bits from right to left cyclically until running out. For example, when  $(p, q, r) = (2, 6, 3)$  the addressing function would be  $(j_5 j_4 j_3 k_2 j_2 k_1 j_1 i_1 k_0 j_0 i_0)_2$ . In particular, Tocher's scheme is optimal.

[But such a mapping is not necessarily best when the page size isn't a power of 2. For example, consider a  $16 \times 16$  matrix; the addressing function  $(j_3 i_3 i_2 i_1 i_0 j_2 j_1 j_0)_2$  is better than  $(j_3 i_3 j_2 i_2 j_1 i_1 j_0 i_0)_2$  for all page sizes from 17 to 62, except for size 32 when they are equally good.]

**87.** Set  $x \leftarrow x \& \sim((x \& \text{"@@@@@@@@"} \gg 1))$ ; each byte  $(a_7 \dots a_0)_2$  is thereby changed to  $(a_7 a_6 (a_5 \wedge \bar{a}_6) a_4 \dots a_0)_2$ . The same transformation works also on 30 additional letters in the Latin-1 supplement to ASCII (for example,  $\mathfrak{x} \mapsto \mathfrak{E}$ ); but there's one glitch,  $\mathfrak{y} \mapsto \mathfrak{B}$ .

[Don Woods used this trick in his original program for the game of Adventure (1976), uppercasing the user's input words before looking them up in a dictionary.]

**88.** Set  $z \leftarrow (x \oplus \bar{y}) \& h$ , then  $z \leftarrow ((x \mid h) - (y \& \bar{h})) \oplus z$ .

table lookup  
Dijkstra  
zip-fastener  
Z order, see zip  
Morton  
quadrees  
Samet  
Raman  
Wise  
Cantor  
analysis of alg  
convex optimization  
Tocher  
Latin-1  
Woods  
game  
Adventure

89.  $t \leftarrow x \mid \bar{y}$ ,  $t \leftarrow t \& (t \gg 1)$ ,  $z \leftarrow (x \& \bar{y} \& \bar{\mu}_0) \mid (t \& \mu_0)$ . [From the “nasty” test program for H. G. Dietz and R. J. Fisher’s SWARC compiler (1998), optimized by T. Dahlheimer.]

90. Insert ‘ $z \leftarrow z \mid ((x \oplus y) \& l)$ ’ either before or after ‘ $z \leftarrow (x \& y) + z$ ’. (The ordering makes no difference, because  $x + y \equiv x \oplus y$  (modulo 4) when  $x + y$  is odd. Therefore MMIX can round to odd at no additional cost, using MOR. Rounding to even in the ambiguous cases is more difficult, and with fixed point arithmetic it is not advantageous.)

91. If  $\frac{1}{2}[x, y]$  denotes the average as in (88), the desired result is obtained by repeating the following operations seven times, then concluding with  $z \leftarrow \frac{1}{2}[x, y]$  once more:

$$z \leftarrow \frac{1}{2}[x, y], \quad t \leftarrow \alpha \& h, \quad m \leftarrow (t \ll 1) - (t \gg 7),$$

$$x \leftarrow (m \& z) \mid (\bar{m} \& x), \quad y \leftarrow (\bar{m} \& z) \mid (m \& y), \quad \alpha \leftarrow \alpha \ll 1.$$

Although rounding errors accumulate through eight levels, the resulting absolute error never exceeds 807/255. Moreover, it is  $\approx 1.13$  if we average over all  $256^3$  cases, and it is less than 2 with probability  $\approx 94.2\%$ . If we round to odd as in exercise 90, the maximum and average error are reduced to 616/255 and  $\approx 0.58$ ; the probability of error  $< 2$  rises to  $\approx 99.9\%$ . Therefore the following MMIX code uses such unbiased rounding:

x GREG ;y GREG ;z GREG	$\left\{ \begin{array}{lll} \text{XOR} & t, x, y & \text{MOR } m, \text{ffhi}, \text{alf} \\ \text{MOR} & z, \text{rodd}, t & \text{PUT } rM, m \\ \text{AND} & t, x, y & \text{MUX } x, z, x \\ \text{ADDU} & z, z, t & \text{MUX } y, y, z \\ & & \text{SLU } \text{alf}, \text{alf}, 1 \end{array} \right\}$
alf GREG ;m GREG ;t IS \$255	
ffhi GREG -1<<56	
1 GREG #0101010101010101	
rodd GREG #4020100804020101	

repeat seven times:

after which the first four instructions are repeated again. The total time for eight  $\alpha$ -blends (67v) is less than the cost of eight multiplications.

92. We get  $z_j = \lceil (x_j + y_j)/2 \rceil$  for each  $j$ . (This fact, noticed by H. S. Warren, Jr., follows from the identity  $x + y = ((x \mid y) \ll 1) - (x \oplus y)$ . See also the next exercise.)

93.  $x - y = (x \oplus y) - ((\bar{x} \& y) \ll 1)$ . (“Borrows” instead of “carries.”)

94.  $(x - l)_j = (x_j - 1 - b_j) \bmod 256$ , where  $b_j$  is the “borrow” from fields to the right. So  $t_j$  is nonzero if and only if  $(x_j \dots x_0)_{256} < (1 \dots 1)_{256} = (256^{j+1} - 1)/255$ . (The answers to the stated questions are therefore “yes” and “no.”)

In general if the constant  $l$  is allowed to have *any* value  $(l_7 \dots l_1 l_0)_{256}$ , operation (90) makes  $t_j \neq 0$  if and only if  $(x_j \dots x_0)_{256} < (l_j \dots l_0)_{256}$  and  $x_j < 128$ .

95. Use (90): Test if  $h \& (t(x \oplus ((x \gg 8) + (x \ll 56))) \mid t(x \oplus ((x \gg 16) + (x \ll 48))) \mid t(x \oplus ((x \gg 24) + (x \ll 40))) \mid t(x \oplus ((x \gg 32) + (x \ll 32)))) = 0$ , where  $t(x) = (x - l) \& \bar{x}$ . (These 28 steps reduce to 20 if cyclic shift is available, or to 11 with MXOR and BDIF.)

96. Suppose  $0 \leq x, y < 256$ ,  $x_h = \lfloor x/128 \rfloor$ ,  $x_l = x \bmod 128$ ,  $y_h = \lfloor y/128 \rfloor$ ,  $y_l = y \bmod 128$ . Then  $[x < y] = \langle \bar{x}_h y_h [x_l < y_l] \rangle$ ; see exercise 7.1.1–106. And  $[x_l < y_l] = [y_l + 127 - x_l \geq 128]$ . Hence  $[x < y] = \lfloor \langle \bar{x} y z \rangle / 128 \rfloor$ , where  $z = (\bar{x} \& 127) + (y \& 127)$ .

It follows that  $t = h \& \langle \bar{x} y z \rangle$  has the desired properties, when  $z = (\bar{x} \& \bar{h}) + (y \& \bar{h})$ . This formula can also be written  $t = h \& \sim \langle x \bar{y} \bar{z} \rangle$ , where  $\bar{z} = \sim((\bar{x} \& \bar{h}) + (y \& \bar{h})) = (x \mid \bar{h}) - (y \& \bar{h})$  by (18).

To get a similar test function for  $[x_j \leq y_j] = 1 - [y_j < x_j]$ , we just interchange  $x \leftrightarrow y$  and take the complement:  $t \leftarrow h \& \sim \langle x \bar{y} z \rangle = h \& \langle \bar{x} y \bar{z} \rangle$ , where  $z = (x \& \bar{h}) + (\bar{y} \& \bar{h})$ .

97. Set  $x' \leftarrow x \oplus \text{*****}$ ,  $y' \leftarrow x \oplus y$ ,  $t \leftarrow h \& (x \mid ((x \mid h) - l)) \& (y' \mid ((y' \mid h) - l))$ ,  $m \leftarrow (t \ll 1) - (t \gg 7)$ ,  $t \leftarrow t \& (x' \mid ((x' \mid h) - l))$ ,  $z \leftarrow (m \& \text{*****}) \mid (\bar{m} \& y)$ . (20 steps.)

98. Set  $u \leftarrow x \oplus y$ ,  $z \leftarrow (\bar{x} \& \bar{h}) + (y \& \bar{h})$ ,  $t \leftarrow h \& (x \oplus (u \mid (x \oplus z)))$ ,  $v \leftarrow ((t \ll 1) - (t \gg 7)) \& u$ ,  $z \leftarrow x \oplus v$ ,  $w \leftarrow y \oplus v$ . [This 14-step procedure invokes answer 96 to compute  $t =$

Dietz  
Fisher  
SWARC compiler  
Dahlheimer  
MMIX  
MOR  
Rounding to even  
round to odd  
MUX  
unbiased rounding  
Warren  
identity  
Borrows  
carries  
borrow  
MXOR  
BDIF  
cyclic shift  
medians

$h \& \langle \bar{x}yz \rangle$ , using the footprint method of Section 7.1.2 to evaluate the median in only three steps when  $x \oplus y$  is known. Of course the MMIX solution is much quicker, if available: `BDIF t,x,y; ADDU z,y,t; SUBU w,x,t.`

**99.** In this potpourri, each of the eight bytes appears to be solving a different kind of problem; we must recast the conditions so that they fit into a common framework:  $f_0 = [x_0 \oplus '1' \leq 0]$ ,  $f_1 = [x_1 \oplus '*' > 0]$ ,  $f_2 = [x_2 \leq 'A' - 1]$ ,  $f_3 = [x_3 > 'z']$ ,  $f_4 = [x_4 > 'a' - 1]$ ,  $f_5 = [x_5 \oplus '0' \leq 9]$ ,  $f_6 = [x_6 \oplus 255 > 86]$ ,  $f_7 = [x_7 \oplus '?' \leq 3]$ . Aha! We can use the formulas in answer 96, adjusting  $d$  to switch between  $\leq$  and  $>$  as needed:  $a = ('?'(255)'0'000'*''!')_{256} = \#3fff300000002a21$ ;  $b = \bar{h} = \#7f7f7f7f7f7f7f7f$ ;  $c = \bar{h} \& \sim(3(86)9('a' - 1)'z'('A' - 1)00)_{256} = \#7c29761f053f7f7f$  (the hardest one);  $d = \#8000800000800080$ ; and  $e = h = \#8080808080808080$ .

**100.** We want  $u_j = x_j + y_j + c_j - 10c_{j+1}$  and  $v_j = x_j - y_j - b_j + 10b_{j+1}$ , where  $c_j$  and  $b_j$  are the “carry” and “borrow” into digit position  $j$ . Set  $u' \leftarrow (x + y + (6 \dots 66)_{16}) \bmod 2^{64}$  and  $v' \leftarrow (x - y) \bmod 2^{64}$ . Then we find  $u'_j = x_j + y_j + c_j + 6 - 16c_{j+1}$  and  $v'_j = x_j - y_j - b_j + 16b_{j+1}$  for  $0 \leq j < 16$ , by induction on  $j$ . Hence  $u'$  and  $v'$  have the same pattern of carries and borrows as if we were working in radix 10, and we have  $u = u' - 6(\bar{c}_{16} \dots \bar{c}_2 \bar{c}_1)_{16}$ ,  $v = v' - 6(b_{16} \dots b_2 b_1)_{16}$ . The following computation schemes therefore provide the desired results (10 operations for addition, 9 for subtraction):

$$\begin{aligned} y' &\leftarrow y + (6 \dots 66)_{16}, & u' &\leftarrow x + y', & v' &\leftarrow x - y, \\ t &\leftarrow \langle \bar{x}y'u' \rangle \& (8 \dots 88)_{16}, & & t &\leftarrow \langle \bar{x}yv' \rangle \& (8 \dots 88)_{16}, \\ u &\leftarrow u' - t + (t \gg 2); & & & v &\leftarrow v' - t + (t \gg 2). \end{aligned}$$

**101.** For subtraction, set  $z \leftarrow x - y$ ; for addition, set  $z \leftarrow x + y + \#e8c4c4fc18$ , where this constant is built from  $256 - 24 = \#e8$ ,  $256 - 60 = \#c4$ , and  $65536 - 1000 = \#fc18$ . Borrows and carries will occur between fields as if mixed-radix subtraction or addition were being performed. The remaining task is to correct for cases in which borrows occurred or carries did not; we can do this easily by inspecting individual digits, because the radices are less than half of the field sizes: Set  $t \leftarrow z \& \#8080808000$ ,  $t \leftarrow (t \ll 1) - (t \gg 7) - ((t \gg 15) \& 1)$ ,  $z \leftarrow z - (t \& \#e8c4c4fc18)$ . [See Stephen Soule, *CACM* **6** (1975), 344–346. We’re lucky that the ‘c’ in ‘fc18’ is even.]

**102.** (a) We assume that  $x = (x_{15} \dots x_0)_{16}$  and  $y = (y_{15} \dots y_0)_{16}$ , with  $0 \leq x_j, y_j < 5$ ; the goal is to compute  $u = (u_{15} \dots u_0)_{16}$  and  $v = (v_{15} \dots v_0)_{16}$ , with components  $u_j = (x_j + y_j) \bmod 5$  and  $v_j = (x_j - y_j) \bmod 5$ . Here’s how:

$$\begin{aligned} u &\leftarrow x + y, & v &\leftarrow x - y + 5l, \\ t &\leftarrow (u + 3l) \& h, & t &\leftarrow (v + 3l) \& h, \\ u &\leftarrow u - ((t - (t \gg 3)) \& 5l); & v &\leftarrow v - ((t - (t \gg 3)) \& 5l). \end{aligned}$$

Here  $l = (1 \dots 1)_{16} = (2^{64} - 1)/15$ ,  $h = 8l$ . (Addition in 7 operations, subtraction in 8.)

(b) Now  $x = (x_{20} \dots x_0)_8$ , etc., and we must be more careful to confine carries:

$$\begin{aligned} t &\leftarrow x + \bar{h}, & z &\leftarrow (x \mid h) - (y \& \bar{h}), \\ z &\leftarrow (t \& \bar{h}) + (y \& \bar{h}), & t &\leftarrow (y \mid \bar{z}) \& \bar{x} \& h, \\ t &\leftarrow (y \mid z) \& t \& h, & v &\leftarrow x - y + t + (t \gg 2). \\ u &\leftarrow x + y - (t + (t \gg 2)); & & & \end{aligned}$$

Here  $h = (4 \dots 4)_8 = (2^{65} - 4)/7$ . (Addition in 11 operations, subtraction in 10.)

Similar procedures work, of course, for other moduli. In fact we can do multibyte arithmetic on the coordinates of toruses in general, with different moduli in each component (see 7.2.1.3–(66)).

footprint method  
median  
MMIX  
BDIF  
potpourri  
carry  
borrow  
medians  
Borrows  
carries  
Soule  
multibyte arithmetic  
toruses

**103.** Let  $h$  and  $l$  be the constants in (87) and (88). Addition is easy:  $u \leftarrow x | ((x \& \bar{h}) + y)$ . For subtraction, take away 1 and add  $x_j \& (1 - y_j)$ :  $t \leftarrow (x \& \bar{l}) \gg 1$ ,  $v \leftarrow t | (t + (x \& (y \oplus l)))$ .

**104.** Yes, in 19: Let  $a = (((1901 \ll 4) + 1) \ll 5) + 1$ ,  $b = (((2099 \ll 4) + 12) \ll 5) + 28$ . Set  $m \leftarrow (x \gg 5) \& \#f$  (the month),  $c \leftarrow \#10 \& \sim((x | (x \gg 1)) \gg 5)$  (the leap year correction),  $u \leftarrow b + \#3 \& ((\#3bbeecc + c) \gg (m + m))$  (the *max\_day* adjustment), and  $t \leftarrow ((x \oplus a \oplus (x - a)) | (x \oplus u \oplus (u - x))) \& \#1000220$  (the test for unwanted carries).

**105.** Exercise 98 explains how to compute bitwise min and max; a simple modification will compute min in some byte positions and max in others. Thus we can “sort by perfect shuffles” as in Section 5.3.4, Fig. 57, if we can permute bytes between  $x$  and  $y$  appropriately. And such permutation is easy, by exercise 1. [Of course there are much simpler and faster ways to sort 16 bytes. But see S. Albers and T. Hagerup, *Inf. and Computation* **136** (1997), 25–51, and M. Thorup, *J. Algorithms* **42** (2002), 205–230, for asymptotic implications of this approach.]

**106.** The  $n$  bits are regarded as  $g$  fields of  $g$  bits each. First the nonzero fields are detected ( $t_1$ ), and we form a word  $y$  that has  $(y_{g-1} \dots y_0)_2$  in each  $g$ -bit field, where  $y_j = [\text{field } j \text{ of } x \text{ is nonzero}]$ . Then we compare each field with the constants  $2^{g-1}, \dots, 2^0$  ( $t_2$ ), and form a mask  $m$  that identifies the most significant nonzero field of  $x$ . After putting  $g$  copies of that field into  $z$ , we test  $z$  as we tested  $y$  ( $t_3$ ). Finally an appropriate sideways addition of  $t_2$  and  $t_3$  ( $g$ -bit-wise) yields  $\lambda$ . (Try the case  $g = 4$ ,  $n = 16$ .)

To compute  $2^\lambda$  without shifting left, replace ‘ $t_2 \ll 1$ ’ by ‘ $t_2 + t_2$ ’, and replace the final line by  $w \leftarrow (((a \cdot (t_3 \oplus (t_3 \gg g))) \bmod 2^n) \gg (n - g)) \cdot l$ ; then  $w \& m$  is  $2^{\lambda x}$ .

```

107. h  GREG #8000800080008000    SLU  q,t,16    OR   t,t,y
      ms GREG #00ff0f0f33335555    ADDU t,t,q    AND  t,t,h
      1H SRU  q,x,32                SLU  q,t,32    5H SLU  q,t,15
      ZSNZ lam,q,32                ADDU t,t,q    ADDU t,t,q
      ADD  t,lam,16                3H ANDN y,t,ms    SLU  q,t,30
      SRU  q,x,t                  4H XOR  t,t,y    ADDU t,t,q
      CSNZ lam,q,t                OR   q,y,h    6H SRU  q,t,60
      2H SRU  t,x,lam              SUBU t,q,t    ADDU lam,lam,q █

```

The total time is  $22v$  (and no mems). [There’s also a mem-less version of (56), costing only  $16v$ , if its last line is replaced by `ADD t,lam,4; SRU y,x,t; CSNZ lam,y,t; SRU y,x,lam; SLU t,y,1; SRU t,[#ffffaa50],t; AND t,t,3; ADD lam,lam,t.`]

**108.** For example, let  $e$  be minimum so that  $n \leq 2^e \cdot 2^{2^e}$ . If  $n$  is a multiple of  $2^e$ , we can use  $2^e$  fields of size  $n/2^e$ , with  $e$  reductions in step B1; otherwise we can use  $2^e$  fields of size  $2^{\lceil \lg n \rceil - e - 1}$ , with  $e + 1$  reductions in step B1. In either case there are  $e$  iterations in steps B2 and B5, so the total running time is  $O(e) = O(\log \log n)$ .

**109.** Start with  $x \leftarrow x \& -x$  and apply Algorithm B. (Step B4 of that algorithm can be slightly simplified in this special case, using a constant  $l$  instead of  $x \oplus y$ .)

**110.** Let  $s = 2^d$  where  $d = 2^e - e$ . We will use  $s$ -bit fields in  $n$ -bit words.

**K1.** [Stretch  $x \bmod s$ .] Set  $y \leftarrow x \& (s - 1)$ . Then set  $t \leftarrow y \& \bar{\mu}_j$  and  $y \leftarrow y \oplus t \oplus (t \ll 2^j(s - 1))$  for  $e > j \geq 0$ . Finally set  $y \leftarrow (y \ll s) - y$ . [If  $x = (x_{2^e-1} \dots x_0)_2$  we now have  $y = (y_{2^e-1} \dots y_0)_{2^s}$ , where  $y_j = (2^s - 1)x_j[j < d]$ .]

**K2.** [Set up minterms.] Set  $y \leftarrow y \oplus (a_{2^e-1} \dots a_0)_{2^s}$ , where  $a_j = \mu_{d,j}$  for  $0 \leq j < d$  and  $a_j = 2^s - 1$  for  $d \leq j < 2^e$ .

**K3.** [Compress.] Set  $y \leftarrow y \& (y \gg 2^j s)$  for  $e > j \geq 0$ . [Now  $y = 1 \ll (x \bmod s)$ . This is the key point that makes the algorithm work.]

leap year  
table lookup by shifting  
bitwise min and max  
perfect shuffles  
Albers  
Hagerup  
Thorup  
sideways addition  
CSNZ  
ZSNZ  
table lookup via shifting  
Stretch  
magic

**K4.** [Finish.] Set  $y \leftarrow y \mid (y \ll 2^j s)$  for  $0 \leq j < e$ . Finally set  $y \leftarrow y \& (\mu_{2^e, j} \oplus -((x \gg j) \& 1))$  for  $d \leq j < 2^e$ . ■

**111.** The  $n$  bits are divided into fields of  $s$  bits each, although the leftmost field might be shorter. First  $y$  is set to flag the all-1 fields. Then  $t = (\dots t_1 t_0)_{2^s}$  contains candidate bits for  $q$ , including “false drops” for certain patterns  $01^k$  with  $s \leq k < r$ . We always have  $\nu t_j \leq 1$ , and  $t_j \neq 0$  implies  $t_{j-1} = 0$ . The bits of  $u$  and  $v$  subdivide  $t$  into two parts so that we can safely compute  $m = (t \gg 1) \mid (t \gg 2) \mid \dots \mid (t \gg r)$ , before making a final test to eliminate the false drops.

**112.** Notice that if  $q = x \& (x \ll 1) \& \dots \& (x \ll (r-1)) \& \sim(x \ll r)$  then we have  $x \& \overline{x+q} = x \& (x \ll 1) \& \dots \& (x \ll (r-1))$ .

If we can solve the stated problem in  $O(1)$  steps, we can also extract the most significant bit of an  $r$ -bit number in  $O(1)$  steps: Apply the case  $n = 2r$  to the number  $2^n - 1 - x$ . Conversely, a solution to the extraction problem can be shown to yield a solution to the  $1^r 0$  problem. Exercise 110 therefore implies a solution in  $O(\log \log r)$  steps.

**113.** Let  $0' = 0$ ,  $x'_0 = x_0$ , and construct  $x'_{i'} = x_i$  for  $1 \leq i \leq r$  as follows: If  $x_i = a \circ_i b$  and  $\circ_i \notin \{+, -, \ll\}$ , let  $i' = (i-1)' + 1$  and  $x'_{i'} = a' \circ_i b'$ , where  $a' = x'_{j'}$  if  $a = x_j$  and  $a' = a$  if  $a = c_i$ . If  $x_i = a \ll c$ , let  $i' = (i-1)' + 2$  and  $(x'_{i'-1}, x'_{i'}) = (a' \& (\lceil 2^{n-c} \rceil - 1), x'_{i'-1} \ll c)$ . If  $x_i = a + b$ , let  $i' = (i-1)' + 6$  and let  $(x'_{(i-1)'+1}, \dots, x'_{i'})$  compute  $((a' \& \bar{h}) + (b' \& \bar{h})) \oplus ((a' \oplus b') \& h)$ , where  $h = 2^{n-1}$ . And if  $x_i = a - b$ , do the similar computation  $((a' \mid h) - (b' \& \bar{h})) \oplus ((a' \equiv b') \& h)$ . Clearly  $r' \leq 6r$ .

**114.** Simply let  $X_i = X_{j(i)} \circ_i X_{k(i)}$  when  $x_i = x_{j(i)} \circ_i x_{k(i)}$ ,  $X_i = C_i \circ_i X_{k(i)}$  when  $x_i = c_i \circ_i x_{k(i)}$ , and  $X_i = X_{j(i)} \circ_i C_i$  when  $x_i = x_{j(i)} \circ_i c_i$ , where  $C_i = c_i$  when  $c_i$  is a shift amount, otherwise  $C_i = (c_i \dots c_i)_{2^n} = (2^{mn} - 1)c_i / (2^n - 1)$ . This construction is possible thanks to the fact that variable-length shifts are prohibited.

[Notice that if  $m = 2^d$ , we can use this idea to simulate  $2^d$  instances of  $f(x, y_i)$ ; then  $O(d)$  further operations allow “quantification.”]

**115.** (a)  $z \leftarrow (\bar{x} \ll 1) \& (x \ll 2)$ ,  $y \leftarrow x \& (x + z)$ . [This problem was posed to the author by Vaughan Pratt in 1977.]

(b) First find  $x_l \leftarrow (x \ll 1) \& \bar{x}$  and  $x_r \leftarrow x \& (\bar{x} \ll 1)$ , the left and right ends of  $x$ ’s blocks; and set  $x'_r \leftarrow x_r \& (x_r - 1)$ . Then  $z_e \leftarrow x'_r \& (x'_r - (x_l \& \bar{\mu}_0))$  and  $z_o \leftarrow x'_r \& (x'_r - (x_l \& \mu_0))$  are the right ends that are followed by a left end in even or odd position, respectively. The answer is  $y \leftarrow x \& (x + (z_e \& \bar{\mu}_0) + (z_o \& \mu_0))$ ; it can be simplified to  $y \leftarrow x \& (x + (z_e \oplus (x'_r \& \mu_0)))$ .

(c) This case is impossible, by Corollary I.

**116.** The language  $L$  is well defined, by Lemma A (except that the presence or absence of the empty string is irrelevant). A language is regular if and only if it can be defined by a finite-state automaton, and a 2-adic integer is rational if and only if it can be defined by a finite-state automaton that ignores its inputs. The identity function corresponds to the language  $L = 1(0 \cup 1)^*$ , and a simple construction will define an automaton that corresponds to the sum, difference, or Boolean combination of the numbers defined by any two given automata acting on the sequence  $x_0 x_1 x_2 \dots$ . Hence  $L$  is regular.

In exercise 115,  $L$  is (a)  $11^*(000^*1(0 \cup 1)^* \cup 0^*)$ ; (b)  $11^*(00(00)^*1(0 \cup 1)^* \cup 0^*)$ .

**117.** Incidentally, the stated language  $L$  corresponds to an inverse Gray binary code: It defines a function with the property that  $f(2x) = \sim f(2x + 1)$ , and  $g(f(2x)) = g(f(2x + 1)) = x$ , where  $g(x) = x \oplus (x \gg 1)$  (see Eq. 7.2.1.1-(g)).

**118.** If  $x = (x_{n-1} \dots x_1 x_0)_2$  and  $0 \leq a_j \leq 2^j$  for  $0 \leq j < n$ , we have  $\sum_{j=0}^{n-1} a_j x_j = \sum_{j=0}^{n-1} (a_j \dot{-} (\bar{x} \& 2^j))$ . Take  $a_j = \lfloor 2^{j-1} \rfloor$  to get  $x \gg 1$ .

extract the most significant bit  
quantification  
Pratt  
finite-state automaton  
Gray binary code

Paterson  
underflow

Conversely, the following argument by M. S. Paterson proves that monus must be used at least  $n - 1$  times: Consider any chain for  $f(x)$  that uses addition, subtraction, bitwise Booleans, and  $k$  occurrences of the “underflow” operation  $y \triangleleft z = (2^n - 1)[y < z]$ . If  $k < n - 1$  there must be two  $n$ -bit numbers  $x'$  and  $x''$  such that  $x' \bmod 2 = x'' \bmod 2 = 0$  and such that all  $k$  of the  $\triangleleft$ 's yield the same result for both  $x'$  and  $x''$ . Then  $f(x') \bmod 2^j = f(x'') \bmod 2^j$  when  $j = \rho(x' \oplus x'')$ . So  $f(x)$  is not the function  $x \gg 1$ .

**119.**  $z \leftarrow x \oplus y$ ,  $f \leftarrow 2^p \& \bar{z} \& (z - 1)$ . (See (90).)

**120.** Generalizing Corollary W, these are the functions such that  $f(x_1, \dots, x_m) \equiv f(y_1, \dots, y_m) \pmod{2^k}$  whenever  $x_j \equiv y_j \pmod{2^k}$  for  $1 \leq j \leq m$ , for  $0 \leq k \leq n$ . The least significant bit is a binary function of  $m$  variables, so it has  $2^{2^m}$  possibilities. The next-to-least is a binary function of  $2m$  variables, namely the bits of  $(x_1 \bmod 4, \dots, x_m \bmod 4)$ , so it has  $2^{2^{2m}}$ ; and so on. Thus the answer is  $2^{2^m + 2^{2m} + \dots + 2^{nm}}$ .

**121.** (a) If  $f$  has a period of length  $pq$ , where  $q > 1$  is odd, its  $p$ -fold iteration  $f^{[p]}$  has a period of length  $q$ , say  $y_0 \mapsto y_1 \mapsto \dots \mapsto y_q = y_0$  where  $y_{j+1} = f^{[p]}(y_j)$  and  $y_1 \neq y_0$ . But then, by Corollary W, we must have  $y_0 \bmod 2^{n-1} \mapsto y_1 \bmod 2^{n-1} \mapsto \dots \mapsto y_q \bmod 2^{n-1}$  in the corresponding  $(n - 1)$ -bit chain. Consequently  $y_1 \equiv y_0 \pmod{2^{n-1}}$ , by induction on  $n$ . Hence  $y_1 = y_0 \oplus 2^{n-1}$ , and  $y_2 = y_0$ , etc., a contradiction.

(b)  $x_1 = x_0 + x_0$ ,  $x_2 = x_0 \gg (p - 1)$ ,  $x_3 = x_1 \mid x_2$ ; a period of length  $p$  starts with the value  $x_0 = (1 + 2^p + 2^{2p} + \dots) \bmod 2^n$ .

**122.** Subtraction is analogous to addition; Boolean operations are even simpler; and constants have only one bit pattern. The only remaining case is  $x_r = x_j \gg c$ , where we have  $S_r = S_j + c$ ; the shift goes left when  $c < 0$ . Then  $V_{pqr} = V_{(p+c)(q+c)j}$ , and

$$x_r \& [2^p - 2^q] = ((x_j \& [2^{p+c} - 2^{q+c}]) \gg c) \& (2^n - 1).$$

Hence  $|X_{pqr}| \leq |X_{(p+c)(q+c)j}| \leq B_j = B_r$  by induction.

**123.** If  $x = (x_{g-1} \dots x_0)_2$ , note first that  $t = 2^{g-1}(x_0 \dots x_{g-1})_{2^g}$  in (104); hence  $y = (x_0 \dots x_{g-1})_2$  as claimed. Theorem P now implies that  $\lfloor \frac{1}{3} \lg g \rfloor$  broadword steps are needed to multiply by  $a_{g+1}$  and by  $a_{g-1}$ . At least one of those multiplications must require  $\lfloor \frac{1}{6} \lg g \rfloor$  or more steps.

**124.** Initially  $t \leftarrow 0$ ,  $x_0 = x$ ,  $U_0 = \{1, 2, \dots, 2^{n-1}\}$ , and  $i' \leftarrow 0$ . When advancing  $t \leftarrow t + 1$ , if the current instruction is  $r_i \leftarrow r_j \pm r_k$  we simply define  $x_t = x_{j'} \pm x_{k'}$  and  $i' \leftarrow t$ . The cases  $r_i \leftarrow r_j \circ r_k$  and  $r_i \leftarrow c$  are similar.

If the current instruction branches when  $r_i \leq r_j$ , define  $x_t = x_{t-1}$  and let  $V_1 = \{x \in U_{t-1} \mid x_{i'} \leq x_{j'}\}$ ,  $V_0 = U_t \setminus V_1$ . Let  $U_t$  be the larger of  $V_0$  and  $V_1$ ; branch if  $U_t = V_1$ . Notice that  $|U_t| \geq |U_{t-1}|/2$  in this case.

If the current instruction is  $r_i \leftarrow r_j \gg r_k$ , let  $W = \{x \in U_{t-1} \mid x \& [2^{\lg n + s} - 2^s] \neq 0 \text{ for some } s \in S_{k'}\}$ , and note that  $|W| \leq |S_{k'}| \lg n \leq 2^{t-1+e+f}$ . Let  $V_c = \{x \in U_{t-1} \setminus W \mid x_{k'} = c\}$  for  $|c| < n$ , and  $V_n = U_{t-1} \setminus W \setminus \bigcup_{|c| < n} V_c$ . Lemma B tells us that at most  $B_{k'} + 1 \leq 2^{2^{t-1}-1} + 1$  of the sets  $V_c$  are nonempty. Let  $U_t$  be the largest; and if it is  $V_c$ , define  $x_t = x_{j'} \gg c$ ,  $i' \leftarrow t$ . In this case  $|U_t| \geq (|U_{t-1}| - 2^{t-1+e+f}) / (2^{2^{t-1}-1} + 1)$ .

Similarly for  $r_i \leftarrow M[r_j \bmod 2^m]$  or  $M[r_j \bmod 2^m] \leftarrow r_i$ , let  $W = \{x \in U_{t-1} \mid x \& [2^{m+s} - 2^s] \neq 0 \text{ for some } s \in S_{j'}\}$ , and  $V_z = \{x \in U_{t-1} \setminus W \mid x_{j'} \bmod 2^m = z\}$ , for  $0 \leq z < 2^m$ . By Lemma B, at most  $B_{j'} \leq 2^{2^{t-1}-1}$  of the sets  $V_z$  are nonempty; let  $U_t = V_z$  be the largest. To write  $r_i$  in  $M[z]$ , define  $x_t = x_{t-1}$ ,  $z'' \leftarrow i'$ ; to read  $r_i$  from  $M[z]$ , set  $i' \leftarrow t$  and put  $x_t = x_{z''}$  if  $z''$  is defined, otherwise let  $x_t$  be the precomputed constant  $M[z]$ . In both cases  $|U_t| \geq (|U_{t-1}| - 2^{t-1}m) / 2^{2^{t-1}-1}$  is sufficiently large.

If  $t < f$  we cannot be sure that  $r_1 = \rho x$ . The reason is that the set  $W = \{x \in U_t \mid x \& [2^{\lg n + s} - 2^s] \neq 0 \text{ for some } s \in S_{1'}\}$  has size  $|W| \leq |S_{1'}| \lg n \leq 2^{t+e+f}$ ,

and  $|U_t \setminus W| \geq 2^{2^{e+f}-2^t+1} - 2^{t+e+f} > 2^{2^t-1} \geq |\{x_1, \dots, \&[2^{\lg n} - 1] \mid x_0 \in U_t \setminus W\}|$ . Two elements of  $U_t \setminus W$  cannot have the same value of  $\rho x = x_1, \dots, \&[2^{\lg n} - 1]$ .

[The same lower bound applies even if we allow the RAM to make arbitrary  $2^{2^t-1}$ -way branches based on the contents of  $(r_1, \dots, r_l)$  at time  $t$ .]

**125.** Start as in answer 124, but with  $U_0 = [0..2^g]$ . Simplifying that argument by eliminating the sets  $W$  will yield sets such that  $|U_t| \geq 2^g / \max(2^m, 2n)^t$ ; for example, at most  $2n$  different shift instructions can occur.

Suppose we can stop at time  $t$  with  $t < \lfloor \lg(h+1) \rfloor$ . The proof of Theorem P yields  $p$  and  $q$  with  $x^R \&[2^p - 2^q]$  independent of  $x \&[2^{p+s} - 2^{q+s}]$ . Hence the hinted extension of Lemma B shows that  $x^R$  takes on at most  $2^{2^t-1} \leq 2^{(h-1)/2}$  different values, for every setting of the other bits  $\{x \&[2^{p+s} - 2^{q+s}] \mid s \in S_t\}$ . Consequently  $r_1 = x_1$  can be the correct value of  $x^R$  for at most  $2^{(h-1)/2+g-h}$  values of  $x$ . But  $2^{(h-1)/2+g-h}$  is less than  $|U_t|$ , by (106).

**126.** M. S. Paterson has proposed a related (but different) conjecture: For every 2-adic chain with  $k$  addition-subtraction operations, there is a (possibly huge) integer  $x$  with  $\nu x = k+1$  such that the chain does not calculate  $2^{\lambda x}$ .

**127.** Johan Håstad [*Advances in Computing Research* **5** (1989), 143–170] has shown that every polynomial-size circuit that computes the parity function from the inputs  $\{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$  with AND and OR gates of unlimited fanin must have depth  $\Omega(\log n / \log \log n)$ .

**128.** (Note also that the suffix parity function  $x^\oplus$  is considered in exercises 36 and 117.)

**130.** If the answer is “no,” the analogous question with *variable*  $a$  suggests itself.

**131.** This program does a typical “breadth-first search,” keeping  $\text{LINK}(q) = r$ . Register  $u$  is the vertex currently being examined;  $v$  is one of its successors.

```

0H LDOU r,q,link    1   r ← LINK(q).      STOU v,q,link    |R|−|Q|  LINK(q) ← v.
   SET u,r          1   u ← r.             STOU r,v,link    |R|−|Q|  LINK(v) ← r.
1H LDOU a,u,arcs    |R|  a ← ARCS(u).      SET q,v          |R|−|Q|  q ← v.
   BZ a,4F          |R|  Is S[u] = ∅?      3H PBNZ a,2B      S      Loop on a.
2H LDOU v,a,tip     S    v ← TIP(a).      4H LDOU u,u,link    |R|    u ← LINK(u).
   LDOU a,a,next     S    a ← NEXT(a).     CMPU t,u,r        |R|    Is u ≠ r?
   LDOU t,v,link     S    t ← LINK(v).     PBNZ t,1B         |R|    If so, continue.
   PBNZ t,3F         S    Is v ∈ R?

```

**132.** (a) We always have  $\tau(U) \subseteq \&_{u \notin U} \delta_u = \sigma(U)$ . And equality holds if and only if  $2^u \subseteq \rho(u')$  for all  $u \in U$  and  $u' \in U$ .

(b) We’ve proved that  $\tau(U) \subseteq \sigma(U)$ ; hence  $T \subseteq U$ . And if  $t \in T$  we have  $2^t \subseteq \rho_u$  for all  $u \in U$ . Therefore  $\sigma(T) \subseteq \tau(T)$ .

(c) Parts (a) and (b) prove that the elements of  $C_n$  represent the cliques.

(d) If  $u \subseteq v$  then  $u \& \rho_k \subseteq v \& \rho_k$  and  $u \& \delta_k \subseteq v \& \delta_k$ ; so we can work entirely with maximal entries. The following algorithm uses cache-friendly sequential (rather than linked) allocation, in a manner analogous to radix exchange sort (Algorithm 5.2.2R).

We assume that  $w_1 \dots w_s$  is a workspace of  $s$  unsigned words, bounded by  $w_0 = 0$  and  $w_{s+1} = 2^n - 1$ . The elements of  $C_{k-1}^+$  appear initially in positions  $w_1 \dots w_m$ , and our goal is to replace them by the elements of  $C_k^+$ .

**M1.** [Initialize.] Terminate if  $\rho_k = 2^n - 1$ . Otherwise set  $v \leftarrow 2^k$ ,  $i \leftarrow 1$ ,  $j \leftarrow m$ .

RAM  
Paterson  
2-adic chain  
Håstad  
suffix parity function  
breadth-first search  
cache-friendly  
sequential  
linked  
radix exchange sort



**M2.** [Partition on  $v$ .] While  $w_i \& v = 0$ , set  $i \leftarrow i + 1$ . While  $w_j \& v \neq 0$ , set  $j \leftarrow j - 1$ . Then if  $i > j$ , go to M3; otherwise swap  $w_i \leftrightarrow w_j$ , set  $i \leftarrow i + 1$ ,  $j \leftarrow j - 1$ , and repeat this step.

**M3.** [Split  $w_i \dots w_m$ .] Set  $l \leftarrow j$ ,  $p \leftarrow s + 1$ . While  $i \leq m$ , do subroutine Q with  $u = w_i$  and set  $i \leftarrow i + 1$ .

**M4.** [Combine maximal elements.] Set  $m \leftarrow l$ . While  $p \leq s$ , set  $m \leftarrow m + 1$ ,  $w_m \leftarrow w_p$ , and  $p \leftarrow p + 1$ . ■

Subroutine Q uses global variables  $j$ ,  $k$ ,  $l$ ,  $p$ , and  $v$ . It essentially replaces the word  $u$  by  $u' = u \& \rho_k$  and  $u'' = u \& \delta_k$ , retaining them if they are still maximal. If so,  $u'$  goes into the upper workspace  $w_p \dots w_s$  but  $u''$  stays below.

**Q1.** [Examine  $u'$ .] Set  $w \leftarrow u \& \rho_k$  and  $q \leftarrow s$ . If  $w = u$ , go to Q4.

**Q2.** [Is it comparable?] If  $q < p$ , go to Q3. Otherwise if  $w \& w_q = w$ , go to Q7. Otherwise if  $w \& w_q = w_q$ , go to Q4. Otherwise set  $q \leftarrow q - 1$  and repeat Q2.

**Q3.** [Tentatively accept  $u'$ .] Set  $p \leftarrow p - 1$  and  $w_p \leftarrow w$ . Memory overflow occurs if  $p \leq m + 1$ . Otherwise go to Q7.

**Q4.** [Prepare for loop.] Set  $r \leftarrow p$  and  $w_{p-1} \leftarrow 0$ .

**Q5.** [Remove nonmaximals.] While  $w \mid w_q \neq w$ , set  $q \leftarrow q - 1$ . While  $w \mid w_r = w$ , set  $r \leftarrow r + 1$ . Then if  $q < r$ , go to Q6; otherwise set  $w_q \leftarrow w_r$ ,  $w_r \leftarrow 0$ ,  $q \leftarrow q - 1$ ,  $r \leftarrow r + 1$ , and repeat this step.

**Q6.** [Reset  $p$ .] Set  $w_q \leftarrow w$  and  $p \leftarrow q$ . Terminate the subroutine if  $w = u$ .

**Q7.** [Examine  $u''$ .] Set  $w \leftarrow u \& \bar{v}$ . If  $w = w_q$  for some  $q$  in the range  $1 \leq q \leq j$ , do nothing. Otherwise set  $l \leftarrow l + 1$  and  $w_l \leftarrow w$ . ■

In practice this algorithm performs reasonably well; for example, when it is applied to the  $8 \times 8$  queen graph (exercise 7-129), it finds the 310 maximal cliques after 306,513 mems of computation, using 397 words of workspace. It finds the 10188 maximal independent sets of that same graph after about 310 megamems, using 15090 words; there are respectively (728, 6912, 2456, 92) such sets of sizes (5, 6, 7, 8), including the 92 famous solutions to the eight queens problem.

*Reference:* N. Jardine and R. Sibson, *Mathematical Taxonomy* (Wiley, 1971), Appendix 5. Many other algorithms for listing maximal cliques have also been published. See, for example, W. Knödel, *Computing* **3** (1968), 239-240, **4** (1969), 75; C. Bron and J. Kerbosch, *CACM* **16** (1973), 575-577; S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, *SICOMP* **6** (1977), 505-517; E. Loukakis, *Computers and Math. with Appl.* **9** (1983), 583-589; D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *Inf. Proc. Letters* **27** (1988), 119-123. See also exercise 5-23.

**133.** (a) An independent set is a clique of  $\bar{G}$ ; so complement  $G$ . (b) A vertex cover is the complement of an independent set; so complement  $G$ , then complement the outputs.

**134.**  $a \mapsto 00$ ,  $b \mapsto 01$ ,  $c \mapsto 11$  is the first mapping of class II.

**135.** The unary operators are simple:  $\neg(x_l x_r) = \bar{x}_r \bar{x}_l$ ;  $\diamond(x_l x_r) = x_r x_r$ ;  $\boxplus(x_l x_r) = x_l x_l$ . And  $x_l x_r \leftrightarrow y_l y_r = (z_l \wedge z_r)(z_l \vee z_r)$ , where  $z_l = (x_l \equiv y_l)$  and  $z_r = (x_r \equiv y_r)$ .

**136.** (a) Classes II, III,  $IV_a$ , and  $IV_c$  all have the optimum cost 4. Curiously the functions  $z_l = x_l \vee y_l \vee (x_r \wedge y_r)$ ,  $z_r = x_r \vee y_r$  work for the mapping  $(a, b, c) \mapsto (00, 01, 11)$  of class II as well as for the mapping  $(a, b, c) \mapsto (00, 01, 1*)$  of class  $IV_c$ . [This operation is equivalent to saturating addition, when  $a = 0$ ,  $b = 1$ , and  $c$  stands for "more than 1."] (b) The symmetry between  $a$ ,  $b$ , and  $c$  implies that we need only try classes I,  $IV_a$ , and  $V_a$ ; and those classes turn out to cost 6, 7, and 8. One winner for class I, with

overflow  
queen graph  
maximal independent sets  
eight queens problem  
Jardine  
Sibson  
Knödel  
Bron  
Kerbosch  
Tsukiyama  
Ide  
Ariyoshi  
Shirakawa  
Loukakis  
Johnson  
Yannakakis  
Papadimitriou  
complement  
complement  
saturating addition

$(a, b, c) \mapsto (00, 01, 10)$ , is  $z_l = v_r \wedge \bar{u}_l$ ,  $z_r = v_l \wedge \bar{u}_r$ , where  $u_l = x_l \oplus y_l$ ,  $u_r = x_r \oplus y_r$ ,  $v_l = y_r \oplus u_l$ , and  $v_r = y_l \oplus u_r$ . [See exercise 7.1.2–60, which gives the same answer but with  $z_l \leftrightarrow z_r$ . The reason is that we have  $(x + y + z) \bmod 3 = 0$  in this problem but  $(x + y - z) \bmod 3 = 0$  in that one; and  $z_l \leftrightarrow z_r$  is equivalent to negation. The binary operation  $z = x \circ y$  in this case can also be characterized by the fact that the elements  $(x, y, z)$  are all the same or all different; thus it is familiar to people who play the game of SET. It is the only binary operation on  $n$ -element sets that has  $n!$  automorphisms and differs from the trivial examples  $x \circ y = x$  or  $x \circ y = y$ .]

(c) Cost 3 is achieved only with class I: Let  $(a, b, c) \mapsto (00, 01, 10)$  and  $z_l = (x_l \vee x_r) \wedge y_l$ ,  $z_r = \bar{x}_r \wedge y_r$ .

**137.** In fact,  $z = (x + 1) \& y$  when  $(a, b, c) \mapsto (00, 01, 10)$ . [It's a contrived example.]

**138.** The simplest case known to the author requires the calculation of *two* binary operations, such as

$$\begin{pmatrix} a & b & b \\ a & b & b \\ c & a & a \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a & b & a \\ a & b & a \\ c & a & c \end{pmatrix};$$

each has cost 2 in class  $V_a$ , but the costs are (3, 2) and (2, 3) in classes I and II.

**139.** The calculation of  $z_2$  is essentially equivalent to exercise 136(b); so the natural representation (111) wins. Fortunately this representation also is good for  $z_1$ , with  $z_{1l} = x_l \wedge y_l$ ,  $z_{1r} = x_r \wedge y_r$ .

**140.** With representation (111), first use full binary adders to compute  $(a_1 a_0)_2 = x_l + y_l + z_l$  and  $(b_1 b_0)_2 = x_r + y_r + z_r$  in  $5 + 5 = 10$  steps. Now the “greedy footprint” method shows how to compute the four desired functions of  $(a_1, a_0, b_1, b_0)$  in eight further steps:  $u_l = a_1 \wedge \bar{b}_0$ ,  $u_r = a_0 \wedge \bar{b}_1$ ;  $t_1 = a_1 \oplus b_0$ ,  $t_2 = a_0 \oplus b_1$ ,  $t_3 = a_1 \oplus t_2$ ,  $t_4 = a_0 \oplus t_1$ ,  $v_l = t_3 \wedge \bar{t}_1$ ,  $v_r = t_4 \wedge \bar{t}_2$ . [Is this method optimum?]

**141.** Suppose we've computed bits  $a = a_0 a_1 \dots a_{2m-1}$  and  $b = b_0 b_1 \dots b_{2m-1}$  such that

$a_s = [s = 1 \text{ or } s = 2 \text{ or } s \text{ is a sum of distinct Ulam numbers } \leq m \text{ in exactly one way}]$ ,

$b_s = [s \text{ is a sum of distinct Ulam numbers } \leq m \text{ in more than one way}]$ ,

for some integer  $m = U_n \geq 2$ . For example, when  $m = n = 2$  we have  $a = 0111$  and  $b = 0000$ . Then  $\{s \mid s \leq m \text{ and } a_s = 1\} = \{U_1, \dots, U_n\}$ ; and  $U_{n+1} = \min\{s \mid s > m \text{ and } a_s = 1\}$ . (Notice that  $a_s = 1$  when  $s = U_{n-1} + U_n$ .) The following simple bitwise operations preserve these conditions:  $n \leftarrow n + 1$ ,  $m \leftarrow U_n$ , and

$$(a_m \dots a_{2m-1}, b_m \dots b_{2m-1}) \leftarrow ((a_m \dots a_{2m-1} \oplus a_0 \dots a_{m-1}) \& \overline{b_m \dots b_{2m-1}}, \\ (a_m \dots a_{2m-1} \& a_0 \dots a_{m-1}) \mid b_m \dots b_{2m-1}),$$

where  $a_s = b_s = 0$  for  $2U_{n-1} \leq s < 2U_n$  on the right side of this assignment.

[See M. C. Wunderlich, *BIT* **11** (1971), 217–224; *Computers in Number Theory* (1971), 249–257. These mysterious numbers, which were first defined by S. Ulam in *SIAM Review* **6** (1964), 348, have baffled number theorists for many years. The ratio  $U_n/n$  appears to converge to a constant,  $\approx 13.52$ ; for example,  $U_{200000000} = 270371127$  and  $U_{400000000} = 540752349$ . Furthermore, D. W. Wilson has observed empirically that the numbers form quasi-periodic “clusters” whose centers differ by multiples of another constant,  $\approx 21.6016$ . Calculations by Jud McCranie and the author for  $U_n < 640000000$  indicate that the largest gap  $U_n - U_{n-1}$  may occur between  $U_{24576523} = 332250401$  and  $U_{24576524} = 332251032$ ; the smallest gap  $U_n - U_{n-1} = 1$  apparently occurs only when  $U_n \in \{2, 3, 4, 48\}$ . Certain small gaps like 6, 11, 14, and 16 have never been observed.]

SET  
greedy footprint  
footprints  
Wunderlich  
Ulam  
Wilson  
McCranie  
Knuth  
gap

**142.** Algorithm E in that exercise performs the following operations on subcubes: (i) Count the \*s in a given subcube  $c$ . (ii) Given  $c$  and  $c'$ , test if  $c \subseteq c'$ . (iii) Given  $c$  and  $c'$ , compute  $c \sqcup c'$  (if it exists). Operation (i) is simple with sideways addition; let's see which of the nine classes of two-bit encodings (119), (123), (124) works best for (ii) and (iii). Suppose  $a = 0$ ,  $b = 1$ ,  $c = *$ ; the symmetry between 0 and 1 means that we need only examine classes I, III, IV<sub>a</sub>, IV<sub>c</sub>, V<sub>a</sub>, and V<sub>c</sub>.

For the asterisks-and-bits mapping  $(0, 1, *) \mapsto (00, 01, 10)$ , which belongs to class I, the truth table for  $c \not\subseteq c'$  is 010\*100\*110\*\*\*\*\* in each component. (For example,  $0 \subseteq *$  and  $* \not\subseteq 1$ . The \*s in this truth table are don't-cares for the unused codes 11.) The methods of Section 7.1.2 tell us that the cheapest such functions have cost 3; for example,  $c \subseteq c'$  if and only if  $((b \oplus b') \mid a) \& \bar{a}' = 0$ . Furthermore the consensus  $c \sqcup c' = c''$  exists if and only if  $\nu z = 1$ , where  $z = (b \oplus b') \& \sim(a \oplus a')$ . And in that case,  $a'' = (a \oplus b \oplus b') \& \sim(a \oplus a')$ ,  $b'' = (b \mid b') \& \bar{z}$ . [The asterisk and bit codes were used for this purpose by M. A. Breuer in *Proc. ACM Nat. Conf.* **23** (1968), 241–250.]

But class III works out better, with  $(0, 1, *) \mapsto (01, 10, 00)$ . Then  $c \subseteq c'$  if and only if  $(\bar{c}_l \& c'_l) \mid (\bar{c}_r \& c'_r) = 0$ ;  $c \sqcup c' = c''$  exists if and only if  $\nu z = 1$  where  $z = x \& y$ ,  $x = c_l \mid c'_l$ ,  $y = c_r \mid c'_r$ ; and  $c''_l = x \oplus z$ ,  $c''_r = y \oplus z$ . We save two operations for each consensus, with respect to class I, compensating for an extra step when counting asterisks.

Classes IV<sub>a</sub>, V<sub>a</sub>, and V<sub>c</sub> turn out to be far inferior. Class IV<sub>c</sub> has some merit, but class III is best.

**143.**  $f(x) = ((x \& m_1) \ll 17) \mid ((x \gg 17) \& m_1) \mid ((x \& m_2) \ll 15) \mid ((x \gg 15) \& m_2) \mid ((x \& m_3) \ll 10) \mid ((x \gg 10) \& m_3) \mid ((x \& m_4) \ll 6) \mid ((x \gg 6) \& m_4)$ , where  $m_1 = \#7f7f7f7f7f7f$ ,  $m_2 = \#fefefefefefe$ ,  $m_3 = \#3f3f3f3f3f3f$ ,  $m_4 = \#fcfcfcfcfcfc$ . [See, for example, *Chess Skill in Man and Machine*, edited by Peter W. Frey (1977), page 59. Five steps suffice to compute  $f(x)$  on MMIX (four MOR operations and one OR), since  $f(x) = q \cdot x \cdot q' \mid q' \cdot x \cdot q$  with  $q = \#40a05028140a0502$  and  $q' = \#2010884422110804$ .]

**144.** Node  $j \oplus (k \ll 1)$ , where  $k = j \& -j$ .

**145.** It names the ancestor of the leaf node  $j \mid 1$  at height  $h$ .

**146.** By (136) we want to show that  $\lambda(j \& -i) = \rho l$  when  $l - 2^{\rho l} < i \leq l \leq j < l + 2^{\rho l}$ . The desired result follows from (35) because  $-l \leq -i < -l + 2^{\rho l}$ .

**147.** (a)  $\pi v_j = \beta v_j = j$ ,  $\alpha v_j = 1 \ll \rho j$ , and  $\tau j = \Lambda$ , for  $1 \leq j \leq n$ .

(b) Suppose  $n = 2^{e_1} + \dots + 2^{e_t}$  where  $e_1 > \dots > e_t \geq 0$ , and let  $n_k = 2^{e_1} + \dots + 2^{e_k}$  for  $0 \leq k \leq t$ . Then  $\pi v_j = j$  and  $\beta v_j = \alpha v_j = n_k$  for  $n_{k-1} < j \leq n_k$ . Also  $\tau n_k = v_{n_{k-1}}$  for  $1 \leq k \leq t$ , where  $v_0 = \Lambda$ ; all other  $\tau j = \Lambda$ .

**148.** Yes, if  $\pi y_1 = 010000$ ,  $\pi y_2 = 010100$ ,  $\pi x_1 = 010101$ ,  $\pi x_2 = 010110$ ,  $\pi x_3 = 010111$ ,  $\beta x_3 = 010111$ ,  $\beta y_2 = 010100$ ,  $\beta x_2 = 011000$ ,  $\beta y_1 = 010000$ , and  $\beta x_1 = 100000$ .

**149.** We assume that  $\text{CHILD}(v) = \text{SIB}(v) = \text{PARENT}(v) = \Lambda$  initially for all vertices  $v$  (including  $v = \Lambda$ ), and that there is at least one nonnull vertex.

**S1.** [Make triply linked tree.] For each of the  $n$  arcs  $u \rightarrow v$  (perhaps  $v = \Lambda$ ), set  $\text{SIB}(u) \leftarrow \text{CHILD}(v)$ ,  $\text{CHILD}(v) \leftarrow u$ ,  $\text{PARENT}(u) \leftarrow v$ . (See exercise 2.3.3–6.)

**S2.** [Begin first traversal.] Set  $p \leftarrow \text{CHILD}(\Lambda)$ ,  $n \leftarrow 0$ , and  $\lambda 0 \leftarrow -1$ .

**S3.** [Compute  $\beta$  in the easy case.] Set  $n \leftarrow n + 1$ ,  $\pi p \leftarrow n$ ,  $\tau n \leftarrow \Lambda$ , and  $\lambda n \leftarrow 1 + \lambda(n \gg 1)$ . If  $\text{CHILD}(p) \neq \Lambda$ , set  $p \leftarrow \text{CHILD}(p)$  and repeat this step; otherwise set  $\beta p \leftarrow n$ .

**S4.** [Compute  $\tau$ , bottom-up.] Set  $\tau \beta p \leftarrow \text{PARENT}(p)$ . Then if  $\text{SIB}(p) \neq \Lambda$ , set  $p \leftarrow \text{SIB}(p)$  and return to S3; otherwise set  $p \leftarrow \text{PARENT}(p)$ .

sideways addition  
don't-cares  
Breuer  
Frey  
MOR  
triply linked tree  
traversal in preorder

- S5.** [Compute  $\beta$  in the hard case.] If  $p \neq \Lambda$ , set  $h \leftarrow \lambda(n \& -\pi p)$ , then  $\beta p \leftarrow ((n \gg h) | 1) \ll h$ , and go back to S4.
- S6.** [Begin second traversal.] Set  $p \leftarrow \text{CHILD}(\Lambda)$ ,  $\lambda 0 \leftarrow \lambda n$ ,  $\pi \Lambda \leftarrow \beta \Lambda \leftarrow \alpha \Lambda \leftarrow 0$ .
- S7.** [Compute  $\alpha$ , top-down.] Set  $\alpha p \leftarrow \alpha(\text{PARENT}(p)) | (\beta p \& -\beta p)$ . Then if  $\text{CHILD}(p) \neq \Lambda$ , set  $p \leftarrow \text{CHILD}(p)$  and repeat this step.
- S8.** [Continue to traverse.] If  $\text{SIB}(p) \neq \Lambda$ , set  $p \leftarrow \text{SIB}(p)$  and go to S7. Otherwise set  $p \leftarrow \text{PARENT}(p)$ , and repeat step S8 if  $p \neq \Lambda$ . ■

traversal in postorder  
 Cartesian trees  
 Vuillemin  
 right-to-left minimum  
 left-to-right minimum  
 triply linked tree  
 Gabow  
 Bentley  
 Tarjan  
 Fischer  
 Heun  
 sum of rho

**150.** We may assume that the elements  $A_j$  are distinct, by regarding them as ordered pairs  $(A_j, j)$ . The hinted binary search tree, which is a special case of the “Cartesian trees” introduced by Jean Vuillemin [*CACM* **23** (1980), 229–239], has the property that  $k(i, j)$  is the nearest common ancestor of  $i$  and  $j$ . Indeed, the ancestors of any given node  $j$  are precisely the nodes  $k$  such that  $A_k$  is a right-to-left minimum of  $A_1 \dots A_j$  or  $A_k$  is a left-to-right minimum of  $A_j \dots A_n$ .

The algorithm of the preceding answer does the desired preprocessing, except that we need to set up a triply linked tree differently on the nodes  $\{0, 1, \dots, n\}$ . Start as before with  $\text{CHILD}(v) = \text{SIB}(v) = \text{PARENT}(v) = 0$  for  $0 \leq v \leq n$ , and let  $\Lambda = 0$ . Assume that  $A_0 \leq A_j$  for  $1 \leq j \leq n$ . Set  $t \leftarrow 0$  and do the following steps for  $v = n, n-1, \dots, 1$ : Set  $u \leftarrow 0$ ; then while  $A_v < A_t$  set  $u \leftarrow t$  and  $t \leftarrow \text{PARENT}(t)$ . If  $u \neq 0$ , set  $\text{SIB}(v) \leftarrow \text{SIB}(u)$ ,  $\text{SIB}(u) \leftarrow 0$ ,  $\text{PARENT}(u) \leftarrow v$ ,  $\text{CHILD}(v) \leftarrow u$ ; otherwise simply set  $\text{SIB}(v) \leftarrow \text{CHILD}(t)$ . Also set  $\text{CHILD}(t) \leftarrow v$ ,  $\text{PARENT}(v) \leftarrow t$ ,  $t \leftarrow v$ .

Continue with step S2 after the tree has been built. The running time is  $O(n)$ , because the operation  $t \leftarrow \text{PARENT}(t)$  is performed at most once for each node  $t$ . [This beautiful way to reduce the range minimum query problem to the nearest common ancestor problem was discovered by H. N. Gabow, J. L. Bentley, and R. E. Tarjan, *STOC* **16** (1984), 137–138, who also suggested the following exercise.]

**151.** For node  $v$  with  $k$  children  $u_1, \dots, u_k$ , define the node sequence  $S(v) = v$  if  $k = 0$ ;  $S(v) = vS(u_1)$  if  $k = 1$ ; and  $S(v) = S(u_1)v \dots vS(u_k)$  if  $k > 1$ . (Consequently  $v$  appears exactly  $\max(k-1, 1)$  times in  $S(v)$ .) If there are  $k$  trees in the forest, rooted at  $u_1, \dots, u_k$ , write down the node sequence  $S(u_1)\Lambda \dots \Lambda S(u_k) = V_1 \dots V_N$ . (The length of this sequence will satisfy  $n \leq N < 2n$ .) Let  $A_j$  be the depth of node  $V_j$ , for  $1 \leq j \leq N$ , where  $\Lambda$  has depth 0. (For example, consider the forest  $(141)$ , but add another child  $K \rightarrow D$  and an isolated node  $L$ . Then  $V_1 \dots V_{15} = CFAGJDHDK\Lambda BEIAL$  and  $A_1 \dots A_{15} = 231342323012301$ .) The nearest common ancestor of  $u$  and  $v$ , when  $u = V_i$  and  $v = V_j$ , is then  $V_{k(i,j)}$  in the range minimum query problem. [See J. Fischer and V. Heun, *Lecture Notes in Comp. Sci.* **4009** (2006), 36–48.]

**152.** Step V1 finds the level above which  $\alpha x$  and  $\alpha y$  have bits that apply to both of their ancestors. (See exercise 148.) Step V2 increases  $h$ , if necessary, to the level where they have a common ancestor, or to the top level  $\lambda n$  if they don’t (namely if  $k = 0$ ). If  $\beta x \neq \beta z$ , step V4 finds the topmost level among  $x$ ’s ancestors that leads to level  $h$ ; hence it knows the lowest ancestor  $\hat{x}$  for which  $\beta \hat{x} = \beta z$  (or  $\hat{x} = \Lambda$ ). Finally in V5, preorder tells us which of  $\hat{x}$  or  $\hat{y}$  is an ancestor of the other.

**153.** That pointer has  $\rho j$  bits, so it ends after  $\rho 1 + \rho 2 + \dots + \rho j = j - \nu j$  bits of the packed string, by (61). [Here  $j$  is even. Navigation piles were introduced in *Nordic Journal of Computing* **10** (2003), 238–262.]

**154.** The gray lines define  $36^\circ$ - $36^\circ$ - $90^\circ$  triangles, ten of which make a pentagon with  $72^\circ$  angles at each vertex. These pentagons tile the hyperbolic plane in such a way that *five* of them meet at each vertex.

**155.** Observe first that  $0 \leq (\alpha 0)_{1/\phi} < \phi^{-1} + \phi^{-3} + \phi^{-5} + \dots = 1$ , since there are no consecutive 1s. Observe next that  $F_{-n}\phi \equiv \phi^{-n}$  (modulo 1), by exercise 1.2.8–11. Now add  $F_{k_1}\phi + \dots + F_{k_r}\phi$ . For example,  $(4\phi) \bmod 1 = \phi^{-5} + \phi^{-2}$ ;  $(-2\phi) \bmod 1 = \phi^{-4} + \phi^{-1}$ .

This argument also proves the interesting formula  $\lfloor N(\alpha)\phi \rfloor = -N(\alpha 0)$ .

**156.** (a) Start with  $y \leftarrow 0$ , and with  $k$  large enough that  $|x| < F_{k+1}$ . If  $x < 0$ , set  $k \leftarrow (k-1) \mid 1$ , and while  $x + F_k > 0$  set  $k \leftarrow k-2$ ; then set  $y \leftarrow y + (1 \ll k)$ ,  $x \leftarrow x + F_{k+1}$ ; repeat. Otherwise if  $x > 1$ , set  $k \leftarrow k \& -2$ , and while  $x - F_k \leq 0$  set  $k \leftarrow k-2$ ; then set  $y \leftarrow y + (1 \ll k)$ ,  $x \leftarrow x - F_{k+1}$ ; repeat. Otherwise set  $y \leftarrow y + x$  and terminate with  $y = (\alpha)_2$ .

(b) The operations  $x_1 \leftarrow a_1$ ,  $y_1 \leftarrow -a_1$ ,  $x_k \leftarrow y_{k-1} + a_k$ ,  $y_k \leftarrow x_{k-1} - x_k$  compute  $x_k = N(a_1 \dots a_k)$  and  $y_k = N(a_1 \dots a_k 0)$ . [Does *every* broadword chain for  $N(a_1 \dots a_n)$  require  $\Omega(n)$  steps?]

**157.** The laws are obvious except for the two cases involving  $(\alpha -)$ . For those we have  $N((\alpha -)0^k) = N(\alpha 0^k) + F_{-k-2}$  for all  $k \geq 0$ , because decrementation never “borrows” at the right. (But the analogous formula  $N((\alpha +)0^k) = N(\alpha 0^k) + F_{-k-1}$  does *not* hold.)

**158.** Incrementation satisfies the rules  $(\alpha 00)+ = \alpha 01$ ,  $(\alpha 10)+ = (\alpha +)00$ ,  $(\alpha 1)+ = (\alpha +)0$ . It can be achieved with six 2-adic operations on the integer  $x = (\alpha)_2$  by setting  $y \leftarrow x \mid (x \gg 1)$ ,  $z \leftarrow y \& \sim(y+1)$ ,  $x \leftarrow (x \mid z) + 1$ .

Decrementation of a nonzero codeword is more difficult. It satisfies  $(\alpha 10^{2k})- = \alpha 0(10)^k$ ,  $(\alpha 10^{2k+1})- = \alpha(01)^{k+1}$ ; hence by Corollary I it cannot be computed by a 2-adic chain. Yet six operations suffice, if we allow monus:  $y \leftarrow x - 1$ ,  $z \leftarrow y \& \bar{x}$ ,  $w \leftarrow z \& \mu_0$ ,  $x \leftarrow y - w + (w \dot{-} (z - w))$ .

**159.** Besides the Fibonacci number system (146) and the negaFibonacci number system (147), there’s also an *odd Fibonacci number system*: Every positive integer  $x$  can be written uniquely in the form

$$x = F_{l_1} + F_{l_2} + \dots + F_{l_s}, \quad \text{where } l_1 \gg l_2 \gg \dots \gg l_s > 0 \text{ and } l_s \text{ is odd.}$$

Given a negaFibonacci code  $\alpha$ , the following 20-step 2-adic chain converts  $x = (\alpha)_2$  to  $y = (\beta)_2$  to  $z = (\gamma)_2$ , where  $\beta$  is the odd codeword with  $N(\alpha) = F(\beta)$  and  $\gamma$  is the standard codeword with  $F(\beta) = F(\gamma 0)$ :  $x^+ \leftarrow x \& \mu_0$ ,  $x^- \leftarrow x \oplus x^+$ ;  $d \leftarrow x^+ - x^-$ ;  $t \leftarrow d \mid x^-$ ,  $t \leftarrow t \& \sim(t \ll 1)$ ;  $y \leftarrow (d \& \bar{\mu}_0) \oplus t \oplus ((t \& x^-) \gg 1)$ ;  $z \leftarrow (y+1) \gg 1$ ;  $w \leftarrow z \oplus (4\mu_0)$ ;  $t \leftarrow w \& \sim(w+1)$ ;  $z \leftarrow z \oplus (t \& (z \oplus ((w+1) \gg 1)))$ .

Corresponding negaFibonacci and odd representations satisfy the remarkable law

$$F_{k_1+m} + \dots + F_{k_r+m} = (-1)^m (F_{l_1-m} + \dots + F_{l_s-m}), \quad \text{for all integers } m.$$

For example, if  $N(\alpha) < 0$  the steps above will convert  $x = (\alpha 0)_2$  to  $y = (\beta)_2$ , where  $F((\beta \gg 2)0) = -N(\alpha)$ . Furthermore  $\beta$  is the odd code for negaFibonacci  $\alpha$  if and only if  $\alpha^R$  is the odd code for negaFibonacci  $\beta^R$ , when  $|\alpha| = |\beta|$  is odd and  $N(\alpha) > 0$ .

No finite 2-adic chain will go the other way, by Corollary I, because the Fibonacci code  $10^k$  corresponds to negaFibonacci  $10^{k+1}$  when  $k$  is odd,  $(10)^{k/2}1$  when  $k$  is even. But if  $\gamma$  is a standard Fibonacci codeword we can compute  $y = (\beta)_2$  from  $z = (\gamma)_2$  by setting  $y \leftarrow z \ll 1$ ,  $t \leftarrow y \& (y-1) \& \bar{\mu}_0$ ,  $y \leftarrow y - t + [t \neq 0]((t-1) \& \mu_0)$ . And then the method above will compute  $\alpha^R$  from  $\beta^R$ . The overall running time for conversion to negaFibonacci form will then be of order  $\log |\gamma|$ , for two string reversals.

**160.** The text’s rules are actually incomplete: They should also define the orientation of each neighbor. Let us stipulate that  $\alpha_{sn} = \alpha$ ;  $\alpha_{en} = \alpha$ ;  $(\alpha 0)_{wn} = \alpha 0$ ,  $(\alpha 1)_{wo} = \alpha 1$ ;  $(\alpha 00)_{ns} = \alpha 00$ ,  $(\alpha 10)_{nw} = \alpha 10$ ,  $(\alpha 1)_{ne} = \alpha 1$ ;  $(\alpha 0)_{oo} = \alpha 0$ ,  $(\alpha 101)_{oo} = \alpha 101$ ,

broadword chain  
2-adic chain  
monus  
magic mask  
odd Fibonacci number system  
2-adic chain  
magic mask  
string reversals

$(\alpha 1001)_{oo} = \alpha 1001$ ,  $(\alpha 0001)_{ow} = \alpha 0001$ . Then a case analysis proves that all cells within  $d$  steps of the starting cell have a consistent labeling and orientation, by induction on the graph distance  $d$ . (Note the identity  $\alpha+ = ((\alpha 0)-) \gg 1$ .) Furthermore the labeling remains consistent when we attach  $y$  coordinates and move when necessary from one strip to another via the  $\delta$ -rules of (153).

**161.** Yes, it is bipartite, because all of its edges are defined by the set of boundary lines. (The hyperbolic *cylinder* cannot be bicolored; but two adjacent strips can.)

**162.** It's convenient to view the hyperbolic plane through another lens, by mapping its points to the upper halfplane  $\Im z > 0$ . Then the “straight lines” become semicircles centered on the  $x$ -axis, together with vertical halfplanes as a limiting case. In this representation, the edges  $|z-1| = \sqrt{2}$ ,  $|z| = r$ , and  $\Re z = 0$  define a  $36^\circ$ - $45^\circ$ - $90^\circ$  triangle if  $r^2 = \phi + \sqrt{\phi}$ . Every triangle  $ABC$  has three neighbors  $CBA'$ ,  $ACB'$ , and  $BAC'$ , obtained by “reflecting” two of its edges about the third, where the reflection of  $|z-c'| = r'$  about  $|z-c| = r$  is  $|z-c-\frac{1}{2}(x_1+x_2)| = \frac{1}{2}|x_1-x_2|$ ,  $x_j = r^2/(c' \pm r' - c)$ .

The mapping  $z \mapsto (z-z_0)/(z-\bar{z}_0)$  takes the upper halfplane into the unit circle; when  $z_0 = \frac{1}{2}(\sqrt{\phi} - 1/\phi)(1 + 5^{1/4}i)$  the central pentagon will be symmetric. Repeated reflections of the initial triangle, using breadth-first search until reaching triangles that are invisible, will lead to Fig. 14. To get just the pentagons (without the gray lines), one can begin with just the central cell and perform reflections about *its* edges, etc.

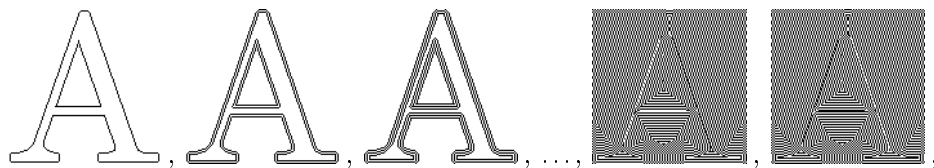
**163.** (This figure can be drawn as in exercise 162, starting with vertices that project to the three points  $ir$ ,  $ir\omega$ , and  $ir\omega^2$ , where  $r^2 = \frac{1}{2}(1+\sqrt{2})(4-\sqrt{2}-\sqrt{6})$  and  $\omega = e^{2\pi i/3}$ . Using a notation devised by L. Schläfli in 1852, it can be described as the infinite tiling with parameters  $\{3, 8\}$ , meaning that eight triangles meet at every vertex; see Schläfli's *Gesammelte Mathematische Abhandlungen* 1 (1950), 212. Similarly, the pentagrid and the tiling of exercise 154 have Schläfli symbols  $\{5, 4\}$  and  $\{5, 5\}$ , respectively.)

**164.** The original definition requires more computation, even though it can be factored:

$$\text{custer}'(X) = X \& \sim(Y_N \& Y \& Y_S), \quad Y = X_W \& X \& X_E.$$

But the main reason for preferring (157) is that it produces a thinner, kingwise connected border. The rookwise connected border that results from the 1957 definition is less attractive, because it's noticeably darker when the border travels diagonally than when it travels horizontally or vertically. (Try some experiments and you'll see.)

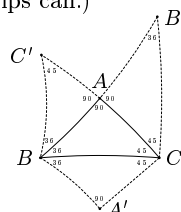
**165.** The first image  $X^{(1)}$  is the “outer” border of the original black pixels. Fingerprint-like whorls are formed thereafter. For example, starting with Fig. 15(a) we get



in a  $120 \times 120$  bitmap, eventually alternating endlessly between two bizarre patterns. (Does *every* nonempty  $M \times N$  bitmap lead to such a 2-cycle?)

**166.** If  $X = \text{custer}(X)$ , the sum of the elements of  $X + (X \hat{\wedge} 1) + (X \ll 1) + (X \gg 1) + (X \vee 1)$  is at most  $4MN + 2M + 2N$ , since it is at most 4 in each cell of the rectangle and at most 1 in the adjacent cells. This sum is also five times the number of black pixels. Hence  $f(M, N) \leq \frac{4}{5}MN + \frac{2}{5}M + \frac{2}{5}N$ . Conversely we get  $f(M, N) \geq \frac{4}{5}MN - \frac{2}{5}$  by

bipartite  
cylinder  
hyperbolic plane  
upper halfplane  
reflection  
breadth-first search  
Schläfli



letting the pixel in row  $i$  and column  $j$  be black unless  $(i + 2j) \bmod 5 = 2$ . (This problem is equivalent to finding a minimum dominating set of the  $M \times N$  grid.)

**167.** (a) With 17 steps we can construct a half adder and three full adders (see 7.1.2–(23)) so that  $(z_1 z_2)_2 = x_{NW} + x_W + x_{SW}$ ,  $(z_3 z_4)_2 = x_N + x_S$ ,  $(z_5 z_6)_2 = x_{NE} + x_E + x_{SE}$ , and  $(z_7 z_8)_2 = z_2 + z_4 + z_6$ . Then  $f = S_1(z_1, z_3, z_5, z_7) \wedge (x \vee z_8)$ , where the symmetric function  $f_1$  needs seven operations by Fig. 9 in Section 7.1.2. [This solution is based on ideas of W. F. Mann and D. Sleator.]

(b) Given  $x^- = X_{j-1}^{(t)}$ ,  $x = X_j^{(t)}$ , and  $x^+ = X_{j+1}^{(t)}$ , compute  $a \leftarrow x^- \& x^+ (= z_3)$ ,  $b \leftarrow x^- \oplus x^+ (= z_4)$ ,  $c \leftarrow x \oplus b$ ,  $d \leftarrow c \gg 1 (= z_6)$ ,  $e \leftarrow c \ll 1 (= z_2)$ ,  $e \leftarrow c \oplus d$ ,  $c \leftarrow c \& d$ ,  $f \leftarrow b \& e$ ,  $f \leftarrow f \mid c (= z_7)$ ,  $e \leftarrow b \oplus e (= z_8)$ ,  $c \leftarrow x \& b$ ,  $c \leftarrow c \mid a$ ,  $b \leftarrow c \ll 1 (= z_5)$ ,  $c \leftarrow c \gg 1 (= z_1)$ ,  $d \leftarrow b \& c$ ,  $c \leftarrow b \mid c$ ,  $b \leftarrow a \& f$ ,  $f \leftarrow a \mid f$ ,  $f \leftarrow d \mid f$ ,  $c \leftarrow b \mid c$ ,  $f \leftarrow f \oplus c (= S_1(z_1, z_3, z_5, z_7))$ ,  $e \leftarrow e \mid x$ ,  $f \leftarrow f \& e$ .

[For excellent summaries of the joys and passions of Life, including a proof that any Turing machine can be simulated, see Martin Gardner, *Wheels, Life and Other Mathematical Amusements* (1983), Chapters 20–22; E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways 4* (A. K. Peters, 2004), Chapter 25.]

*At last I've got what I wanted — an apparently unpredictable law of genetics.*

*... Overpopulation, like underpopulation, tends to kill.*

*A healthy society is neither too dense nor too sparse.*

— JOHN H. CONWAY, letter to Martin Gardner (March 1970)

**168.** The following algorithm, which uses four  $n$ -bit registers  $x^-$ ,  $x$ ,  $x^+$ , and  $y$ , works properly even when  $M = 1$  or  $N = 1$ . It needs only about two reads and two writes per raster word to transform  $X^{(t)}$  to  $X^{(t+1)}$  in (158):

**C1.** [Loop on  $k$ .] Do step C2 for  $k = 1, 2, \dots, N'$ ; then go to C5.

**C2.** [Loop on  $j$ .] Set  $x \leftarrow A_{(M-1)k}$ ,  $x^+ \leftarrow A_{0k}$ , and  $A_{Mk} \leftarrow x^+$ . Then perform steps C3 and C4 for  $j = 0, 1, \dots, M - 1$ .

**C3.** [Move down.] Set  $x^- \leftarrow x$ ,  $x \leftarrow x^+$ , and  $x^+ \leftarrow A_{(j+1)k}$ . (Now  $x = A_{jk}$ , and  $x^-$  holds the former value of  $A_{(j-1)k}$ .) Compute the bitwise function values  $y \leftarrow f(x^- \gg 1, x^-, x \ll 1, x \gg 1, x \ll 1, x^+ \gg 1, x^+, x^+ \ll 1)$ .

**C4.** [Update  $A_{jk}$ .] Set  $x^- \leftarrow A_{j(k-1)} \& -2$ ,  $y \leftarrow y \& (2^{n-1} - 1)$ ,  $A_{j(k-1)} \leftarrow x^- + (y \gg (n-2))$ ,  $A_{jk} \leftarrow y + (x^- \ll (n-2))$ .

**C5.** [Wrap around.] For  $0 \leq j < M$ , set  $x \leftarrow A_{jN'} \& -2^{n-1-d}$ ,  $A_{jN'} \leftarrow x + (A_{j1} \gg d)$ , and  $A_{j1} \leftarrow A_{j1} + (x \ll d)$ , where  $d = 1 + (N - 1) \bmod (n - 2)$ . ■

[An  $M \times N$  torus is equivalent to an  $(M - 1) \times (N - 1)$  array surrounded by zeros, in many cases like (157) and (159) and even (161). For exercise 173 we can clean an  $(M - 2) \times (N - 2)$  array that is bordered by two rows and columns of zeros. But Life images (exercise 167) can grow without bound; they can't safely be confined to a torus.]

**169.** It quickly morphs into a rabbit, which proceeds to explode. Beginning at time 278, all activity stabilizes to a two-cycle formed from a set of traffic lights and three additional blinkers, together with three still lifes (tub, boat, and bee hive).

**170.** If  $M \geq 2$  and  $N \geq 2$ , the first step blanks out the top row and the rightmost column. Then if  $M \geq 3$  and  $N \geq 3$ , the next step blanks out the bottom row and the leftmost column. So in general we're left after  $t = \min(M, N) - 1$  steps with a single row or column of black pixels: The first  $\lceil t/2 \rceil$  rows, the last  $\lceil t/2 \rceil$  columns, the last  $\lfloor t/2 \rfloor$  rows, and the first  $\lfloor t/2 \rfloor$  columns have been set to zero. The automaton will stop after making two more (nonproductive) cycles.

dominating set  
grid  
half adder  
full adders  
Mann  
Sleator  
Turing machine  
Gardner  
Berlekamp  
Conway  
Guy  
CONWAY  
Gardner  
clean

**171.** Without (160):  $x_1 \leftarrow x_{SE} \& \bar{x}_N$ ,  $x_2 \leftarrow x_N \& \bar{x}_{SE}$ ,  $x_3 \leftarrow x_E \& \bar{x}_1$ ,  $x_4 \leftarrow x_{NE} \& \bar{x}_2$ ,  
 $x_5 \leftarrow x_3 \mid x_4$ ,  $x_6 \leftarrow x_W \& \bar{x}_5$ ,  $x_7 \leftarrow x_1 \& \bar{x}_{NE}$ ,  $x_8 \leftarrow x_7 \& \bar{x}_{NW}$ ,  $x_9 \leftarrow x_E \mid x_{SW}$ ,  
 $x_{10} \leftarrow x_8 \& x_9$ ,  $x_{11} \leftarrow x_{10} \mid x_6$ ,  $x_{12} \leftarrow x_S \& x_{11}$ ,  $x_{13} \leftarrow x_2 \& \bar{x}_E$ ,  $x_{14} \leftarrow x_{13} \& x_W$ ,  
 $x_{15} \leftarrow x_N \& x_{NE}$ ,  $x_{16} \leftarrow x_{SW} \& x_W$ ,  $x_{17} \leftarrow x_{15} \mid x_{16}$ ,  $x_{18} \leftarrow x_{NE} \& x_{SW}$ ,  $x_{19} \leftarrow x_{17} \& \bar{x}_{18}$ ,  
 $x_{20} \leftarrow x_E \mid x_{SE}$ ,  $x_{21} \leftarrow x_{20} \mid x_S$ ,  $x_{22} \leftarrow x_{NW} \& \bar{x}_{21}$ ,  $x_{23} \leftarrow x_{22} \& x_{19}$ ,  $x_{24} \leftarrow x_{12} \mid x_{14}$ ,  
 $g \leftarrow x_{23} \mid x_{24}$ . With (160), set  $x_4 \leftarrow x_{NE} \& \bar{x}_N$  and leave everything else the same.

**172.** The statement isn't quite true; consider the following examples:



The 'I' and 'H' at the left show that pixels are sometimes left intact where paths join, and that rotating by  $90^\circ$  can make a difference. The next two examples illustrate a quirky influence of left-right reflection. The diamond example demonstrates that very thick images can be unthinnable; none of its black pixels can be removed without changing the number of holes. The final examples, one of which was inspired by the answer to exercise 166, were processed first without (160), in which case they are unchanged by the transformation. But with (160) they're thinned dramatically.

**173.** (a) If  $X$  and  $Y$  are closed,  $X \& Y$  is closed; if  $X$  and  $Y$  are open,  $X \mid Y$  is open. Thus  $X^D$  is closed and  $X^L$  is open; the hinted statement follows. Furthermore  $X^{DD} = X^D$  and  $X^{LL} = X^L$ . (In fact we have  $X^L = \sim(\sim X)^D$ , because the definitions are dual, obtained by swapping black with white.) Now  $X^{DL} \subseteq X^D$ , so  $X^{DL D} \subseteq X^{DD} = X^D$ . And dually,  $X^L \subseteq X^{LDL}$ . We conclude that there's no reason to launder a clean picture:  $X^{DL D L} = (X^{DL D})^L \subseteq X^{DL} \subseteq (X^D)^{LDL} = X^{DL D L}$ .

(b) We have  $X^D = (X \mid X_W \mid X_{NW} \mid X_N) \& (X \mid X_N \mid X_{NE} \mid X_E) \& (X \mid X_E \mid X_{SE} \mid X_S) \& (X \mid X_S \mid X_{SW} \mid X_W)$ . Furthermore, in analogy with answer 167(b), this function can be computed from  $x^-$ ,  $x$ , and  $x^+$  in ten broadword steps:  $f \leftarrow x \mid (x \gg 1) \mid ((x^- \mid (x^- \gg 1)) \& (x^+ \mid (x^+ \gg 1)))$ ,  $f \leftarrow f \& (f \ll 1)$ . [This answer incorporates ideas of D. R. Fuchs.]

To get  $X^L$ , just interchange  $\mid$  and  $\&$ . [For further discussion, see C. Van Wyk and D. E. Knuth, Report STAN-CS-79-707 (Stanford Univ., 1979), 15–36.]

**174.** Three-dimensional digital topology has been studied by R. Malgouyres, *Theoretical Computer Science* **186** (1997), 1–41.

**175.** There are 25 in the outline, 2 + 3 in the eyes, 1 + 1 in the ears, 4 in the nose, and 1 in the smile, totalling 37. (All white pixels are connected kingwise to the background.)

**176.** (a) If  $v$  isn't isolated, there are eight easy cases to consider, depending on what kind of neighbor  $v$  has in  $G$ .

(b) There's a vertex of  $G'$  adjacent to each vertex of  $(N_u \cup N_v) \setminus G'$ . (Four cases.)

(c) Yes. In fact, by definition (161), we always have  $|S'(v')| \geq 2$ .

(d) Let  $N'_{v'} = \{v \mid v' \in N_v\}$ . If  $v'$  is the east neighbor of  $u'$ , call it  $u'_E$ , either  $u' \in G$  or  $u'_S \in G$ ; this element is equal-or-adjacent to every vertex of  $N'_{u'} \cup N'_{v'}$ . A similar argument applies when  $v' = u'_N$ . If  $v' = u'_{NE}$ , there's no problem if  $u' \in G$ . Otherwise  $u'_W \in G$ ,  $u'_S \in G$ , and either  $u'_N \in G$  or  $u'_E \in G$ ; hence  $N'_{u'} \cup N'_{v'}$  is connected in  $G$ . Finally if  $v' = u'_{SE}$ , the proof is easy if  $u'_S \in G$ ; otherwise  $u' \in G$  and  $v' \in G$ .

(e) Given a nontrivial component  $C$  of  $G$ , with  $v \in C$  and  $v' \in S(v)$ , let  $C'$  be the component of  $G'$  that contains  $v'$ . This component  $C'$  is well defined, by (a) and (b). Given a component  $C'$  of  $G'$ , with  $v' \in C'$  and  $v \in S'(v')$ , let  $C$  be the component of  $G$  that contains  $v$ . This component  $C$  is nontrivial and well defined, by (c) and (d). Finally, the correspondence  $C \leftrightarrow C'$  is one-to-one.

dual  
Fuchs  
Van Wyk  
Knuth  
Malgouyres



**177.** Now the vertices of  $G$  are the *white* pixels, adjacent when they are *rook*-neighbors. So we define  $N_{(i,j)} = \{(i,j), (i-1,j), (i,j+1)\}$ . Arguments like those of answer 176, but simpler, establish a one-to-one correspondence between the nontrivial components of  $G$  and the components of  $G'$ .

**178.** Observe that in adjacent rows of  $X^*$ , two pixels of the same value are kingwise neighbors only if they are rookwise connected.

**179.** The pixels of each row  $x_1 \dots x_N$  can be “runlength encoded” as a sequence of integers  $0 = c_0 < c_1 < \dots < c_{2m+1} = N + 2$  so that  $x_j = 0$  for  $j \in [c_0 \dots c_1) \cup [c_2 \dots c_3) \cup \dots \cup [c_{2m} \dots c_{2m+1})$  and  $x_j = 1$  for  $j \in [c_1 \dots c_2) \cup \dots \cup [c_{2m-1} \dots c_{2m})$ . (The number of runs per row tends to be reasonably small in most images. Notice that the background condition  $x_0 = x_{N+1} = 0$  is implicitly assumed.)

The algorithm below uses a modified encoding with  $a_j = 2c_j - (j \bmod 2)$  for  $0 \leq j \leq 2m+1$ . For example, the second row of the Cheshire cat has  $(c_1, c_2, c_3, c_4, c_5) = (5, 8, 23, 25, 32)$ ; we will use  $(a_1, a_2, a_3, a_4, a_5) = (9, 16, 45, 50, 63)$  instead. The reason is that white runs of adjacent rows are rookwise adjacent if and only if the corresponding intervals  $[a_j \dots a_{j+1})$  and  $[b_k \dots b_{k+1})$  overlap, and exactly the same condition characterizes when black runs of adjacent rows are kingwise adjacent. Thus the modified encoding nicely unifies both cases (see exercise 178).

We construct a triply linked tree of components, where each node has several fields: **CHILD**, **SIB**, and **PARENT** (tree links); **DORMANT** (a circular list of all children that aren’t connected to the current row); **HEIR** (a node that has absorbed this one); **ROW** and **COL** (location of the first pixel); and **AREA** (the total number of pixels in the component).

The algorithm traverses the tree in *double order* (see exercise 2.3.1–18), using pairs of pointers  $(P, P')$ , where  $P' = P$  when  $P$  is traversed the first time,  $P' = \text{PARENT}(P)$  when  $P$  is traversed the second time. The successor of  $(P, P')$  is  $(Q, Q') = \text{next}(P, P')$ , determined as follows: If  $P = P'$  and  $\text{CHILD}(P) \neq \Lambda$ , then  $Q \leftarrow Q' \leftarrow \text{CHILD}(P)$ ; otherwise  $Q \leftarrow P$  and  $Q' \leftarrow \text{PARENT}(Q)$ . If  $P \neq P'$  and  $\text{SIB}(P) \neq \Lambda$ , then  $Q \leftarrow Q' \leftarrow \text{SIB}(P)$ ; otherwise  $Q \leftarrow \text{PARENT}(P)$  and  $Q' \leftarrow \text{PARENT}(Q)$ .

When there are  $m$  black runs, the tree will have  $m+1$  nodes, not counting nodes that are dormant or have been absorbed. Moreover, the primed pointers  $P'_1, \dots, P'_{2m+1}$  of the double traversal  $(P_1, P'_1), \dots, (P_{2m+1}, P'_{2m+1})$  are precisely the components of the current row, in left-to-right order. For example, in (163) we have  $m = 5$ ; and  $(P'_1, \dots, P'_{11})$  point respectively to  $\textcircled{0}, \textcircled{B}, \textcircled{1}, \textcircled{B}, \textcircled{0}, \textcircled{C}, \textcircled{0}, \textcircled{A}, \textcircled{2}, \textcircled{A}, \textcircled{0}$ .

- I1.** [Initialize.] Set  $t \leftarrow 1$ ,  $\text{ROOT} \leftarrow \text{LOC}(\text{NODE}(0))$ ,  $\text{CHILD}(\text{ROOT}) \leftarrow \text{SIB}(\text{ROOT}) \leftarrow \text{PARENT}(\text{ROOT}) \leftarrow \text{DORMANT}(\text{ROOT}) \leftarrow \text{HEIR}(\text{ROOT}) \leftarrow \Lambda$ ; also  $\text{ROW}(\text{ROOT}) \leftarrow \text{COL}(\text{ROOT}) \leftarrow 0$ ,  $\text{AREA}(\text{ROOT}) \leftarrow N + 2$ ,  $s \leftarrow 0$ ,  $a_0 \leftarrow b_0 \leftarrow 0$ ,  $a_1 \leftarrow 2N + 3$ .
- I2.** [Input a new row.] Terminate if  $s > M$ . Otherwise set  $b_k \leftarrow a_k$  for  $k = 1, 2, \dots$ , until  $b_k = 2N + 3$ ; then set  $b_{k+1} \leftarrow b_k$  as a “stopper.” Set  $s \leftarrow s + 1$ . If  $s > M$ , set  $a_1 \leftarrow 2N + 3$ ; otherwise let  $a_1, \dots, a_{2m+1}$  be the modified runlength encoding of row  $s$  as discussed above. (This encoding can be obtained with the help of the  $\rho$  function; see (43).) Set  $j \leftarrow k \leftarrow 1$  and  $P \leftarrow P' \leftarrow \text{ROOT}$ .
- I3.** [Gobble up short  $b$ ’s.] If  $b_{k+1} \geq a_j$ , go to I9. Otherwise set  $(Q, Q') \leftarrow \text{next}(P, P')$ ,  $(R, R') \leftarrow \text{next}(Q, Q')$ , and do a four-way branch to (I4, I5, I6, I7) according as  $2[\mathbf{Q} \neq \mathbf{Q}'] + [\mathbf{R} \neq \mathbf{R}'] = (0, 1, 2, 3)$ .
- I4.** [Case 0.] (Now  $Q = Q'$  is a child of  $P'$ , and  $R = R'$  is the first child of  $Q'$ . Node  $Q$  will remain a child of  $P'$ , but it will be preceded by any children of  $R$ .) Absorb  $R$  into  $P'$  (see below). Set  $\text{CHILD}(Q) \leftarrow \text{SIB}(R)$  and  $Q' \leftarrow \text{CHILD}(R)$ . If  $Q' \neq \Lambda$ ,

runlength encoded  
runlength encoding, see also edge transitions  
Cheshire cat  
triply linked tree  
circular list  
double order  
ruler function

Lutz

- set  $R \leftarrow Q'$ , and while  $R \neq \Lambda$  set  $PARENT(R) \leftarrow P'$ ,  $R' \leftarrow R$ ,  $R \leftarrow SIB(R)$ ; then  $SIB(R) \leftarrow Q$ ,  $Q \leftarrow Q'$ . Set  $CHILD(P) \leftarrow Q$  if  $P = P'$ ,  $SIB(P) \leftarrow Q$  if  $P \neq P'$ . Go to I8.
- I5.** [Case 1.] (Now component  $Q = R$  is surrounded by  $P' = R'$ .) If  $P = P'$ , set  $CHILD(P) \leftarrow SIB(Q)$ ; otherwise set  $SIB(P) \leftarrow SIB(Q)$ . Set  $R \leftarrow DORMANT(R')$ . Then if  $R = \Lambda$ , set  $DORMANT(R') \leftarrow SIB(Q) \leftarrow Q$ ; otherwise  $SIB(Q) \leftarrow SIB(R)$  and  $SIB(R) \leftarrow Q$ . Go to I8.
- I6.** [Case 2.] (Now  $Q'$  is the parent of both  $P'$  and  $R$ . Either  $P = P'$  is childless, or  $P$  is the last child of  $P'$ .) Absorb  $R$  into  $P'$  (see below). Set  $SIB(P') \leftarrow SIB(R)$  and  $R \leftarrow CHILD(R)$ . If  $P = P'$ , set  $CHILD(P) \leftarrow R$ ; otherwise  $SIB(P) \leftarrow R$ . While  $R \neq \Lambda$ , set  $PARENT(R) \leftarrow P'$  and  $R \leftarrow SIB(R)$ . Go to I8.
- I7.** [Case 3.] (Node  $P' = Q$  is the last child of  $Q' = R$ , which is a child of  $R'$ .) Absorb  $P'$  into  $R'$  (see below). If  $P = P'$ , set  $P \leftarrow R$ . Otherwise set  $P' \leftarrow CHILD(P')$ , and while  $P' \neq \Lambda$  set  $PARENT(P') \leftarrow R'$ ,  $P' \leftarrow SIB(P')$ ; also set  $SIB(P) \leftarrow SIB(Q')$  and  $SIB(Q') \leftarrow CHILD(Q)$ . If  $Q = CHILD(R)$ , set  $CHILD(R) \leftarrow \Lambda$ . Otherwise set  $R \leftarrow CHILD(R)$ , then  $R \leftarrow SIB(R)$  until  $SIB(R) = Q$ , then  $SIB(R) \leftarrow \Lambda$ . Finally set  $P' \leftarrow R'$ .
- I8.** [Advance  $k$ .] Set  $k \leftarrow k + 2$  and return to step I3.
- I9.** [Update the area.] Set  $AREA(P') \leftarrow AREA(P') + \lceil a_j/2 \rceil - \lceil a_{j-1}/2 \rceil$ . Then go back to I2 if  $a_j = 2N + 3$ .
- I10.** [Gobble up short  $a$ .] If  $a_{j+1} \geq b_k$ , go to I11. Otherwise set  $Q \leftarrow LOC(NODE(t))$  and  $t \leftarrow t + 1$ . Set  $PARENT(Q) \leftarrow P'$ ,  $DORMANT(Q) \leftarrow HEIR(Q) \leftarrow \Lambda$ ; also  $ROW(Q) \leftarrow s$ ,  $COL(Q) \leftarrow \lceil a_j/2 \rceil$ ,  $AREA(Q) \leftarrow \lceil a_{j+1}/2 \rceil - \lceil a_j/2 \rceil$ . If  $P = P'$ , set  $SIB(Q) \leftarrow CHILD(P)$  and  $CHILD(P) \leftarrow Q$ ; otherwise set  $SIB(Q) \leftarrow SIB(P)$  and  $SIB(P) \leftarrow Q$ . Finally set  $P \leftarrow Q$ ,  $j \leftarrow j + 2$ , and return to I3.
- I11.** [Move on.] Set  $j \leftarrow j + 1$ ,  $k \leftarrow k + 1$ ,  $(P, P') \leftarrow next(P, P')$ , and go to I3. ■

To “absorb  $P$  into  $Q$ ” means to do the following things: If  $(ROW(P), COL(P))$  is less than  $(ROW(Q), COL(Q))$ , set  $(ROW(Q), COL(Q)) \leftarrow (ROW(P), COL(P))$ . Set  $AREA(Q) \leftarrow AREA(P) + AREA(Q)$ . If  $DORMANT(Q) = \Lambda$ , set  $DORMANT(Q) \leftarrow DORMANT(P)$ ; otherwise if  $DORMANT(P) \neq \Lambda$ , swap  $SIB(DORMANT(P)) \leftrightarrow SIB(DORMANT(Q))$ . Finally, set  $HEIR(P) \leftarrow Q$ . (The  $HEIR$  links could be used on a second pass to identify the final component of each pixel. Notice that the  $PARENT$  links of dormant nodes are not kept up to date.)

[A similar algorithm was given by R. K. Lutz in *Comp. J.* **23** (1980), 262–269.]

**180.** Let  $F(x, y) = x^2 - y^2 + 13$  and  $Q(x, y) = F(x - \frac{1}{2}, y - \frac{1}{2}) = x^2 - y^2 - x + y + 13$ . Apply Algorithm T to digitize the hyperbola from  $(\xi, \eta) = (-6, 7)$  to  $(\xi', \eta') = (0, \sqrt{13})$ ; hence  $x = -6$ ,  $y = 7$ ,  $x' = 0$ ,  $y' = 4$ . The resulting edges are  $(-6, 7) \text{ --- } (-5, 7) \text{ --- } (-5, 6) \text{ --- } (-4, 6) \text{ --- } (-4, 5) \text{ --- } (-3, 5) \text{ --- } (-3, 4) \text{ --- } \dots \text{ --- } (0, 4)$ . Then apply it again with  $\xi = 0$ ,  $\eta = \sqrt{13}$ ,  $\xi' = 6$ ,  $\eta' = 7$ ,  $x = 0$ ,  $y = 4$ ,  $x' = 6$ ,  $y' = 7$ ; the same edges are found (in reverse order), but with negated  $x$  coordinates.

**181.** Subdivide at points  $(\xi, \eta)$  where  $F_x(\xi, \eta) = 0$  or  $F_y(\xi, \eta) = 0$ , namely at the real roots of  $\{Q(-(b\eta + d)/(2a), \eta + \frac{1}{2}) = 0, \xi = -(b\eta + d)/(2a) - \frac{1}{2}\}$  or the real roots of  $\{Q(\xi + \frac{1}{2}, -(b\xi + e)/(2c)) = 0, \eta = -(b\xi + e)/(2c) - \frac{1}{2}\}$ , if they exist.

**182.** By induction on  $|x' - x| + |y' - y|$ . Consider, for example, the case  $x > x'$  and  $y < y'$ . We know from (iii) that  $(\xi, \eta)$  lies in the box  $x - \frac{1}{2} \leq \xi < x + \frac{1}{2}$  and  $y - \frac{1}{2} \leq \eta < y + \frac{1}{2}$ , and from (ii) that the curve travels monotonically as it moves from  $(\xi, \eta)$  to  $(\xi', \eta')$ . It must therefore exit the box at the edge  $(x - \frac{1}{2}, y - \frac{1}{2}) \text{ --- } (x - \frac{1}{2}, y + \frac{1}{2})$  or  $(x - \frac{1}{2}, y + \frac{1}{2}) \text{ --- } (x + \frac{1}{2}, y + \frac{1}{2})$ . The latter holds if and only if  $F(x - \frac{1}{2}, y + \frac{1}{2}) < 0$ , because the curve can't intersect that edge twice when  $x' < x$ . And  $F(x - \frac{1}{2}, y + \frac{1}{2})$  is

the value  $Q(x, y + 1)$  that is tested in step T4, because of the initialization in step T1. (We assume that the curve doesn't go *exactly* through  $(x - \frac{1}{2}, y + \frac{1}{2})$ , by implicitly adding a tiny positive amount to the function  $F$  behind the scenes.)

**183.** Consider, for example, the ellipse defined by  $F(x - \frac{1}{2}, y - \frac{1}{2}) = Q(x, y) = 13x^2 + 7xy + y^2 - 2 = 0$ ; this ellipse is a cigar-shaped curve that extends roughly between  $(-2, 5)$  and  $(1, -6)$ . Suppose we want to digitize its upper right boundary. Hypotheses (i)–(iv) of Algorithm T hold with

$$\xi = \sqrt{\frac{8}{3}} - \frac{1}{2}, \quad \eta = -\sqrt{\frac{98}{3}} - \frac{1}{2}, \quad \xi' = -\sqrt{\frac{98}{39}} - \frac{1}{2}, \quad \eta' = \sqrt{\frac{104}{3}} - \frac{1}{2},$$

$x = 1, y = -6, x' = -2, y' = 5$ . Step T1 sets  $Q \leftarrow Q(1, -5) = 1$ , which causes step T4 to move left (L); in fact, the resulting path is  $L^3U^{11}$ , while the correct digitization according to (164) is  $U^3LU^4LU^3LU$ . Failure occurred because  $Q(x, y) = 0$  has two roots on the edge  $(1, -5) - (2, -5)$ , namely  $((35 \pm \sqrt{29})/26, -5)$ , causing  $Q(1, -5)$  to have the same sign as  $Q(2, -5)$ . (One of those roots is on the boundary we are *not* trying to draw, but it's still there.) Similar failure occurs with the parabola defined by  $Q(x, y) = 9x^2 + 6xy + y^2 - y = 0$ ,  $\xi = -5/12, \eta = -1/4, \xi' = -5/2, \eta' = -19/2, x = 0, y = 0, x' = -2, y' = 9$ . Hyperbolas can fail too (consider  $6x^2 + 5xy + y^2 = 1$ ).

Algorithms for discrete geometry are notoriously delicate; unusual cases tend to drive them berserk. Algorithm T works properly for portions of any ellipse or parabola whose maximum curvature is less than 2. The maximum curvature of an ellipse with semiaxes  $\alpha \geq \beta$  is  $\alpha/\beta^2$ ; the cigar-shaped example has maximum curvature  $\approx 42.5$ . The maximum curvature of the parabola  $y = \alpha x^2$  is  $\alpha/2$ ; the anomalous parabola above has maximum curvature  $\approx 5.27$ . “Reasonable” conics don't make such sharp turns.

To make Algorithm T work correctly *without* hypothesis (v), we need to slow it down a bit, by changing the tests ‘ $Q < 0$ ’ to ‘ $Q < 0$  or  $X$ ’, where  $X$  is a test on the sign of a derivative. Namely,  $X$  is respectively ‘ $S > c$ ’, ‘ $R > a$ ’, ‘ $R < -a$ ’, ‘ $S < -c$ ’, in steps T2, T3, T4, T5.

**184.** Let  $Q'(x, y) = -1 - Q(x, y)$ . The key point is that  $Q(x, y) < 0$  if and only if  $Q'(x, y) \geq 0$ . (Curiously the algorithm makes the same decisions, backwards, although it probes the values of  $Q'$  and  $Q$  in different places.)

**185.** Find a positive integer  $h$  so that  $d = (\eta - \eta')h$  and  $e = (\xi' - \xi)h$  are integers and  $d + e$  is even. Then carry out Algorithm T with  $x = \lfloor \xi + \frac{1}{2} \rfloor, y = \lfloor \eta + \frac{1}{2} \rfloor, x' = \lfloor \xi' + \frac{1}{2} \rfloor, y' = \lfloor \eta' + \frac{1}{2} \rfloor$ , and  $Q(x, y) = d(x - \frac{1}{2}) + e(y - \frac{1}{2}) + f$ , where

$$f = \lfloor (\eta'\xi - \xi'\eta)h \rfloor - [d > 0 \text{ and } (\eta'\xi - \xi'\eta)h \text{ is an integer}].$$

(The ‘ $d > 0$ ’ term ensures that the opposite straight line, from  $(\xi', \eta')$  back to  $(\xi, \eta)$ , will have precisely the same edges; see exercise 183.) Steps T1 and T6–T9 become much simpler than they were in the general case, because  $R = d$  and  $S = e$  are constant.

(F. G. Stockton [CACM **6** (1963), 161, 450] and J. E. Bresenham [IBM Systems Journal **4** (1965), 25–30] gave similar algorithms, but with diagonal edges permitted.)

**186.** (a)  $B(\epsilon) = z_0 + 2\epsilon(z_1 - z_0) + O(\epsilon^2)$ ;  $B(1 - \epsilon) = z_2 - 2\epsilon(z_2 - z_1) + O(\epsilon^2)$ .

(b) Every point of  $S(z_0, z_1, z_2)$  is a convex combination of  $z_0, z_1$ , and  $z_2$ .

(c) Obviously true, since  $(1 - t)^2 + 2(1 - t)t + t^2 = 1$ .

(d) The collinear condition follows from (b). Otherwise, by (c), we need only consider the case  $z_0 = 0$  and  $z_2 - 2z_1 = 1$ , where  $z_1 = x_1 + iy_1$  and  $y_1 \neq 0$ . In that case all points lie on the parabola  $4x = (y/y_1)^2 + 4yx_1/y_1$ .

(e) Note that  $B(u\theta) = (1 - u)^2 z_0 + 2u(1 - u)((1 - \theta)z_0 + \theta z_1) + u^2 B(\theta)$  for  $0 \leq u \leq 1$ .

ellipse  
cigar-shaped curve  
parabola  
Hyperbolas  
curvature  
Stockton  
Bresenham

[S. N. Bernshtein introduced  $B_n(z_0, z_1, \dots, z_n; t) = \sum_k \binom{n}{k} (1-t)^{n-k} t^k z_k$  in *Soobshcheniia Khar'kovskoe matematicheskoe obshchestvo* (2) **13** (1912), 1–2.]

**187.** We can assume that  $z_0 = (x_0, y_0)$ ,  $z_1 = (x_1, y_1)$ , and  $z_2 = (x_2, y_2)$ , where the coordinates are (say) fixed-point numbers represented as 16-bit integers divided by 32.

If  $z_0$ ,  $z_1$ , and  $z_2$  are collinear, use the method of exercise 185 to draw a straight line from  $z_0$  to  $z_2$ . (If  $z_1$  doesn't lie between  $z_0$  and  $z_2$ , the other edges will cancel out, because edges are implicitly XORed by a filling algorithm.) This case occurs if and only if  $D = x_0 y_1 + x_1 y_2 + x_2 y_0 - x_1 y_0 - x_2 y_1 - x_0 y_2 = 0$ .

Otherwise the points  $(x, y)$  of  $S(z_0, z_1, z_2)$  satisfy  $F(x, y) = 0$ , where

$$F(x, y) = ((x - x_0)(y_2 - 2y_1 + y_0) - (y - y_0)(x_2 - 2x_1 + x_0))^2 - 4D((x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0))$$

and  $D$  is defined above. We multiply by  $32^4$  to obtain integer coefficients; then negate this formula and subtract 1, if  $D < 0$ , to satisfy condition (iv) of Algorithm T and the reverse-order condition. (See exercise 184.)

The monotonicity condition (ii) holds if and only if  $(x_1 - x_0)(x_2 - x_1) > 0$  and  $(y_1 - y_0)(y_2 - y_1) > 0$ . If necessary, we can use the recurrence of exercise 186(e) to break  $S(z_0, z_1, z_2)$  into at most three monotonic subquines; for example, setting  $\theta = (x_0 - x_1)/(x_0 - 2x_1 + x_2)$  will achieve monotonicity in  $x$ . (A slight rounding error may occur during this fixed point arithmetic, but the recurrence can be performed in such a way that the subquines are definitely monotonic.)

*Notes:* When  $z_0$ ,  $z_1$ , and  $z_2$  are near each other, a simpler and faster method based on exercise 186(e) with  $\theta = \frac{1}{2}$  is adequate for most practical purposes, if one doesn't care about making the exactly correct choice between local edge sequences like “up-then-left” versus “left-then-up.” In the late 1980s, Sampo Kaasila chose to use squines as the basic method of shape specification in the TrueType font format, because they can be digitized so rapidly. The METAFONT system achieves greater flexibility with cubic Bézier splines [see D. E. Knuth, *METAFONT: The Program* (Addison-Wesley, 1986)], but at the cost of extra processing time. A fairly fast “six-register algorithm” for the resulting cubic curves was, however, developed subsequently by John Hobby [*ACM Trans. on Graphics* **9** (1990), 262–277]. Vaughan Pratt introduced *conic splines*, which are sort of midway between squines and Bézier cubics, in *Computer Graphics* **9**, 3 (July 1985), 151–159. Conic spline segments can be elliptical and hyperbolic as well as parabolic, hence they require fewer intermediate points and control points than squines; furthermore, they can be handled by Algorithm T.

**188.** If the rows of the bitmap are  $(X_0, X_1, \dots, X_{63})$ , do the following operations for  $k = 0, 1, \dots, 5$ : For all  $i$  such that  $0 \leq i < 64$  and  $i \& 2^k = 0$ , let  $j = i + 2^k$  and either (a) set  $t \leftarrow (X_i \oplus (X_j \gg 2^k)) \& \mu_{6,k}$ ,  $X_i \leftarrow X_i \oplus t$ ,  $X_j \leftarrow X_j \oplus (t \ll 2^k)$ ; or (b) set  $t \leftarrow X_i \& \bar{\mu}_{6,k}$ ,  $u \leftarrow X_j \& \mu_{6,k}$ ,  $X_i \leftarrow ((X_i \ll 2^k) \& \bar{\mu}_{6,k}) \mid u$ ,  $X_j \leftarrow ((X_j \gg 2^k) \& \mu_{6,k}) \mid t$ .

[The basic idea is to transform  $2^k \times 2^k$  submatrices for increasing  $k$ , as in exercise 5–12. Speedups are possible with MMIX, using MOR and MUX as in exercise 208, and using LDTU/STTU when  $k = 5$ . See L. J. Guibas and J. Stolfi, *ACM Transactions on Graphics* **1** (1982), 204–207; M. Thorup, *J. Algorithms* **42** (2002), 217. Incidentally, Theorem P and answer 54 show that  $\Omega(n \log n)$  operations on  $n$ -bit numbers are needed to transpose an  $n \times n$  bit matrix. An application that needs frequent transpositions might therefore be better off using a redundant representation, maintaining its matrices in both normal and transposed form.]

**189.** The following big-endian program assumes that  $n \leq 74880$ .

Bernshtein  
fixed-point  
Kaasila  
squines  
TrueType  
Bézier  
Knuth  
METAFONT  
six-register algorithm  
Hobby  
Pratt  
conic splines  
elliptical  
hyperbolic  
magic  
MOR  
MUX  
Guibas  
Stolfi  
Thorup  
lower bounds  
redundant representation  
big-endian

	LOC	Data_Segment		LDO	k, Initk		Hunt
BITMAP	LOC	@+M*N/8		OH	SET s, N/64		DVIPAGE
base	GREG	@		1H	SET a, h	A trick (see below)	Knuth
GRAYMAP	LOC	@+M*N/64			SET r, 8		trick
GTAB	BYTE	255, 252, 249, 246, 243		2H	LDOU t, base, k		MOR
	BYTE	240, 236, 233, 230, 227			MOR u, c1, t		pigeonhole principle
	BYTE	224, 221, 217, 214, 211			SUBU t, t, u	(Nypwise sums)	
	BYTE	208, 204, 201, 198, 194			MOR u, c2, t		
	BYTE	191, 188, 184, 181, 178			AND t, t, mu1		
	BYTE	174, 171, 167, 164, 160			ADDU t, t, u	(Nybblewise sums)	
	BYTE	157, 153, 150, 146, 142			MOR u, c3, t		
	BYTE	139, 135, 131, 128, 124			AND t, t, mu2		
	BYTE	120, 116, 112, 108, 104			ADDU t, t, u	(Bytewise sums)	
	BYTE	100, 96, 92, 88, 84			ADDU a, a, t		
	BYTE	79, 75, 70, 66, 61			INCL k, N/8	Move to next row.	
	BYTE	56, 52, 46, 41, 36			SUB r, r, 1		
	BYTE	30, 24, 18, 10, 0			PBNZ r, 2B	Repeat 8 times.	
Initk	OCTA	BITMAP-GRAYMAP		3H	SRU t, a, 56		
corr	GREG	N-8			LDBU t, gtab, t		
c1	GREG	#4000100004000100			SLU a, a, 8		
c2	GREG	#2010000002010000			STBU t, z, 0		
c3	GREG	#0804020100000000			INCL z, 1		
mu1	GREG	#3333333333333333			PBN a, 3B	(The trick)	
mu2	GREG	#0f0f0f0f0f0f0f0f			SUB k, k, corr		
h	GREG	#8080808080808080			SUB s, s, 1		
gtab	GREG	GTAB-#80			PBNZ s, 1B	Loop on columns.	
	LOC	#100			INCL k, 7*N/8	Loop on groups	
MakeGray	LDA	z, GRAYMAP			PBN k, 0B	of 8 rows. ■	

[Inspired by Neil Hunt's DVIPAGE, the author used such graymaps extensively when preparing new editions of *The Art of Computer Programming* in 1992–1998.]

190. (a) We must have  $\alpha_{j+1} = f(\alpha_j) \oplus \alpha_{j-1}$  for  $j \geq 1$ , where  $\alpha_0 = 0 \dots 0$  and  $f(\alpha) = ((\alpha \ll 1) \& 1 \dots 1) \oplus \alpha \oplus (\alpha \gg 1)$ . The elements of the bottom row  $\alpha_m$  satisfy the parity condition if and only if this rule makes  $\alpha_{m+1}$  entirely zero.

(b) True. The parity condition on matrix entries  $a_{ij}$  is  $a_{ij} = a_{(i-1)j} \oplus a_{i(j-1)} \oplus a_{(j+1)i} \oplus a_{(i+1)j}$ , where  $a_{ij} = 0$  if  $i = 0$  or  $i = m+1$  or  $j = 0$  or  $j = n+1$ . If two matrices  $(a_{ij})$  and  $(b_{ij})$  satisfy this condition, so does  $(c_{ij})$  when  $c_{ij} = a_{ij} \oplus b_{ij}$ .

(c) The upper left submatrix consisting of all rows that precede the first all-zero row (if any) and all columns that precede the first all-zero column (if any) is perfect. And this submatrix determines the entire matrix, because the pattern on the other side of a row or column of zeros is the top/bottom or left/right reflection of its neighbor. For example, if  $\alpha_{m'+1}$  is zero, then  $\alpha_{m'+1+j} = \alpha_{m'+1-j}$  for  $1 \leq j \leq m'$ .

(d) Starting with a given vector  $\alpha_1$  and using the rule in (a) will always lead to a row with  $\alpha_{m+1} = 0 \dots 0$ . Proof: We must have  $(\alpha_j, \alpha_{j+1}) = (\alpha_k, \alpha_{k+1})$  for some  $0 \leq j < k \leq 2^n$ , by the pigeonhole principle. If  $j > 0$  we also have  $(\alpha_{j-1}, \alpha_j) = (\alpha_{k-1}, \alpha_k)$ , because  $\alpha_{j-1} = f(\alpha_j) \oplus \alpha_{j+1} = f(\alpha_k) \oplus \alpha_{k+1} = \alpha_{k-1}$ . Therefore the first repeated pair begins with a row  $\alpha_k$  of zeros. Furthermore we have  $\alpha_i = \alpha_{k-i}$  for  $0 \leq i \leq k$ ; hence the first all-zero row  $\alpha_{m+1}$  occurs when  $m$  is  $k-1$  or  $k/2-1$ .

Rows  $\alpha_1, \dots, \alpha_m$  will form a perfect pattern unless there is a column of 0s. There are  $t > 0$  such columns if and only if  $t+1$  is a divisor of  $n+1$  and  $\alpha_1$  has the form  $\alpha 0 \alpha^R 0 \dots 0 \alpha$  ( $t$  even) or  $\alpha 0 \alpha^R 0 \dots 0 \alpha^R$  ( $t$  odd), where  $|\alpha|+1 = (n+1)/(t+1)$ .

(e) This starting vector does not have the form forbidden in (d).

**191.** (a) The former is  $\alpha_1, \alpha_2, \dots$  if and only if the latter is  $0\alpha_1 0\alpha_1^R, 0\alpha_2 0\alpha_2^R, \dots$ .

(b) Let the binary string  $a_0 a_1 \dots a_{N-1}$  correspond to the polynomial  $a_0 + a_1 x + \dots + a_{N-1} x^{N-1}$ , and let  $y = x^{-1} + 1 + x$ . Then  $\alpha_0 = 0 \dots 0$  corresponds to  $F_0(y)$ ;  $\alpha_1 = 10 \dots 0$  corresponds to  $F_1(y)$ ; and by induction  $\alpha_j$  corresponds to  $F_j(y)$ , mod  $x^N + 1$  and mod 2. For example, when  $N = 6$  we have  $\alpha_2 = 110001 \leftrightarrow 1 + x + x^5$  because  $x^{-1} \bmod (x^6 + 1) = x^5$ , etc.

(c) Again, induction on  $j$ .

(d) The identity in the hint holds by induction on  $m$ , because it is clearly true when  $m = 1$  and  $m = 2$ . Working mod 2, this identity yields the simple equations

$$F_{2k}(y) = yF_k(y)^2; \quad F_{2k-1}(y) = (F_{k-1}(y) + F_k(y))^2.$$

So we can go from the pair  $P_k = (F_{k-1}(y) \bmod (x^N + 1), F_k(y) \bmod (x^N + 1))$  to the pair  $P_{k+1}$  in  $O(n)$  steps, and to the pair  $P_{2k}$  in  $O(n^2)$  steps. We can therefore compute  $F_j(y) \bmod (x^N + 1)$  after  $O(\log j)$  iterations. Multiplying by  $f_\alpha(x) + f_\alpha(x^{-1})$  and reducing mod  $x^N + 1$  then allows us to read off the value of  $\alpha_j$ .

Incidentally,  $F_{n+1}(x)$  is the special case  $K_n(x, x, \dots, x)$  of a continuant polynomial; see Eq. 4.5.3–(4). We have  $F_{n+1}(x) = \sum_{k=0}^n \binom{n-k}{k} x^{n-2k} = i^{-n} U_n(ix/2)$ , where  $U_n$  is the classical Chebyshev polynomial defined by  $U_n(\cos \theta) = \sin((n+1)\theta)/\sin \theta$ .

**192.** (a) By exercise 191(c),  $c(q)$  is the least  $j > 0$  such that  $(x + x^{-1})F_j(x^{-1} + 1 + x) \equiv 0$  (modulo  $x^{2q} + 1$ ), using polynomial arithmetic mod 2. Equivalently, it's the smallest positive  $j$  for which  $F_j(y)$  is a multiple of  $(x^{2q} + 1)/(x^2 + 1) = (1 + x + \dots + x^{q-1})^2$ , when  $y = x^{-1} + 1 + x$ .

(b) Use the method of exercise 191(d) to evaluate  $((x + x^{-1})F_j(y)) \bmod (x^{2q} + 1)$  when  $j = M/p$ , for all prime divisors  $p$  of  $M$ . If the result is zero, set  $M \leftarrow M/p$  and repeat the process. If no such result is zero,  $c(q) = M$ .

(c) We want to show that  $c(2^e)$  is a divisor of  $3 \cdot 2^{e-1}$  but not of  $3 \cdot 2^{e-2}$  or  $2^{e-1}$ . The latter holds because  $F_{2^{e-1}}(y) = y^{2^{e-1}-1}$  is relatively prime to  $x^{2^{e+1}} + 1$ . The former holds because

$$F_{3 \cdot 2^{e-1}}(y) = y^{2^{e-1}-1} F_3(y)^{2^{e-1}} = y^{2^{e-1}-1} (1+y)^{2^e} = y^{2^{e-1}-1} (x^{-1} + x)^{2^e},$$

which is  $\equiv 0$  modulo  $x^{2^{e+1}} + 1$  but not modulo  $x^{2^{e+2}} + 1$ .

(d)  $F_{2^e-1}(y) = \sum_{k=1}^e y^{2^e-2^k}$ . Since  $y = x^{-1}(1+x+x^2)$  is relatively prime to  $x^q + 1$ , we have  $y^{-1} \equiv a_0 + a_1 x + \dots + a_{q-1} x^{q-1}$  (modulo  $x^q + 1$ ) for some coefficients  $a_i$ ; hence  $y^{-2^k} \equiv a_0 + a_1 x^{2^k} + \dots + a_{q-1} x^{2^k(q-1)} \equiv a_0 + a_1 x^{2^{k+e}} + \dots + a_{q-1} x^{2^{k+e}(q-1)} \equiv y^{-2^{k+e}}$  (modulo  $x^q + 1$ ) for  $0 \leq k < e$ , and it follows that  $F_{2^{2^e-1}}(y)$  is a multiple of  $x^{2^q} + 1$ .

(e) In this case  $c(q)$  divides  $4(2^{2^e} - 1)$ . Proof: Let  $x^q + 1 = f_1(x)f_2(x) \dots f_r(x)$  where  $f_1(x) = x + 1$ ,  $f_2(x) = x^2 + x + 1$ , and each  $f_i(x)$  is irreducible mod 2. Since  $q$  is odd, these factors are distinct. Therefore, in the finite field of polynomials mod  $f_j(x)$  for  $j \geq 3$ , we have  $y^{-2^k} = y^{-2^{k+e}}$  as in (d). Consequently  $F_{2^{2^e-1}}(y)$  is a multiple of  $f_3(x) \dots f_r(x) = (x^q + 1)/(x^3 + 1)$ . So  $F_{2^{2^e-1}}(y) = y F_{2^{2^e-1}}(y)^2$  is a multiple of  $(x^{2^q} + 1)/(x^2 + 1)$  as desired.

(f) If  $F_{c(q)}(y)$  is a multiple of  $x^{2^q} + 1$ , it's easy to see that  $c(2q) = 2c(q)$ . Otherwise  $F_{3c(q)}(y)$  is a multiple of  $F_3(y) = (1+y)^2 = x^{-2}(1+x)^4$ ; hence  $F_{6c(q)}(y)$  is a multiple of  $x^{4q} + 1$  and  $c(2q)$  divides  $6c(q)$ . The latter case can happen only when  $q$  is odd.

*Notes:* Parity patterns are related to a popular puzzle called “Lights Out,” which was invented in the early 1980s by Dario Uri, also invented independently about

continuant polynomial  
Chebyshev polynomial  
finite field  
Lights Out  
Uri

the same time by László Meerö and called **XL25**. [See David Singmaster's *Cubic Circular*, issues 7&8 (Summer 1985), 39–42; Dieter Gebhardt, *Cubism For Fun* **69** (March 2006), 23–25.] Klaus Sutner has pursued further aspects of this theory in *Theoretical Computer Science* **230** (2000), 49–73.

**193.** Let  $b_{(2i)(2j)} = a_{ij}$ ,  $b_{(2i+1)(2j)} = a_{ij} \oplus a_{(i+1)j}$ ,  $b_{(2i)(2j+1)} = a_{ij} \oplus a_{i(j+1)}$ , and  $b_{(2i+1)(2j+1)} = 0$ , for  $0 \leq i \leq m$  and  $0 \leq j \leq n$ , where we regard  $a_{ij} = 0$  when  $i = 0$  or  $i = m + 1$  or  $j = 0$  or  $j = n + 1$ . We don't have  $(b_{(2i)1}, b_{(2i)2}, \dots, b_{(2i)(2n+1)}) = (0, 0, \dots, 0)$  because  $(a_{i1}, \dots, a_{in}) \neq (0, \dots, 0)$  for  $1 \leq i \leq m$ . And we don't have  $(b_{(2i+1)1}, b_{(2i+1)2}, \dots, b_{(2i+1)(2n+1)}) = (0, 0, \dots, 0)$  because adjacent rows  $(a_{i1}, \dots, a_{in})$  and  $(a_{(i+1)1}, \dots, a_{(i+1)n})$  always differ for  $0 \leq i \leq m$  when  $m$  is odd.

**194.** Set  $\beta_i \leftarrow (1 \ll (n-i)) \mid (1 \ll (i-1))$  for  $1 \leq i \leq m$ , where  $m = \lceil n/2 \rceil$ . Also set  $\gamma_i \leftarrow (\beta_1 \& \alpha_{i1}) + (\beta_2 \& \alpha_{i2}) + \dots + (\beta_m \& \alpha_{im})$ , where  $\alpha_{ij}$  is the  $j$ th row of the parity pattern that begins with  $\beta_i$ ; vector  $\gamma_i$  records the diagonal elements of such a matrix. Then set  $r \leftarrow 0$  and apply subroutine N of answer 195 for  $i \leftarrow 1, 2, \dots, m$ . The resulting vectors  $\theta_1, \dots, \theta_r$  are a basis for all  $n \times n$  parity patterns with 8-fold symmetry.

To test if any such pattern is perfect, let the pattern starting with  $\theta_i$  first be zero in row  $c_i$ . If any  $c_i = n + 1$ , the answer is yes. If  $\text{lcm}(c_1, \dots, c_r) < n$ , the answer is no. If neither of these conditions decides the matter, we can resort to brute-force examination of  $2^r - 1$  nonzero linear combinations of the  $\theta$  vectors.

For example, when  $n = 9$  we find  $\gamma_1 = 111101111$ ,  $\gamma_2 = \gamma_3 = 010101010$ ,  $\gamma_4 = 000000000$ ,  $\gamma_5 = 001010100$ ; then  $r = 0$ ,  $\theta_1 = 011000110$ ,  $\theta_2 = 000101000$ ,  $c_1 = c_2 = 5$ . So there is no perfect solution.

In the author's experiments for  $n \leq 3000$ , “brute force” was needed only when  $n = 1709$ . Then  $r = 21$  and the values of  $c_i$  were all equal to 171 or 855 except that  $c_{21} = 342$ . The solution  $\theta_1 \oplus \theta_{21}$  was found immediately.

The answers for  $1 \leq n \leq 383$  are 4, 5, 11, 16, 23, 29, 30, 32, 47, 59, 62, 64, 65, 84, 95, 101, 119, 125, 126, 128, 131, 154, 164, 170, 185, 191, 203, 204, 239, 251, 254, 256, 257, 263, 314, 329, 340, 341, 371, 383.

[A fractal similar to Fig. 20, called the “mikado pattern,” appears in a paper by H. Eriksson, K. Eriksson, and J. Sjöstrand, *Advances in Applied Math.* **27** (2001), 365. See also S. Wolfram, *A New Kind of Science* (2002), rule 150R on page 439.]

**195.** Set  $\beta_i \leftarrow 1 \ll (m - i)$  and  $\gamma_i \leftarrow \alpha_i$  for  $1 \leq i \leq m$ ; also set  $r \leftarrow 0$ . Then perform the following subroutine for  $i = 1, 2, \dots, m$ :

**N1.** [Extract low bit.] Set  $x \leftarrow \gamma_i \& -\gamma_i$ . If  $x = 0$ , go to N4.

**N2.** [Find  $j$ .] Find the smallest  $j \geq 1$  such that  $\gamma_j \& x \neq 0$  and  $\gamma_j \& (x - 1) = 0$ .

**N3.** [Dependent?] If  $j < i$ , set  $\gamma_i \leftarrow \gamma_i \oplus \gamma_j$ ,  $\beta_i \leftarrow \beta_i \oplus \beta_j$ , and return to N1. (These operations preserve the matrix equation  $C = BA$ .) Otherwise terminate the subroutine (because  $\gamma_i$  is linearly independent from  $\gamma_1, \dots, \gamma_{i-1}$ ).

**N4.** [Record a solution.] Set  $r \leftarrow r + 1$  and  $\theta_r \leftarrow \beta_i$ . ■

At the conclusion, the  $m - r$  nonzero vectors  $\gamma_i$  are a basis for the vector space of all linear combinations of  $\alpha_1, \dots, \alpha_m$ ; they're characterized by their low bits.

**196.** (a) #0a; #cea3; #e7ae97; #f09d8581.

(b) If  $\lambda x = \lambda x'$ , the result is clear because  $l = l'$ . Otherwise we have either  $\alpha_1 < \alpha'_1$  or  $(\alpha_1 = \alpha'_1 \text{ and } \alpha_2 < \alpha'_2)$ ; the latter case can occur only when  $x \geq 2^{16}$ .

(c) Set  $j \leftarrow k$ ; while  $\alpha_j \oplus \#80 < \#40$ , set  $j \leftarrow j - 1$ . Then  $\alpha(x^{(i)})$  begins with  $\alpha_j$ .

Meerö  
XL25  
Singmaster  
Gebhardt  
Sutner  
Knuth  
mikado pattern  
Eriksson  
Eriksson  
Sjöstrand  
Wolfram  
vector space

197. (a) #000a; #03a3; #7b97; #d834dd41.

(b) Lexicographic order is *not* preserved when, say,  $x = \text{\#ffff}$  and  $x' = \text{\#10000}$ .

(c) To answer this question properly one needs to know that the 2048 integers in the range  $\text{\#d800} \leq x < \text{\#e000}$  are not legal codepoints of UCS; they are called *surrogates*. With this understanding,  $\beta(x^{(i)})$  begins at  $\beta_k$  if  $\beta_k \oplus \text{\#dc00} \geq \text{\#0400}$ , otherwise it begins at  $\beta_{k-1}$ .

198.  $a = \text{\#e50000}$ ,  $b = 3$ ,  $c = \text{\#16}$ . (We could let  $b = 0$ , but then  $a$  would be huge. This trick was suggested by P. Raynaud-Richard in 1997. The stated constants, suggested by R. Pournader in 2008, are the smallest possible.)

199. We want  $\alpha_1 > \text{\#c1}$ ;  $2^8\alpha_1 + \alpha_2 < \text{\#f490}$ ; and either  $(\alpha_1 \& -\alpha_1) + \alpha_1 < \text{\#100}$  or  $\alpha_1 + \alpha_2 > \text{\#17f}$ . These conditions hold if and only if

$$(\text{\#c1} - \alpha_1) \& (2^8\alpha_1 + \alpha_2 - \text{\#f490}) \& (((\alpha_1 \& -\alpha_1) + \alpha_1 - \text{\#100}) | (\text{\#17f} - \alpha_1 - \alpha_2)) < 0.$$

Markus Kuhn suggests adding the further clause ' $\& (\text{\#20} - ((2^8\alpha_1 + \alpha_2) \oplus \text{\#eda0}))$ ', to ensure that  $\alpha_1\alpha_2$  doesn't begin the encoding of a surrogate.

200. If  $\$0 = (x_7 \dots x_1x_0)_{256}$  then  $\$3$  is set to the symmetric function  $S_2(x_7, x_4, x_2)$ .

201. `MOR x, c, x`, where  $c = \text{\#f0f0f0f0f0f0f0f0f}$ .

202. `MOR x, x, c`, where  $c = \text{\#c0c030300c0c0303}$ ; then `MOR x, mone, x`. (See answer 209.)

203.  $a = \text{\#0008000400020001}$ ,  $b = \text{\#0f0f0f0f0f0f0f0f}$ ,  $c = \text{\#0606060606060606}$ ,  $d = \text{\#0000002700000000}$ ,  $e = \text{\#2a2a2a2a2a2a2a2a}$ . (The ASCII code for 0 is  $6 + \text{\#2a}$ ; the ASCII code for  $a$  is  $6 + \text{\#2a} + 10 + \text{\#27}$ .)

204.  $p = \text{\#8008400420021001}$ ,  $q = \text{\#8020080240100401}$  (the transpose of  $p$ ),  $r = \text{\#4080102004080102}$  (a symmetric matrix), and  $m = \text{\#aa55aa55aa55aa55}$ .

205. Shuffle, but with  $p \leftrightarrow q$ ,  $r = \text{\#0804020180402010}$ ,  $m = \text{\#f0f0f0f0f0f0f0f0f}$ .

206. Just change  $p$  to  $\text{\#0880044002200110}$ . (Incidentally, these shuffles can also be defined as permutations on  $z = (z_{63} \dots z_1z_0)_2$  in another way: The outshuffle maps  $z_j \mapsto z_{(2j) \bmod 63}$ , for  $0 \leq j < 63$ , while the inshuffle maps  $z_j \mapsto z_{(2j+1) \bmod 65}$ .)

207. Do `MOR y, p, x`; `MOR y, y, p`; `MOR t, y, q`; `PUT rM, m1`; `MUX y, y, t`; `MOR t, t, q`; `PUT rM, m2`; `MUX y, y, t`. In both cases  $p = \text{\#2004801002400801}$ ; for triple-zip,  $q = \text{\#402010080402018}$ ,  $m_1 = \text{\#4949494949494949}$ ,  $m_2 = \text{\#dbdbdbdbdbdbdbdbdb}$ ; for the inverse,  $q = \text{\#0402018040201008}$ ,  $m_1 = \text{\#0707070707070707}$ ,  $m_2 = \text{\#3f3f3f3f3f3f3f}$ .

208. (Solution by H. S. Warren, Jr.) The text's 7-swap, 14-swap, 28-swap method can be implemented with only 12 instructions:

```
MOR t, x, c1; MOR t, c1, t; PUT rM, m1; MUX y, x, t;
MOR t, y, c2; MOR t, c2, t; PUT rM, m2; MUX y, y, t;
MOR t, y, c3; MOR t, c3, t; PUT rM, m3; MUX y, y, t;
```

here  $c1 = \text{\#4080102004080102}$ ,  $c2 = \text{\#2010804002010804}$ ,  $c3 = \text{\#0804020180402010}$ ,  $m1 = \text{\#aa55aa55aa55aa55}$ ,  $m2 = \text{\#cccc3333cccc3333}$ ,  $m3 = \text{\#f0f0f0f0f0f0f0f0f}$ .

209. Four instructions suffice: `MXOR y, p, x`; `MXOR x, mone, x`; `MXOR x, x, q`; `XOR x, x, y`; here  $p = \text{\#80c0e0f0f8fcfeff}$ ,  $mone = -1$ , and  $q = \bar{p}$ .

210. `SLU x, one, x`; `MOR x, b, x`; `AND x, x, a`; `MOR x, x, #ff`; here register `one` = 1.

211. In general, element  $ij$  of the Boolean matrix product  $AXB$  is  $\bigvee \{x_{kl} \mid a_{ik}b_{lj} = 1\}$ . For this problem we choose  $a_{ik} = [i \subseteq k]$  and  $b_{lj} = [l \supseteq j]$ ; the answer is '`MOR t, f, a`; `MOR t, b, t`' where  $a = \text{\#80c0a0f088ccaaff}$  and  $b = \text{\#ff5533110f050301} = a^T$ .

surrogates  
Raynaud-Richard  
Pournader  
Kuhn  
surrogate  
symmetric function  
Warren  
swap  
MUX  
Boolean matrix product



(Notice that this trick gives a simple test  $[f = \hat{f}]$  for monotonicity. Furthermore, the 64-bit result  $(t_{63} \dots t_1 t_0)_2$  gives the coefficients of the multilinear representation

$$f(x_1, \dots, x_6) = (t_{63} + t_{62}x_6 + \dots + t_1x_1x_2x_3x_4x_5 + t_0x_1x_2x_3x_4x_5x_6) \bmod 2,$$

if we substitute **MXOR** for **MOR**, by the result of exercise 7.1.1–11.)

**212.** If  $\cdot$  denotes **MXOR** as in (183) and  $b = (\beta_7 \dots \beta_1 \beta_0)_{256}$  has bytes  $\beta_j$ , we can evaluate  $c = (a \cdot B_0^L) \oplus ((a \ll 8) \cdot (B_1^L + B_0^U)) \oplus ((a \ll 16) \cdot (B_2^L + B_1^U)) \oplus \dots \oplus ((a \ll 56) \cdot (B_7^L + B_6^U))$ , where  $B_j^U = (q\beta_j) \& m$ ,  $B_j^L = (((q\beta_j) \ll 8) + \beta_j) \& \overline{m}$ ,  $q = \#0080402010080402$ , and  $m = \#7f3f1f0f07030100$ . (Here  $q\beta_j$  denotes *ordinary* multiplication of integers.)

**213.** In this big-endian computation, register **nn** holds  $-n$ , and register **data** points to the octabyte following the given bytes  $\alpha_{n-1} \dots \alpha_1 \alpha_0$  in memory (with  $\alpha_{n-1}$  first). The constants **aa** =  $\#8381808080402010$  and **bb** =  $\#339bcf6530180c06$  correspond to matrices  $A$  and  $B$ , found by computing the remainders  $x^k \bmod p(x)$  for  $72 \leq k < 80$ .

SET	c,0	$c \leftarrow 0$ .	LDOU	t,data,nn	$t \leftarrow \text{next octa.}$
LDOU	t,data,nn	$t \leftarrow \text{next octa.}$	XOR	u,u,c	$u \leftarrow u \oplus c$ .
ADD	nn,nn,8	$n \leftarrow n - 8$ .	SLU	c,v,56	$c \leftarrow v \ll 56$ .
BZ	nn,2F	Done if $n = 0$ .	SRU	v,v,8	$v \leftarrow v \gg 8$ .
1H MXOR	u,aa,t	$u \leftarrow t \cdot A$ .	XOR	u,u,v	$u \leftarrow u \oplus v$ .
MXOR	v,bb,t	$v \leftarrow t \cdot B$ .	XOR	t,t,u	$t \leftarrow t \oplus u$ .
ADD	nn,nn,8	$n \leftarrow n - 8$ .	PBN	nn,1B	Repeat if $n > 0$ . ■

A similar method finishes the job, with no auxiliary table needed:

2H SET	nn,8	$n \leftarrow 8$ .	SRU	v,v,8	$v \leftarrow v \gg 8$ .
3H AND	x,t,ffff	$x \leftarrow \text{high byte.}$	XOR	t,t,v	$t \leftarrow t \oplus v$ .
MXOR	u,aaa,x	$u \leftarrow x \cdot A'$ .	SUB	nn,nn,1	$n \leftarrow n - 1$ .
MXOR	v,bbb,x	$v \leftarrow x \cdot B'$ .	PBP	nn,3B	Repeat if $n > 0$ .
SLU	t,t,8	$t \leftarrow t \ll 8$ .	XOR	t,t,c	$t \leftarrow t \oplus c$ .
XOR	t,t,u	$t \leftarrow t \oplus u$ .	SRU	crc,t,48	Return $t \gg 48$ . ■

Here **aaa** =  $\#8381808080808080$ , **bbb** =  $\#0383c363331b0f05$ , and **ffff** =  $\#ff00\dots00$ .

*The Books of the Big-Endians have been long forbidden.*

— LEMUEL GULLIVER, *Travels Into Several Remote Nations of the World* (1726)

**214.** By considering the irreducible factors of the characteristic polynomial of  $X$ , we must have  $X^n = I$  where  $n = 2^3 \cdot 3^2 \cdot 5 \cdot 7 \cdot 17 \cdot 31 \cdot 127 = 168661080$ . Neill Clift has shown that  $l(n-1) = 33$  and found the following sequence of 33 **MXOR** instructions to compute  $Y = X^{-1} = X^{n-1}$ : **MXOR t,x,x**; **MXOR \$1,t,x**; **MXOR \$2,t,\$1**; **MXOR \$3,\$2,\$2**; **MXOR t,\$3,\$3**;  $S^6$ ; **MXOR t,t,\$2**;  $S^3$ ; **MXOR \$1,t,\$1**; **MXOR t,\$1,\$3**;  $S^{13}$ ; **MXOR t,t,\$1**;  $S$ ; **MXOR y,t,x**; here  $S$  stands for ‘**MXOR t,t,t**’. To test if  $X$  is nonsingular, do **MXOR t,y,x** and compare **t** to the identity matrix  $\#8040201008040201$ .

**215.** **SADD \$0,x,0**; **SADD \$1,x,a**; **NEG \$0,32,\$0**; **2ADDU \$1,\$1,\$0**; **SLU \$0,b,\$1**; then **BN \$0,Yes**; here  $a = \#aaaaaaaaaaaaaaaa$  and  $b = \#2492492492492492$ .

**216.** Start with  $s_k \leftarrow 0$  and  $t_k \leftarrow -1$  for  $0 \leq k < m$ . Then do the following for  $1 \leq k \leq m$ : If  $x_k \neq 0$  and  $x_k < 2^m$ , set  $l \leftarrow \lambda x_k$  and  $s_l \leftarrow s_l + x_k$ ; if  $t_l < 0$  or  $t_l > x_k$ , also set  $t_l \leftarrow x_k$ . Finally, set  $y \leftarrow 1$  and  $k \leftarrow 0$ ; while  $y \geq t_k$  and  $k < m$ , set  $y \leftarrow y + s_k$  and  $k \leftarrow k + 1$ . Double precision  $n$ -bit arithmetic is sufficient for  $y$  and  $s_k$ . [This pleasant algorithm appeared in D. Eppstein’s blog, 2008.03.22.]

**217.** See R. D. Cameron, *U.S. Patent 7400271* (15 July 2008); *Proc. ACM Symp. Principles and Practice of Parallel Programming* **12** (2008), 91–98.

multilinear representation  
MXOR  
big-endian  
GULLIVER  
SWIFT  
characteristic polynomial  
Clift  
MXOR  
identity matrix  
SADD  
NEG  
2ADDU  
table lookup by shifting  
Eppstein  
Cameron  
Patent

# 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.

- 0–1 matrices, 67–70, *see also* Bitmaps.
  - multiplication of, 50–51, 56, 107.
  - transposing, 15, 56, 67, 69, 80.
  - triangularizing, 68.
- 0–1 principle, 54.
- 1 (the constant  $(\dots 111)_2$ ), 3, 8, 9, 50, 71, 76, 107.
- 2-adic chains, 23–27, 37, 61, 91, 96.
- 2-adic fractions, 9, 75.
- 2-adic integers: Infinite binary strings  $(\dots x_2x_1x_0)_2$  subject to arithmetic and bitwise operations, 2, 8, 16, 21, 53, 55, 61.
  - as a metric space, 74.
- 2-bit encoding for 3-state data, 28–31, 63.
- 2-cube equivalence, 29–30.
- 2-dimensional data allocation, 16.
- 2ADDU (times 2 and add unsigned), 79, 84, 108.
- 3-valued logic, 31, 63.
- 4-neighbors, *see* Rook-neighbors, 40.
- 4ADDU (times 4 and add unsigned), 77, 79.
- 8-neighbors, *see* King-neighbors, 40.
- 8ADDU (times 8 and add unsigned), 79.
- 16ADDU (times 16 and add unsigned), 79.
- $\infty$  (infinity), 8, 55.
- $\delta$ -maps, 84.
- $\delta$ -shifts, 16, 57.
  - cyclic, 17, 58.
- $\delta$ -swaps, 13–16, 50, 55–56, 107.
- $\lambda x ([\lg x])$ , *see* Binary logarithm.
- $\mu$  (average memory access time), 118.
- $\mu_k$  and  $\mu_{d,k}$ , *see* Magic masks.
- $\nu x$ , *see* Sideways addition.
- $\pi$  (circle ratio), as “random” example, 17.
- $\rho x$ , *see* Ruler function.
- $v$  (instruction cycle time), 118.
- Absorption laws, 3.
- Abstract RISC (reduced-instruction-set computer) model, 26.
- Ackland, Bryan David, 44.
- Acyclic digraph, 33.
- Addition, 3, 19.
  - bitwise, 19, 87.
  - modulo 5, 60.
  - scattered, 18, 57.
  - sideways, 2, 11–12, 55, 62, 79, 88, 94.
  - unary, 60.
- Adjacency lists, 62.
- Adjacency matrices of graphs, 28, 62.
- Adventure game, 85.
- Agrawal, Dharma Prakash (धर्म प्रकाश अग्रवाल), 71.
- Albers, Susanne, 88.
- Allouche, Jean-Paul, 78.
- Alpha channels, 59.
- Alphabetic data, 20, 59.
- Analysis of algorithms, 55, 85.
- Ancestors in a forest, 33.
  - nearest common, 33–35, 64.
- AND (bitwise conjunction), 2–3.
- Animating functions, 53, 56.
- Arc lists, 62.
- Ariyoshi, Hiromu (有吉弘), 92.
- Arndt, Jörg Uwe, 76, 84.
- Array storage allocation, 16, 22, 54, 59.
- ASCII: American Standard Code for Information Interchange, iv, 59, 69, 118.
- Associative laws, 3, 72.
- Asterisk codes for subcubes, 18, 63.
- Averages, bitwise, 19, 59.
- Background of an image, 42–43.
- Balanced branching functions, 53.
- Balanced ternary notation, 63, 79.
- Banyan networks, 81.
- Basic RAM (random-access machine)
  - model, 26–27, 62, 91.
- Baumgart, Bruce Guenther, 12.
- Bays, John Carter, 77.
- BDIF (byte difference), 20, 86–87.
- Beneš, Václav Edvard, 13.
- Bentley, Jon Louis, 95.
- Berlekamp, Elwyn Ralph, 21, 73, 98.
- Bernshtein, Sergei Natanovich (Бернштейн, Сергей Натанович), 103.
- BESM-6 (БЕСМ-6) computer, 83.
- Beyer, Wendell Terry, 42.
- Bézier, Pierre Etienne, splines, 48, 66–67, 103.
- Big-endian convention, 6–8, 12, 20, 77, 103–104, 108.
- Binary basis, 71.
- Binary logarithm ( $\lambda x = [\lg x]$ ), 10–11, 21–22, 25, 33–35, 55–56, 60–61, 64, 70.
- Binary recurrence relations, 8, 10, 11, 55.
- Binary search trees, 64, 79.
- Binary tree structures, 32.
- Binary valuation, *see* Ruler function.
- Binary-coded decimal notation, 60.
- Bipartite graphs, 14–15, 97.
- Bit boards, 32, 63.
- Bit codes for subcubes, 18, 63.
- Bit permutations, 13–17, 25, 50.

- Bit reversal, 12–13, 25, 27, 55, 56, 96, 97.
- Bit slices, 19, 70.
- Bitmaps, 39–48, 64–68.
  - cleaning, 65.
  - drawing on, 48.
  - filling contours in, 44–48, 66–67.
  - rotation and transposition of, 67.
- Bitwise manipulations, 1–108.
- Black pixels, 4, 40–41, 47–48, 67.
- Bolyai, János, 36.
- Bookworm problem, 54.
- Boolean matrices, 50, 69, *see also* Bitmaps.
  - multiplication of, 50–51, 56, 107.
- Borkowski, Ludwik Stefan, 31.
- Borrows, 86–87, 96.
- Boundary curves, digitized, 44–48.
- Bouton, Charles Leonard, 71.
- Branch instructions, 26, 90, *see also* Branchless computation.
- Branching functions, 53, 56.
- Branchless computation, 10, 23–27, 48–49, 61, 69, 70.
- Braymore, Caroline, 1.
- Breadth-first search, 91, 97.
- Brent, Richard Peirce, 83.
- Bresenham, Jack Elton, 102.
- Breuer, Melvin Allen, 94.
- Broadword chains, 23–27, 60–62, 65, 96.
  - strong, 61.
- Broadword computations, 21–27, 60–62, 65, 99.
- Brodal, Gerth Stølting, 22.
- Brodnik, Andrej, 27.
- Bron, Coenraad, 92.
- Brooker, Ralph Anthony, 2.
- Brown, David Trent, 51.
- Bruijn, Nicolaas Govert de, cycles, 10.
- Büchi, Julius Richard, 75.
- Butterfly networks, 56.
- Byte: An 8-bit quantity, 7–8.
- Byte permutations, 50.
- Bytes, parallel operations on, *see* Multibyte processing.
- Cache memory, 5, 35, 49, 77, 91.
- Cache-oblivious array addressing, *see* Zip.
- Cahn, Leonard, 40.
- Cameron, Robert Douglas, 108.
- Cancellation law, 72.
- Cantor, Georg Ferdinand Ludwig Philipp, 85.
- Cardinality of a set, 11.
- Carries, 18–19, 25, 86–87.
- Cartesian coordinates, 44.
- Cartesian trees, 79, 95.
- Cellular automata, 40–43, 65–66.
- Characteristic polynomial of a matrix, 108.
- Chebyshev (= Tschebyscheff), Pafnutii Lvovich (Чебышев, Пафнутий Львович), polynomials, 105.
- Cheshire cat, 42–43, 65, 66, 100.
- Chessboards, 32, 63.
- Chung, Kin-Man (鍾建民), 17, 58.
- Cigar-shaped curve, 102.
- Circles, digitized, 44, 47.
- Circular lists, 62, 100.
- Cleaning images, 65, 98.
- Clift, Neill Michael, 108.
- Cliques, maximal, 62–63.
- Closed bitmaps, 65.
- Colex ordering of integers, 79.
- Collation of bits, 2.
- Colman, George, the younger, 1.
- Combinations, 75–76.
- Commutative laws, 3, 71.
- Comparator modules, 58.
- Comparison of bytes, 21, 60.
- Complementation, 3, 52, 92.
- Complete binary trees, 33, 74.
  - infinite, 53.
- Composition of permutations, 53, 56–57.
- Compression of scattered bits, 16, 57, 83.
- Conditional-set instructions, 9–10, 48–49, 77–79, 88.
- Conic sections, digitizing, 44–48, 66–67.
- Conic splines, 103.
- Conjunction, in 3-valued logic, 31.
- Connectivity structure of an image, 41–43, 65–66.
- Consensus of subcubes, 63.
- Continuant polynomials, 105.
- Control points, 48.
- Convex optimization, 85.
- Conway, John Horton, 40, 73, 74, 98.
  - field, 52.
- CRC (cyclic redundancy check), 51, 70.
- Crossbar modules, 14–15, 58.
- CSNZ (conditional set if nonzero), 10, 48–49, 88.
- CSOD (conditional set if odd), 79.
- CSZ (conditional set if zero), 9, 77, 78.
- Curvature: Reciprocal of the radius, 102.
- Custering, 39, 44, 64–65.
- Cycles in a graph, 15.
- Cyclic redundancy checking, 51, 70.
- Cyclic shifts, 17, 56, 86.
- Cylinder, hyperbolic, 39, 97.
- Dahlheimer, Thorsten, 86.
- Dallos, József, 9.
- Dates, packed, 4, 60.
- de Bruijn, Nicolaas Govert, cycles, 10.
- Depth of a Boolean function, 13.
- Descartes, René, coordinates, 44.
- Dietz, Henry Gordon, 19, 86.
- Digitization of contours, 44–48, 66–67.
- Dijkstra, Edsger Wybe, 85.
- Dilated numbers, *see* Scattered arithmetic, Zip.

- Dirichlet, Johann Peter Gustav Lejeune, generating function, 78.
- Discrete logarithm, *see* Binary logarithm.
- Disjointness testing, 58.
- Disjunction, in 3-valued logic, 31.
- Distance between 2-adic integers, 74.
- Distinct bytes, testing for, 59.
- Distribution networks, *see* Mapping networks.
- Distributive laws, 3, 72.
- Divide and conquer paradigm, 12, 16.
- Divisibility by 3, 70.
- Division, 54.
  - avoiding, 4, 54.
  - by 10, 24.
  - by powers of 2, 3–4.
  - in Conway's field, 52.
  - of 2-bit numbers, 59.
- Dominating sets, minimum, 98.
- Don't-cares, 18, 29–30, 81, 94.
- Dot-minus operation ( $x \dot{-} y$ ), v, 20, 24, 61, 82, 96.
- Double order for traversing trees, 100–101.
- Dovetailing, 16, 59, *see also* Perfect shuffles, Zip.
- Drawing on a bitmap, 48.
- Duality between 0 and 1, 99.
- Duguid, Andrew Melville, 13.
- DVIPAGE program, 104.
- Edges between pixels, 44–48, 66–67.
- EDSAC computer, 2, 11.
- Eight queens problem, 92.
- Ellipses, 44–47, 102, 103.
- Encoding of ternary data, 28–31, 63.
- Eofill (even/odd filling), 47.
- Eppstein, David Arthur, 108.
- Equality of bytes, 20, 59, 60.
- Equivalence, in 3-valued logic, 63.
- Eratosthenes of Cyrene (Ἐρατοσθένης ὁ Κυρηναῖος); sieve (κόσκινον), 5, 54.
- Eriksson, Henrik, 106.
- Eriksson, Kimmo, 106.
- Escher, Giorgio Arnaldo (= George Arnold), 37.
- Escher, Maurits Cornelis, 37.
- Euclid (Εὐκλείδης), 36.
- Extracting bits, 2, 4, 8.
  - and compressing them, 16–17, 57, 83.
  - the least significant only ( $2^{\rho x}$ ), 8–10, 18, 21.
  - the most significant only ( $2^{\lambda x}$ ), 11, 55, 60–62, 89.
- Fast Fourier transforms, 56.
- Ferranti Mercury computer, 2.
- Fibonacci, Leonardo, of Pisa (= Leonardo filio Bonacii Pisano), numbers, 36.
- Fibonacci number system, 36, 64; *see also* NegaFibonacci number system.
  - odd, 96.
- Fibonacci polynomials, 67–68.
- Fields, algebraic, 50, 52, 105.
- Fields of data, *see* Packing of data.
- Filling a contour in a bitmap, 44–48, 66–67.
- Fingerprints, 40.
- Finite fields, 50, 52, 105.
- Finite-state automata, 89.
- Fischer, Johannes Christian, 95.
- Fisher, Randall James, 19, 86.
- Fixed point arithmetic, 86, 103.
- Flag: A 1-bit indicator, 20, 59, 60.
- Floating point arithmetic, 10, 78.
- Floyd, Robert W, 58.
- Footprints, 87, 93.
- Fractals, 68, 78.
- Fractional precision, 4, 69.
- Fragmented fields, 18, 58.
- Fredman, Michael Lawrence, 22, 60.
- Freed, Edwin Earl, 55.
- Frey, Peter William, 94.
- Fuchs, David Raymond, 99.
- Full adders, 98.
  - for balanced ternary numbers, 63.
- Gabow, Harold Neil, 95.
- Games, 40, 52, 63, 65, 85, 93.
- Gaps, between prime numbers, 77.
  - between Ulam numbers, 93.
  - in a scattered accumulator, 85.
- Garbage collection, 27.
- Gardner, Martin, 40, 98.
- Gathering bits, 83.
- Gauß (= Gauss), Johann Friderich Carl (= Carl Friedrich), 36.
- Gebhardt, Dieter, 106.
- Generating functions, 55, 57.
  - Dirichlet, 78.
- Gill, Stanley, 11.
- Gillies, Donald Bruce, 11.
- Gladwin, Harmon Timothy, 8.
- Gosper, Ralph William, Jr., v, 56, 70.
  - hack, 4, 54.
- Graphs, 14–15.
  - algorithms on, 27–28, 62–63.
- Gray, Frank, binary code, 73, 89.
- Gray levels in image data, 59, 67.
- Greedy-footprint heuristic, 87, 93.
- GREG (global register definition), 9, 12.
- Grid structure, 36, 98.
- Group of functions, 53.
- Groupoids, multiplication tables for, 31, 63.
- Grundy, Patrick Michael, 71.
- Guibas, Leonidas John (Γκίμπας, Λεωνίδας Ιωάννου), v, 103.
- Gulliver, Lemuel, 108.
- Guo, Zicheng Charles (郭自成), 41, 65.
- Guy, Richard Kenneth, 73, 98.

- Hacks, 1–70.
- Hagerup, Torben, 88.
- HAKMEM, 26, 71, 75.
- Half adders, 98.
  - for balanced ternary numbers, 63.
- Hall, Richard Wesley, Jr., 41, 65.
- Hamburg, Michael Alexander, 75.
- Hardy, Godfrey Harold, 75.
- Harel, Dov (דב הרצל), 33.
- Håstad, Johan Torkel, 91.
- Heaps, 32.
  - sideways, 32–35, 63–64.
- Heckel, Paul Charles, 82.
- Herrmann, Francine, 36.
- Heun, Volker, 95.
- Hexadecimal constants, v.
- Hexadecimal digits, 69.
- Hobby, John Douglas, 48, 103.
- Holes in images, 42–43.
- Hollis, Jeffrey John, 62.
- Hudson, Richard Howard, 77.
- Hunt, Neil, 104.
- Hyperbolas, 44, 66, 75, 102, 103.
- Hyperbolic plane geometry, 35–39, 47, 64, 97.
- Hyperfloor function ( $2^{\lambda x}$ ), 11, 55, 60–62, 74, 89.
- Ide, Mikio (井手幹生), 92.
- Identities for bitwise operations, 3, 4, 52, 53, 55, 75, 77, 86.
- Identity matrix, 108.
- ILLIAC I computer, 11.
- Implication, in 3-valued logic, 31.
- Implicit data structures, 32–39, 63–64.
- Independent sets, maximal, 63.
- Infinite binary trees, 53.
- Infinite exclusive-or operation, 74.
- Infinite-precision numbers, 2, 4, 52.
- Inorder of nodes, 33.
- Inshuffles, 69, 80.
- Inside of a curve, 44.
- Interchanging selected bits, 71.
- Interchanging two bits, 55.
- Interleaving bits, 16, 59, *see also* Perfect shuffles, Zip.
- Internet, ii, iii.
- Inverse of a binary matrix, 70.
- Inverse of a permutation, 50, 80.
- Isometries, 73–74.
- Jardine, Nicholas, 92.
- Johnson, David Stifter, 92.
- Jordan, Marie Ennemond Camille,
  - curve theorem, 44.
- Kaas, Robert, 32.
- Kaasila, Sampo Juhani, 103.
- Katajainen, Jyrki Juhani, 35.
- Kerbosch, Joseph (= Joep) August
  - Gérard Marie, 92.
- King-neighbors, 40, 65.
- Kingwise connected components, 41–43, 65–66, 97.
- Kirsch, Russell Andrew, 40, 65.
- Knight moves, 63.
- Knödel, Walter, 92.
- Knuth, Donald Ervin (高德纳), i, v, 22, 77, 78, 93, 99, 103, 104, 106.
- Kuhn, Markus Günther, 107.
- Lakhtakia, Akhlesh (अखिलेश लखटकिया), 75.
- Lamport, Leslie B., 19, 20, 59.
- Lander, Leon Joseph, 77.
- Large megabytes:  $2^{20}$  bytes, 77.
- Largest element of a set, 11.
- Larvala, Samuli Kristian, 83.
- Latin-1 supplement to ASCII, 85.
- Läuter, Martin, 10.
- Lawrie, Duncan Hamish, 81.
- LDTU (load tetra unsigned), 103.
- Le Corre, J., 13.
- Leap year, 88.
- Least common ancestors, *see* Nearest common ancestors.
- Least significant 1 bit ( $2^{\rho x}$ ), 8–9, 21.
- Lee, Ruby Bei-Loh (李佩露), 83.
- Left-to-right minimum, 95.
- Leftmost bits, 10–11, 22, 55.
- Lehmer, Derrick Henry, 4.
- Leiserson, Charles Eric, 10, 55.
- Lenfant, Jacques, 80.
- Lenstra, Hendrik Willem, Jr., 52, 73.
- Levialdi Ghiron, Stefano, 42–43.
- Lexicographic order, 18, 68.
- lg, *see* Binary logarithm.
- Life game, 40, 65.
- Lights Out puzzle, 105–106.
- Linked allocation, 91.
- Little-endian convention, 6–8, 12, 20, 28, 76, 77.
- Littlewood, John Edensor, 75.
- Lobachevsky, Nikolai Ivanovich
  - (Лобачевский, Николай Иванович), 36.
- Loukakis, Emmanuel (Λουκάκης, Μανώλης), 92.
- Lower bounds, 23–27, 61–62, 103.
- Lowercase letters, 59.
- Lowest common ancestors, *see* Nearest common ancestors.
- Loyd, Samuel, 77.
- Lukasiewicz, Jan, 31, 63.
- Lutz, Rüdiger Karl (= Rudi), 101.
- Lynch, William Charles, 11.
- Magic masks ( $\mu_k$  and  $\mu_{d,k}$ ), 9, 11–13, 16, 22, 37, 54, 71, 75, 76, 78–82, 84, 86, 88, 89, 96, 103.

- Majority function, 27, *see also* Median function.  
 Malgouyres, Rémy, 99.  
 Manchester Mark I computer, 2.  
 Mann, William Fredrick, 98.  
 Mapping modules, 58, 81.  
 Mapping networks, 58, 81.  
 Mapping three items into two-bit codes, 28–31, 63.  
 Mappings of bits, 17, 58, 81.  
 Margenstern, Maurice, 36.  
 Mark II computer (Manchester/Ferranti), 2.  
 Martin, Monroe Harnish, 10.  
 Mask: A bit pattern with 1s in key positions, 9, 12–13, 16–18, 20, 49, 50, 69.  
 Masked integers, *see* Scattered arithmetic.  
 Masking: ANDing with a mask, 31.  
 Matrices of 0s and 1s, 67–70, *see also* Bitmaps.  
     multiplication of, 50–51, 56.  
     transposing, 15, 56, 67, 69, 80.  
     triangularizing, 68.  
 Matrix multiplication, 50–51, 56.  
 Matrix transposition, 15, 56, 67, 69, 80.  
 max (maximum) function, 2, 31, 60.  
 Maximal cliques, 62–63.  
 Maximal independent sets, 63, 92.  
 Maximal proper subsets, 58.  
 Maybe, 31.  
 McCranie, Judson Shasta, 93.  
 Median function, v, 21, 86, 87.  
 Meerö, László, 106.  
 Mems: Memory accesses.  
 Merge sorting, 49.  
 METAFONT, 103.  
 mex (minimal excludant) function, 52.  
 Mikado pattern, 106.  
 Miller, Jeffrey Charles Percy, 11.  
 Miltersen, Peter Bro, 27.  
 min (minimum) function, 2, 31, 60.  
 Minimal excludant, 52.  
 Minimum element in subarray, 64.  
 Minsky, Marvin Lee, 66.  
 Missing subset sum, 70.  
 Mixed-radix representation, 60.  
 MMIX, ii, iv, 5, 7–10, 12, 19, 20, 28, 48–51, 54, 55, 57, 59, 60, 62, 67, 69, 70, 73, 77, 79, 84, 86, 87, 94, 103, 118.  
 mod (remainder) function, 4.  
 Mod-5 arithmetic, 60.  
 Modal logic, 31, 63.  
 none, 76, *see* –1.  
 Monotone Boolean functions, 70.  
 Monotonic portions of curves, 45–47, 66.  
 Monus operation ( $x \dot{-} y$ ), v, 20, 24, 61, 82, 96.  
 Moody, John Kenneth Montague (= Ken), 62.  
 MOR (multiple or), 12, 19, 50–51, 56, 69–70, 94, 103, 104, 107–108.  
 Morton, Guy Macdonald, 85.  
 Most significant 1 bit ( $2^{\lambda x}$ ), 2, 11, 60–62, 89.  
 MP3 (MPEG-1 Audio Layer III), 51.  
 Muller, David Eugene, 11.  
 Multibyte encoding, 68–69.  
 Multibyte processing, 19–23, 59–61.  
     addition, 19, 60, 87.  
     comparison, 20–21.  
     max and min, 60, 88.  
     modulo 5, 60.  
     potpourri, 87.  
     subtraction, 59, 60, 87.  
 Multilinear representation of a Boolean function, 108.  
 Multiple-precision arithmetic, 6.  
 Multiplication, 4, 10–11, 22, 61, 78.  
     avoiding, 21, 22, 59, 78.  
     by powers of 2, 3, 78.  
     in Conway's field, 52.  
     in groupoids, 31, 63.  
     lower bound for, 22, 26, 62.  
     of 0–1 matrices, 56; *see also* MOR and MXOR.  
     of polynomials mod 2, 70.  
     of signed bits, 29–30.  
 Munro, James Ian, 27.  
 MUX (multiplex), 50, 83, 86, 103, 107.  
 MXOR (multiple xor), 50–51, 56, 69–70, 73, 86, 107–108.  
 Mycroft, Alan, 20.  
 Navigation piles, 35, 64.  
 Nearest common ancestors, 33–35, 64.  
 Necessity, in 3-valued logic, 63.  
 NEG (negation), 49, 76, 108.  
 Negabinary number system, 52.  
 Negadecimal number system, 37.  
 NegaFibonacci number system, 36–39, 64.  
 Negation, 3, 52, 63.  
 Nested parentheses, 54.  
 Newline symbol, 20.  
 Nicely, Thomas Ray, 77.  
 Nim, 2, 52.  
     addition, 2, 52.  
     division, 52.  
     multiplication, 52, 73.  
     second-order, 52.  
 Noisy data, 65.  
 Non-Euclidean geometry, 35–36, 97.  
 Nonzero bytes, testing for, 20–21.  
 Nonzero register, converted to mask, 49.  
 NOT (bitwise complementation), 2.  
 Notational conventions, v, 81.  
      $\langle xyz \rangle$  (median of three), v.  
      $u \rightarrow^* v$  (transitive closure), 27.  
      $\bar{x}$  or  $\sim x$  (bitwise complement), 3.  
      $x^\oplus$  (suffix parity), 55.  
      $x \& y$  (bitwise AND), 3.  
      $x \mid y$  (bitwise OR), 3.

- $x \oplus y$  (bitwise XOR), 3.
- $x \ll y$  (bitwise left shift), 3.
- $x \gg y$  (bitwise right shift), 3.
- $x \ddagger y$  (zipper function), *see* Zip.
- $x - y$  ( $\max(x-y, 0)$ ), v, 20, 24, 61, 82, 96.
- $x?y:z = xy + \bar{x}z$  (mux), 60, 62.
- $z \cdot \chi$  (sheep-and-goats), 17–18, 57–58.
- NP-hard problems, 57.
- Null spaces, 68.
- NXOR (not xor), 79.
- Nybble: A 4-bit quantity, 12.
- Nyp: A 2-bit quantity, 12.
- Objects in images, 42.
- Octabyte or octa: A 64-bit quantity, 7–8.
- Odd Fibonacci number system, 96.
- Ofman, Yuri Petrovich (Офман, Юрий Петрович), 84.
- Omega network for routing, 56–57.
- One-to-many mapping, 17, 30.
- Ones counting, *see* Sideways addition.
- Online algorithms, 42–43, 66.
- Open bitmaps, 65.
- Optical character recognition, 40, 65.
- OR (bitwise disjunction), 2–3.
- Ordinal numbers, 73.
- Oriented forests and trees, 33–34, 42–43.
- Oriented paths, 27, 64.
- Outshuffles, 56, 69.
- Outside of a curve, 44.
- Overflowing memory, 92.
- Packed data, operating on, 4, 19–21, 31, 59–60, 63, 69.
- Packing of data, 4–6, 16, 31, 54, 64, 69, 70, 83.
- Page faults, 59.
- Paley, Raymond Edward Alan Christopher, 54, 75.
- Papadimitriou, Christos Harilaos (Παπαδημητρίου, Χρίστος Χαριλάου), 92.
- Papert, Seymour Aubrey, 66.
- Parabolas, 44, 66, 102.
- Parallel processing of subwords, 19–23, 59–61, 70.
- Parenthesis traces, 54.
- Parity function, 27, 62, 73, 79.
  - suffix, 55, 69, 91.
- Parity patterns, 67–68.
- Parkin, Thomas Randall, 77.
- Patents, 79, 83, 108.
- Paterson, Michael Stewart, 22, 90, 91.
- Pattern recognition, 40.
- Patterns, searching for, 20–22, 61.
- Pentagrid, 36–39, 64, 97.
- Perez, Aram, 51.
- Perfect hash functions, 78.
- Perfect parity patterns, 67–68.
- Perfect shuffles, 16, 50, 56, 57, 69, 80, 88.
  - 3-way, 69, 85.
- Period length, 62.
- Permutation matrices, 50.
- Permutation networks, 13–15, 56–58, 81.
- Permutations,
  - induced by index digits, 56.
  - of bits within a word, 13–17, 25, 50.
  - of bytes within a word, 50.
  - of the 2-adic integers, 53.
  - Omega-routable, 56–57.
- Perpendicular lines, 36.
- Peterson, William Wesley, 51.
- Phi ( $\phi$ ), 64.
- Pi ( $\pi$ ), as “random” example, 17.
- Pickover, Clifford Alan, 75.
- Pigeonhole principle, 104.
- Pipelined machine, 48–49.
- Pitteway, Michael Lloyd Victor, 45.
- Pixel algebra, v, 40.
- Pixel patterns, 4, 53.
- Pixels, 39–48, 64–68.
  - gray, 59, 67.
- Pólya, György (= George), 75.
- Polynomials modulo 2, 51, 57.
  - multiplication of, 70.
  - remainders of, 51, 57, 67–68.
- Polynomials modulo 5, 60.
- Population count, 11, *see* Sideways addition.
- Portability, 7–8.
- Possibility, in 3-valued logic, 63.
- Pournader, Roozbeh (روزبه پورنادر), 107.
- Pratt, Vaughan Ronald, 54, 58, 81, 84, 89, 103.
- Prefix problem, *see* Suffix parity function.
- Preorder of nodes, 33–35.
- Presume, Livingstone Irving, 55.
- Prime implicants, 63.
- Prime numbers, 5, 54.
- Printing, 39.
- Priority queues, 35.
- Pritchard, Paul Andrew, 77.
- Prodinger, Helmut, 78.
- Program counter, 26.
- Projection functions, 9.
- Prokop, Harald, 10, 55.
- Quadratic forms, 45–47, 66–67.
- Quadrees, 85.
- Quantifications, 74, 89.
- Queen graph, 92.
- Quick, Jonathan Horatio, 53, 58.
- Quilt, 4.
- Rabin, Michael Oser (מיכאל עוזר רבין), 84.
- Radix  $-2$ , 52.
- Radix conversion, 60.
- Radix exchange sort, 91.
- RAM (random-access machine), 26–27, 62, 91.
- Raman, Rajeev, 85.

- Ramshaw, Lyle Harold, 21.
- Randall, Keith Harold, 10, 55.
- Randomized data structures, 79.
- Range checking, 60.
- Range minimum query problem, 64.
- Rank of a binary matrix, 68.
- Rasters, 39, *see* Bitmaps.
- Rational 2-adic numbers, 61.
- Ray, Louis Charles, 40.
- Raynaud-Richard, Pierre, 107.
- Reachability problem, 27–28, 33.
- Rearrangeable networks, *see* Permutation networks.
- Recurrence relations, 8, 10, 11, 37, 51, 55, 67.
- Recursive processes, 15, 17, 32, 52, 72, 80, 84.
- Redundant representations, 103.
- Reflection of bits, 12–13, 25, 55, 56, 96, 97.
- Regular languages, 61.
- Reitwiesner, George Walter, 55.
- Remainders mod  $2^n - 1$ , 11.
- Remainders mod  $2^n$ , 4.
- Remainders of polynomials mod 2, 51, 57, 67–68.
- Removal of bits, 8.
- Replication of bits, 17, 58, 88.
- Representation,
  - of graphs, 27, 62.
  - of permutations, 57.
  - of sets as integers, 11, 18, 28, 58, 62–63, 75.
  - of three states with two bits, 28–31, 63.
- Reversal of bits, 12–13, 25, 27, 55, 56, 96, 97.
- Right-to-left minimum, 95.
- Rightmost bits, 8–10, 54.
- Rochdale, Simon, 1.
- Rokicki, Tomas Gerhard, v, 55, 79.
- Rook-neighbors, 40, 100.
- Rookwise connected components, 41–43, 66, 97.
- Rosenfeld, Azriel (עזריאל רוזנפלד), 41.
- Rotation of square bitmaps, 67.
- Rote, Günter (= Rothe, Günther Alfred Heinrich), 66.
- Rounding, 33, 86.
  - to an odd number, 2, 59, 86.
- Ruler function ( $\rho x$ ), 8, 20, 21, 25, 26, 28, 32, 33, 35, 53, 55, 56, 60, 64, 78, 100.
  - summed, 95.
- Runlength encoding, 100, *see also* Edges between pixels.
- Runs of 1s, 8, 11, 22–23, 55, 61.
- Rutovitz, Denis, 40.
- S, the letter, 48.
- S1S, 74–75.
- Saccheri, Giovanni Girolamo, 36.
- SADD (sideways addition), 9, 28, 76, 78, 79, 108.
- Samet, Hanan (חנן סמט), 85.
- Saturating addition, 92.
- Saturating subtraction, *see* Monus operation.
- Scattered arithmetic, 18, 58.
  - addition, 18, 57.
  - shifting, 58.
  - subtraction, 58.
- Scattering bits, 83.
- Schieber, Baruch Menachem (ברוך מנחם שיבר), 33.
- Schläfli, Ludwig, 97.
- Schroeppe, Richard Crabtree, 26, 52, 82.
- Seal, David, 10.
- Second-order logic, 74–75.
- Security holes, 69.
- Segmented broadcasting, *see* Stretching bits.
- Segmented sieves, 77.
- Sequential allocation, 91.
- SET, the game, 93.
- Sets, represented as integers, 11, 18, 27–28, 58, 62–63, 75.
  - maximal proper subsets of, 58.
- Shades of gray, 67.
- Shallit, Jeffrey Outlaw, 78.
- Sheep-and-goats operation, 17–18, 57–58.
- Shi, Zhi-Jie Jerry (史志杰), 83.
- Shift instructions, 3, 19, 21, 52, 61.
  - signed, 10, 49, 76, 78.
  - table lookup via, 5, 23, 69, 77, 88.
- Shift sets, 24–25.
- Shirakawa, Isao (白川功), 92.
- Shrinking of images, 42–43, 66.
- Shuffle network for routing, 56.
- Sibling links, 32, 63.
- Sibson, Robin, 92.
- Sideways addition, 2, 11–12, 55, 62, 79, 94.
  - bitwise, 11, 88.
  - function  $\nu x$ , 11, 27, 55, 78.
  - summed, 55, 82.
- Sideways heaps, 32–35, 63–64.
- Sieve of Eratosthenes, 5, 54.
- Signed bits, representation of, 29, 55.
- Signed right shifts, 10, 49, 76, 78.
- SIMD (single instruction, multiple data) architecture, 19.
- Simply connected components, 43.
- Singmaster, David Breyer, 106.
- Six-register algorithm, 103.
- Sjöstrand, Jonas Erik, 106.
- Slanina, Matteo, 74.
- Sleator, Daniel Dominic Kaplan, 4, 98.
- Slepian, David, 13.
- SLU (shift left unsigned), 5, 107.
- Smallest element of a set, 11.
- Smearing bits to the right, 8, 11, 78.
- Sorted data, 54.



- Sorting, 60, 75, 83.
  - networks for, 58.
- Soule, Stephen Parke, 87.
- Sprague, Roland Percival, 71.
- Squaring a polynomial, 57.
- Squines, 48, 66, 103.
- SR (shift right, preserving the sign), 10, 49, 76, 78.
- SRU (shift right unsigned), 5, 9, 78.
- Standard networks of comparators, 58.
- Stanford GraphBase, ii, iii.
- Steele, Guy Lewis, Jr., v, 16, 57, 80, 83.
- Sterne, Laurence, iii.
- Stockmeyer, Larry Joseph, 81, 84.
- Stockton, Fred G., 102.
- Stolfi, Jorge, v, 103.
- Storage allocation, 16, 22, 54, 59.
- Strachey, Christopher, 12.
- Straight lines, digitizing, 66.
- Stretching bits, 58, 88.
- Strings, searching for special bytes in, 20, 70.
- Strong broadword chains, 61.
- STTU (store tetra unsigned), 103.
- Subcubes, 18, 63.
- Subset sum, first missing, 70.
- Subsets, 11, 27–28, 62–63, 75.
  - generating all, 18.
  - maximal proper, 58.
- Subtraction, 3, 52, 59.
  - bitwise, 59, 87.
  - modulo 5, 60.
  - saturating, *see* Monus operation.
  - scattered, 58.
  - unary, 60.
- Suffix parity function, 55, 69, 91.
- Sum of bits, *see* Sideways addition.
  - weighted, 55.
- Surrogates, 107.
- Surroundedness tree, 43, 66.
- Sutner, Klaus, 106.
- Swapping bits, 12–15, 55–56, 107.
  - between variables, 71.
- SWAR methods, 19–23, 59–61.
- SWARC compiler, 86.
- Swift, Jonathan, 108.
- Sylow, Peter Ludvig Mejdell, 2-subgroup, 74.
- Symmetric functions, Boolean, 62, 107.
- Symmetric group, 74.
- Symmetric order of nodes, 33.
- Table lookup, 9, 10, 85.
  - by shifting, 5, 23, 69, 77, 88.
- Tarjan, Robert Endre, 33, 95.
- Ternary vectors, 31.
- Tessellation, 36, 47, 64.
- Tetrabyte or tetra: A 32-bit quantity, 7–8.
- Text processing, 19–21, 59–60, 69–70.
- Theory meets practice, 21–22.
- Thinning an image, 40–41, 65.
- Thompson, Kenneth Lane, 68.
- Thorup, Mikkel, 88, 103.
- Three-register algorithm, 45–48, 66–67.
- Three-state encodings, 28–31, 63.
- Three-valued logic, 31, 63.
- Tiling, 36, 47, 64.
- Time, mixed-radix representation of, 60.
- Tocher, Keith Douglas, 2, 59, 85.
- Toruses, 65, 87.
- Trailing zeros, 8, *see* Ruler function.
- Transdichotomous methods, *see* Broadword computations.
- Transitive closure, 27, 33.
- Transposed allocation, 70, 77.
- Transposing a 0–1 matrix, 15, 56, 67, 69, 80.
- Traversal in postorder, 95, 100–101.
- Traversal in preorder, 94, 100–101.
- Treaps, 79.
- Triangularizing a 0–1 matrix, 68.
- Tricks versus techniques, 2, 104.
- Trinomials, 57.
- Triple zipper function, 69, 85.
- Tripily linked trees, 94–95, 100.
- TrueType, 103.
- Truth tables, 9, 29–30, 70.
- Tsukiyama, Shuji (築山修治), 92.
- Turing, Alan Mathison, 2.
  - machines, 98.
- Two's complement notation, 2, 26, 71.
- Typesetting, 39.
- UCS (Universal Character Set), 69.
- Ulam, Stanisław Marcin, 93.
  - numbers, 63.
- Ultraparallel lines, 36.
- Unary notation, 60.
- Unbiased rounding, 59, 86.
- Uncompressing bits, 57.
- Underflow mask, 90.
- Unger, Stephen Herbert, 19.
- Unicode, 69.
- Universal Character Set, 69.
- Unpacking of data, 2, 4–6, 57, 83.
- Unsigned 2-adic integers, 71.
- Unsolvable problems, 75.
- Upper halfplane, 97.
- Uppercase letters, 59.
- Urban, Genevieve Hawkins, 40.
- Uri, Dario, 105.
- UTF-8: 8-bit UCS Transformation
  - Format, 69.
- UTF-16: 16-bit UCS Transformation
  - Format, 69.

- van Emde Boas, Peter, 32.  
 Van Wyk, Christopher John, 99.  
 Variance, 57.  
 Veblen, Oswald, 44.  
 Vector space, basis for, 74, 106.  
 Vertex covers, minimal, 63.  
 Vishkin, Uzi Yehoshua (עוזי יהושע ויסקין), 33.  
 Vitale, Fabio, 35.  
 Vuillemin, Jean Etienne, 95.  
  
 Wada, Eiiti (和田英一), 76.  
 Warren, Henry Stanley, Jr., v, 8, 11, 12, 25, 51, 52, 71, 78, 83, 86, 107.  
 Wegner, Peter (= Weiden, Puttilo Leonovich = Вейден, Путтило Леонович), 8, 12.  
 Weighted sum of bits, 55.  
 Welter, Cornelis P., 74, 75.  
 Weste, Neil Harry Earle, 44.  
 Wheeler, David John, 11.  
 White pixels, 4, 40, 67.  
 Wilkes, Maurice Vincent, 11.  
 Willard, Dan Edward, 22, 60.  
 Wilson, David Whitaker, 93.  
 Wise, David Stephen, 85.  
 Wolfram, Stephen, 106.  
 Wong, Chak-Kuen (黃澤權), 17, 58.  
  
 Woodrum, Luther Jay, 77.  
 Woods, Donald Roy, 85.  
 Wraparound parity patterns, 67.  
 Wunderlich, Charles Marvin, 93.  
 Wyde: A 16-bit quantity, 7–8.  
  
 XL25 game, 106.  
 XOR (bitwise exclusive-or), 1–2.  
     identities involving, 3, 53, 55, 75.  
  
 Yannakakis, Mihalis (Γιαννακάκης, Μιχάλης), 92.  
  
 Z order, *see* Zip.  
 Zero-byte test, 20–21, 59.  
 Zero-one principle, 54.  
 Zero-or-set instructions, 9, 10, 88.  
 Zeta function, 78.  
 Zijlstra, Erik, 32.  
 Zimmermann, Paul Vincent Marie, 83.  
 Zip: The zipper function, 16, 50, 57, 66, 77, 80, 83, 85.  
     triple (three-way), 69, 85.  
 Zip-fastener method, 85.  
 ZSNZ (zero or set if nonzero), 10, 88.  
 ZSZ (zero or set if zero), 9.

# ASCII CHARACTERS

	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	
#2x		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	#2x
#3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	#3x
#4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	#4x
#5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	#5x
#6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	#6x
#7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	■	#7x
	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	

# MMIX OPERATION CODES

	# 0	# 1	# 2	# 3	# 4	# 5	# 6	# 7	
# 0x	TRAP 5v	FCMP v	FUN v	FEQL v	FADD 4v	FIX 4v	FSUB 4v	FIXU 4v	# 0x
	FLOT[I] 4v		FLOTU[I] 4v		SFLOT[I] 4v		SFLOTU[I] 4v		
# 1x	FMUL 4v	FCMPE 4v	FUNE v	FEQLE 4v	FDIV 40v	FSQRT 40v	FREM 4v	FINT 4v	# 1x
	MUL[I] 10v		MULU[I] 10v		DIV[I] 60v		DIVU[I] 60v		
# 2x	ADD[I] v		ADDU[I] v		SUB[I] v		SUBU[I] v		# 2x
	2ADDU[I] v		4ADDU[I] v		8ADDU[I] v		16ADDU[I] v		
# 3x	CMP[I] v		CMPU[I] v		NEG[I] v		NEGU[I] v		# 3x
	SL[I] v		SLU[I] v		SR[I] v		SRU[I] v		
# 4x	BN[B] v+π		BZ[B] v+π		BP[B] v+π		BOD[B] v+π		# 4x
	BNN[B] v+π		BNZ[B] v+π		BNP[B] v+π		BEV[B] v+π		
# 5x	PBN[B] 3v−π		PBZ[B] 3v−π		PBP[B] 3v−π		PBOD[B] 3v−π		# 5x
	PBNN[B] 3v−π		PBNZ[B] 3v−π		PBNP[B] 3v−π		PBEV[B] 3v−π		
# 6x	CSN[I] v		CSZ[I] v		CSP[I] v		CSOD[I] v		# 6x
	CSNN[I] v		CSNZ[I] v		CSNP[I] v		CSEV[I] v		
# 7x	ZSN[I] v		ZSZ[I] v		ZSP[I] v		ZSOD[I] v		# 7x
	ZSNN[I] v		ZSNZ[I] v		ZSNP[I] v		ZSEV[I] v		
# 8x	LDB[I] μ+v		LDBU[I] μ+v		LDW[I] μ+v		LDWU[I] μ+v		# 8x
	LDT[I] μ+v		LDTU[I] μ+v		LDO[I] μ+v		LDOU[I] μ+v		
# 9x	LDSF[I] μ+v		LDHT[I] μ+v		CSWAP[I] 2μ+2v		LDUNC[I] μ+v		# 9x
	LDVTS[I] v		PRELD[I] v		PREGO[I] v		GO[I] 3v		
# Ax	STB[I] μ+v		STBU[I] μ+v		STW[I] μ+v		STWU[I] μ+v		# Ax
	STT[I] μ+v		STTU[I] μ+v		STO[I] μ+v		STOU[I] μ+v		
# Bx	STSF[I] μ+v		STHT[I] μ+v		STCO[I] μ+v		STUNC[I] μ+v		# Bx
	SYNCD[I] v		PREST[I] v		SYNCID[I] v		PUSHGO[I] 3v		
# Cx	OR[I] v		ORN[I] v		NOR[I] v		XOR[I] v		# Cx
	AND[I] v		ANDN[I] v		NAND[I] v		NXOR[I] v		
# Dx	BDIF[I] v		WDIF[I] v		TDIF[I] v		ODIF[I] v		# Dx
	MUX[I] v		SADD[I] v		MOR[I] v		MXOR[I] v		
# Ex	SETH v	SETMH v	SETML v	SETL v	INCH v	INCMH v	INCM L v	INCL v	# Ex
	ORH v	ORMH v	ORML v	ORL v	ANDNH v	ANDNMH v	ANDNML v	ANDNL v	
# Fx	JMP[B] v		PUSHJ[B] v		GETA[B] v		PUT[I] v		# Fx
	POP 3v	RESUME 5v	[UN]SAVE 20μ+v		SYNC v	SWYM v	GET v	TRIP 5v	
	# 8	# 9	# A	# B	# C	# D	# E	# F	

$\pi = 2v$  if the branch is taken,  $\pi = 0$  if the branch is not taken