Note to readers:
Please ignore these
side notes; they’re just
hints to myself for
preparing the index,
and they’re often flaky!

KNUTH

THE ART OF
COMPUTER PROGRAMMING

VOLUME 4  PRE-FASCICLE 5B

INTRODUCTION
TO
BACKTRACKING

DONALD E. KNUTH  Stanford University

ADDISON–WESLEY

October 4, 2015

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 MIMIX computer.

Copyright © 2015 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 -95), 01 October 2015

October 4, 2015
PREFACE

Begin at the beginning, and do not allow yourself to gratify
a mere idle curiosity by dipping into the book, here and there.
This would very likely lead to your throwing it aside,
with the remark "This is much too hard for me."
and thus losing the chance of adding a very large item
to your stock of mental delights.
— LEWIS CARROLL, in Symbolic Logic (1896)

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,
3, and 4A 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 portion of fascicle 5 previews the opening
pages of Section 7.2.2 of The Art of Computer Programming, entitled "Backtrack
programming." The preceding section, 7.2.1, was about "Generating basic
combinatorial patterns" — namely tuples, permutations, combinations, partitions,
and trees. Now it's time to consider the non-basic patterns, the ones that have
a much less uniform structure. For these we generally need to make tentative
choices and then to back up when those choices need revision. Several subsections
(7.2.2.1, 7.2.2.2, etc.) will follow this introductory material.

* * *

The explosion of research in combinatorial algorithms since the 1970s has
meant that I cannot hope to be aware of all the important ideas in this field.
I've tried my best to get the story right, yet I fear that in many respects I'm woefully ignorant. So I beg expert readers to steer me in appropriate directions.

October 4, 2015
Please look, for example, at the exercises that I’ve classed as research problems (rated with difficulty level 46 or higher), namely exercises 14, . . . ; I’ve also implicitly mentioned or posed additional unsolved questions in the answers to exercises 6, 8, 42, 45, . . . . 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 31(b), 33, 44, . . . . Furthermore I’ve credited exercises . . . to unpublished work of . . . . Have any of those results ever appeared in print, to your knowledge?

I’ve got a historical question too: Have you any idea who originated the idea of “stamping” in data structures? (See Eq. 7.2.2–(26). It is quite different from the so-called time stamps in persistent data structures, although some programmers use the name “time stamp” for this kind of stamp.) It’s a technique that I’ve often seen in the programs that I’ve been reading during recent decades, but I wonder if it ever appeared in a book or paper that was published before, say, 1980.

* * *

Special thanks are due to . . . for their detailed 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
99 Unbrary 2015

D. E. K.

October 4, 2015
PREFACE 

v
Part of the Preface to Volume 4B

During the years that I’ve been preparing Volume 4, I’ve often run across basic techniques of probability theory that I would have put into Section 1.2 of Volume 1 if I’d been clairvoyant enough to anticipate them in the 1960s. Finally I realized that I ought to collect most of them together in one place, near the beginning of Volume 4B, because the story of these developments is too interesting to be broken up into little pieces scattered here and there.

Therefore this volume begins with a special section entitled “Mathematical Preliminaries Redux,” and future sections use the abbreviation ‘MPR’ to refer to its equations and its exercises.
MATHEMATICAL PRELIMINARIES REDUX

Many parts of this book deal with discrete probabilities, namely with a finite or countably infinite set \( \Omega \) of atomic events \( \omega \), each of which has a given probability \( \Pr(\omega) \), where

\[
0 \leq \Pr(\omega) \leq 1 \quad \text{and} \quad \sum_{\omega \in \Omega} \Pr(\omega) = 1.
\] (1)

\[\ldots\]

\[\text{For the complete text of the special MPR section, please see Pre-Fascicle 5a.}\]

Incidentally, Section 7.2.2 intentionally begins on a left-hand page, and its illustrations are numbered beginning with Fig. 68, because Section 7.2.1 ended on a right-hand page and its final illustration was Fig. 67. The editor has decided to treat Chapter 7 as a single unit, even though it will be split across several physical volumes.
7.2.2. Backtrack Programming

Now that we know how to generate simple combinatorial patterns such as tuples, permutations, combinations, partitions, and trees, we’re ready to tackle more exotic patterns that have subtler and less uniform structure. Instances of almost any desired pattern can be generated systematically, at least in principle, if we organize the search carefully. Such a method was christened “backtrack” by R. J. Walker in the 1950s, because it is basically a way to examine all fruitful possibilities while exiting gracefully from situations that have been fully explored.

Most of the patterns we shall deal with can be cast in a simple, general framework: We seek all sequences \(x_1x_2\ldots x_n\) for which some property \(P_k(x_1, x_2, \ldots, x_n)\) holds, where each item \(x_k\) belongs to some given domain \(D_k\) of integers. The backtrack method, in its most elementary form, consists of inventing intermediate “cutoff” properties \(P_i(x_1, \ldots, x_l)\) for \(1 \leq l < n\), such that

\[
P_i(x_1, \ldots, x_l) \text{ is true whenever } P_{i+1}(x_1, \ldots, x_{l+1}) \text{ is true}; \quad (1)
\]

\[
P_i(x_1, \ldots, x_l) \text{ is fairly easy to test, if } P_{i-1}(x_1, \ldots, x_{l-1}) \text{ holds.} \quad (2)
\]

(We assume that \(P_0()\) is always true. Exercise 1 shows that all of the basic patterns studied in Section 7.2.1 can easily be formulated in terms of domains \(D_k\) and cutoff properties \(P_i\).) Then we can proceed lexicographically as follows:

**Algorithm B (Basic backtrack).** Given domains \(D_k\) and properties \(P_i\) as above, this algorithm visits all sequences \(x_1x_2\ldots x_n\) that satisfy \(P_k(x_1, x_2, \ldots, x_n)\).

**B1.** [Initialize.] Set \(l \leftarrow 1\), and initialize the data structures needed later.

**B2.** [Enter level \(l\).] (Now \(P_{l-1}(x_1, \ldots, x_{l-1})\) holds.) If \(l > n\), visit \(x_1x_2\ldots x_n\) and go to B5. Otherwise set \(x_l \leftarrow \min D_l\), the smallest element of \(D_l\).

**B3.** [Try \(x_l\).] If \(P_l(x_1, \ldots, x_l)\) holds, update the data structures to facilitate testing \(P_{l+1}\), set \(l \leftarrow l + 1\), and go to B2.

**B4.** [Try again.] If \(x_l \neq \max D_l\), set \(x_l\) to the next larger element of \(D_l\) and return to B3.

**B5.** [Backtrack.] Set \(l \leftarrow l - 1\). If \(l > 0\), downdate the data structures by undoing the changes recently made in step B3, and return to B4. (Otherwise stop.)

The main point is that if \(P_l(x_1, \ldots, x_l)\) is false in step B3, we needn’t waste time trying to append any further values \(x_{l+1} \ldots x_n\). Thus we can often rule out huge regions of the space of all potential solutions. A second important point is that very little memory is needed, although there may be many, many solutions.

For example, let’s consider the classic problem of \(n\) queens: In how many ways can \(n\) queens be placed on an \(n \times n\) board so that no two are in the same
row, column, or diagonal? We can suppose that one queen is in each row, and that the queen in row \( k \) is in column \( x_k \), for \( 1 \leq k \leq n \). Then each domain \( D_k \) is \( \{1, 2, \ldots, n\} \); and \( P_n(x_1, \ldots, x_n) \) is the condition that

\[
x_j \neq x_k \quad \text{and} \quad |x_k - x_j| \neq k - j, \quad \text{for } 1 \leq j < k \leq n.
\]

(If \( x_j = x_k \) and \( j < k \), two queens are in the same column; if \( |x_k - x_j| = k - j \), they're in the same diagonal.)

This problem is easy to set up for Algorithm B, because we can let property \( P_1(x_1, \ldots, x_l) \) be the same as (3) but restricted to \( 1 \leq j < k \leq l \). Condition (1) is clear; and so is condition (2), because \( P_l \) requires testing (3) only for \( k = l \) when \( P_{l-1} \) is known. Notice that \( P_1(x_1) \) is always true in this case.

One of the best ways to learn about backtracking is to execute Algorithm B by hand in the special case \( n = 4 \) of the \( n \) queens problem: First we set \( x_1 \leftarrow 1 \). Then when \( l = 2 \) we find \( P_2(1, 1) \) and \( P_2(1, 2) \) false; hence we don't get to \( l = 3 \) until trying \( x_2 \leftarrow 3 \). Then, however, we're stuck, because \( P_3(1, 3, x) \) is false for \( 1 \leq x \leq 4 \). Backtracking to level 2, we now try \( x_2 \leftarrow 4 \); and this allows us to set \( x_3 \leftarrow 2 \). However, we're stuck again, at level 4; and this time we must back up all the way to level 1, because there are no further valid choices at levels 3 and 2. The next choice \( x_1 \leftarrow 2 \) does, happily, lead to a solution without much further ado, namely \( x_1, x_2, x_3, x_4 = 2, 4, 1, 3 \). And one more solution (3142) turns up before the algorithm terminates.

The behavior of Algorithm B is nicely visualized as a tree structure, called a search tree or backtrack tree. For example, the backtrack tree for the four queens problem has just 17 nodes,

![Backtrack Tree](image)

corresponding to the 17 times step B2 is performed. Here \( x_l \) is shown as the label of an edge from level \( l - 1 \) to level \( l \) of the tree. (Level \( l \) of the algorithm actually corresponds to the tree's level \( l - 1 \), because we've chosen to represent patterns by writing \( x_1 x_2 \ldots x_n \) instead of \( x_0 x_1 \ldots x_{n-1} \) in this discussion.) The profile \( (p_0, p_1, \ldots, p_n) \) of this particular tree—the number of nodes at each level—is \( (1, 4, 6, 4, 2) \); and we see that the number of solutions, \( p_n = p_4 \), is 2.

Figure 68 shows the corresponding tree when \( n = 8 \). This tree has 2057 nodes, distributed according to the profile \( (1, 8, 42, 140, 344, 568, 550, 312, 92) \). Thus the early cutoffs facilitated by backtracking have allowed us to find all 92 solutions by examine only 0.01% of the \( 8^8 = 16,777,216 \) possible sequences \( x_1 \ldots x_8 \). (And \( 8^8 \) is only 0.38% of the \( \binom{64}{8} = 4,426,165,368 \) ways to put eight queens on the board.)
Fig. 68. The problem of placing eight nonattacking queens has this backtrack tree.

Notice that, in this case, Algorithm B spends most of its time in the vicinity of level 5. Such behavior is typical: The backtrack tree for \( n = 16 \) queens has 1,141,190,303 nodes, and its profile is \((1, 16, 210, 2236, 19688, 141812, 838816, 3808456, 15324708, 46358876, 108478966, 193892860, 260303408, 253897632, 171158018, 72002088, 14772512)\), concentrated near level 12.

**Data structures.** Backtrack programming is often used when a huge tree of possibilities needs to be examined. Thus we want to be able to test property \( P_l \) as quickly as possible in step B3.

One way to implement Algorithm B for the \( n \) queens problem is to avoid auxiliary data structures and simply to make a bunch of sequential comparisons in that step: “Is \( x_i - x_j \in \{ j - l, l - j \} \) for some \( j < l \)?” Assuming that we access memory whenever referring to \( x_j \), given a trial value \( x_i \) in a register, such an implementation performs approximately 112 billion memory accesses when \( n = 16 \); that’s about 98 mens per node.

We can do better by introducing three simple arrays. Property \( P_l \) says essentially that the numbers \( x_k \) are distinct, and so are the numbers \( x_k + k \), and so are the numbers \( x_k - k \). Therefore we can use auxiliary Boolean arrays \( a_1 \ldots a_n, b_1 \ldots b_{n-1}, \text{ and } c_1 \ldots c_{2n-1} \), where \( a_j \) means ‘some \( x_k = j \)’, \( b_j \) means ‘some \( x_k + k - 1 = j \)’, and \( c_j \) means ‘some \( x_k - k + n = j \)’. Those arrays are readily updated and downdated if we customize Algorithm B as follows:

**B1**, [Initialize.] Set \( a_1 \ldots a_n \leftarrow 0 \ldots 0, b_1 \ldots b_{n-1} \leftarrow 0 \ldots 0, c_1 \ldots c_{2n-1} \leftarrow 0 \ldots 0 \), and \( t \leftarrow 1 \).

**B2**, [Enter level \( l \).] (Now \( P_{l-1}(x_1, \ldots, x_{l-1}) \) holds.) If \( l > n \), visit \( x_1 x_2 \ldots x_n \) and go to B5*. Otherwise set \( t \leftarrow 1 \).

**B3**, [Try \( t \).] If \( a_t = 1 \) or \( b_{t+l-1} = 1 \) or \( c_{t+l+n} = 1 \), go to B4*. Otherwise set \( a_t \leftarrow 1, b_{t+l-1} \leftarrow 1, c_{t+l+n} \leftarrow 1, x_l \leftarrow t, l \leftarrow l + 1 \), and go to B2*.

**B4**, [Try again.] If \( t < n \), set \( t \leftarrow t + 1 \) and return to B3*.

**B5**, [Backtrack.] Set \( l \leftarrow l - 1 \). If \( l > 0 \), set \( t \leftarrow x_l, c_{t+l+n} \leftarrow 0, b_{t+l-1} \leftarrow 0, a_t \leftarrow 0 \), and go to B4*. (Otherwise stop.)

Notice how step B5* neatly undoes the updates that step B3* had made, in the reverse order. Reverse order for downdating is typical of backtrack algorithms,
although there is some flexibility; we could, for example, have restored $a_t$ before $b_{t+l-1}$ and $c_{t-l+n}$, because those arrays are independent.

The auxiliary arrays $a$, $b$, $c$ make it easy to test property $P_t$ at the beginning of step $B^3*$, but we must also access memory when we update them and downdate them. Does that cost us more than it saves? Fortunately, no: the running time for $n = 16$ goes down to about 34 billion mems, roughly 30 mems per node.

Furthermore, we could keep the bit vectors $a$, $b$, $c$ entirely in registers, on a machine with 64-bit registers, assuming that $n \leq 32$. Then there would be just two memory accesses per node, namely to store $x_t \leftarrow t$ and later to fetch $t \leftarrow x_t$; however, quite a lot of in-register computation would become necessary.

**Walker’s method.** The 1950s-era programs of R. J. Walker organized backtracking in a somewhat different way. Instead of letting $x_t$ run through all elements of $D_t$, he calculated and stored the set

$$S_t = \{ x \in D_t \mid P_t(x_1, \ldots, x_{t-1}, x) \text{ holds} \}$$

upon entry to each node at level $l$. This computation can often be done efficiently all at once, instead of piecemeal, because some cutoff properties make it possible to combine steps that would otherwise have to be repeated for each $x \in D_t$. In essence, he used the following variant of Algorithm B:

**Algorithm W (Walker’s backtrack).** Given domains $D_t$ and cutoffs $P_t$ as above, this algorithm visits all sequences $x_1 x_2 \ldots x_n$ that satisfy $P_n(x_1, x_2, \ldots, x_n)$.

**W1.** [Initialize.] Set $l \leftarrow 1$, and initialize the data structures needed later.

**W2.** [Enter level $l$.] (Now $P_{t-1}(x_1, \ldots, x_{t-1})$ holds.) If $l > n$, visit $x_1 x_2 \ldots x_n$ and go to W4. Otherwise determine the set $S_l$ as in (5).

**W3.** [Try to advance.] If $S_l$ is nonempty, set $x_t \leftarrow \min S_l$, update the data structures to facilitate computing $S_{t+1}$, set $l \leftarrow l + 1$, and go to W2.

**W4.** [Backtrack.] Set $l \leftarrow l - 1$. If $l > 0$, downdate the data structures by undoing changes made in step W3, set $S_l \leftarrow S_l \setminus x$, and go back to W3.

Walker applied this method to the $n$ queens problem by computing $S_l = U \setminus A_t \setminus B_t \setminus C_t$, where $U = D_t = \{ 1, \ldots, n \}$ and

$$A_t = \{ x_j \mid 1 \leq j < l \}, \quad B_t = \{ x_j + j - l \mid 1 \leq j < l \}, \quad C_t = \{ x_j - j + l \mid 1 \leq j < l \}.$$  \hspace{1cm} (6)

He represented these auxiliary sets by bit vectors $a$, $b$, $c$, analogous to (but different from) the bit vectors of Algorithm $B^*$ above. Exercise 9 shows that the updating in step W3 is easy, using bitwise operations on $n$-bit numbers; furthermore, no downdating is needed in step W4. The corresponding run time when $n = 16$ turns out to be just 9.1 gigamems, or 8 mems per node.

Let $Q(n)$ be the number of solutions to the $n$ queens problem. Then we have

$$n = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16$$

$$Q(n) = 1 \quad 1 \quad 0 \quad 0 \quad 2 \quad 10 \quad 4 \quad 40 \quad 92 \quad 352 \quad 724 \quad 2680 \quad 14200 \quad 73712 \quad 365596 \quad 2279184 \quad 14772512$$

and the values for $n \leq 11$ were computed independently by several people during the nineteenth century. Small cases were relatively easy; but when T. B. Sprague
had finished computing $Q(11)$ he remarked that “This was a very heavy piece of work, and occupied most of my leisure time for several months. ... It will, I imagine, be scarcely possible to obtain results for larger boards, unless a number of persons co-operate in the work.” [See Proc Edinburgh Math. Soc 17 (1899), 43-68. Sprague was the leading actuary of his day.] Nevertheless, H. Oomen went on to evaluate $Q(12) = 14,200$—an astonishing feat of hand calculation—in 1910. [See W. Ahrens, Math. Unterhaltungen und Spiele 2, second edition (1918), 344.]

All of these hard-won results were confirmed in 1960 by R. J. Walker, using the SWAC computer at UCLA and the method of exercise 9. Walker also computed $Q(13)$; but he couldn’t go any further with the machine available to him at the time. The next step, $Q(14)$, was computed by Michael D. Kennedy at the University of Tennessee in 1963, commandeering an IBM 1620 for 120 hours. S. R. Bunch evaluated $Q(15)$ in 1974 at the University of Illinois, using about two hours on an IBM System 360-75; then J. R. Bitner found $Q(16)$ after about three hours on the same computer, but with an improved method.

Computers and algorithms have continued to get better, of course, and such results are now obtained almost instantly. Hence larger and larger values of $n$ lie at the frontier. The current record as of 2015 is $Q(26) = 22,317,699,616,364,044$, found in 2009 by Thomas B. Preufler of the University of Dresden. (His distributed computation occupied a dynamic cluster of up to 26 diverse FPGA devices for 270 days; those devices provided a total peak of 550 custom-designed hardware solvers to handle 25,204,802 subproblems individually.)

**Permutations and Langford pairs.** Every solution $x_1 \ldots x_n$ to the $n$ queens problem is a permutation of $\{1, \ldots, n\}$, and many other problems have the same property. Indeed, we’ve already seen Algorithm 7.2.1.2X, which is an elegant backtrack procedure specifically designed for special kinds of permutations. When that algorithm begins to choose the value of $x_1$, it makes all of the appropriate elements $\{1, 2, \ldots, n\} \setminus \{x_1, \ldots, x_{n-1}\}$ conveniently accessible in a linked list.

We can get further insight into such data structures by returning to the problem of Langford pairs, which was discussed at the very beginning of Chapter 7. That problem can be reformulated as the task of finding all permutations of $\{1, 2, \ldots, n\} \cup \{-1, -2, \ldots, -n\}$ with the property that

\[ x_j = k \quad \text{implies} \quad x_{j+k+1} = -k, \quad \text{for} \ 1 \leq j \leq 2n \text{ and } 1 \leq k \leq n. \]  

(7)

For example, when $n = 4$ there are two solutions, namely $23421314$ and $41321432$. (As usual we find it convenient to write $1$ for $-1$, $2$ for $-2$, etc.) Notice that whenever $x = x_1 x_2 \ldots x_{2n}$ is a solution, its “dual” $-x = (-x_2) \ldots (-x_2)(-x_1)$ is also a solution.

Here’s a Langford-inspired adaptation of Algorithm 7.2.1.2X, with the former notation modified slightly to match Algorithms B and W: We want to maintain pointers $p_0 p_1 \ldots p_n$ such that, if the positive integers not already present in $x_1 \ldots x_{n-1}$ are $k_1 < k_2 < \cdots < k_t$ when we’re choosing $x_i$, we have the linked list

\[ p_0 = k_1, \ p_{k_1} = k_2, \ldots, \ p_{k_{t-1}} = k_t, \ p_{k_t} = 0. \]  

(8)

Such a condition turns out to be easy to maintain.
Algorithm L (Langford pairs). This algorithm generates all solutions \(x_1 \ldots x_{2n}\) to (7) in lexicographic order, using pointers \(p_0 p_1 \ldots p_n\) that satisfy (8), and also using an auxiliary array \(y_1 \ldots y_{2n}\) for backtracking.

**L1. Initialize.** Set \(x_1 \ldots x_{2n} \leftarrow 0 \ldots 0,\ p_k \leftarrow k + 1\) for \(0 \leq k < n,\ p_n \leftarrow 0,\ l \leftarrow 1.\)

**L2. Enter level \(l.\)** Set \(k \leftarrow p_k.\) If \(k = 0,\) visit \(x_1x_2 \ldots x_{2n}\) and go to L5. Otherwise set \(j \leftarrow 0,\) and while \(x_1 < 0\) set \(l \leftarrow l + 1.\)

**L3.** [Try \(x_l = k.\) (At this point we have \(k = p_j.\)] If \(l + k + 1 > 2n,\) go to L5. Otherwise, if \(x_{l+k+1} = 0,\) set \(x_l \leftarrow k,\ x_{l+k+1} \leftarrow -k,\ y_l \leftarrow j,\ p_j \leftarrow p_k,\ l \leftarrow l + 1,\) and return to L2.

**L4.** [Try again.] (We’ve found all solutions that begin with \(x_1 \ldots x_{l-1} k\) or something smaller.) Set \(j \leftarrow k\) and \(k \leftarrow p_j,\) then go to L3 if \(k \neq 0.\)

**L5.** [Backtrack.] Set \(l \leftarrow l - 1.\) If \(l > 0\) do the following: While \(x_l < 0,\) set \(l \leftarrow l - 1.\) Then set \(k \leftarrow x_l,\ x_l \leftarrow 0,\ x_{l+k+1} \leftarrow 0,\ j \leftarrow y_l,\ p_j \leftarrow k,\) and go back to L4. Otherwise terminate the algorithm.

Careful study of these steps will reveal how everything fits together nicely. Notice that, for example, step L3 removes \(k\) from the linked list (8) by simply setting \(p_j \leftarrow p_k.\) That step also sets \(x_{l+k+1} \leftarrow -k,\) in accordance with (7), so that we can skip over position \(l + k + 1\) when we encounter it later in step L2.

The main point of Algorithm L is the somewhat subtle way in which step L5 undoes the deletion operation by simply setting \(p_j \leftarrow k.\) The pointer \(p_k\) still retains the appropriate link to the next element in the list, because \(p_k\) has not been changed by any of the intervening updates. (Think about it.) This is the germ of an idea called “dancing links” that we will explore in Section 7.2.2.1.

To draw the search tree corresponding to a run of Algorithm L, we can label the edges with the positive choices of \(x_l\) as we did in (4), while labeling the nodes with any previously set negative values that are passed over in step L2. For instance the tree for \(n = 4\) is

\[ \text{Solutions appear at depth } n \text{ in this tree, even though they involve } 2n \text{ values } x_1x_2 \ldots x_{2n}. \]

Algorithm L sometimes makes false starts and doesn’t realize the problem until probing further than necessary. Notice that the value \(x_l = k\) can appear only when \(l + k + 1 \leq 2n;\) hence if we haven’t seen \(k\) by the time \(l\) reaches \(2n - k - 1,\) we’re forced to choose \(x_l = k.\) For example, the branch 12 in (9) needn’t be pursued, because 4 must appear in \(\{x_1, x_2, x_3\}.\) Exercise 20 explains how to incorporate this cutoff principle into Algorithm L. When \(n = 17,\) it reduces the number of nodes in the search tree from 1.29 trillion to 330 billion,

October 4, 2015
and reduces the running time from 25.0 teramems to 8.1 teramems. (The amount of work has gone from 19.4 mems per node to 24.4 mems per node, because of the extra tests for cutoffs, yet there’s a significant overall reduction.)

Furthermore, we can “break the symmetry” by ensuring that we don’t consider both a solution and its dual. This idea, exploited in exercise 21, reduces the search tree to just 160 billion nodes and costs just 3.94 teramems—that’s 24.6 mems per node.

**Word rectangles.** Let’s look next at a problem where the search domains $D_i$ are much larger. An $m \times n$ word rectangle is an array of $n$-letter words* whose columns are $m$-letter words. For example,

\[
\begin{align*}
\text{status} & \\
\text{lowest} & \\
\text{utopia} & \\
\text{making} & \\
\text{sledge}
\end{align*}
\]

is a $5 \times 6$ word rectangle whose columns all belong to \textsc{Words(5757)}, the collection of 5-letter words in the Stanford GraphBase. To find such patterns, we can suppose that column $i$ contains the $x_i$th most common 5-letter word, where $1 \leq x_i \leq 5757$ for $1 \leq i \leq 6$; hence there are $5757^6 = 36, 406, 369, 848, 837, 732, 146, 649$ ways to choose the columns. In (10) we have $x_1 \ldots x_6 = 1446, 185, 1021, 2537, 66, 255$.

Of course very few of those choices will yield suitable rows; but backtracking will hopefully help us to find all solutions in a reasonable amount of time.

We can set this problem up for Algorithm B by storing the $n$-letter words in a trie (see Section 6.3), with one trie node of size $26$ for each $l$-letter prefix of a legitimate word, $0 \leq l \leq n$. For example, such a trie for $n = 6$ represents 15727 words with 23667 nodes. The prefix $st$ corresponds to node number 200, whose 26 entries are

\[(484, 0, 0, 0, 0, 1589, 0, 0, 0, 0, 2609, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1280, 0, 0, 251, 0, 0, 563, 0, 0, 0, 0, 1621, 0); \quad (11)\]

This means that $sta$ is node 484, $ste$ is node 1589, $..$, $sty$ is node 1621, and there are no 6-letter words beginning with $stb$, $stc$, $..$, $stz$. A slightly different convention is used for prefixes of length $n - 1$; for example, the entries for node 580, ‘corn’ are

\[(3879, 0, 0, 0, 3878, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0); \quad (12)\]

meaning that corn, corned, cornel, corner, and cornet are ranked 3879, 3878, 9002, 171, and 5013 in the list of 6-letter words.

* Whenever five-letter words are used in the examples of this book, they're taken from the 5757 Stanford GraphBase words as explained at the beginning of Chapter 7. Words of other lengths are taken from The Official SCRABBLE® Players Dictionary, fourth edition (Hasbro, 2005), because those words have been incorporated into many widely available computer games. Such words have been ranked according to the British National Corpus of 2007—where *the* occurs 5,405,633 times and the next-most common word, ‘of’, occurs roughly half as often (3,021,525). The OSPD4 list includes respectively (101, 1004, 4002, 8887, 15727, 23667, 29718, 20130, 22314) words of lengths (2, 3, . . ., 10), of which (97, 121, 1451, 4474, 6910, 8852, 9205, 8252, 6656) occur at least six times in the British National Corpus.
Suppose $x_1$ and $x_2$ specify the 5-letter column-words slums and total as in (10). Then the trie tells us that the next column-word $x_3$ must have the form $c_1c_2e_3c_4c_5$ where $c_1 \in \{a, e, i, o, r, u, y\}$, $c_2 \notin \{e, h, j, k, y, z\}$, $c_3 \in \{a, e, i, o, t, \}$, $c_4 \notin \{a, b, o\}$, and $c_5 \in \{a, e, i, o, u, y\}$. (There are 221 such words.)

Let $a_1\ldots a_m$ be the trie nodes corresponding to the prefixes of the first $l$ columns of a partial solution to the word rectangle problem. This auxiliary array enables Algorithm B to find all solutions, as explained in exercise 24. It turns out that there are exactly 625,415 valid $5 \times 6$ word rectangles, according to our conventions; and the method of exercise 24 needs about 19 teramens of computation to find them all. In fact, the profile of the search tree is

$$\begin{align*}
(1, 5757, 245830, 360728099, 579940198, 29621728, 625415),
\end{align*}$$

indicating for example that just 360,728,099 of the 57573 = 190,804,533,093 choices for $x_1x_2x_3$ will lead to valid prefixes of 6-letter words.

With care, exercise 24’s running time can be significantly decreased, once we realize that every node of the search tree for $1 \leq l \leq n$ requires testing 5757 possibilities for $x_l$ in step B3. If we build a more elaborate data structure for the 5-letter words, so that it becomes easy to run through all words that have a specific letter in a specific position, we can refine the algorithm so that the average number of possibilities per level that need to be investigated becomes only

$$\begin{align*}
(5757.0, 1607.9, 844.1, 273.5, 153.5, 100.8);
\end{align*}$$

the total running time then drops to 1.15 teramens. Exercise 25 has the details. And exercise 28 discusses a method that’s faster yet.

Commafree codes. Our next example deals entirely with four-letter words. But it’s not obscure: it’s an intriguing question of coding theory. The problem is to find a set of four-letter words that can be decoded even if we don’t put spaces or other delimiters between them: if we take any message that’s formed from words of the set by simply concatenating them together, like this, and if we look at any seven consecutive letters $\ldots x_1x_2x_3x_4x_5x_6x_7 \ldots$, exactly one of the four-letter substrings $x_1x_2x_3x_4$, $x_2x_3x_4x_5$, $x_3x_4x_5x_6$, $x_4x_5x_6x_7$ will be a codeword. Equivalently, if $x_1$, $x_2$, $x_3$, and $x_4$ are codewords, then $x_2x_3x_4x_5$ and $x_3x_4x_5x_6$ and $x_4x_5x_6x_7$ aren’t. (For example, ike isn’t.) Such a set is called a “commafree code” or a “self-synchronizing block code” of length four.

Commafree codes were introduced by F. H. C. Crick, J. S. Griffith, and L. E. Orgel Proc National Acad. Sci. 43 (1957), 416–421, and studied further by S. W. Golomb, B. Gordon, and L. R. Welch [Canadian Journal of Mathematics 10 (1958), 202–209], who considered the general case of $m$-letter alphabets and $n$-letter words. They constructed optimum commafree codes for all $m$ when $n = 2, 3, 5, 7, 9, 11, 13,$ and 15; and optimum codes for all $m$ were subsequently found also for $n = 17, 19, 21, \ldots$ (see exercise 32). We will focus our attention on the four-letter case here ($n = 4$), partly because that case is still very far from being resolved, but mostly because the task of finding such codes is especially instructive. Indeed, our discussion will lead us naturally to an understanding of several significant techniques that are important for backtrack programming in general.
To begin, we can see immediately that a commafree codeword cannot be "periodic," like doce or gaga. Such a word already appears within two adjacent copies of itself. Thus we’re restricted to aperiodic words like item, of which there are \( m^4 - m^2 \). Notice further that if item has been chosen, we aren’t allowed to include any of its cyclic shifts temi, emit, or mite, because they all appear within item item. Hence the maximum number of codewords in our commafree code cannot exceed \( (m^4 - m^2)/4 \).

For example, consider the binary case, \( m = 2 \), when this maximum is 3. Can we choose three four-bit “words,” one from each of the cyclic classes

\[
\begin{align*}
[0001] &= \{0001, 0010, 0100, 1000\}, \\
[0011] &= \{0011, 0110, 1100, 1001\}, \\
[0111] &= \{0111, 1110, 1011, 1011\},
\end{align*}
\]

so that the resulting code is commafree? Yes: One solution in this case is simply to choose the smallest word in each class, namely 0001, 0011, and 0111. (Alert readers will recall that we studied the smallest word in the cyclic class of any acyclic string in Section 7.2.1.1, where such words were called prime strings and where some of their remarkable properties were proved.)

That trick doesn’t work when \( m = 3 \), however, when there are \((81 - 9)/4 = 18\) cyclic classes. Then we cannot include 1112 after we’ve chosen 0001 and 0011. Indeed, a code that contains 0001 and 1112 can’t contain either 0011 or 0111.

We could systematically backtrack through 18 levels, choosing \( x_1 \) in [0001] and \( x_2 \) in [0011], etc., and rejecting each \( x_i \) as in Algorithm B whenever we discover that \( \{x_1, x_2, \ldots, x_i\} \) isn’t commafree. For example, if \( x_1 = 0010 \) and we try \( x_2 = 1001 \), this approach would backtrack because \( x_1 \) occurs inside \( x_2 x_1 \).

But a na"ive strategy of that kind, which recognizes failure only after a bad choice has been made, can be vastly improved. If we had been clever enough, we could have looked a little bit ahead, and never even considered the choice \( x_2 = 1001 \) in the first place. Indeed, after choosing \( x_1 = 0010 \), we can automatically exclude all further words of the form *001, such as 2001 when \( m \geq 3 \) and 3001 when \( m \geq 4 \).

Even better pruning occurs if, for example, we’ve chosen \( x_1 = 0001 \) and \( x_2 = 0011 \). Then we can immediately rule out all words of the forms 1*** or ***0, because \( x_1 *** \) includes \( x_2 \) and \( *** x_2 \) includes \( x_1 \). Already we could then deduce, in the case \( m \geq 3 \), that classes [0002], [0021], [0111], [0211], and [1112] must be represented by 0002, 0021, 0111, 0211, and 2111, respectively; each of the other three possibilities in those classes has been wiped out!

Thus we see the desirability of a lookahead mechanism.

**Dynamic ordering of choices.** Furthermore, we can see from this example that it’s not always good to choose \( x_1 \), then \( x_2 \), then \( x_3 \), and so on when trying to satisfy a general property \( P_n(x_1, x_2, \ldots, x_n) \) in the setting of Algorithm B. Maybe the search tree will be much smaller if we first choose \( x_0 \), say, and then turn next to some other \( x_j \), depending on the particular value of \( x_0 \) that was selected. Some orderings might have much better cutoff properties than others, and every branch of the tree is free to choose its variables in any desired order.
Indeed, our comma-free coding problem for ternary 4-tuples doesn't dictate any particular ordering of the 18 classes that is likely to keep the search tree small. Therefore, instead of calling those choices \( x_1, x_2, \ldots, x_{18} \), it's better to identify them by the various class names, namely \( x_{0001}, x_{0002}, x_{0011}, x_{0012}, x_{0021}, x_{0022}, x_{0102}, x_{0111}, x_{0112}, x_{0122}, x_{0211}, x_{0212}, x_{0221}, x_{0222}, x_{1122}, x_{1222} \). (Algorithm 7.2.1.1F is a good way to generate those names.) At every node of the search tree we can then choose a convenient variable on which to branch, based on previous choices. After beginning with \( x_{0001} \leftarrow 0001 \) at level 1 we might decide to try \( x_{0011} \leftarrow 0011 \) at level 2; and then, as we've seen, the choices \( x_{0002} \leftarrow 0002, x_{0021} \leftarrow 0021, x_{0111} \leftarrow 0111, x_{0211} \leftarrow 0211, \) and \( x_{1112} \leftarrow 2111 \) are forced, so we should make them at levels 3 through 7.

Furthermore, after those forced moves are made, it turns out that they don't force any others. But only two choices for \( x_{0012} \) will remain, while \( x_{0122} \) will have three. Therefore it will probably be wiser to branch on \( x_{0012} \) rather than on \( x_{0122} \) at level 8. (Incidentally, it also turns out that there is no comma-free code with \( x_{0001} = 0001 \) and \( x_{0011} = 0011 \), except when \( m = 2 \).)

It's easy to adapt Algorithms B and W to allow dynamic ordering. Every node of the search tree can be given a “frame” in which we record the variable being set and the choice that was made. This choice of variable and value can be called a “move” made by the backtrack procedure.

Dynamic ordering can be helpful also after backtracking has taken place. If we continue the example above, where \( x_{0001} = 0001 \) and we've explored all cases in which \( x_{0011} = 0011 \), we need not continue by trying another value for \( x_{0011} \). We do want to remember that \( 0011 \) should no longer be considered legal, until \( x_{0001} \) changes; but we could decide to explore next a case such as \( x_{0002} = 2000 \) at level 2. In fact, \( x_{0002} = 2000 \) is quickly seen to be impossible in the presence of \( 0001 \) (see exercise 34). An even more efficient choice at level 2, however, is \( x_{0012} = 0012 \), because that branch immediately forces \( x_{0002} = 0002, x_{0022} = 0022, x_{0122} = 0122, x_{0222} = 0222, x_{1222} = 1222, \) and \( x_{0112} = 1001 \).

**Sequential allocation redux.** The choice of a variable and value on which to branch is a delicate tradeoff. We don't want to devote more time to planning than we'll save by having a good plan.

If we're going to benefit from dynamic ordering, we'll need efficient data structures that will lead to good decisions without much deliberation. On the other hand, elaborate data structures need to be updated whenever we branch to a new level, and they need to be downgraded whenever we return from that level. Algorithm L illustrates an efficient mechanism based on linked lists; but sequentially allocated lists are often even more appealing, because they are cache-friendly and they involve fewer accesses to memory.

Assume then that we wish to represent a set of items as an unordered sequential list. The list begins in a cell of memory pointed to by \texttt{HEAD}, and \texttt{TAIL} points just beyond the end of the list. For example,

\begin{verbatim}
... HEAD 3 9 1 4 TAIL ...
\end{verbatim}

October 4, 2015
is one way to represent the set \( \{1, 3, 4, 9\} \). The number of items currently in the set is \( \text{TAIL} - \text{HEAD} \); thus \( \text{TAIL} = \text{HEAD} \) if and only if the list is empty. If we wish to insert a new item \( x \), knowing that \( x \) isn’t already present, we simply set

\[
\text{MEM}[\text{TAIL}] \leftarrow x, \quad \text{TAIL} \leftarrow \text{TAIL} + 1.
\]  

Conversely, if \( \text{HEAD} \leq P < \text{TAIL} \), we can easily delete \( \text{MEM}[P] \):

\[
\text{TAIL} \leftarrow \text{TAIL} - 1; \quad \text{if } P \neq \text{TAIL}, \text{set } \text{MEM}[P] \leftarrow \text{MEM}[\text{TAIL}].
\]

(We’ve tacitly assumed in (17) that \( \text{MEM}[\text{TAIL}] \) is available for use whenever a new item is inserted. Otherwise we would have had to test for memory overflow.)

Sometimes, of course, we want to delete an item from a list without knowing its \( \text{MEM} \) location — although we do know that it is somewhere in that list. In such cases we can maintain an “inverse list,” assuming that all items \( x \) lie in the range \( 0 \leq x < M \). For example, (16) becomes the following, if \( M = 10 \):

\[
\text{HEAD} \quad \text{TAIL}
\]

\[
\begin{array}{cccccccccc}
\ldots & 3 & 9 & 1 & 4 & & & \ldots \\
\ldots & & & & & \text{HEAD} & & \ldots \\
\end{array}
\]

(Shaded cells have undefined contents.) With this setup, insertion (17) becomes

\[
\text{MEM}[\text{TAIL}] \leftarrow x, \quad \text{MEM}[\text{HEAD} + x] \leftarrow \text{TAIL}, \quad \text{TAIL} \leftarrow \text{TAIL} + 1,
\]

and \( \text{TAIL} \) will never exceed \( \text{HEAD} + M \). Similarly, deletion of \( x \) becomes

\[
P \leftarrow \text{MEM}[\text{HEAD} + x], \quad \text{TAIL} \leftarrow \text{TAIL} - 1;
\]

\[
\text{if } P \neq \text{TAIL}, \text{set } y \leftarrow \text{MEM}[\text{TAIL}], \text{MEM}[P] \leftarrow y, \text{MEM}[\text{HEAD} + y] \leftarrow P.
\]

For example, after deleting ‘9’ from (19) we would obtain this:

\[
\text{HEAD} \quad \text{TAIL}
\]

\[
\begin{array}{cccccccccc}
\ldots & 3 & 4 & 1 & & & \ldots \\
\ldots & & & & & \text{HEAD} & & \ldots \\
\end{array}
\]

In more elaborate situations we also want to test whether or not a given item \( x \) is present. If so, we can keep more information in the inverse list. A particularly useful variation arises when the list that begins at \( \text{IHEAD} \) contains a complete permutation of the values \( \{\text{HEAD}, \text{HEAD} + 1, \ldots, \text{HEAD} + M - 1\} \), and the memory cells beginning at \( \text{HEAD} \) contain the inverse permutation — although only the first \( \text{TAIL} - \text{HEAD} \) elements of that list are considered to be “active.”

For example, in our comma-free code problem with \( m = 3 \), we can begin by putting items representing the \( M = 18 \) cycle classes \( \{0001\}, \{0002\}, \ldots, \{1222\} \)
into memory cells \texttt{HEAD} through \texttt{HEAD} + 17. Initially they're all active, with \texttt{TAIL} = \texttt{HEAD} + 18 and \texttt{MEM[IHEAD} + 0\texttt{] = HEAD} + 0 for 0 \leq c < 18. Then whenever we decide to choose a codeword for class c, we delete c from the active list by using a souped-up version of (21) that maintains full permutations:

\begin{verbatim}
P ← \texttt{MEM[IHEAD} + c\texttt{]}, \quad \texttt{TAIL} ← \texttt{TAIL} − 1;
if \texttt{P} \neq \texttt{TAIL}, \texttt{set y ← MEM[TAIL], MEM[TAIL] ← c, MEM[P] ← y,}
\texttt{MEM[IHEAD} + c\texttt{] ← TAIL, MEM[IHEAD} + y\texttt{] ← P. (23)}
\end{verbatim}

Later on, after backtracking to a state where we once again want c to be considered active, we simply set \texttt{TAIL} ← \texttt{TAIL} + 1, because c will already be in place!

\textbf{Lists for the comma-free problem.} The task of finding all four-letter comma-free codes is not difficult when \(m = 3\) and only 18 cycle classes are involved. But it already becomes challenging when \(m = 4\), because we must then deal with \((4^4 - 4^2)/4 = 60\) classes. Therefore we'll want to give it some careful thought as we try to set it up for backtracking.

The example scenarios for \(m = 3\) considered above suggest that we'll repeatedly want to know the answers to questions such as, “How many words of the form 02** are still available for selection as codewords?” Redundant data structures oriented to queries of that kind appear to be needed. Fortunately, we shall see that there's a nice way to provide them, using sequential lists as in (19)–(23).

In Algorithm C below, each of the \(m^4\) four-letter words is given one of three possible states during the search for comma-free codes. A word is \texttt{green} if it’s part of the current set of tentative codewords. It is \texttt{red} if it’s not currently a candidate for such status, either because it is incompatible with the existing words or because the algorithm has already examined all scenarios in which it is green in their presence. Every other word is \texttt{blue}, and sort of in limbo; the algorithm might or might not decide to make it green. All words are initially blue—except for the \(m^2\) periodic words, which are permanently red.

We'll use the Greek letter \(\alpha\) to stand for the integer value of a four-letter word in radix \(m\). For example, if \(m = 3\) and if \(x\) is the word 0102, then \(\alpha = (0102)_3 = 11\). The current state of word \(x\) is kept in \texttt{MEM[alpha]}, using one of the arbitrary internal codes 2 (\texttt{GREEN}), 0 (\texttt{RED}), or 1 (\texttt{BLUE}).

The most important feature of the algorithm is that every blue word \(x = x_1x_2x_3x_4\) is potentially in one of seven different lists, called \texttt{P1(x)}, \texttt{P2(x)}, \texttt{P3(x)}, \texttt{S1(x)}, \texttt{S2(x)}, \texttt{S3(x)}, and \texttt{CL(x)}, where

- \texttt{P1(x)}, \texttt{P2(x)}, \texttt{P3(x)} are the blue words matching \texttt{x1**}, \texttt{x1x2**}, \texttt{x1x2x3*};
- \texttt{S1(x)}, \texttt{S2(x)}, \texttt{S3(x)} are the blue words matching \texttt{**x4}, \texttt{**x3x4}, \texttt{**x2x3x4};
- \texttt{CL(x)} hosts the blue words in \(\{x_1x_2x_3x_4, x_2x_3x_4x_1, x_3x_4x_1x_2, x_4x_1x_2x_3\}\).

These seven lists begin respectively in \texttt{MEM} locations \texttt{P1OFF} + \texttt{p1(\alpha)}, \texttt{P2OFF} + \texttt{p2(\alpha)}, \texttt{P3OFF} + \texttt{p3(\alpha)}, \texttt{S1OFF} + \texttt{s1(\alpha)}, \texttt{S2OFF} + \texttt{s2(\alpha)}, \texttt{S3OFF} + \texttt{s3(\alpha)}, and \texttt{CLOFF} + \texttt{cl(\alpha)}; here \texttt{P1OFF}, \texttt{P2OFF}, \texttt{P3OFF}, \texttt{S1OFF}, \texttt{S2OFF}, \texttt{S3OFF}, \texttt{CLOFF} are respectively \(2m^4\), \(5m^4\), \(8m^4\), \(11m^4\), \(14m^4\), \(17m^4\), \(20m^4\).

We define \(p1((x_1x_2x_3x_4)_m) = (x_1)_m\), \(p2((x_1x_2x_3x_4)_m) = (x_1x_2)_m\), \(p3((x_1x_2x_3x_4)_m) = (x_1x_2x_3)_m\), \(s1((x_1x_2x_3x_4)_m) = (x_4)_m\), \(s2((x_1x_2x_3x_4)_m) = (x_3x_4)_m\), \(s3((x_1x_2x_3x_4)_m) = (x_2x_3)_m\); and finally...
Table 1

| lists used by algorithm C \( (m = 2) \), entering level 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | a | b | c | d | e | f |
| 10 | 0001 | 20 | 21 | 22 | 0101 | 23 | 24 | 1101 | 2d | 29 | 1010 | 1110 | 1011 |
| 20 | 0001 | 26 | 21 | 25 | 0111 | 23 | 24 | 1101 | 2d | 29 | 1010 | 1110 | 1011 |
| 30 | 0001 | 50 | 51 | 52 | 0011 | 53 | 54 | 0001 | 5a | 59 | 0010 | 0010 | 0010 |
| 40 | 0001 | 90 | 91 | 92 | 93 | 0011 | 94 | 95 | 0001 | 96 | 0010 | 0010 | 0010 |
| 50 | 0001 | 80 | 81 | 82 | 83 | 0011 | 84 | 85 | 0001 | 86 | 0010 | 0010 | 0010 |
| 60 | 0001 | 70 | 71 | 72 | 73 | 0011 | 74 | 75 | 0001 | 76 | 0010 | 0010 | 0010 |
| 70 | 0001 | 60 | 61 | 62 | 63 | 0011 | 64 | 65 | 0001 | 66 | 0010 | 0010 | 0010 |
| 80 | 0001 | 50 | 51 | 52 | 53 | 0011 | 54 | 55 | 0001 | 56 | 0010 | 0010 | 0010 |
| 90 | 0001 | 40 | 41 | 42 | 43 | 0011 | 44 | 45 | 0001 | 46 | 0010 | 0010 | 0010 |

This table shows MEM locations 0000 through 110f, using hexadecimal notation. (For example, MEM[40d] = 5e; see exercise 36.) Blank entries are unused by the algorithm.

\( d((x_1 x_2 x_3 x_4)_m) \) is an internal number between 0 and \((m^4 - m^2)/4 - 1\) assigned to each class. The seven MEM locations where \( x \) appears in these seven lists are respectively kept in inverse lists that begin in MEM locations P10FF = \( m^4 + \alpha \), P20FF = \( m^4 + \alpha \) etc. The TAIL pointers, which indicate the current list sizes as in (19)-(23), are respectively kept in MEM locations P11FF + \( m^4 + \alpha \), P20FF + \( m^4 + \alpha \) etc. (Whew; got that?)

This vast apparatus, which occupies 22m⁴ cells of MEM, is illustrated in Table 1, at the beginning of the computation for the case \( m = 2 \). Fortunately it's not really as complicated as it may seem at first. Nor is it especially vast: After all, 22m⁴ is only 13,750 when \( m = 5 \).

(A close inspection of Table 1 reveals incidentally that the words 0100 and 1000 have been colored red, not blue. That's because we can assume without loss of generality that class [0001] is represented either by 0001 or by 0010. The other two cases are covered by left-right reflection of all codewords.)

Algorithm C finds these lists invaluable when it is deciding where next to branch. But it has no further use for a list in which one of the items has become green. Therefore it declares such lists "closed"; and it saves most of the work of list maintenance by updating only the lists that remain open. A closed list is represented internally by setting its TAIL pointer to HEAD - 1.

For example, Table 2 shows how the lists in MEM will have changed just after \( x = 0010 \) has been chosen to be a tentative codeword. The elements \{0001, 0010, 0011, 0110, 0111\} of P1(\( x \)) are effectively hidden, because the tail...
Table 2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>RED</td>
<td>BLUE</td>
<td>RED</td>
<td>RED</td>
<td>BLUE</td>
<td>RED</td>
<td>RED</td>
<td>RED</td>
<td>RED</td>
<td>BLUE</td>
<td>RED</td>
<td>RED</td>
<td>BLUE</td>
<td>RED</td>
<td>RED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

The word 0010 has become green, thus closing its seven lists and making 0011 red. The logic of Algorithm C has also made 1001 red. Hence 0001 and 1001 have been deleted from the open lists in which they appeared (see exercise 37).

| pointer MEM[30] = 1f = 20-1 marks that list as closed. (Those list elements actually do still appear in MEM locations 200 through 204, just as they did in Table 1. But there's no need to look at that list while any word of the form 0*** is green.) |

A general mechanism for doing and undoing. We're almost ready to finalize the details of Algorithm C and get on with the search for commasfree codes, but a big problem still remains: The state of computation at every level of the search involves all of the marvelous lists that we've just specified, and those lists aren't tiny. They occupy more than 5000 cells of MEM when m = 4, and they can change substantially from level to level.

We could make a new copy of the entire state, whenever we advance to a new node of the search tree. But that's a bad idea, because we don't want to perform thousands of memory accesses per node. A much better strategy would be to stick with a single instance of MEM, and to update and downstate the lists as the search progresses, if we could only think of a simple way to do that.

And we're in luck: There is such a way, first formulated by R. W. Floyd in his classic paper “Nondeterministic algorithms” [JACM 14 (1967), 636–644]. Floyd's original idea, which required a special compiler to generate forward and backward versions of every program step, can in fact be greatly simplified when all of the changes in state are confined to a single MEM array. All we need to do is to replace every assignment operation of the form \( \text{MEM}[a] \leftarrow v \) by the
slightly more cumbersome operation

\[ \text{store}(a, v) : \text{Set } \text{UND}0[a] \leftarrow (a, \text{MEM}[a]), \text{MEM}[a] \leftarrow v, \text{ and } u \leftarrow u + 1. \quad (24) \]

Here \text{UND}0 is a sequential stack that holds \((a, v)\) pairs; in our application we could say \(\text{UND}0[a] \leftarrow (a \leq 16, v)\), because the cell addresses and values never exceed 16 bits. Of course we'll also need to check that the stack pointer \(u\) doesn't get too large, if the number of assignments has no a priori limit.

Later on, when we want to undo all changes to \text{MEM} since the time when \(u\) had reached a particular value \(u_0\), we simply do this:

\[ \text{unstore}(u_0) : \text{While } u > u_0, \text{ set } u \leftarrow u - 1, \]
\[ (a, v) \leftarrow \text{UND}0[a], \text{ and } \text{MEM}[a] \leftarrow v. \quad (25) \]

In our application this unstacking operation \(\langle a, v \rangle \leftarrow \text{UND}0[a]\) could become \(a \leftarrow \text{UND}0[a] \gg 16, v \leftarrow \text{UND}0[a] \& \text{ffff}\).

A useful refinement of this technique is often advantageous, based on the idea of "stamping" that is part of the folklore of programming. It puts only one item on the \text{UND}0 stack when the same memory address is updated more than once in the same round.

\[ \text{store}(a, v) : \text{If } \text{STAMP}[a] \neq \sigma, \text{ set } \text{STAMP}[a] \leftarrow \sigma, \]
\[ \text{UND}0[a] \leftarrow (a, \text{MEM}[a]), \text{ and } u \leftarrow u + 1. \]
\[ \text{Then set } \text{MEM}[a] \leftarrow v. \quad (26) \]

Here \text{STAMP} is an array with one entry for each address in \text{MEM}. It's initially all zero, and \(\sigma\) is initially 1. Whenever we come to a fallback point, where the current stack pointer will be remembered as the value \(u_0\) for some future undoing, we "bump" the current stamp by setting \(\sigma \leftarrow \sigma + 1\). Then (26) will continue to do the right thing. (In programs that run for a long time, we must be careful when integer overflow causes \(\sigma\) to be bumped to zero; see exercise 38).

Notice that the combination of (24) and (25) will perform five memory accesses for each assignment and its undoing. The combination of (26) and (25) will cost seven items for the first assignment to \text{MEM}[a], but only two items for every subsequent assignment to the same address. So (26) wins if multiple assignments exceed one-time-only assignments.

**Backtracking through commafree codes.** OK, we're now equipped with enough basic knowhow to write a pretty good backtrack program for the problem of generating all commafree four-letter codes.

Algorithm C below incorporates one more key idea, which is a lookahead mechanism that is specific to commafree backtracking; we'll call it the "poison list." Every item on the poison list is a pair, consisting of a suffix and a prefix that the commafree rule forbids from occurring together. Every green word \(x_1x_2x_3x_4\) — that is, every word that will be a final codeword in the current branch of our backtrack search — contributes three items to the poison list, namely

\[ (**x_1x_2x_3, x_4**), (**x_1x_2, x_3x_4**), \text{ and (**x_1, x_2x_3x_4**).} \quad (27) \]

October 4, 2015
If there's a green word on both sides of a poison list entry, we're dead: The comma-free condition fails and we must backtrack. If there's a green word on one side but not the other, we can kill off all blue words on the other side by making them red. And if either side of a poison list entry corresponds to an empty list, we can remove this entry from the poison list because it will never affect the outcome. (Blue words become red or green, but red words stay red.)

For example, consider the transition from Table 1 to Table 2. When word 0010 becomes green, the poison list receives its first three items:

\[ (**001,0**), (**000,10**), (**000,010**). \]

The first of these kills off the 001 list, because 0** contains the green word 0010. That makes 1001 red. The last of these, similarly, kills off the 010 list; but that list is empty when \( m = 2 \). The poison list now reduces to a single item, (**010, 10**), which remains poisonous because list **00** contains the blue word 1100 and 10** contains the blue word 1011.

We'll maintain the poison list at the end of MEM, following the CL lists. It obviously will contain at most \( 3(m^4 - m^2)/4 \) entries, and in fact it usually turns out to be quite small. No inverse list is required; so we shall adopt the simple method of (17) and (18), but with two cells per entry so that TAIL will change by \( \pm 2 \) instead of by \( \pm 1 \). The value of TAIL will be stored in MEM at key times so that temporary changes to it can be undone.

The case \( m = 4 \), in which each codeword consists of four quaternary digits \( \{0, 1, 2, 3\} \), is particularly interesting, because an early backtrack program by Lee Laxdal found that no such comma-free code can make use of all 60 of the cycle classes \{0001\}, \{0002\}, ... , \{2333\}. [See B. H. Jiggs, Canadian Journal of Math. 15 (1963), 178–187.] Laxdal's program also reportedly showed that at least three of those classes must be omitted; and it found several valid 57-word sets. Further details were never published, because the proof that 60 codewords are impossible depended on what Jiggs called a "quite time-consuming" computation.

Because 60 is impossible, our algorithm cannot simply assume that a move such as 1001 is forced when the other words 0011, 0110, 1100 of its class have been ruled out. We must also consider the possibility that class \{0011\} is entirely absent from the code. Such considerations add an interesting further twist to the problem, and Algorithm C describes one way to cope with it.

**Algorithm C (Four-letter comma-free codes).** Given an alphabet size \( m \leq 7 \) and a goal \( g \) in the range \( L - m(m - 1) \leq g \leq L \), where \( L = (m^4 - m^2)/4 \), this algorithm finds all sets of \( g \) four-letter words that are comma-free and include either 0001 or 0010. It uses an array MEM of \( M = \lfloor 23.5m^4 \rfloor \) 16-bit numbers, as well as several more auxiliary arrays: ALF of size \( 16^3 m \); STAMP of size \( M \); X, C, S, U, of size \( L + 1 \); FREE and IFREE of size \( L \); and a sufficiently large array called UNDO whose maximum size is difficult to guess.

**C1. (Initialize.)** Set \( \text{ALF}(abcd)_m \leftarrow (abcd)_m \) for \( 0 \leq a, b, c, d < m \). Set \( \text{STAMP}(k) \leftarrow 0 \) for \( 0 \leq k < M \) and \( \sigma \leftarrow 0 \). Put the initial prefix, suffix, and class lists into MEM, as in Table 1. Also create an empty poison list by...
setting MEM[PP] ← POISON, where POISON = 22m^4 and PP = POISON − 1.
Set FREE[k] ← IFREE[k] ← k for 0 ≤ k < L. Then set l ← 1, x ← *0001,
c ← 0, s ← L − g, f ← L, u ← 0, and go to step C3. (Variable l is the
level, x is a trial word, c is its class, s is the “slack,” f is the number of free
classes, and u is the size of the UNDO stack.) Go to C3.

C2. [Enter level l.] If l > L, visit the solution x_1 . . . x_L and go to C6. Otherwise
choose a candidate word x and class c as described in exercise 39.

C3. [Try the candidate.] Set V[I] ← u and σ ← σ + 1. If x < 0, go to C6 if s = 0
or l = 1, otherwise set s ← s − 1. If x ≥ 0, update the data structures
to make x green, as described in exercise 40, escaping to C5 if trouble arises.

C4. [Make the move.] Set X[I] ← x, C[I] ← c, S[U] ← s, p ← IFREE[c], f ←
f − 1. If p ̸= f, set y ← FREEL[f], FREEL[y] ← y, IFREE[y] ← p, FREEL[f] ←
c, IFREE[c] ← f. (This is (23).) Then set l ← l + 1 and go to C2.

C5. [Try again.] While u > V[I], set u ← u − 1 and MEM[UNDO[u] ≥ 16] ←
UNDO[u] & *ffff. (Those operations restore the previous state, as in (25).)
Then σ ← σ + 1 and reden x (see exercise 40). Go to C2.

C6. [Backtrack.] Set l ← l − 1, and terminate if l = 0. Otherwise set x ← X[I],
c ← C[I], f ← f − 1. If x < 0, repeat this step (class c was omitted from
the code). Otherwise set s ← S[I] and go back to C5.

Exercises 39 and 40 provide the instructive details that flesh out this skeleton.

Algorithm C needs just 13, 177, and 2380 megamems to prove that no solutions exist for m = 4 when g is 60, 59, and 58. It needs about 22800 megamems
to find the 1152 solutions for g = 57; see exercise 44. There are roughly (14,
240, 3700, 38000) thousand nodes in the respective search trees, with most of
the activity taking place on levels 30 ± 10. The height of the UNDO stack never
exceeds 2804, and the poison list never contains more than 12 entries at a time.

* * *

Watch this space for more good stuff . . .

* * *
Historical notes. The origins of backtrack programming are obscure. Equivalent ideas must have occurred to many people, yet there was hardly any reason to write them down until computers existed. We can be reasonably sure that James Bernoulli used such principles in the 17th century, when he successfully solved the "Tot tibi sunt dotes" problem that had eluded so many others (see Section 7.2.1.7), because traces of the method exist in his exhaustive list of solutions.

The eight queens problem was first proposed by Max Bezzel [Schachzeitung 3 (1848), 363; 4 (1849), 40] and by Franz Nauck [Illustrierte Zeitung 14, 361 (1 June 1850), 352; 15, 377 (21 September 1850), 182], perhaps independently. C. F. Gauss saw the latter publication, and wrote several letters about it to his friend H. C. Schumacher. Gauss's letter of 27 September 1850 is especially interesting, because it explained how to find all the solutions by backtracking—which he called "Tatominen", from a French term meaning "to feel one's way." He also listed the lexicographically first solutions of each equivalence class under reflection and rotation: 15863724, 16837245, 24683175, 25713864, 25741863, 26174835, 26831475, 27368514, 27581463, 35281746, 35841726, and 36258174.

Computers arrived a hundred years later, and people began to use them for combinatorial problems. The time was therefore ripe for backtrack to be described as a general technique, and Robert J. Walker rose to the occasion [Proc. Symposia in Applied Math. 10 (1960), 91–94]. His brief note introduced Algorithm W in machine-oriented form, and mentioned that the procedure could readily be extended to find variable-length patterns $x_1 \ldots x_n$ where $n$ is not fixed.

The next milestone was a paper by Solomon W. Golomb and Leonard D. Baumert [JACM 12 (1965), 516–524], . . .

Please bear with me as I continue to assemble this info.
EXERCISES

1. [22] Explain how the tasks of generating (i) n-tuples, (ii) permutations, (iii) combinations, (iv) integer partitions, (v) set partitions, and (vi) nested parentheses can all be regarded as special cases of backtracking programming, by presenting suitable domains \( D_k \) and cutoff properties \( P_k(x_1, \ldots, x_k) \) that satisfy (1) and (2).

2. [16] True or false: We can choose \( D_1 \) so that \( P_1(x_1) \) is always true.

3. [16] Using a chessboard and eight markers, one can obey Algorithm B and essentially traverse the tree of Fig. 68 by hand in about three hours. Invent a trick to save half of the work.

4. [20] Reformulate Algorithm B as a recursive procedure called \( \text{try}(l) \), having global variables \( n \) and \( x_1 \ldots x_n \), to be invoked by saying \( \text{try}(1) \). Can you imagine why the author of this book decided not to present the algorithm in such a recursive form?

5. [20] Given \( r \), with \( 1 \leq r \leq n \), in how many ways can \( 7 \) nonattacking queens be placed on an \( 8 \times 8 \) chessboard, if no queen is placed in row \( r \)?

6. [20] (T. B. Sprague, 1890.) Are there \( n \)-queen solutions for any \( n > 5 \) that are "framed" with \( x_1 = 2, x_2 = n, x_{n-1} = 1, \) and \( x_n = n - 1 \)?

7. [20] Are there two \( n \)-queen placements with the same \( x_1 x_2 x_3 x_4 x_5 \)?

8. [21] Can a \( 4m \)-queen placement have \( 3m \) queens on "white" squares?

9. [22] Adapt Algorithm W to the \( n \)-queen problem, using bitwise operations on \( n \)-bit numbers as suggested in the text.

10. [M25] (W. Ahrens, 1930.) Both solutions of the \( n \)-queen problem when \( n = 4 \) have chiral symmetry: Rotation by 90° leaves them unchanged, but reflection doesn't.

  a) Can the \( n \)-queen problem have a solution with reflection symmetry?

  b) Show that chiral symmetry is impossible when \( n \) mod 4 \( \in \{2, 3\} \).

  c) Sometimes the solution to an \( n \)-queen problem contains four queens that form the corners of a tilted square, as shown here. Prove that we can always get another solution by tilting the square the other way (but leaving the other \( n - 4 \) queens in place).

  d) Let \( C_n \) be the number of chirally symmetric solutions, and suppose \( c_n \) of them have \( x_k > k \) for \( 1 \leq k \leq n/2 \). Prove that \( C_n = 2^{n/4} c_n \).

11. [M28] (Wraparound queen.) Replace \( \lbrack x_i - x_j \rbrack \) in (3) by the stronger condition \( \lbrack x_i - x_j \rbrack \) mod \( n \neq n - k - j \) and \( (x_i - x_j) \) mod \( n \neq k - j \). (The \( n \times n \) grid becomes a torus.) Prove that the resulting problem is solvable if and only if \( n \) is not divisible by 2 or 3.

12. [M30] For which \( n \geq 0 \) does the \( n \)-queen problem have at least one solution?

13. [M25] If exercise 11 has \( T(n) \) toroidal solutions, show that \( Q(mn) \geq Q(m)T(n) \).

14. [HM47] Does \( \ln Q(n) / (n \ln n) \) approach a positive constant as \( n \to \infty \)?

15. [21] Let \( H(n) \) be the number of ways that \( n \) queen bees can occupy \( n \times n \) honeycomb so that no two are in the same line. (For example, one of the \( H(4) = 7 \) ways is shown here.) Compute \( H(n) \) for small \( n \).

16. [15] J. H. Quick (a student) noticed that the loop in step L2 of Algorithm 1 can be changed from 'while \( x_i < 0 \)' to 'while \( x_i \neq 0 \)', because \( x_i \) cannot be positive at that point of the algorithm. So he decided to eliminate the minus signs and just set \( x_{i+1} \leftarrow k \) in step L3. Was it a good idea?
17. [17] Suppose \( n = 4 \) and Algorithm 1 has reached step 1.2 with \( l = 4 \), \( x_1x_2x_3 = 241 \). What are the current values of \( x_4x_5x_6x_7 \) and \( y_1y_2y_3 \) and the domains \( D_i \) in Langford's problem (7)?

19. [M19] What are the domains \( D_i \) in Langford's problem (7)?

20. [21] Extend Algorithm 1 so that it forces \( x_i \leftarrow k \) whenever \( k \not \in \{ x_1, \ldots, x_{i-1} \} \).

21. [M21] If \( x = x_1x_2 \ldots x_{2n} \), let \( x^D = (-x_{2n}) \ldots (-x_2)\{ -x_1 \} = -x^H \) be its dual.

a) Show that if \( n \) is odd and \( x \) solves Langford's problem (7), we have \( x_i = n \) for some \( k \leq \lfloor n/2 \rfloor \) if and only if \( x_i^D = n \) for some \( k > \lfloor n/2 \rfloor \).

b) Find a similar rule that distinguishes \( x \) from \( x^D \) when \( n \) is even.

c) Consequently the algorithm of exercise 20 can be modified so that exactly one of each dual pair of solutions \( \{ x, x^D \} \) is visited.

22. [M25] Explore "loose Langford pairs": Replace \( j + k + 1 \) in (7) by \( j + \lfloor 3k/2 \rfloor \).

23. [17] We can often obtain one word rectangle from another by changing only one letter or two. Can you think of any \( 5 \times 6 \) rectangles that almost match (10)?

24. [20] Customize Algorithm B so that it will find all \( 5 \times 6 \) word rectangles.

25. [25] Explain how to use orthogonal lists, as in Fig. 13 of Section 2.26, so that it’s easy to visit all \( 5 \)-letter words whose \( k \)th character is \( c \), given \( 1 \leq k \leq 5 \) and \( a \leq c \leq z \). Use those sublists to speed up the algorithm of exercise 24.

26. [21] Can you find nice word rectangles of sizes \( 5 \times 7 \), \( 5 \times 8 \), \( 5 \times 9 \), \( 5 \times 10 \)?

27. [22] What profile and average node costs replace (13) and (14) when we ask the algorithm of exercise 25 for \( 6 \times 5 \) word rectangles instead of \( 5 \times 6 \)?

28. [29] The method of exercises 24 and 25 does \( n \) levels of backtracking to fill the cells of an \( m \times n \) rectangle in one column at a time, using a trie to detect illegal prefixes in the rows. Devise a method that does \( mn \) levels of backtracking and fills just one cell per level, using tries for both rows and columns.

29. [15] What’s the largest commafree subset of the following words?

\[ \text{aced babe bade beef cafe code dado dead deaf face fade feed} \]

30. [22] Let \( w_1, w_2, \ldots, w_m \) be four-letter words on an \( m \)-letter alphabet. Design an algorithm that accepts or rejects each \( w_j \), according as \( w_j \) is commafree or not with respect to the accepted words of \( \{ w_1, \ldots, w_{j-1} \} \).

31. [M22] A two-letter block code on an \( m \)-letter alphabet can be represented as a digraph \( D \) on \( m \) vertices, with \( a \rightarrow b \) if and only if \( ab \) is a codeword.

a) Prove that the code is commafree \( \iff \) \( D \) has no oriented paths of length 3.

b) How many arcs can be in a digraph with no oriented paths of length \( r \)?

32. [M28] (W. L. Eastman, 1965.) The following elegant construction yields a commafree code of maximum size for any odd block length \( n \), over any alphabet. Given a sequence of \( x = x_0x_1 \ldots x_{n-1} \) of nonnegative integers, where \( x \) differs from each of its other cyclic shifts \( x_{n-1}x_0 \ldots x_{i-1} \) for \( 0 < k < n \), the procedure outputs a cyclic shift \( x^* \) with the property that the set of all such \( x^* \) codes is commafree.

We regard \( x \) as an infinite periodic sequence \( \langle x_n \rangle \) with \( x_k = x_{k+n} \) for all \( k > n \). Each cyclic shift then has the form \( x=x_{i+n} \ldots x_{i+n-1} \). The simplest nontrivial example occurs when \( n = 3 \), where \( x = x_0x_1x_2x_3 \) and we don’t have \( x_0 = x_1 = x_2 \). In this case the algorithm outputs \( x_{i}x_{i+1}x_{i+2} \) where \( x_i > x_{i+1} \leq x_{i+2} \); and the set of all such triples clearly satisfies the commafree condition.

October 4, 2015
The key idea is to think of $x$ as partitioned into $t$ substrings by boundary markers $b_j$, where $0 \le b_0 < b_1 < \cdots < b_{t-1} < n$ and $b_j = b_{j-1} + n$ for $j \ge 1$. Then substring $y_j = x_{b_j}x_{b_j+1}\cdots x_{b_{j+1}-1}$. The number $t$ of substrings is always odd. Initially $t = n$ and $b_j = j$ for all $j$; ultimately $t = 1$, and $\sigma x = y_0$ is the desired output.

Eastman’s algorithm is based on comparison of adjacent substrings $y_{j-1}$ and $y_j$. If these substrings have the same length, we use lexicographic comparison; otherwise we declare that the longer substring is bigger.

It’s convenient to describe the algorithm using terminology based on the topography of Nevada: Say that $i$ is a basin if the substrings satisfy $y_{i-1} > y_i \le y_{i+1}$. There must be at least one basin; otherwise all the $y_j$ would be equal, and $x$ would equal one of its cyclic shifts. We look at consecutive basins, $i$ and $j$; this means that $i < j$ and that $i$ and $j$ are basins, and that $i + 1$ through $j - 1$ are not basins. If there’s only one basin we have $j = i + t$. The indices between consecutive basins are called ranges.

Since $t$ is odd, there’s an odd number of consecutive basins for which $j - i$ is odd. Each round of Eastman’s algorithm retains exactly one boundary point in the range between such basins, and deletes all the others. The retained point is the smallest $k = i + 2t$ such that $y_k > y_{k+1}$. At the end of a round, we reset $t$ to the number of retained boundary points, and we begin another round if $t > 1$. 

- Play through the algorithm by hand when $n = 19$ and $x = 3141592653589793238$.
- Show that the number of rounds is at most $\lceil \log_2 n \rceil$.
- Exhibit a binary $x$ that achieves this worst-case bound when $n = 3^e$.
- Implement the algorithm with full details. (It’s surprisingly short!)

33. [HM26] What is the probability that Eastman’s algorithm finishes in one round? (Assume that $x$ is a random $m$-ary string of odd length $n > 1$, unequal to any of its other cyclic shifts. Use a generating function to express the answer.)

34. [18] Why can’t a comma-free code of length $(m^2 - m^4) / 4$ contain 0001 and 0000?

35. [15] Why do you think sequential data structures such as (16)--(23) weren’t featured in Section 2.2.2 of this series of books entitled “Sequential Allocation”?

36. [17] What’s the significance of (a) $MEM[40] = 5e$ and (b) $MEM[904] = 63$ in Table 1?

37. [18] Why is (a) $MEM[18] = 57$ and (b) $MEM[104] = 9a$ in Table 2?

38. [20] Suppose you’re using the undoing scheme (26) and the operation $\sigma \leftarrow \sigma + 1$ has just bumped the current stamp $\sigma$ to zero. What should you do?

39. [25] Spell out the low-level implementation details of the candidate selection process in step C2 of Algorithm C. Use the routine $store(a, v)$ of (26) whenever changing the contents of $MEM$, and use the following selection strategy:

   - Find a class $c$ with the least number $r$ of blue words.
   - If $r = 0$, set $x \leftarrow -1$; otherwise set $x$ to a word in class $c$.
   - If $r > 1$, use the poison list to find an $x$ that maximizes the number of blue words that could be killed on the other side of the prefix or suffix list that contains $x$.

40. [28] Continuing Exercise 39, spell out the details of step C3 when $x \ge 0$.

   a) What updates should be done to $MEM$ when a blue word $x$ becomes red?
   b) What updates should be done to $MEM$ when a blue word $x$ becomes green?
   c) Step C3 begins its job by making $x$ green as in part (b). Explain how it should finish its job by updating the poison list.

42. [M30] Is there a binary ($m = 2$) comma-free code with one codeword in each of the $(\sum_{d | n} \mu(d) 2^{n/d}) / n$ cycle classes, for every word length $n$?
44. [HM29] A commafree code on \( m \) letters is equivalent to \( 2m! \) such codes if we permute the letters and/or replace each codeword by its left-right reflection.

Determine all of the nonisomorphic commafree codes of length 4 on \( m \) letters when \( m \) is (a) 2 (b) 3 (c) 4 and there are (a) 3 (b) 18 (c) 57 codewords.

45. [M41] Find a maximum-size commafree code of length 4 on \( m = 5 \) letters.
Table 666

TWENTY QUESTIONS (SEE EXERCISE 90)

<table>
<thead>
<tr>
<th>Question</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The first question whose answer is A is:</td>
<td>(A) 1</td>
</tr>
<tr>
<td>(B) 2</td>
<td>(C) 3</td>
</tr>
<tr>
<td>(D) 4</td>
<td>(E) 5</td>
</tr>
<tr>
<td>2. The next question with the same answer as this one is:</td>
<td>(A) 4</td>
</tr>
<tr>
<td>(B) 6</td>
<td>(C) 8</td>
</tr>
<tr>
<td>(D) 10</td>
<td>(E) 12</td>
</tr>
<tr>
<td>3. The only two consecutive questions with identical answers are:</td>
<td>(A) 15</td>
</tr>
<tr>
<td>and 16</td>
<td>(B) 16</td>
</tr>
<tr>
<td>and 17</td>
<td>(C) 17</td>
</tr>
<tr>
<td>and 18</td>
<td>(D) 18</td>
</tr>
<tr>
<td>and 19</td>
<td>(E) 19</td>
</tr>
<tr>
<td>4. The answer to this question is the same as the answers to questions:</td>
<td>(A) 10</td>
</tr>
<tr>
<td>and 13</td>
<td>(B) 14</td>
</tr>
<tr>
<td>and 16</td>
<td>(C) 7</td>
</tr>
<tr>
<td>and 20</td>
<td>(D) 1</td>
</tr>
<tr>
<td>and 15</td>
<td>(E) 8</td>
</tr>
<tr>
<td>5. The answer to question 14 is:</td>
<td>(A) B</td>
</tr>
<tr>
<td>(B) E</td>
<td>(C) C</td>
</tr>
<tr>
<td>(D) A</td>
<td>(E) D</td>
</tr>
<tr>
<td>6. The answer to this question is:</td>
<td>(A) A</td>
</tr>
<tr>
<td>(B) B</td>
<td>(C) C</td>
</tr>
<tr>
<td>(D) D</td>
<td>(E) none of those</td>
</tr>
<tr>
<td>7. An answer that appears most often is:</td>
<td>(A) A</td>
</tr>
<tr>
<td>(B) B</td>
<td>(C) C</td>
</tr>
<tr>
<td>(D) D</td>
<td>(E) E</td>
</tr>
<tr>
<td>8. Ignoring answers that appear equally often, the least common answer is:</td>
<td>(A) A</td>
</tr>
<tr>
<td>(B) B</td>
<td>(C) C</td>
</tr>
<tr>
<td>(D) D</td>
<td>(E) E</td>
</tr>
<tr>
<td>9. The sum of all question numbers whose answers are correct, and the same as this one is:</td>
<td>(A) (\in [50..62])</td>
</tr>
<tr>
<td>(B) (\in [52..55])</td>
<td>(C) (\in [44..49])</td>
</tr>
<tr>
<td>(D) (\in [41..67])</td>
<td>(E) (\in [44..53])</td>
</tr>
<tr>
<td>10. The answer to question 17 is:</td>
<td>(A) D</td>
</tr>
<tr>
<td>(B) B</td>
<td>(C) A</td>
</tr>
<tr>
<td>(D) E</td>
<td>(E) wrong</td>
</tr>
<tr>
<td>11. The number of questions whose answer is D is:</td>
<td>(A) 2</td>
</tr>
<tr>
<td>(B) 3</td>
<td>(C) 4</td>
</tr>
<tr>
<td>(D) 5</td>
<td>(E) 6</td>
</tr>
<tr>
<td>12. The number of other questions with the same answer as this one is the same as the number of questions with answer:</td>
<td>(A) B</td>
</tr>
<tr>
<td>(B) C</td>
<td>(C) D</td>
</tr>
<tr>
<td>(D) E</td>
<td>(E) none of those</td>
</tr>
<tr>
<td>13. The number of questions whose answer is E is:</td>
<td>(A) 5</td>
</tr>
<tr>
<td>(B) 4</td>
<td>(C) 3</td>
</tr>
<tr>
<td>(D) 2</td>
<td>(E) 1</td>
</tr>
<tr>
<td>14. No answer appears exactly this many times:</td>
<td>(A) 2</td>
</tr>
<tr>
<td>(B) 3</td>
<td>(C) 4</td>
</tr>
<tr>
<td>(D) 5</td>
<td>(E) none of those</td>
</tr>
<tr>
<td>15. The set of odd-numbered questions with answer A is:</td>
<td>(A) 7</td>
</tr>
<tr>
<td>(B) 9</td>
<td>(C) not {11}</td>
</tr>
<tr>
<td>(D) {13}</td>
<td>(E) {15}</td>
</tr>
<tr>
<td>16. The answer to question 8 is the same as the answer to question:</td>
<td>(A) 3</td>
</tr>
<tr>
<td>(B) 2</td>
<td>(C) 13</td>
</tr>
<tr>
<td>(D) 18</td>
<td>(E) 20</td>
</tr>
<tr>
<td>17. The answer to question 10 is:</td>
<td>(A) C</td>
</tr>
<tr>
<td>(B) D</td>
<td>(C) B</td>
</tr>
<tr>
<td>(D) A</td>
<td>(E) correct</td>
</tr>
<tr>
<td>18. The number of prime-numbered questions whose answers are vowels is:</td>
<td>(A) prime</td>
</tr>
<tr>
<td>(B) square</td>
<td>(C) odd</td>
</tr>
<tr>
<td>(D) even</td>
<td>(E) zero</td>
</tr>
<tr>
<td>19. The last question whose answer is B is:</td>
<td>(A) 14</td>
</tr>
<tr>
<td>(B) 15</td>
<td>(C) 16</td>
</tr>
<tr>
<td>(D) 17</td>
<td>(E) 18</td>
</tr>
<tr>
<td>20. The maximum score that can be achieved on this test is:</td>
<td>(A) 18</td>
</tr>
<tr>
<td>(B) 19</td>
<td>(C) 20</td>
</tr>
<tr>
<td>(D) indeterminate</td>
<td>(E) achievable only by getting this question wrong</td>
</tr>
</tbody>
</table>

90. [M29] (Donald R. Woods, 2000.) Find all ways to maximize the number of correct answers to the questionnaire in Table 666. Each question must be answered with a letter from A to E. *Hint:* Begin by clarifying the exact meaning of this exercise. What answers are best for the following two-question, two-letter "warmup problem"?

1. (A) Answer 2 is B.  (B) Answer 1 is A.
2. (A) Answer 1 is correct.  (B) Either answer 2 is wrong or answer 1 is A, but not both.

91. [HM28] Show that exercise 90 has a surprising, somewhat paradoxical answer if two changes are made to Table 666: 9(E) becomes \{'\in [30..43]\'; 15(C) becomes \{'\{11\}'.
999. [M00] this is a temporary exercise (for dummies)
SECTION 7.2.2

1. Although many answers are possible, the following may be the nicest: (i) \( D_k \) is arbitrary (but hopefully finite), and \( P_1 \) is always true. (ii) \( D_k = \{1, 2, \ldots, n\} \) and \( P_k = \{x_j \neq x_i \text{ for } 1 \leq j < k \leq l\} \). (iii) For combinations of \( n \) things from \( N \), \( D_k = \{1, \ldots, N+1-k\} \) and \( P_k = \{x_k < \cdots < x_1\} \). (iv) \( D_k = \{0, 1, \ldots, [n/k]\} \); \( P_k = \{x_k \geq \cdots \geq x_1 \text{ and } n = \frac{n-1}{k} x_1 + \cdots + x_l \leq n\} \). (v) For restricted growth strings, \( D_k = \{0, \ldots, k-1\} \) and \( P_k = \{x_{j+1} \leq 1 + \max(x_1, \ldots, x_j) \text{ for } 1 \leq j < l\} \). (vi) For the indices of left parentheses (see 7.2.1.6-(8)), \( D_k = \{1, 2, 2k-1\} \) and \( P_k = \{x_l < \cdots < x_1\} \).

2. True. (If not, set \( D_1 = D_2 \cap \{x \mid P_1(x)\} \).

3. We can restrict \( D_1 \) to \{1, 2, 3, 4\}, because the reflection \((9-x_1) \ldots (9-x_k)\) of every solution \( x_1 \ldots x_n \) is also a solution. (H. C. Schumacher made this observation in a letter to Gauss, 24 September 1850.) Notice that Fig. 68 is left-right symmetric.

4. \( \text{try}(l) = \text{if } l > n, \text{visit } x_1, \ldots, x_n. \text{ Otherwise, for } x_l = \min D_l, \min D_l + 1, \ldots, \max D_l, \text{ if } P_l(x_1, \ldots, x_l) \text{ call } \text{try}(l + 1). \)

This formulation is elegant, and fine for simple problems. But it doesn’t give any clue about why the method is called “backtrack.” Nor does it yield efficient code for important problems where the inner loop is performed billions of times. We will see that the key to efficient backtracking is to provide good ways to update and downvote the data structures that speed up the testing of property \( P_l \). The overhead of recursion can get in the way, and the actual iterative structure of Algorithm B isn’t difficult to grasp.

5. Excluding cases with \( j = r \) or \( k = r \) from (3) yields respectively (312, 396, 430, 458, 430, 396, 312) solutions. (With column \( r \) also omitted there are just \( 40, 46, 42, 80, 42, 42, 46, 40, 40, 40 \).

6. Yes, probably for all \( n \geq 16 \). One such is \( x_1 x_2 \ldots x_{17} = 217 12 10 7 14 3 5 9 13 15 4 11 6 1 16 \). [See Proc. Edinburgh Math. Soc. 8 (1895), 43 and Fig. 52.]

7. Yes: (4273615, 42736851); also therefore (57263148, 57263184).

8. Yes, at least when \( m = 4 \); e.g., \( x_1 \ldots x_{10} = 5 8 13 16 3 7 15 11 6 2 10 14 1 4 9 12 \). There are no solutions when \( m = 5 \), but \( 7 10 13 20 17 24 3 6 23 11 16 21 4 9 14 2 19 22 1 8 5 12 15 18 \) works for \( m = 6 \). (Are there solutions for all even \( m \geq 4 ? \)) C. F. de Jaenisch, Traité des applications de l’analyse mathématique au jeu des échecs 2 (1862), 132–153, noted that all 8-queen solutions have four of each color. He proved that the number of white queens must be even, because \( \sum_{i=1}^{4m} (x_i + k) \) is even.

9. Let bit vectors \( a_i, b_i, c_i \) represent the “useful” elements of the sets in (6), with \( a_i = \sum \{2^{j-1} \mid x \in A_i\} \), \( b_i = \sum \{2^{j-1} \mid x \in B_i \cap [1 \ldots n]\} \), \( c_i = \sum \{2^{j-1} \mid x \in C_i \cap [1 \ldots n]\} \). Then step W2 sets \( s_i \leftarrow \mu \& \bar{a}_i \& \bar{b}_i \& \bar{c}_i \), where \( \mu \) is the mask \( 2^n - 1 \).

In step W3 we can set \( t_s \leftarrow s_i \& \neg s_i, \text{ at } t \leftarrow a_i - 1 \), \( b_i \leftarrow (b_{i-1} + \ell) \gg 1 \), \( c_i \leftarrow ((c_{i-1} + \ell) \ll 1) \& \mu; \text{ and it’s also convenient to set } s_i \leftarrow s_i - t \text{ at this time, instead of deferring this change to step W4. There’s no need to store } x_i \text{ in memory, or even to compute } x_i \text{ in step W3 as an integer in } [1 \ldots n], \text{ because } x_i \text{ can be deduced from } a_i = a_{i-1} \text{ when a solution is found.} \)

10. (a) Only when \( n = 1 \), because reflected queens can capture each other.

(b) Queens not in the center must appear in groups of four.

(c) The four queens occupy the same rows, columns, and diagonals in both cases.

(d) In each solution counted by \( c_n \) we can independently tilt (or not) each of the \( \lceil n/4 \rceil \) groups of four. [Mathematische Unterhaltungen und Spiele 1, second edition (Leipzig: Teubner, 1910), 249–258.]

October 4, 2015
11. Suppose the $x_i$ are distinct. Then $\sum_{i=1}^{n} (x_i + k) = 2 \binom{n+1}{2} \equiv 0 \pmod{n}$. If the numbers $(x_i + k) \mod n$ are also distinct, we have also $\sum_{i=1}^{n} k \equiv \binom{n+1}{2}$. But that is impossible when $n$ is even.

Now suppose further that the numbers $(x_i - k) \mod n$ are distinct. Then we have $\sum_{i=1}^{n} (x_i - k) \equiv \sum_{i'=1}^{n'} k' = n(n+1)(2n+1)/6$. And we also have $\sum_{i=1}^{n} (x_i + k)^2 - \sum_{i=1}^{n} (x_i - k)^2 = 4n^2(2n+1)/6 \equiv 2n/3$, which is impossible when $n$ is a multiple of 3. [See W. Ahrens, Mathematische Unterhaltungen und Spiele 2, second edition (1918), 364–366, where G. Pólya cites a more general result of A. Hurwitz that applies to wraparound diagonals of other slopes.]

Conversely, if $n$ isn’t divisible by 2 or 3, we can let $x_n = n$ and $x_i = (2k)$ mod $n$ for $1 \leq k < n$. (The rule $x_k = (3k)$ mod $n$ also works. See Édouard Lucas, Récréations Mathématiques 1 (1882), 84–86.)

Another nice solution was found by J. Franel [L’Intermédiaire des Mathématiciens 1 (1894), 140–141] when $n \mod 6 \in \{2, 4\}$. Let $x_i = (n/2 + 2k - 3[2k \leq n]) \mod n + 1$, for $1 \leq k \leq n$. With this setup we find that $x_i - x_j = \pm (k - j)$ and $1 \leq j < k \leq n$ implies $(1 or 3)(k - j) + (0 or 3) \equiv 0 \pmod{n}$; hence $k - j = n - 1 \pmod{3}$. But the values of $x_1, x_2, x_3, x_{n-2}, x_{n-1}, x_n$ give no attacking queens except when $n = 2$. Franel’s solution has empty diagonals, so it provides solutions also for $n \mod 6 \in \{3, 5\}$. We conclude that only $n = 2$ and $n = 3$ are impossible.

[For a more complicated construction for all $n > 3$ had been given earlier by E. Paul, in Deutsche Schachzeitung 29 (1874), 129–134, 257–267. Pauls also explained how to find all solutions, in principle, by building the tree level by level (not backtracking).]

13. For $1 \leq j \leq n$, let $x_1^{(j)} \ldots x_n^{(j)}$ be a solution for $m$ queens, and let $y_1 \ldots y_n$ be a solution for $n$ toroidal queens. Then $X_{1+1,n+j} = (x_i^{(j)} - 1) n + y_j$ (for $1 \leq i \leq m$ and $1 \leq j \leq n$) is a solution for $mn$ queens. [I. Rivin, I. Vardi, and P. Zimmermann, AMM 101 (1994), 629–639, Theorem 2.]

14. [Rivin, Vardi, and Zimmermann, in the paper just cited, observe that in fact the sequence $(\ln Q(n)) / (\ln n)$ appears to be in $	ext{crossing.}$]

15. Let the queen in row $k$ be in cell $k$. Then we have a “relaxation” of the $n$ queens problem, with $|x_i - x_j|$ becoming $|x_i - x_j|$ in (3); so we can ignore the $b$ vector in Algorithm B* or in exercise 9. We get

$$n = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14$$

$$H(n) = 1 1 1 1 3 7 23 83 405 2113 12657 82927 596853 4698655 4001743 36785483$$


16. It fails spectacularly in step L5. The minus signs, which mark decisions that were previously forced, are crucial tags for backtracking.

17. $x_4 \ldots x_8 = 2 \hat{0} \hat{4} \hat{0} \hat{6} \hat{0} \hat{8} \hat{0} \hat{6} \hat{0} \hat{8} \hat{0} \hat{6} \hat{0} \hat{8} \hat{0} \hat{6} \hat{0} \hat{8}$, and $y_1 y_2 y_3 = 130$. (If $x_i \leq 0$ the algorithm will never look at $y_i$; hence the current state of $y_1 \ldots y_6$ is irrelevant. But $y_4 y_5$ happens to be 20, because of past history; $y_6$, $y_7$, and $y_8$ haven’t yet been touched.)

19. We could say $D_k = \{-n, \ldots, -2, 1, 2, \ldots, n\}$, or $\{k \mid k \neq 0 \text{ and } 2 - l \leq k \leq 2n - l - 1\}$, or anything in between. (But this observation isn’t very useful.)
20. First we add a Boolean array $a_1, \ldots, a_n$, where $a_i$ means \textit{“k has appeared,”} as in Algorithm B*. It’s 0 \ldots 0 in step L1; we set $a_k \leftarrow 1$ in step L3, $a_k \leftarrow 0$ in step L5.

The loop in step L2 becomes \textit{“while $x_i < 0$, go to L5 if $l \geq n - 1$ and $a_{2b_i - 1} = 0$, otherwise set $l \leftarrow l + 1”.\text{”} After finding $l + k + 1 \leq 2n$ in L3, and before testing $x_{i+k+1}$ for 0, insert this: \textit{“if $l \geq n - 1$ and $a_{2b_i - 1} = 0$, while $l + k + 1 \neq 2n$ set $j \leftarrow k$, $k \leftarrow p_i.”}\text{”}

21. (a) In any solution $x_k = n$ $\iff x_{k+n+1} = -n$ $\iff x_{i-k} = n$.

(b) $x_k = n - 1$ for some $k \leq n/2$ if and only if $x_0 = n - 1$ for some $k > n/2$.

(c) Let $n = n - [n \text{ is even}]$. Change \textit{“if $l \geq n - 1$ and $a_{2b_i - 1} = 0$’ into the modified step L2 to \textit{“if $l = [n/2]$ and $a_{2b_i - 1} = 0$”}. Insert the following before the other insertion into step L3: \textit{“if $l = [n/2]$ and $a_{2b_i - 1} = 0$, while $k \neq n’$ set $j \leftarrow k$ and $k \leftarrow n’”}\text{. And in step L5—this subtext is needed when $n$ is even—go to L5 instead of L4 if $l = [n/2]$ and $k = n’$.

22. The solutions $\overline{1}$ and $\overline{2112}$ for $n = 1$ and $n = 2$ are self-dual; the solutions for $n = 4$ and $n = 5$ are 43121312, 21435453, 45123153, and their duals. The total number of solutions for $n = 1, 2, \ldots$ is 1, 1, 0, 2, 4, 20, 0, 156, 516, 2008, 0, 52336, 297800, 1767792, 0, 756756756, \ldots; there are none when $n \text{ mod } 4 = 3$, by a parity argument.

Algorithm I needs only obvious changes. To compute solutions by a streamlined method like Algorithm 21, use $n’ = n - (0, 1, 2, 0)$ and substitute \textit{“if $l = [n/2]$ and $a_{2b_i - 1} = 0$” for \textit{“if $l = [n/2]$”}, when $n \text{ mod } 4 = (0, 1, 2, 3)$; also replace \textit{“if $l \geq n - 1$ and $a_{2b_i - 1} = 0$’ by \textit{“if $l \geq [n/2]$ and $a_{1(4(b_i + 2 - n)/3)} = 0$”’}. The case $n = 15$ is proved impossible with 397 million nodes and 9.93 gigamans.

23. slums $\rightarrow$ stuff, sluff, slump, slurs, slurp, or slute; (slums.total) $\rightarrow$ (slums.tonal).

24. Build the list of 6-letter words and the trie of 6-letter words in step B1; also set $a_1 = a_2 = a_3 = a_4 = 0$ $\iff$ 00000. Use $m = 1$ in step B2 and max $D_4 = 5757$ in step B4.

Testing $P_3$ in step B3, if word $x_3 = c_1 c_2 c_3 c_4 c_5$, consists of forming $a_{11} \ldots a_{15}$, where $a_{12} = \text{trie}[a_{13} \ldots a_{15}]$ for $1 \leq k \leq 5$; but jump to B4 if any $a_{1k}$ is zero.

25. There are $5 \times 26$ singly linked lists, accessed from pointers $h_{l_k}$, all initially zero. The zhth word $c_1 c_2 c_3 c_4 c_5$, for $1 \leq x \leq 5757$, belongs to 5 lists and has five pointers $l_1, l_2, l_3, l_4, l_5$. To insert it, set $l_{x_k} \leftarrow h_{l_{x_k} x_k}, h_{l_{x_k} x_k} \leftarrow x$, and $k_{x_k} \leftarrow k_{x_k} + 1$, for $1 \leq k \leq 5$. (Thus $k_{x_k}$ will be the length of the list accessed from $h_{l_{x_k}}$.)

We can store a \textit{“signature”} $\sum_{v=1}^{2^n} 2^{v-1} \left\{\text{trie}[a, c] \neq 0\right\}$ with each node $a$ of the trie. For example, the signature for node 350 is $2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 31$, according to (11); here $a \leftrightarrow 1, \ldots , z \leftrightarrow 26$.

The process of running through all $x$ that match a given signature $y$ with respect to $z$, as needed in steps B2 and B4, now takes the following form: (i) Set $i \leftarrow 0$. (ii) While $2^i \neq y$, set $i \leftarrow i + 1$. (iii) Set $x \leftarrow h_{x_{i+1}}$; go to (iv) if $x \neq 0$. (iv) Visit $x$. (v) Set $x \leftarrow l_{x}$; go to (iv) if $x \neq 0$. (vi) Set $i \leftarrow i + 1$; go to (ii) if $2^i \leq y$.

Let $\text{trie}[a, b]$ be the signature of node $a$. We choose $z$ and $y = \text{trie}[a_{b-1}, 0]$ in step B2 so that the number of nodes to visit, $\sum_{v=0}^{2^n} s_{v}[2^{v-1} \& y \neq 0]$, is minimum for $1 \leq z \leq 5$. For example, when $l = 3$, $x_1 = 1446$, and $x_2 = 185$ as in (20), that sum for $z = 1$ is $8_{12} + 8_{15} + 8_{19} + 8_{21} + 8_{23} = 256 + 129 + 74 + 108 + 268 + 75 + 47 = 997$; and the sums for $z = 2, 3, 4, 5$ are 4722, 1370, 5057, and 1646. Hence we choose $z = 1$ and $y = 8_{1144111}$; only 997 words, not 5757, need be tested for $x_3$.

The values $y_1$ and $z_1$ are maintained for use in backtracking. (In practice we keep $x, y, z$ in registers during most of the computation. Then we set $x_1 \leftarrow x, y_1 \leftarrow y, z_1 \leftarrow z$ before increasing $l \leftarrow l + 1$ in step B3; and we set $x \leftarrow x_1, y \leftarrow y_1, z \leftarrow z_1$ in...
step B3. We also keep $i$ in a register, while traversing the sublists as above; this value
is restored in step B5 by setting it to the $i$th letter of word $x$, decreased by $^3a^3$.)

26. Here are the author's favorite 5 $\times$ 7 and 5 $\times$ 8, and the only 5 $\times$ 9's:

\begin{itemize}
  \item smash
  \item grandest
  \item pastelist
  \item varisters
  \item partial
  \item renounce
  \item accident
  \item agentival
  \item immene
  \item episodes
  \item mortgage
  \item coelomate
  \item emerged
  \item basement
  \item proreform
  \item undeleted
  \item sadness
  \item eyesores
  \item andeyes
  \item oysterers
\end{itemize}

No 5 $\times$ 10 word rectangles exist, according to our ground rules.

27. (1, 15727, 8072679, 630467290, 99062681, 625415) and (15727, 0, 43216, 17497, 430.4, 286.0). Total time $\approx 18.3$ teramems. (In Section 7.2.2.1 we'll study a method that
is symmetrical between rows and columns.)

28. Build a separate trie for the $m$-letter words; but instead of having trie nodes of
size 26 as in (11), it's better to convert this trie into a \emph{compressed} representation that
omits the zeros. For example, the compressed representation of the node for prefix
\textquoteleft core\textquoteright{} in (12) consists of five consecutively stored pairs of entries ($^3a^3$, 3879), ($^3d^3$, 3878), ($^3t^3$, 9002), ($^3t^3$, 171), ($^3t^3$, 5013), followed by (0, 0). Similarly, each shorter prefix
with $c$ descendants is represented by $c$ consecutive pairs (character, link), followed by
(0, 0) to mark the end of the node. Steps B3 and B4 are now very convenient.

Level $i$ corresponds to row $i = 1 + (i - 1) \bmod m$ and column $j = 1 + [(i - 1) / m]$. For backtracking we store the $n$-trie pointer $a_{i,j}$, as before, together with an index $x_i$ into the compressed $m$-trie.

This method was suggested by Bernard Gattingo in 1986 (unpublished). It finds
all 5 $\times$ 6 word rectangles in just 400 gigamems; and its running time for \textquoteleft transposed\textquoteright{} 6 $\times$ 5 rectangles turns out to be slightly less (380 gigamems). Notice that only one mem
is needed to access each (character, link) pair in the compressed trie.

29. Leave out face and (of course) dead; the remaining eleven are fine.

30. Keep tables $p_i$, $p'_i$, $p''_i$, $s_i$, $d_i$, $d''_i$, for $0 \leq i, j, k < m$, each capable of storing a
ternary digit. Also keep a table $x_0$, $x_1$, \ldots{} of tentatively accepted words. Begin with
$g \leftarrow 0$. Then for each input $w_j = abcd$, where $0 \leq a, b, c, d < m$, set $x_0 \leftarrow abcd$ and
also do the following: Set $p_a \leftarrow p_a + 1$, $p'_a \leftarrow p'_a + 1$, $p''_a \leftarrow p''_a + 1$, $s_a \leftarrow s_a + 1$,
$s''_a \leftarrow s''_a + 1$, $d_a \leftarrow d_a + 1$, $d'_a \leftarrow d'_a + 1$, $d''_a \leftarrow d''_a + 1$, where $x + y = \min(2, x + y)$ denotes saturating ternary
addition. Then if $s_a P_{a,b,c,d} + s'_a P_{a,b,c,d} + s''_a P_{a,b,c,d} = 0$ for all $x_k = a' b' c' d'$, where
$0 \leq k \leq g$, set $g \leftarrow g + 1$. Otherwise reject $w_j$ and set $p_a \leftarrow p_a - 1$, $p'_a \leftarrow p'_a - 1$, $p''_a \leftarrow p''_a - 1$, $s_a \leftarrow s_a - 1$, $s''_a \leftarrow s''_a - 1$, $d_a \leftarrow d_a - 1$, $d'_a \leftarrow d'_a - 1$, $d''_a \leftarrow d''_a - 1$.

31. (a) The word be appears in message abcd if and only if $a \rightarrow b$, $b \rightarrow c$, and $c \rightarrow d$.

(b) For $0 \leq k < r$, put vertex $v$ into class $k$ if the longest path from $v$ has
length $k$. Given any such partition, we can include all arcs from class $k$ to class $j < k$
without increasing the path lengths. So it's a question of finding the maximum of
$\sum_{0 \leq i < j < r} p_j p_i$ subject to $p_0 + p_1 + \cdots + p_{r-1} = m$. The values $p_j = \lceil (m+j) / r \rceil$ achieve
this (see exercise 7.2.1.4-68(a)). When $r = 3$ the maximum simplifies to $\lfloor m^2 / 3 \rfloor$.

32. (a) The first-round basins are 1, 3, 6, 9, 13, 16 (and 20): retain boundaries 5, 8, 15.
The second-round substrings $y_0 = 926$, $y_1 = 5358979$, $y_2 = 333834115$ have different
lengths, so lexicographic comparison is unnecessary; the answer is $y_2 y_0 y_1$.

(b) Each substring consists of at least three substrings of the previous round.

(c) Let $a_0 = 0$, $b_0 = 1$, $a_{n+1} = a_n a_n b_n$, $b_{n+1} = a_n b_n b_n$; use $a_n$ or $b_n$ when $n = 3^n$.
(d) We use an auxiliary subroutine ‘compare(i)’, which returns \([y_{i-1} > y_i]\), given
\(i > 0\): If \(b_i - b_{i-1} \neq b_{i+1} - b_i\), return \([b_i - b_{i-1} > b_{i+1} - b_i]\). Otherwise, for \(j = 0, 1, \ldots, \)
while \(b_i + j < b_{i+j}\), if \(x_{b_{i+j-1}} \neq x_{b_{i+j}}\) return \([x_{b_{i+j-1}} > x_{b_{i+j}}]\). Otherwise return 0.

E1. [Initialize] Set \(x_j \leftarrow x_{j-n}\) for \(n \leq j < 3n\), \(b_j \leftarrow j\) for \(0 \leq j < 3n\), and \(t \leftarrow n\).

E2. [Begin a round] Set \(t' \leftarrow 0\). Find the smallest \(i > 0\) such that \(i \leq n\) and compare(i) = 1. (If no such \(i\) exists, report an error: The input \(x\) was equal to one of its cyclic shifts.) Then, while compare(i + 1) = 0, set \(i \leftarrow i + 1\).

(Now \(i\) is the first positive index of a basin.)

E3. [Climb the range.] Set \(q \leftarrow i + 1\). While compare(q + 1) = 0, set \(q \leftarrow q + 1\).

E4. [Advance to next basin] Set \(j \leftarrow q + 1\). While compare(j + 1) = 1, set \(j \leftarrow j + 1\). If \(j = i\) is even, go to E6.

E5. [Retain a boundary] If \(q = i\) is odd, set \(q \leftarrow q + 1\). Then if \(q < j\), set \(b_j \leftarrow b_q\); otherwise set \(b_j \leftarrow b_j\) for \(t' \geq k > 0\) and \(b_j \leftarrow b_j\). Finally set \(t' \leftarrow t' + 1\).

E6. [Done with round?] If \(j \leq t\), set \(i \leftarrow j\) and return to E3. Otherwise, if \(t' = 1\), terminate; \(\sigma\) begins at \(x_{t'}\). Otherwise set \(t \leftarrow t'\), \(b_k \leftarrow b_k + n\) for \(0 \leq k < 3n\), and \(b_k \leftarrow b_k + n\) for \(k \geq t\) while \(b_{k-1} < 2n\). Return to E2.

[Eastman proved the commutative property in IEEE Trans. IT-11 (1965), 263–267.]

33. Call \(x_{i_1}, \ldots, x_{i_s}\) a “range sequence” if \(x_0 > x_1 \leq \cdots \leq x_{i_s-1}\), and let \(f_s(m)\) be the number of range sequences for which \(m > x_0\) and \(x_{i_s-1} < m\). The number of such sequences with \(x_1 = j\) is \((m-j-1)^{m+j+3} = (k-1)^{m+j+3}\), summing over \(0 \leq j < m\) gives \(f_s(m) = (k-1)^{m+j+3}\). Thus \(G_m(z) = \sum_{j=0}^{m} f_s(m)z^j = (m+1)/(1-z)^m\).

Algorithm E finishes in one round if and only if some cyclic shift of \(x\) has the form \(w_1 \ldots w_{m-1}w_m\), where each \(w_j\) is a range sequence and \([w_j] \text{ mod } 2 = [j = p]\). The number of aperiodic \(x\) that finish in one round is therefore \(n[x^n] G_m(z)\), where

\[
G_m(z) = \frac{F_m(z) - F_m(-z)}{F_m(-z) + F_m(z)} = \frac{(1 + mz)(1 - z)^m - (1 - mz)(1 + z)^m}{(1 + mz)(1 - z)^m + (1 - mz)(1 + z)^m}.
\]

To get the stated probability, divide by \(\sum_{d | m} u(d) m^{n/d}\), the number of aperiodic \(x\).

(See Eq. 7.2.1.1–(60).) For \(n = 3, 5, 7, 9\) these probabilities are 1, 1, 1, and \(1 - 3/(m^3 - 1)\).

34. If so, it couldn’t have been 0111, 0110, 1100, or 1001.

35. That section considered such representations of stacks and queues, but not unordered sets, because large blocks of sequential memory were either nonexistent or ultra-expensive in oden days. Linked lists were the only decent option for families of variable-size sets, because they made more readily adapt to the limited high-speed memory.

36. (a) The blue word \(x\) with \(a = 4\) (namely 1101) appears in its P2 list at location 5e.

(b) The P3 list for words of the form 010\* is empty. (Both 0100 and 0101 are red.)

37. (a) The S2 list of 0010 has become closed (hence 0110 and 1110 are hidden).

(b) Word 1101 moved to the former position of 1001 in its S2 list, when 1001 became red. (Previously 1011 had moved to the former position of 0001.)

38. In this case, which of course happens rarely, it’s safe to set all elements of STAMP to zero and set \(\sigma \leftarrow 1\). (Do not be tempted to save one line of code by setting all STAMP elements to \(-1\) and failure \(\sigma = 0\). That might fail when \(\sigma\) reaches the value \(-1\).)

39. (a) Set \(r \leftarrow m + 1\). Then for \(k = 0, 1, \ldots, f - 1\), set \(t \leftarrow \text{FREE} [k], j \leftarrow \text{MEM} [\text{CLOFF} + 4t + m^4] - \text{CLOFF} + 4t\), and if \(j < r\) set \(r \leftarrow j\), \(c \leftarrow t\); break out of the loop if \(r = 0\).
(b) If \( r > 0 \) set \( x \leftarrow \text{MEM} [\text{CLOFF} + 4d(\text{ALF}[x])] \).

(c) If \( r > 1 \) set \( q \leftarrow 0 \), \( p' \leftarrow \text{MEM}[\text{PP}] \), and \( p \leftarrow \text{POISON} \). While \( p < p' \) do the following steps: Set \( y \leftarrow \text{MEM}[p] \), \( z \leftarrow \text{MEM}[p + 1] \), \( y' \leftarrow \text{MEM}[y + m^3] \), and \( z' \leftarrow \text{MEM}[z + m^2] \). (Here \( y \) and \( z \) point to the heads of prefix or suffix lists; \( y' \) and \( z' \) point to the tails.) If \( y = y' \) or \( z = z' \), delete entry \( p \) from the poison list; this means, as in (28), to set \( p' \leftarrow p' - 2 \), and if \( p \neq p' \) to store \( p, \text{MEM}[y'] \) and store \( p + 1, \text{MEM}[p'] \) and \( p \leftarrow \text{MEM}[z] \); otherwise set \( p \leftarrow p + 2 \); if \( y' - y \geq z' - z \) and \( y'-y > q \), set \( q \leftarrow y'-y \) and \( x \leftarrow \text{MEM}[y] \); if \( y'-y < z' - z \) and \( z'-z > q \), set \( q \leftarrow z'-z \) and \( x \leftarrow \text{MEM}[z] \). Finally, after \( p \) has become equal to \( p' \), store \( \text{PP}, p' \) and set \( c \leftarrow cl(\text{ALF}[x]) \). (Experiments show that this "max kill" strategy for \( r > 1 \) slightly outperforms a selection strategy based on \( r \) alone.)

40. (a) First there’s a routine \( \text{rem}(\alpha, \delta, o) \) that removes an item from a list, following the protocol (21): Set \( p \leftarrow \delta + o \) and \( q \leftarrow \text{MEM}[p + m^2] \). If \( q \geq p \) (meaning that list \( p \) isn’t closed or being killed), store \( (p + m^3, q) \), set \( t \leftarrow \text{MEM}[\alpha + o - m^2] \); and if \( t \neq q \) also set \( y \leftarrow \text{MEM}[q] \), store \( (t, y) \), and store \( (\text{ALF}[y] + o - m^2, t) \).

Now, to redden \( x \) we set \( \alpha \leftarrow \text{ALF}[x] \), store \( \alpha, \text{RED} \); then \( \text{rem}(\alpha, p_1(\alpha), \text{P1OFF}) \), \( \text{rem}(\alpha, p_2(\alpha), \text{P2OFF}) \), \ldots , \( \text{rem}(\alpha, s_3(\alpha), \text{S3OFF}) \), and \( \text{rem}(\alpha, 4d(\alpha), \text{CLOFF}) \).

(b) A simple routine \( \text{close}(\delta, o) \) closes list \( \delta + o \): Set \( p \leftarrow \delta + o \) and \( q \leftarrow \text{MEM}[p + m^2] \); if \( q \neq p - 1 \), store \( (p + m^4, p - 1) \).

Now, to green \( x \) we set \( \alpha \leftarrow \text{ALF}[x] \), store \( \alpha, \text{GREEN} \); then \( \text{close}(p_1(\alpha), \text{P1OFF}) \), \( \text{close}(p_2(\alpha), \text{P2OFF}) \), \ldots , \( \text{close}(s_3(\alpha), \text{S3OFF}) \), and \( \text{close}(4d(\alpha), \text{CLOFF}) \). Finally, for \( p \leq r < q \) (using the \( p \) and \( q \) that were just set within ‘close’), if \( \text{MEM}[r] \neq x \) redden \( \text{MEM}[r] \).

(c) First set \( p' \leftarrow \text{MEM}[\text{PP}] + 6 \), and store \( p'-6, p_1(\alpha) + \text{S1OFF} \), store \( p'-5, s_2(\alpha) + \text{P3OFF} \), store \( p'-4, p_2(\alpha) + \text{S2OFF} \), store \( p'-3, s_3(\alpha) + \text{P2OFF} \), store \( p'-2, p_3(\alpha) + \text{S3OFF} \), store \( p'-1, s_1(\alpha) + \text{P1OFF} \); this adds the three poison items (27).

Then set \( p \leftarrow \text{POISON} \) and do the following while \( p < p' \): Set \( y, z, y', z' \) as in answer 30(c), and delete poison entry \( p \) if \( y = y' \) or \( z = z' \). Otherwise if \( y' < y \) and \( z' < z \) set \( x \rightarrow \text{C6} \) (a poisoned suffix-prefix pair is present). Otherwise if \( y' > y \) and \( z' > z \) set \( p \leftarrow p + 2 \). Otherwise if \( y' < y \) and \( z' > z \), redden \( x \) for \( z \leq r < z' \), and delete poison entry \( p \). Otherwise \( y' > y \) and \( z' < z \), redden \( x \) for \( y \leq r < y' \), and delete poison entry \( p \).

Finally, after \( p \) has become equal to \( p' \), store \( \text{PP}, p' \).

42. Exercise 32 exhibits such codes explicitly for all odd \( n \). The earliest papers on the subject gave solutions for \( n = 2, 4, 6, 8 \). Y. Niño subsequently found a code for \( n = 10 \) [IEEE Trans. IT-19 (1973), 580–581].

This problem can readily be encoded in CNF and given to a SAT solver. The case \( n = 10 \) involves 990 variables and 8.6 million clauses, and is solved by Algorithm 7.222C in 10.5 gigamems. The case \( n = 12 \) involves 4020 variables and 175 million clauses. After being split into seven independent subproblems (by appending mutually exclusive unit clauses), it was proved unsatisfiable by that algorithm after about 86 teramems of computation.

So the answer is “No.” The maximum-size code for \( n = 12 \) remains unknown.

44. (a) There are 28 comma-free binary codes of size 3 and length 4; Algorithm C produces half of them, because it assumes that cycle class \([0001]\) is represented by \([0001, 0010] \). They form eight equivalence classes, two of which are symmetric under the operation of complementation and reflection; representatives are \([0001, 0011, 0111]\) and \([0010, 0111, 1011]\). The other six are represented by \([0001, 0110, 0111] \) and \([0010, 1001, 1011\) or \([1011]\).\([0001, 1100, 1110]\), \([0011, 0011, 1101]\), \([0010, 0011, 1101]\).
(b) Algorithm C produces half of the 144 solutions, which form twelve equivalence classes. Eight are represented by \{0000, 0002, 1000, 1002, 2001, 2000, 2002, 2011, 2012, 2102, 2112, 2122 or 2212\} and \{(0102, 1011, 1012) or (1020, 1101, 2101)\} and \{(1202, 2202, 2111) or (2021, 2022, 1112)\}; four are represented by \{0001, 0020, 0021, 0022, 1001, 1020, 1021, 1022, 1121, 1120, 1122, 1221, 1220, 2001, 2012, 2022\} and \{(1011, 1012, 2221) or (1101, 2012, 1222)\}.

(c) Algorithm C yields half of the 2304 solutions, which form 48 equivalence classes. Twelve classes have unique representatives that omit cycle classes \{0123\}, \{0103\}, \{1213\}, one such being the code \{0010, 0020, 0030, 0110, 0112, 0120, 0121, 0122, 0130, 0131, 0132, 0133, 0210, 0212, 0223, 0230, 0310, 0312, 0313, 0320, 0322, 0330, 0332, 0333, 1110, 1112, 1113, 2010, 2030, 2110, 2112, 2123, 2210, 2212, 2213, 2230, 2310, 2312, 2313, 2320, 2322, 2323, 3110, 3112, 3113, 3120, 3122, 3210, 3212, 3230, 3310, 3312, 3313\). The others each have two representatives that omit classes \{0123\}, \{0103\}, \{1213\}, one such being the code \{0001, 0002, 0003, 0201, 0202, 1001, 1002, 1003, 1011, 1021, 1022, 1023, 1031, 1032, 1033, 1201, 1203, 1211, 1213, 1221, 1223, 1231, 1233, 1311, 1321, 1323, 1331, 1332, 2001, 2002, 2003, 2021, 2022, 2031, 2032, 2033, 2031, 2032, 2033, 2201, 2203, 2221, 2223, 2201, 2202, 2203, 2200, 2212, 2213, 2210, 2212, 2213, 2230, 2310, 2312, 2313, 2320, 2322, 2323, 3110, 3112, 3113, 3120, 3122, 3210, 3212, 3230, 3310, 3312, 3313\} and its isomorphic image under reflection and \{01\}(23).

45. (The maximum size of such a code is currently unknown. Algorithm C isn’t fast enough to solve this problem on a single computer, but a sufficiently large cluster of machines and/or an improved algorithm should be able to discover the answer. The case \(m = 3\) and \(n = 6\) is also currently unsolved.)

90. Suppose there are \(n\) questions, whose answers each lie in a given set \(S\). A student supplies an answer list \(a = a_1 \ldots a_n\), with each \(a_j \in S\); a grader supplies a Boolean vector \(\beta = x_1 \ldots x_n\). There is a Boolean function \(f_{\beta}(a, \beta)\) for each \(j \in \{1, \ldots, n\}\) and \(s \in S\). A graded answer list \((a, \beta)\) is valid if and only if \(F(a, \beta) = \text{true}\), where

\[
F(a, \beta) = F(a_1 \ldots a_n, x_1 \ldots x_n) = \bigwedge_{j=1}^{n} (a_j \Rightarrow x_j \equiv f_{\beta}(a, \beta)).
\]

The maximum score is the largest value of \(x_1 + \ldots + x_n\) over all graded answer lists \((a, \beta)\) that are valid. A perfect score is achieved if and only if \(F(a, 1 \ldots 1) = \text{true}\).

Thus, in the warmup problem we have \(n = 2\), \(S = \{A, B\}\); \(f_{\beta} = [a_1 = A, B]; f_{\beta} = [a_1 = A]; f_{\beta} = [a_1 = A]; f_{\beta} = [a_1 = A]\). The four possible answer lists are:

- **AA**: \(F = ([a_1 = [A = B]] \land [a_2 = x_1])\)
- **AB**: \(F = ([a_1 = [B = B]] \land [a_2 = \overline{x}_2 \land [A = A]])\)
- **BA**: \(F = ([a_1 = [B = A]] \land [a_2 = x_1])\)
- **BB**: \(F = ([a_1 = [B = A]] \land [a_2 = \overline{x}_2 \land [B = A]])\)

Thus AA and BA must be graded 00; AB can be graded either 10 or 11; and BB has no valid grading. Only AB can achieve the maximum score, 2; but 2 isn’t guaranteed.

In Table 666 we have, for example, \(f_{1C} = [a_1 \neq A] \land [a_2 \neq A] \land [a_3 = A]; f_{2D} = [a_1 = D] \land [a_2 = D] \land [a_3 = D]; f_{2A} = [\Sigma_1, \Sigma_2 = 1 \land \Sigma_3 = 0]\), where \(\Sigma_j = \sum_{1 \leq j \leq 3} \in [a_j = s]\). It’s amusing to note that \(f_{2A} = [[\Sigma_1 = 1, \Sigma_2 = 1, \Sigma_3 = 0]] = [2, 3, 4, 5, 6]\).

The other cases are similar (although often more complicated) Boolean functions—except for 20D and 20E, which are discussed further in exercise 91.

Notice that an answer list that contains both 10E and 17E must be discarded: It can’t be graded, because 10E says \(x_{10} \equiv \overline{x}_1\), while 17E says \(x_{17} \equiv x_{10}\).
By suitable backtrack programming, we can prove that no perfect score is possible. Indeed, if we consider the answers in the order \(3, 15, 20, 19, 2, 1, 17, 10, 5, 4, 16, 11, 13, 14, 7, 18, 6, 8, 12, 9\), many cases can quickly be ruled out. For example, suppose \(a_3 = C\). Then we must have \(a_1 \neq a_2 \cdots \neq a_{10} \neq a_{17} = a_{18} \neq a_{20} \neq a_{20}\), and early cutoffs are often possible. We might reach a node where the remaining choices for answers 5, 6, 7, 8, 9 are respectively \{C, D\}, \{A, C\}, \{B, D\}, \{A, B, E\}, \{B, C, D\}, say. Then if answer 8 is forced to be B, answer 7 can only be D; hence answer 6 is also forced to be A. Also answer 9 can no longer be B. An instructive little propagation algorithm will make such deductions nicely at every node of the search tree. On the other hand, difficult questions like 7, 8, 9, are best not handled with complicated mechanisms; it’s better just to wait until all twenty answers have been tentatively selected, and to check such hard cases only when the checking is easy and fast. In this way the author’s program showed the impossibility of a perfect score by exploring just 52859 nodes, after only 3.4 megamems of computation.

The next task was to try for score 19 by asserting that only \(x_j\) is false. This turned out to be impossible for \(1 \leq j \leq 18\), based on very little computation whatsoever (especially of course, when \(j = 6\)). The hardest case, \(j = 15\), needed just 56 nodes and fewer than 5 kilobits. Then, however, there are three solutions found: One for \(j = 19\) (385 kilonodes, 11 megamems) and two for \(j = 20\) (131 kilonodes, 8 megamems), namely

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

(The incorrect answers are shown here as lowercase letters. The first two solutions establish the truth of 20B and the falsity of 20E.)

91. Now there’s only one list of answers with score \(\geq 19\), namely (iii). But that is paradoxical—because it claims 20E is false; hence the maximum score cannot be 19!

Paradoxical situations are indeed possible when the global function \(F\) of answer 90 is used recursively within one or more of the local functions \(f_j\). Let’s explore a bit of recursive thinking by considering the following two-question, two-letter example:

1. (A) Answer 1 is incorrect.  
   (B) Answer 2 is incorrect.

2. (A) Some answers can’t be graded consistently.  
   (B) No answers achieve a perfect score.

Here we have \(f_{1A} = \overline{x}_1; f_{1B} = \overline{x}_2; f_{2A} = \exists x_1 \exists x_2 \forall x_1 \forall x_2 \neg F(a_1, a_2, x_1, x_2); f_{2B} = \forall x_1 \forall x_2 \neg F(a_1, a_2, 11)\). (Formulas quantified by \(\exists x\) or \(\forall x\) expand into \([S] F\) terms, while \(\exists x\) or \(\forall x\) expand into two; for example, \(\exists x \forall x g(a, x) = (g(A, 0) \lor g(A, 1)) \lor (g(B, 0) \lor g(B, 1))\) when \(S = \{A, B\}\).) Sometimes the expansion is undefined, because it has more than one “fixed point”, but in this case there’s no problem because \(f_{2A}\) is true; Answer AA can’t be graded, since 1A implies \(x_1 = \overline{x}_1\). Also \(f_{2B}\) is true, because both BA and BB imply \(x_1 = \overline{x}_2\). Thus we get the maximum score 1 with either BA or BB and grades 01.

On the other hand the simple one-question, one-letter questionnaire ‘1. (A) The maximum score is 1 has an indeterminate maximum score. For in this case \(f_{1A} = F(A, 1)\). We find that if \(F(A, 1) = 0\), only \((A, 0)\) is a valid grading, so the only possible score is 0; similarly, if \(F(A, 1) = 1\), the only possible score is 1.

OK, suppose that the maximum score for the modified Table 666 is \(m\). We know that \(m < 19\); hence (iii) isn’t a valid grading. It follows that 20E is true, which means that every valid graded list of score \(m\) has \(x_{20}\) false. And we can conclude that \(m = 18\),

October 4, 2015
because of the following two solutions (which are the only possibilities with 20C false):

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

But wait: If \( m = 18 \), we can score 18 with 20A true and two errors, using (say)

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]
or 47 other answer lists. This \textit{contradicts} \( m = 18 \), because \( x_{20} \) is true.

End of story? No. This argument has implicitly been predicated on the assumption that 20D is false. What if \( m \) is indeterminate? Then a new solution arises

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]
of score 19. With (iii) it yields \( m = 19 \)! If \( m \) is determinate, we've shown that \( m \)
cannot actually be defined consistently; but if \( m \) is indeterminate, it's definitely 19.

\textit{Question 20 was designed to create difficulties. [-]}\]

— DONALD R. WOODS (2001)

999. ...
INDEX AND GLOSSARY

He writes indexes to perfection.
— OLIVER GOLDSMITH, Citizen of the World (1762)

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.

Carroll, Lewis (= Dodgson, Charles Lutwidge), iii.
Dodgson, Charles Lutwidge, iii.
Jiggs, B. H. (pen name of Baumert, Hales, Jewett, Imaginary, Colomb, Gordon, and Selfridge), ???.

King, Benjamin Franklin, Jr., 2.
Lennon, John Winston Ono, 2.
MPR: Mathematical Preliminaries Redux, v.
Nothing else is indexed yet (sorry).

Preliminary notes for indexing appear in the upper right corner of most pages.

If I've mentioned somebody’s name and forgotten to make such an index note, it’s an error (worth $2.56).