THE ART OF COMPUTER PROGRAMMING
VOLUME 4  PRE-FASCICLE 5B

INTRODUCTION TO BACKTRACKING

DONALD E. KNUTH  Stanford University

ADDISON–WESLEY  ∨

December 22, 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 MMIX 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 -85), 22 December 2015

December 22, 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 we need 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.
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, 50, 51, 95, . . . . 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 7.2.2–(26). This concept is quite different from the so-called time stamps in persistent data structures, and quite different from the so-called time stamps in depth-first search algorithms, and quite different from the so-called time stamps in cryptology, although many programmers do use the name "time stamp" for those kinds of stamp.) It's a technique that I've seen often, in programs that have come to my attention 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 Umbrary 2015

D. E. K.

December 22, 2015
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

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

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.
Nowhere to go but out,
Nowhere to come but back.
— BEN KING, in The Sum of Life (c. 1893)

No one I think is in my tree.
— JOHN LENNON, in Strawberry Fields Forever (1967)

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 $P_n(x_1, x_2, \ldots, x_n)$ 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 $P_l(x_1, \ldots, x_l)$ for $1 \leq l < n$, such that

\[ P_l(x_1, \ldots, x_l) \text{ is true whenever } P_{l+1}(x_1, \ldots, x_{l+1}) \text{ is true;} \]
\[ P_l(x_1, \ldots, x_l) \text{ is fairly easy to test, if } P_{l-1}(x_1, \ldots, x_{l-1}) \text{ holds.} \]

(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 $P_l$.) Then we can proceed lexicographically as follows:

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

**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 $l > 0$, downgrade 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

$$x_j \neq x_k \quad \text{and} \quad |x_k - x_j| \neq k - j, \quad \text{for} \quad 1 \leq j < k \leq n. \quad (3)$$

(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 $P_1$ requires testing (3) only for $k = l$
when $P_{l-1}$ is known. Notice that $P_1(x_1)$ is always true in this example.

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 $x_1 x_2 x_3 x_4 = 2413$. 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,

$$\text{(4)}$$

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 using subscripts from 1 to $n$ instead of from 0 to $n - 1$ 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.)

December 22, 2015
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 \((1, 16, 210, 2236, 19688, 141812, 838816, 3908456, 15324708, 46358876, 108478966, 193892860, 260303408, 253897632, 171158018, 72002088, 14772512)\), 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 \( x_i - x_j \in \{ j - l, 0, l - j \} \) 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 mems per node.

We can do better by introducing three simple arrays. Property \( P_l \) in (3) 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 \( a_1 \ldots a_n, b_1 \ldots b_{n-1}, \) 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 \( c_{l+n} = 1 \), go to B4*. Otherwise set \( a_t \leftarrow 1, b_{t+l-1} \leftarrow 1, c_{l+n} \leftarrow 1, x_l \leftarrow t, l \leftarrow l + 1, \) 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 \( t \leftarrow x_l, c_{l+n} \leftarrow 0, b_{t+l-1} \leftarrow 0, a_t \leftarrow 0, \) and return to B4*. (Otherwise stop.)

Notice how step B5* nearly 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 $a_i$ before $b_{i+1-1}$ and $c_{i+1+n}$, because those arrays are independent.

The auxiliary arrays $a$, $b$, $c$ make it easy to test property $P_l$ 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 $x_l \leftarrow t$ and later to fetch $t \leftarrow x_l$; 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 $x_l$ run through all elements of $D_l$, he calculated and stored the set

$$S_l \leftarrow \{ x \in D_l \mid P_l(x_1, \ldots, x_{l-1}, x) \} \quad (5)$$

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 $x \in D_l$. In essence, he used the following variant of Algorithm B:

**Algorithm W (Walker’s backtracking).** Given domains $D_k$ and cutoffs $P_l$ as above, this algorithm visits all sequences $x_1x_2\ldots x_n$ 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 $P_{l-1}(x_1, \ldots, x_{l-1})$ holds.) If $l > n$, visit $x_1x_2\ldots x_n$ and go to W4. Otherwise determine the set $S_l$ as in (5).

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

Walker applied this method to the $n$ queens problem by computing $S_l = U \setminus A_l \setminus B_l \setminus C_l$, where $U = D_l = \{1, \ldots, n\}$ and

$$A_l = \{x_j | 1 \leq j < l\}, \quad B_l = \{x_j + j - l | 1 \leq j < l\}, \quad C_l = \{x_j - j + l | 1 \leq j < l\}. \quad (6)$$

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 downvating 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

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

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

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
Such a condition turns out to be easy to maintain. In particular, whenever our data are present in the form $x_1, x_2, \ldots, x_n$, we have the following properties:

1. If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
2. If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For instance, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For instance, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For instance, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For instance, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For instance, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.

For example, when $i, j \leq n$, the equation $x_i = x_j$ implies $x_i = x_j = k$, for $1 \leq i, j \leq n$. Such equations are solved by some kind of algorithm. We have the following properties:

- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_1$ for some integer $k_1$.
- If $x_i = x_j, i \neq j$, then $x_i = x_j = k_2$ for some integer $k_2$.
**Algorithm L (Langford pairs).** This algorithm visits 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 \(k \leftarrow p_0\). If \(k = 0\), visit \(x_1 x_2 \ldots x_{2n}\) and go to L5. Otherwise set \(j \leftarrow 0\), and while \(x_l < 0\) 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 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

![Search Tree for n = 4](image)

Solutions appear at depth \(n\) in this tree, even though they involve \(2n\) values \(x_1 x_2 \ldots x_{2n}\).

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,
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 \( D_1 \) 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 **WORDS(5757)**, the collection of 5-letter words in the Stanford GraphBase. To find such patterns, we can suppose that column \( l \) contains the \( x_l \)th most common 5-letter word, where \( 1 \leq x_l \leq 5757 \) for \( 1 \leq l \leq 6 \); hence there are \( 5757^6 = 36,406,369,848,837,732,146,649 \) ways to choose the columns. In (10) we have \( x_1 \cdots x_6 = 1446 \ 185 \ 1021 \ 2537 \ 66 \ 255 \).

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 23067 nodes. The prefix \( at \) corresponds to node number 200, whose 26 entries are

\[
(484, 0, 0, 0, 1589, 0, 0, 0, 2609, 0, 0, 0, 0, 0, 1251, 0, 0, 563, 0, 0, 0, 1621, 0); \quad (11)
\]

this means that \( ata \) is node 484, \( ate \) is node 1589, \( . . . \), \( sty \) is node 1621, and there are no 6-letter words beginning with \( atb, atc, . . . , atx, atz \). A slightly different convention is used for prefixes of length \( n - 1 \); for example, the entries for node 580, ‘corn\( e \)’, are

\[
(3879, 0, 0, 3878, 0, 0, 0, 0, 0, 0, 9602, 0, 0, 0, 0, 0, 171, 0, 5013, 0, 0, 0, 0, 0, 0); \quad (12)
\]

meaning that \( corn, corn, corn, corn, corn, \) and \( corn \) are ranked 3879, 3878, 9602, 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 *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, ‘\( it \)’, occurs roughly half as often (3,021,525). The OSPD4 list includes respectively \( (101, 1001, 4002, 8887, 15727, 23069, 25718, 29130, 22314) \) words of lengths \( (2, 3, . . . , 10) \), of which (97, 771, 2451, 4474, 6910, 8882, 9205, 8255, 6056) occur at least six times in the British National Corpus.
7.2.2 BACKTRACK PROGRAMMING

Suppose \( x_1 \) and \( x_2 \) specify the 5-letter column-words \( \text{slums} \) and \( \text{total} \) as in (10). Then the trie tells us that the next column-word \( x_3 \) must have the form \( c_1c_2c_3c_4c_5 \) where \( c_1 \in \{ \text{a, e, i, o, r, u, y} \} \), \( c_2 \notin \{ \text{a, h, j, k, y, z} \} \), \( c_3 \in \{ \text{e, m, o, t} \} \), \( c_4 \notin \{ \text{a, b, o} \} \), and \( c_5 \in \{ \text{a, e, i, o, u, y} \} \). (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 teramems of computation to find them all. In fact, the profile of the search tree is

\[
(1, 5757, 245830, 360728099, 579940198, 29621728, 623415),
\]

indicating for example that just 300,728,099 of the 5757 \(^3 \) = 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 \( x_1 \) 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

\[
(5757.0, 1697.9, 844.1, 273.5, 153.5, 100.8);
\]

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

**Commafree codes.** Our next example deals entirely with \( 4 \)-letter words. But it's not obscure: it's an intriguing question of coding theory. The problem is to find a set of \( 4 \)-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 \( 4 \)-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 \( x_1x_2x_3x_4 \) and \( x_5x_6x_7 \) are codewords, then \( x_2x_3x_4x_5 \) and \( x_3x_4x_5x_6 \) and \( x_4x_5x_6x_7 \) aren't. (For example, \( \text{like isn't} \).) 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 do do 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 mitem, 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

\[
\begin{align*}
0001 &= \{0001, 0010, 0100, 1000\}, \\
0011 &= \{0011, 0110, 1100, 1001\}, \\
0111 &= \{0111, 1110, 1011, 1101\},
\end{align*}
\]

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 aperiodic string in Section 7.2.1.1, where such words were called prime strings and where some of the remarkable properties of prime strings 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 0111 or 0011.

We could systematically backtrack through 18 levels, choosing \(x_1\) in \{0001\} and \(x_2\) in \{0011\}, etc., and rejecting each \(x_1\) as in Algorithm B whenever we discover that \(\{x_1, x_2, \ldots, x_k\}\) isn't commafree. For example, if \(x_1 = 0010\) and we try \(x_2 = 1001\), this approach would backtrack because \(x_1\) occurs inside \(x_2x_1\).

But a naive 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 \(\ast 001\), such as 0001 when \(m \geq 3\) and 0010 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 \(1\ast\ast\ast\) or \(\ast\ast0\), because \(x_1\ldots\ast\ast\ast\) includes \(x_2\) and \(\ast\ast\ast0x_2\) 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 would be 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 \(x_{0001}, x_{0002}, x_{0011}, x_{0012}, x_{0021}, x_{0022}, x_{0102}, x_{0111}, x_{0112}, x_{0122}, x_{0211}, x_{0212}, x_{0221}, x_{0222}, x_{1112}, x_{1122}, x_{1222}\). (Algorithm 7.2.1.1F is a good way to generate those names.) At every node of the search tree we then can 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 \(x_{1122} \leftarrow 2111\) 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 \(x_{0001} = 0011\), we aren’t obliged to continue by trying another value for \(x_{0001}\). 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 \(x_{0002} = 0002, x_{0022} = 0022, x_{0122} = 0122, x_{0222} = 0222, x_{1122} = 1222,\) and \(x_{0111} = 1001\).

**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 downvoted 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 \(\text{HEAD}\), and \(\text{TAIL}\) points just beyond the end of the list. For example,

\[
\begin{array}{cccccccc}
\text{HEAD} & 3 & 9 & 1 & 4 & & & & \text{TAIL}
\end{array}
\]
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
\[
\text{MEM}[\text{TAIL}] \leftarrow x, \quad \text{TAIL} \leftarrow \text{TAIL} + 1.
\]
(17)
Conversely, if \( \text{HEAD} \leq p < \text{TAIL} \), we can easily delete \( \text{MEM}[p] \):
\[
\text{TAIL} \leftarrow \text{TAIL} - 1; \quad \text{if } p \neq \text{TAIL}, \text{set } \text{MEM}[p] \leftarrow \text{MEM}[\text{TAIL}].
\]
(18)
(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.)

We can’t delete an item from a list without knowing its \( \text{MEM} \) location. Thus we will often want to maintain an “inverse list,” assuming that all items \( x \) lie in the range \( 0 \leq x < M \). For example, (16) becomes the following, if \( M = 10 \):

![Diagram of a list with HEAD and TAIL pointers and shaded cells with undefined contents.]

(19)

(Shaded cells have undefined contents.) With this setup, insertion (17) becomes
\[
\text{MEM}[\text{TAIL}] \leftarrow x, \quad \text{MEM}[\text{IHEAD} + x] \leftarrow \text{TAIL}, \quad \text{TAIL} \leftarrow \text{TAIL} + 1.
\]
and \( \text{TAIL} \) will never exceed \( \text{HEAD} + M \). Similarly, deletion of \( x \) becomes
\[
\text{P} \leftarrow \text{MEM}[\text{IHEAD} + x], \quad \text{TAIL} \leftarrow \text{TAIL} - 1; \quad \text{if } p \neq \text{TAIL}, \quad \text{set } y \leftarrow \text{MEM}[\text{TAIL}], \quad \text{MEM}[p] \leftarrow y, \quad \text{MEM}[\text{IHEAD} + y] \leftarrow p.
\]
(21)

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

![Diagram of a list with HEAD and TAIL pointers and shaded cells with undefined contents.]

(22)

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 commafree code problem with \( m = 3 \), we can begin by putting items representing the \( M = 18 \) cycle classes \( [0001], [0002], \ldots, [1222] \) into memory cells \( \text{HEAD} \) through \( \text{HEAD} + 17 \). Initially they’re all active, with
TAIL = HEAD + 18 and MEM[HEAD + c] = HEAD + c for 0 ≤ 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:

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

\[
\text{if } P \neq \text{TAIL}, \text{set } y \leftarrow \text{MEM[TAIL]}, \text{MEM[TAIL]} \leftarrow c, \text{MEM[P]} \leftarrow y,
\]

\[
\text{MEM}[\text{HEAD} + c] \leftarrow \text{TAIL}, \text{MEM}[\text{HEAD} + y] \leftarrow P. \quad (23)
\]

Later on, after backtracking to a state where we once again want c to be consid-
ered active, we simply set TAIL ← TAIL + 1, because c will already be in place!

**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 repeat-
edly 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 green if it’s part
of the current set of tentative codewords. It is red if it’s not currently a candidate
for such status, either because it is incompatible with the existing green words
or because the algorithm has already examined all scenarios in which it is green
in their presence. Every other word is blue, and sort of in limbo; the algorithm
might or might not decide to make it red or 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 \( x \) 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 MEM[\( \alpha \)], using one of
the arbitrary internal codes (GREEN), 0 (RED), or 1 (BLUE).

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

- P1(\( x \)), P2(\( x \)), P3(\( x \)) are the blue words matching \( x_1** \), \( x_1x_2** \), \( x_1x_2x_3 \);
- S1(\( x \)), S2(\( x \)), S3(\( x \)) are the blue words matching \( **x_4 \), \( **x_3x_4 \), \( **x_2x_3x_4 \);
- CL(\( x \)) hosts the blue words in \{\( x_1x_2x_3 \), \( x_1x_2x_3x_4 \), \( x_3x_4 \), \( x_2x_3x_4 \), \( x_4x_1x_2x_3 \)\}.

These seven lists begin respectively in MEM locations P1OFF + \( p_1(\alpha) \), P2OFF + \( p_2(\alpha) \),
P3OFF + \( p_3(\alpha) \), S1OFF + \( s_1(\alpha) \), S2OFF + \( s_2(\alpha) \), S3OFF + \( s_3(\alpha) \), and CLOFF + \( d(\alpha) \); here
(\( P1OFF, P2OFF, P3OFF, S1OFF, S2OFF, S3OFF, CLOFF \)) are respectively \( (2m^4, 5m^4, 8m^4, 11m^4, 14m^4, 17m^4, 20m^4) \). We define \( p_1(\( x_1x_2x_3x_4 \)) = (x_1)_m, \)
\( p_2(\( x_1x_2x_3x_4 \)) = (x_1x_2)_m, p_3(\( x_1x_2x_3x_4 \)) = (x_1x_2x_3)_m, s_1(\( x_1x_2x_3x_4 \)) =
(\( x_1x_2x_3 \), s_2(\( x_1x_2x_3x_4 \)) = (x_3x_4), s_3(\( x_1x_2x_3x_4 \)) = (x_2x_3x_4), \) and finally
d(\( x_1x_2x_3x_4 \)) is an internal number between 0 and \( (m^4 - m^2)/4 - 1 \) assigned

December 22, 2015
### Table 1

LISTS USED BY ALGORITHM C \((m = 2)\), ENTERING LEVEL 1

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RED</td>
<td>BLUE</td>
<td>BLUE</td>
<td>RED</td>
<td>BLUE</td>
<td>BLUE</td>
</tr>
<tr>
<td>10</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>0111</td>
<td>1100</td>
</tr>
<tr>
<td>20</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>0111</td>
<td>1100</td>
</tr>
<tr>
<td>50</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>0111</td>
<td>1100</td>
</tr>
<tr>
<td>80</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>0110</td>
<td>1110</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
</tr>
<tr>
<td>110</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>112</td>
<td>114</td>
<td>116</td>
<td>0001</td>
<td>0100</td>
<td>1101</td>
</tr>
<tr>
<td>140</td>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table shows MEM locations 0000 through 150f, using hexadecimal notation. (For example, MEM[40D] = 5e; see exercise 36.) Blank entries are unused by the algorithm.

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 - \(m^4 + \alpha\), \ldots, CLOFF - \(m^4 + \alpha\). And the TAIL pointers, which indicate the current list sizes as in \((19)-(23)\), are respectively kept in MEM locations P10FF + \(m^4 + \alpha\), P20FF + \(m^4 + \alpha\), \ldots, CLOFF + \(m^4 + \alpha\). (Whew; got that?)

This vast apparatus, which occupies 22\(m^4\) cells of MEM, is illustrated in Table 1, at the beginning of the computation for the case \(m = 2\). Fortunately it’s not really so complicated as it may seem at first. Nor is it especially vast: After all, 22\(m^4\) 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 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 pointer MEM[30] = 1f = 20 - 1 marks that list as closed. (Those list elements ac-

December 22, 2015
Table 2  
LISTS USED BY ALGORITHM C \((m = 2)\), ENTERING LEVEL 2

<table>
<thead>
<tr>
<th>RED</th>
<th>RED</th>
<th>RED</th>
<th>BLUE</th>
<th>RED</th>
<th>RED</th>
<th>RED</th>
<th>RED</th>
<th>RED</th>
<th>RED</th>
<th>BLUE</th>
<th>BLUE</th>
<th>BLUE</th>
<th>BLUE</th>
<th>BLUE</th>
<th>RED</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>0</td>
<td>1f</td>
<td>2f</td>
<td>3f</td>
<td>4f</td>
<td>5f</td>
<td>6f</td>
<td>7f</td>
<td>8f</td>
<td>9f</td>
<td>af</td>
<td>0b</td>
<td>1b</td>
<td>2b</td>
<td>3b</td>
<td>4b</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>0a</td>
<td>1a</td>
<td>2a</td>
<td>3a</td>
<td>4a</td>
<td>5a</td>
<td>6a</td>
</tr>
<tr>
<td>110</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>150</td>
<td>160</td>
<td>170</td>
<td>180</td>
<td>190</td>
<td>000</td>
<td>001</td>
<td>002</td>
<td>003</td>
<td>004</td>
<td>005</td>
<td>006</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 formerly appeared (see exercise 37).

Eventually 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 to get on with the search for commafree 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

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

Here \text{UNDO} is a sequential stack that holds (address, value) pairs; in our application we could say \(\text{UNDO}[u] \leftarrow (a \ll 16) + \text{MEM}[a]\), 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:

\[
\text{unstore}(u_0) : \text{While } u > u_0, \text{ set } u \leftarrow u - 1,
\]

\[
(a, v) \leftarrow \text{UNDO}[u], \text{ and } \text{MEM}[a] \leftarrow v. \tag{25}
\]

In our application the unstacking operation \(\langle a, v \rangle \leftarrow \text{UNDO}[u]\) here could be implemented by saying \(a \leftarrow \text{UNDO}[u] \gg 16, v \leftarrow \text{UNDO}[u] \& \#ffffff\).

A useful refinement of this reversible-memory 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{UNDO} stack when the same memory address is updated more than once in the same round.

\[
\text{store}(a, v) : \text{If } \text{STAMP}[a] \neq \sigma, \text{ set } \text{STAMP}[a] \leftarrow \sigma,
\]

\[
\text{UNDO}[u] \leftarrow (a, \text{MEM}[a]), \text{ and } u \leftarrow u + 1.
\]

Then set \text{MEM}[a] \leftarrow v. \tag{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 mems for the first assignment to \text{MEM}[a], but only two mems 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 know how 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

\[
(*x_1x_2x_3x_4**), (**x_1x_2x_3x_4***), \text{ and } (**x_1x_2x_3x_4**). \tag{27}
\]
If there's a green word on both sides of a poison list entry, we're dead: The commafree 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:

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

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, (**000, 10**), which remains poisonous because list **000 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 commafree 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 58 codewords are impossible depended on what Jiggs called a "quite time-consuming" computation.

Because size 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 [100] 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 commafree 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 commafree and include either 0001 or 0010. It uses an array MEM of \(M = \lceil 23.5m^4 \rceil\) 16-bit numbers, as well as several more auxiliary arrays: ALF of size \(16^3m\); STAMP of size \(M\); X, C, S, and \(U\) of size \(L + 1\); FREE and 1FREE of size \(L\); and a sufficiently large array called UNDO whose maximum size is difficult to guess.

**C1. Initialize.** Set \(ALF[(abcd)_{16}] \leftarrow (abcd)_m\) for \(0 \leq a, b, c, d < m\). Set \(STAMP[X] \leftarrow 0\) 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 $\text{MEM}[PP] \leftarrow \text{POISON}$, where \text{POISON} = 22m^4 and $PP = \text{POISON} - 1$. Set $\text{FREE}[k] \leftarrow 1$ for $0 \leq k < L$. Then set $l \leftarrow 1$, $x \leftarrow *0001$, $c \leftarrow 0$, $s \leftarrow L - g$, $f \leftarrow L$, $u \leftarrow 0$, 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.)

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

**C3.** [Try the candidate.] Set $U[l] \leftarrow u$ and $\sigma \leftarrow \sigma + 1$. If $x < 0$, go to C6 if $s = 0$ or $l = 1$, otherwise set $s \leftarrow s - 1$. If $x \geq 0$, update the data structures to make $x$ green, as described in exercise 40, escaping to C5 if trouble arises.

**C4.** [Make the move.] Set $X[l] \leftarrow x$, $C[l] \leftarrow c$, $S[l] \leftarrow s$, $p \leftarrow \text{FREE}[c]$, $f \leftarrow f - 1$. If $p \neq f$, set $y \leftarrow \text{FREE}[f]$, $\text{FREE}[y] \leftarrow p$, $\text{FREE}[f] \leftarrow c$, $\text{FREE}[c] \leftarrow f$. (This is (23).) Then set $l \leftarrow l + 1$ and go to C2.

**C5.** [Try again.] While $u > U[l]$, set $u \leftarrow u - 1$ and $\text{MEM}[\text{UNDO}[u]] \leftarrow \text{UNDO}[u] \& *fff$. (These operations restore the previous state, as in (25).) Then $\sigma \leftarrow \sigma + 1$ and redden $x$ (see exercise 40). Go to C2.

**C6.** [Backtrack.] Set $l \leftarrow l - 1$, and terminate if $l = 0$. Otherwise set $x \leftarrow X[l]$, $c \leftarrow C[l]$, $f \leftarrow f - 1$. If $x < 0$, repeat this step (class $c$ was omitted from the code). Otherwise set $s \leftarrow S[l]$ 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 megamens to prove that no solutions exist for $m = 4$ when $g$ is 60, 59, and 58. It needs about 22800 megamens 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.

**Running time estimates.** Backtrack programs are full of surprises. Sometimes they produce instant answers to a supposedly difficult problem. But sometimes they spin their wheels endlessly, trying to traverse an astronomically large search tree. And sometimes they deliver results just about as fast as we might expect.

Fortunately, we needed sit in the dark. There’s a simple Monte Carlo algorithm by which we can often tell in advance whether or not a given backtrack strategy will be feasible. This method, based on random sampling, can actually be worked out by hand before writing a program, in order to help decide whether to invest further time while following a particular approach. In fact, the very act of carrying out this pleasant pencil-and-paper method often suggests useful cutoff strategies and/or data structures that will be valuable later when a program is being written. For example, the author developed Algorithm C above after first doing some armchair experiments with random choices of potential commafree codewords, and noticing that a family of lists such as those in Tables 1 and 2 would be quite helpful when making further choices.

To illustrate the method, let’s consider the $n$ queens problem again, as represented in Algorithm B* above. When $n = 8$, we can obtain a decent “ballpark
7.2.2 BACKTRACK PROGRAMMING

Fig. 69. Four random attempts to solve the 8 queens problem. Such experiments help to estimate the size of the backtrack tree in Fig. 68. The branching degrees are shown at the right of each diagram, while the random bits used for sampling appear below. Cells have been shaded in gray if they are attacked by one or more queens in earlier rows.

estimate” of the size of Fig. 68 by examining only a few random paths in that search tree. We start by writing down the number \( D_1 \leftarrow 8 \), because there are eight ways to place the queen in row 1. (In other words, the root node of the search tree has degree 8.) Then we use a source of random numbers—say the binary digits of \( \pi \mod 8 = (0.001001000011...)_2 \)—to select one of those placements. Eight choices are possible, so we look at three of those bits; we shall set \( X_1 \leftarrow 2 \), because 001 is the second of the eight possibilities (000, 001, ..., 111).

Given \( X_1 = 2 \), the queen in row 2 can’t go into columns 1, 2, or 3. Hence five possibilities remain for \( X_2 \), and we write down \( D_2 \leftarrow 5 \). The next three bits of \( \pi \) lead us to set \( X_2 \leftarrow 5 \), since 5 is the second of the available columns (4, 5, 6, 7, 8) and 001 is the second value of (000, 001, ..., 100). If \( \pi \) had continued with 101 or 110 or 111 instead of 001, we would incidentally have used the “rejection method” of Section 3.4.1 and moved to the next three bits; see exercise 47.

Continuing in this way leads to \( D_3 \leftarrow 4 \), \( X_3 \leftarrow 1 \); then \( D_4 \leftarrow 3 \), \( X_4 \leftarrow 4 \). (Here we used the two bits 00 to select \( X_3 \), and the next two bits 00 to select \( X_4 \).) The remaining branches are forced: \( D_5 \leftarrow 1 \), \( X_5 \leftarrow 7 \); \( D_6 \leftarrow 1 \), \( X_6 \leftarrow 3 \); \( D_7 \leftarrow 1 \), \( X_7 \leftarrow 6 \); and we’re stuck when we reach level 8 and find \( D_8 \leftarrow 0 \).

These sequential random choices are depicted in Fig. 69(a), where we’ve used them to place each queen successively into an unshaded cell. Parts (b), (c), and (d) of Fig. 69 correspond in the same way to choices based on the binary digits of \( e \mod 1 \), \( \phi \mod 1 \), and \( \gamma \mod 1 \). Exactly 10 bits of \( \pi \), 20 bits of \( e \), 13 bits of \( \phi \), and 13 bits of \( \gamma \) were used to generate these examples.

In this discussion the notation \( D_k \) stands for a branching degree, not for a domain of values. We’ve used uppercase letters for the numbers \( D_1 \), \( X_1 \), \( D_2 \), etc., because those quantities are random variables. Once we’ve reached \( D_8 = 0 \) at some level, we’re ready to estimate the overall cost, by implicitly assuming that the path we’ve taken is representative of all root-to-leaf paths in the tree.

The cost of a backtrack program can be assessed by summing the individual amounts of time spent at each node of the search tree. Notice that every node on level \( l \) of that tree can be labeled uniquely by a sequence \( x_1 \ldots x_{l-1} \), which defines the path from the root to that node. Thus our goal is to estimate the sum of all \( c(x_1 \ldots x_{l-1}) \), where \( c(x_1 \ldots x_{l-1}) \) is the cost associated with node \( x_1 \ldots x_{l-1} \).

December 22, 2015
For example, the four queens problem is represented by the search tree (4), and its cost is the sum of 17 individual costs
\[ c() + c(1) + c(13) + c(14) + c(142) + c(2) + c(24) + \cdots + c(413) + c(42). \] (28)
If \( C(x_1 \ldots x_t) \) denotes the total cost of the subtree rooted at \( x_1 \ldots x_t \), then
\[ C(x_1 \ldots x_t) = c(x_1 \ldots x_t) + C(x_1 \ldots x_t x_{t+1}^{(1)}) + \cdots + C(x_1 \ldots x_t x_{t+1}^{(d)}) \] (29)
when the choices for \( x_{t+1} \) at node \( x_1 \ldots x_t \) are \( \{x_{t+1}^{(1)}, \ldots, x_{t+1}^{(d)}\} \). For instance in (4) we have \( C(1) = c(1) + C(13) + C(14) \); \( C(13) = c(13) \); and \( C() = c() + C(1) + C(2) + C(3) + C(4) \) is the overall cost (28).

In these terms a Monte Carlo estimate for \( C() \) is extremely easy to compute:

**Theorem E.** Given \( D_1, X_1, D_2, X_2, \ldots \) as above, the cost of backtracking is
\[ C() = E[c() + D_1(c(X_1) + D_2(c(X_1 X_2) + D_3(c(X_1 X_2 X_3) + \cdots)))]. \] (30)

**Proof.** Node \( x_1 \ldots x_t \), with branch degrees \( d_1, \ldots, d_t \) above it, is reached with probability \( 1/d_1 \ldots d_t \); so it contributes \( d_1 \ldots d_t c(x_1 \ldots x_t) / d_1 \ldots d_t = c(x_1 \ldots x_t) \) to the expected value in this formula.

For example, the tree (4) has six root-to-leaf paths, and they occur with respective probabilities \( 1/8, 1/8, 1/4, 1/4, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8 \). The first one contributes 1/8 times \( c() + 4c(1) + 2c(13) \), namely \( c() / 8 + c(1)/2 + c(13) \), to the expected value. The second contributes \( c() / 8 + c(1)/2 + c(14) + c(142) \); and so on.

A special case of Theorem E, with all \( c(x_1 \ldots x_t) = 1 \), tells us how to estimate the total size of the tree, which is often a crucial quantity:

**Corollary E.** The number of nodes in the search tree, given \( D_1, D_2, \ldots, \) is
\[ E(1 + D_1 + D_1 D_2 + \cdots) = E(1 + D_1 (1 + D_2 (1 + D_3 (1 + \cdots)))) \] (31)

For example, Fig. 69 gives us four estimates for the size of the tree in Fig. 68, using the numbers \( D_j \) at the right of each 8 \( \times \) 8 diagram. The estimate from Fig. 69(a) is \( 1 + 8 (1 + 5 (1 + 4 (1 + 3 (1 + 1 (1 + 1)))))) = 2129 \); and the other three are respectively 2689, 1489, 2039. None of them is extremely far from the true number, 2057, although we can't expect to be so lucky all the time.

The detailed study in exercise 51 shows that the estimate (31) in the case of 8 queens turns out to be quite well behaved:
\[ (\min 489, \ \text{ave} \ 2057, \ \max 7409, \ \text{dev} \ \sqrt{1148640} \approx 1071). \] (32)

The analogous problem for 16 queens has a much less homogeneous search tree:
\[ (\min 2597105, \ \text{ave} \ 11411900303, \ \max 131048318769, \ \text{dev} \ \approx 1234000000). \] (33)

Still, this standard deviation is roughly the same as the mean, so we'll usually guess the correct order of magnitude. (For example, ten independent experiments predicted .632, .866, .237, 1.027, 4.006, .982, .143, .140, 3.402, and .510 billion nodes, respectively. The mean of these is 1.195.) A thousand trials with \( n = 64 \) suggest that the problem of 64 queens will have about \( 3 \times 10^{65} \) nodes in its tree.
Let's formulate this estimation procedure precisely, so that it can be performed conveniently by machine as well as by hand:

Algorithm E (Estimated cost of backtrack). Given domains \( D_k \) and properties \( P_l \) as in Algorithm B, together with node costs \( c(x_1 \ldots x_l) \) as above, this algorithm computes the quantity \( S \) whose expected value is the total cost \( C() \) in (39). It uses an auxiliary array \( y_{l+2} \ldots \) whose size should be \( \geq \max(|D_1|, \ldots, |D_n|) \).

**E1. Initialize.** Set \( l \leftarrow D \leftarrow 1, S \leftarrow 0, \) and initialize any data structures needed.

**E2. Enter level \( l \).** (At this point \( P_{l-1}(X_1, \ldots, X_{l-1}) \) holds.) Set \( S \leftarrow S + D \cdot c(X_1 \ldots X_{l-1}) \). If \( l > n \), terminate the algorithm. Otherwise set \( d \leftarrow 0 \) and set \( x \leftarrow \min D_l \), the smallest element of \( D_l \).

**E3. Test \( x \).** If \( P_l(X_1, \ldots, X_{l-1}, x) \) holds, set \( y_d \leftarrow x \) and \( d \leftarrow d + 1 \).

**E4. Try again.** If \( x \neq \max D_l \), set \( x \) to the next larger element of \( D_l \) and return to step E3.

**E5. Choose and try.** If \( d = 0 \), terminate. Otherwise set \( D \leftarrow D \cdot d \) and \( X_l \leftarrow y_d \), where \( I \) is a uniformly random integer in \( \{0, \ldots, d - 1\} \). Update the data structures to facilitate testing \( P_{l+1} \), set \( l \leftarrow l + 1 \), and go back to E2. 

Although Algorithm E looks rather like Algorithm B, it never backtracks.

Of course we can't expect this algorithm to give decent estimates in cases where the backtrack tree is wildly erratic. The expected value of \( S \), namely \( E(S) \), is indeed the true cost; but the probable values of \( S \) might be quite different.

An extreme example of bad behavior occurs if property \( P_l \) is the simple condition \( x_1 > \cdots > x_l \) and all domains are \( \{1, \ldots, n\} \). Then there's only one solution, \( x_1 \ldots x_n = n \ldots 1 \); and backtracking is a particularly stupid way to find it!

The search tree for this somewhat ridiculous problem is, nevertheless, quite interesting. It is none other than the binomial tree \( T_n \) of Eq. 7.2.1.3-(21), which has \( \binom{n}{l} \) nodes on level \( l + 1 \) and \( 2^n \) nodes in total. If we set all costs to 1, the expected value of \( S \) is therefore \( 2^n = e^{nln2} \). But exercise 50 proves that \( S \) will almost always be much smaller, less than \( e^{(lnn)^2lnn} \). Furthermore the average value of \( l \) when Algorithm E terminates with respect to \( T_n \) is only \( H_n + 1 \). When \( n = 100 \), for example, the probability that \( l \geq 20 \) on termination is only 0.00000000027, while the vast majority of the nodes are near level 51.

Many refinements of Algorithm E are possible. For example, exercise 52 shows that the choices in step E5 need not be uniform. We shall discuss improved estimation techniques in Section 7.2.2.9, after having seen numerous examples of backtracking in practice.

*Estimating the number of solutions.* Sometimes we know that a problem has more solutions than we could ever hope to generate, yet we still want to know roughly how many there are. Algorithm E will tell us the approximate number, in cases where the backtrack process never reaches a dead end—that is, if it never terminates with \( d = 0 \) in step E5. There may be another criterion for successful termination in step E2 even though \( l \) might still be \( \leq n \). The expected final value of \( D \) is exactly the total number of solutions, because every solution \( X_1 \ldots X_l \) constructed by the algorithm is obtained with probability \( 1/D \).
For example, suppose we want to know the number of different paths by which a king can go from one corner of a chessboard to the opposite corner, without revisiting any square. One such path, chosen at random using the bits of \( \pi \) for guidance as we did in Fig. 60(a), is shown here. Starting in the upper left corner, we have 3 choices for the first move. Then, after moving to the right, there are 4 choices for the second move. And so on. We never make a move that would disconnect us from the goal; in particular, two of the moves are actually forced. (Exercise 58 explains one way to avoid fatal mistakes.)

The probability of obtaining this particular path is exactly \( \frac{1}{D} \), where \( D = 3 \times 4 \times 6 \times 6 \times 2 \times 6 \times 7 \times \ldots \times 2 = 1^2 \times 2^4 \times 3^6 \times 4^{10} \times 5^3 \times 6^6 \times 7^1 \approx 8.7 \times 10^{20} \). Thus, we can reasonably guess, at least tentatively, that there are \( 10^{22} \) such paths, more or less.

Of course that guess, based on a single random sample, rests on very shaky grounds. But we know that the average value \( M_N = (D^{(1)} + \ldots + D^{(N)}) / N \) of \( N \) guesses, in \( N \) independent experiments, will almost surely approach the correct number.

How large should \( N \) be, before we can have any confidence in the results? The actual values of \( D \) obtained from random king paths tend to vary all over the map. Figure 70 plots typical results, as \( N \) varies from 1 to 10000. For each value of \( N \) we can follow the advice of statistics textbooks and calculate the sample variance \( V_N = S_N^2 / (N - 1) \) as in Eq. 4.2.2-(16); then \( M_N \pm \sqrt{V_N/N} \) is the textbook estimate. The top diagram in Fig. 70 shows these “error bars” in gray, surrounding black dots for \( M_N \). This sequence \( M_N \) does appear to settle down after \( N \) reaches 3000 or so, and to approach a value near \( 5 \times 10^{25} \). That’s much higher than our first guess, but it has lots of evidence to back it up.

On the other hand, the bottom chart in Fig. 70 shows the distribution of the logarithms of the 10000 values of \( D \) that were used to make the top chart. Almost half of those values were totally negligible—less than \( 10^{20} \). About 75% of them were less than \( 10^{24} \). But some of them exceeded \( 10^{28} \). Can we really rely on a result that’s based on such chaotic behavior? Is it really right to throw away most of our data and to trust almost entirely on observations that were obtained from comparatively few rare events?

Yes, we’re okay! Some of the justification appears in exercise MPR–124, which is based on theoretical work by P. Diaconis and S. Chatterjee. In the paper cited with that exercise, they defend a simple measure of quality,

\[
Q_N = \max(D^{(1)}, \ldots, D^{(N)}) / (N M_N) = \frac{\max(D^{(1)}, \ldots, D^{(N)})}{D^{(1)} + \ldots + D^{(N)}},
\]

\[
Q_N = \frac{\max(D^{(1)}, \ldots, D^{(N)})}{D^{(1)} + \ldots + D^{(N)}}.
\]

* Four of the actual values that led to Fig. 70 were larger than \( 10^{28} \); the largest, \( \approx 2.1 \times 10^{30} \), came from a path of length 57. The smallest estimate, \( 13891644 \), came from a path of length 10.
arguing that a reasonable policy in most experiments such as these is to stop sampling when $Q_N$ gets small. (Values of this statistic $Q_N$ have been plotted in the middle of Fig. 70.)

Furthermore we can estimate other properties of the solutions to a backtrack problem, instead of merely counting those solutions. For example, the expected value of $ID$ on termination of the random king's path algorithm is the total length of such paths. The data underlying Fig. 70 suggests that this total is $(2.66 \pm 1.4) \times 10^{27}$; hence the average path length appears to be about 53. The samples also indicate that about 34% of the paths pass through the center; about 46% touch the upper right corner; about 22% touch both corners; and about 7% pass through the center and both corners.

For this particular problem we don’t actually need to rely on estimates, because the ZDD technology of Section 7.1.4 allows us to compute the true values. (See exercise 59.) The total number of simple corner-to-corner king paths on a chessboard is exactly 50,819,542,770,311,581,600,906,543; this value lies almost within the error bars of Fig. 70 for all $N \geq 230$, except for a brief interval near $N = 1400$. And the total length of all these paths turns out to be exactly 2,700,911,171,651,251,701,712,099,831, which is a little higher than our estimate. The true average length is therefore $\approx 53.15$. The true probabilities of hitting the center, a given corner, both corners, and all three of those spots are respectively about $38.96\%$, $50.32\%$, $25.32\%$, and $9.86\%$.

The total number of corner-to-corner king paths of the maximum length, 63, is 2,811,002,302,704,446,906,926. This is a number that can not be estimated well by a method such as Algorithm E without additional heuristics.

The analogous problem for corner-to-corner knight paths, of any length, lies a bit beyond ZDD technology because many more ZDD nodes are needed. Using Algorithm E we can estimate that there are about $(8.6 \pm 1.2) \times 10^{19}$ such paths.
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.

Backtrack programs typically traverse the tree of possibilities by using what is now called depth-first search, a general graph exploration procedure that Édouard Lucas credited to a student named Trémaux [Récitations Mathématiques 1 (Paris; Gauthier-Villars, 1882), 47–50].

The eight queens problem was first proposed by Max Bezzel [Sächsische Zeitung 3 (1848), 303; 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 ‘Tatominren’, 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, 16837425, 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 backtracking to be described as a general technique, and Robert J. Walker wrote 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], who formulated the general problem carefully and presented a variety of examples. In particular, they discussed the search for maximum commafree codes, and noted that backtracking can be used to find successively better and better solutions to combinatorial optimization problems. They introduced certain kinds of lookahead, as well as the important idea of dynamic ordering by branching on variables with the fewest remaining choices.


Monte Carlo estimates of the cost of backtracking were first described briefly by M. Hall, Jr., and D. E. Knuth in Computers and Computing, AMM 72, 2, part 2, Slaught Memorial Papers No. 10 (February 1965), 21–28. Knuth gave a much more detailed exposition a decade later, in Math. Comp. 29 (1975), 121–136. Such methods can be considered as special cases of so-called “importance sampling”; see J. M. Hammersley and D. C. Handscomb, Monte Carlo Methods.
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 backtrack programming, by presenting suitable domains $D_k$ and cutoff properties $P_k(x_1, \ldots, x_k)$ that satisfy (1) and (2).

2. [10] True or false: We can choose $D_1$ so that $P_1(x_1)$ is always true.

3. [16] Using a chessboard and eight coins to represent queens, one can follow the steps of 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 $try(l)$, having global variables $n$ and $x_1, \ldots, x_n$, to be invoked by saying ‘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, 1990.) Are there any values $n > 5$ for which the $n$ queens problem has a “framed” solution with $x_1 = 2, x_2 = n, x_{n-1} = 1$, and $x_n = n - 1$?

7. [20] Are there two $8$-queen placements with the same $x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8$?

8. [21] Can a $4n$-queen placement have $3n$ queens on “white” squares?

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

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

   a) Can the $n$ queens problem have a solution with reflection symmetry?
   b) Show that chiral symmetry is impossible when $n \mod 4 \not\in \{2, 3\}.
   c) Sometimes the solution to an $n$ queens 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 $x_i \equiv k \mod n$ for $1 \leq k \leq n/2$. Prove that $C_n = 2^{n/4} c_n$.

11. [M28] (Wraparound queens.) Replace (g) by the stronger conditions $x_j \not\equiv x_i$, $(x_i - x_j) \mod n \not\equiv k - j$, $(x_i - x_j) \mod n \not\equiv k - j$. (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 $n$ queens problem have at least one solution?

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

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

15. [21] Let $H(n)$ be the number of ways that $n$ queen bees can occupy an $n \times n$ 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 $x_k < 0$’ to ‘while $x_k \neq 0$’, because $x_k$ 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 that $n = 4$ and Algorithm I has reached step L2 with $l = 4$ and $x_1x_2x_3 = 241$. What are the current values of $x_1x_2x_3x_4$, $p_0p_1p_2p_3p_4$, and $y_1y_2y_3$?

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

20. [21] Extend Algorithm I so that it forces $x_i \leftarrow k$ whenever $k \notin \{x_1, \ldots, x_{i-1}\}$.

21. [M25] If $x = x_1x_2 \ldots x_{2n}$, let $x^D = (-x_2) \ldots (-x_{2n})(-x_1) = -x^B$ be its dual.
   a) Show that if $n$ is odd and $x$ solves Langford’s problem (7), we have $x_i = n$ for some $k \leq [n/2]$ if and only if $x^D_i = n$ for some $k > [n/2]$.
   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.


23. [17] We can often obtain one word rectangle from another by changing only a 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. [23] The method of exercises 24 and 25 does $n$ levels of backtracking to fill the cells of an $m \times n$ rectangle 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?

   aced bade bear beef cafe code dada 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. [M30] (W. L. Eastman, 1955.) 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 $x_{i+1}x_i \ldots x_0$ for $0 < k < n$, the procedure outputs a cyclic shift $x_k$ with the property that the set of all such $x_k$ is commafree.

   We regard $x$ as an infinite periodic sequence $\langle x_k \rangle$ with $x_k = x_{k+n}$ for all $k \geq n$. Each cyclic shift then has the form $x_kx_{k+1} \ldots x_{k+n-1}$. The simplest nontrivial example occurs when $n = 3$, where $x = x_0x_1x_2x_0x_3x_2x_0 \ldots$ and we don’t have $x_0 = x_1 = x_2$. In this case the algorithm outputs $x_kx_{k+1}x_{k+2}$ where $x_k > x_{k+1} \leq x_{k+2}$; and the set of all such triples clearly satisfies the commafree condition.
One key idea is to think of $x$ as partitioned into $t$ substrings by boundary markers $b_j$, where $0 \leq b_0 < b_1 < \cdots < b_{t-1} < n$ and $b_j = b_{j-1} + n$ for $j \geq t$. Then substring $y_j$ is $x_{b_j}x_{b_j+1}\ldots x_{b_{j+1}-1}$. 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 those substrings have the same length, we use lexicographic comparison; otherwise we declare that the longer substring is bigger.

The second key idea is the notion of “dips,” which are substrings of the form $z = z_1\ldots z_k$ where $k \geq 2$ and $z_1 \geq \cdots \geq z_{k-1} < z_k$. It’s easy to see that any string $y = y_0y_1\ldots$ in which we have $y_i < y_{i+1}$ for infinitely many $i$ can be factored into a sequence of dips, $y = z_0z_1\ldots$, and this factorization is unique. For example,

$$314159265358979323846263338 \ldots = 314 15 9 26 535 89 79 323 846 26 3338 3 \ldots .$$

Furthermore, if $y$ is a periodic sequence, its factorization into dips is also ultimately periodic, although some of the initial factors may not occur in the period. For example,

$$1234435501234435501234435501234435 \ldots = 12 34 435 501 23 4435 501 23 4435 \ldots .$$

Given a periodic, nonconstant sequence $y$ described by boundary markers as above, where the period length $t$ is odd, its periodic factorization will contain an odd number of odd-length dips. Each round of Eastman’s algorithm simply retains the boundary points of the left of those odd-length dips. Then $t$ is reset to the number of retained boundary points, and another round begins if $t > 1$.

a) Play through the algorithm by hand when $n = 19$ and $x = 3141592653589793238$.
b) Show that the number of rounds is at most $\lceil \log_4 n \rceil$.
c) Exhibit a binary $x$ that achieves this worst-case bound when $n = 3^4$.
d) Implement the algorithm with full details. (It’s surprisingly short!)
e) Explain why the algorithm yields a common factor.

33. [HM88] 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 common factor code of length $(m^4 - m^2)/4$ contain 0001 and 2000?

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) $\text{MEM}[404] = 58$ and (b) $\text{MEM}[904] = 84$ in Table 1?

37. [18] Why is (a) $\text{MEM}[16] = 87$ and (b) $\text{MEM}[a0d] = 88$ 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 $\text{MEM}$, and use the following selection strategy:

a) Find a class $c$ with the least number $r$ of blue words.
b) If $r = 0$, set $x \leftarrow -1$; otherwise set $x$ to a word in class $c$.
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 30, spell out the details of step C3 when $x \geq 0$.

a) What updates should be done to $\text{MEM}$ when a blue word $x$ becomes red?
b) What updates should be done to $\text{MEM}$ 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 \((m = 2)\) commafree code with one codeword in each of the \(\left(\sum_{\nu \in \Sigma} \mu(d) 2^{m - d}\right)/n\) cycle classes, for every word length \(n\)?

43. [HM] 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. [M22] Find a maximum-size commafree code of length \(4\) on \(m = 5\) letters.

47. [20] Explain how the choices in Fig. 69 were determined from the “random” bits that are displayed. For instance, why was \(X_2\) set to 1 in Fig. 69(b)?

48. [M15] Interpret the value \(E(D_1 \ldots D_k)\), in the text’s Monte Carlo algorithm.

49. [M22] What’s a simple commafree that corresponds to Theorem E?

50. [HM25] Elmo uses Algorithm E with \(D = \{1, \ldots, n\}, P_t = [x_1 > \ldots > x_t]\), \(c = 1\).

- Alice flips \(n\) coins independently, where coin \(k\) yields “heads” with probability \(1/k\).
- True or false: She obtains exactly \(I\) heads with probability \(\binom{n}{I}/n!\).
- Let \(Y_1, Y_2, \ldots, Y_I\) be the numbers on the coins that come up heads. (Thus \(Y_1 = 1\), and \(Y_2 = 2\) with probability \(1/2\).) Show that \(Pr(Alice\text{ obtains } Y_1, Y_2, \ldots, Y_I) = Pr(Elmo\text{ obtains } X_1 = Y_1, X_2 = Y_{i-1}, \ldots, X_I = Y_I)\).
- Prove that Alice q.s. obtains at most \((\ln n)/(\ln \ln n)\) heads.
- Consequently Elmo’s \(S\) is q.s. less than \(\exp((\ln n)^{1/(\ln \ln n)}))\).

51. [M30] Extend Algorithm B so that it also computes the minimum, maximum, mean, and variance of the Monte Carlo estimates \(S\) produced by Algorithm E.

52. [M24] Instead of choosing each \(y_i\) in step E5 with probability \(1/d\), we could use a biased distribution where \(Pr(I = i) = \frac{p_{x_1, \ldots, x_{i-1}}(y_i)}{p_{x_1, \ldots, x_I}} > 0\). How should the estimate \(S\) be modified so that its expected value in this general scheme is still \(C^I()\)?

53. [M20] If all costs \(c(x_1, \ldots, x_I)\) are positive, show that the biased probabilities of exercise 52 can be chosen in such a way that the estimate \(S\) is always exact.

55. [M25] The commafree code search procedure in Algorithm C doesn’t actually fit the mold of Algorithm E, because it incorporates lookahead, dynamic ordering, reversible memory, and other enhancements to the basic backtrack paradigm. How could its running time be reliably estimated with Monte Carlo methods?

57. [M30] Algorithm E can potentially follow \(M\) different paths \(X_1, \ldots, X_{I_{\ell}}\) before it terminates, where \(M\) is the number of leaves of the backtrack tree. Suppose the final values of \(D\) at those leaves are \(D^{(1)}, \ldots, D^{(M)}\). Prove that \(\left(D^{(1)} \ldots D^{(M)}\right)^{1/M} \geq M\).

58. [27] The text’s king path problem is a special case of the general problem of counting simple paths from vertex \(s\) to vertex \(t\) in a given graph.

We can generate such paths by random walks from \(s\) that don’t get stuck, if we maintain a table of values \(DIST(v)\) for all vertices \(v\) not yet in the path, representing the shortest distance from \(v\) to \(t\) through unused vertices. For with such a table we can simply move at each step to a vertex for which \(DIST(v) < \infty\).

Devise a way to update the \(DIST\) table dynamically without unnecessary work.

59. [26] A ZDD with 3,174,197 nodes can be constructed for the family of all simple corner-to-corner king paths on a chessboard, using the method of exercise 7.1.4–225. Explain how to use this ZDD to compute (a) the total length of all paths; (b) the number of paths that touch any given subset of the center and/or corner points.

December 22, 2015
60. [20] Experiment with biased random walks (see exercise 52), weighting each non-dead-end king move to a new vertex \( v \) by \( 1 + \text{DIST}(v)^2 \) instead of choosing every such move with the same probability. Does this strategy improve on Fig. 70?

<table>
<thead>
<tr>
<th>Table 666</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWENTY QUESTIONS (SEE EXERCISE 90)</td>
</tr>
<tr>
<td>1. The first question whose answer is A is:</td>
</tr>
<tr>
<td>(A) 1    (B) 2    (C) 3    (D) 4    (E) 5</td>
</tr>
<tr>
<td>2. The next question with the same answer as this one is:</td>
</tr>
<tr>
<td>(A) 4    (B) 6    (C) 8    (D) 10   (E) 12</td>
</tr>
<tr>
<td>3. The only two consecutive questions with identical answers are questions:</td>
</tr>
<tr>
<td>(A) 15 and 16 (B) 16 and 17 (C) 17 and 18 (D) 18 and 19 (E) 19 and 20</td>
</tr>
<tr>
<td>4. The answer to this question is the same as the answers to questions:</td>
</tr>
<tr>
<td>(A) 10 and 13 (B) 14 and 16 (C) 7 and 20 (D) 1 and 15 (E) 8 and 12</td>
</tr>
<tr>
<td>5. The answer to question 14 is:</td>
</tr>
<tr>
<td>(A) B    (B) E    (C) C    (D) A    (E) D</td>
</tr>
<tr>
<td>6. The answer to this question is:</td>
</tr>
<tr>
<td>(A) A    (B) B    (C) C    (D) D    (E) None of those</td>
</tr>
<tr>
<td>7. An answer that appears most often is:</td>
</tr>
<tr>
<td>(A) A    (B) B    (C) C    (D) D    (E) E</td>
</tr>
<tr>
<td>8. Ignoring answers that appear equally often, the least common answer is:</td>
</tr>
<tr>
<td>(A) A    (B) B    (C) C    (D) D    (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>
</tr>
<tr>
<td>(A) [50 \ldots 62] (B) [52 \ldots 55] (C) [44 \ldots 49] (D) [81 \ldots 67] (E) [44 \ldots 53]</td>
</tr>
<tr>
<td>10. The answer to question 17 is:</td>
</tr>
<tr>
<td>(A) D    (B) B    (C) A    (D) E    (E) Wrong</td>
</tr>
<tr>
<td>11. The number of questions whose answer is D is:</td>
</tr>
<tr>
<td>(A) 2    (B) 3    (C) 4    (D) 5    (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>
</tr>
<tr>
<td>(A) B    (B) C    (C) D    (D) E    (E) None of those</td>
</tr>
<tr>
<td>13. The number of questions whose answer is E is:</td>
</tr>
<tr>
<td>(A) 5    (B) 4    (C) 3    (D) 2    (E) 1</td>
</tr>
<tr>
<td>14. No answer appears exactly this many times:</td>
</tr>
<tr>
<td>(A) 2    (B) 3    (C) 4    (D) 5    (E) None of those</td>
</tr>
<tr>
<td>15. The set of odd-numbered questions with answer A is:</td>
</tr>
<tr>
<td>(A) {7} (B) {9} (C) {not} {11} (D) {13} (E) {15}</td>
</tr>
<tr>
<td>16. The answer to question 8 is the same as the answer to question:</td>
</tr>
<tr>
<td>(A) 3    (B) 2    (C) 13   (D) 18   (E) 20</td>
</tr>
<tr>
<td>17. The answer to question 10 is:</td>
</tr>
<tr>
<td>(A) C    (B) D    (C) B    (D) A    (E) Correct</td>
</tr>
<tr>
<td>18. The number of prime-numbered questions whose answers are vowels is:</td>
</tr>
<tr>
<td>(A) prime (B) square (C) odd (D) even (E) Zero</td>
</tr>
<tr>
<td>19. The last question whose answer is B is:</td>
</tr>
<tr>
<td>(A) 14   (B) 15   (C) 16   (D) 17   (E) 18</td>
</tr>
<tr>
<td>20. The maximum score that can be achieved on this test is:</td>
</tr>
<tr>
<td>(A) 18   (B) 19   (C) 20   (D) Indeterminate (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.

\[\text{December 22, 2015}\]
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 \( \not\in [30 \ldots 43] \); 15(C) becomes \( \{11\} \).

95. [HM26] Let \( P_n \) be the number of integer sequences \( x_1 \ldots x_n \) such that \( x_1 = 1 \) and \( 1 \leq x_{k+1} \leq 2x_k \) for \( 1 \leq k < n \). (The first few values are 1, 2, 6, 26, 166, 1626, \ldots; this sequence was introduced by A. Cayley in Philosophical Magazine (4) 13 (1857), 245–248, who showed that \( P_n \) enumerates the partitions of \( 2^n - 1 \) into powers of 2.)

a) Show that \( P_n \) is the number of different profiles that are possible for a binary tree of height \( n \).

b) Find an efficient way to compute \( P_n \) for large \( n \). \textit{Hint:} Consider the more general sequence \( P_n^{(m)} \), defined similarly but with \( x_1 = m \).

c) Use the estimation procedure of Theorem E to prove that \( P_n \geq 2^{(n)}/(n - 1)! \).

999. [M00] this is a temporary exercise (for dummies)
SECTION 7.2.2

1. Although many formulations 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 $P_1 = \lnot x_j \neq x_i$ for $1 \leq j < k \leq l$. (iii) For combinations of $n$ things from $N$, $D_k = \{1, \ldots, N + 1 - k\}$ and $P_1 = x_1 < \cdots < x_l$. (iv) $D_k = \{0, 1, \ldots, [n/k]\}$; $P_1 = x_1 \geq \cdots \geq x_n$ and $n = (n - l) x_1 + \cdots + x_l \leq n!$. (v) For restricted growth strings, $D_k = \{0, \ldots, k - 1\}$ and $P_1 = x_{j+1} \leq 1 + \max(x_1, \ldots, x_j)$ for $1 \leq j < l$. (vi) For the indices of left parentheses (see 7.2.16–(8)), $D_k = \{1, \ldots, 2k - 1\}$ and $P_1 = x_1 < \cdots < x_l$.

2. True. (If not, set $D_1 \leftarrow D_1 \cap \{x \mid P_l(x)\}$.)

3. We can restrict $D_1$ to $\{1, 2, 3, 4\}$, because the reflection (9–$x_1$) … (9–$x_6$) of every solution $x_1 \ldots x_6$ 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. $try(l) = \{l \mid l > n, \text{visit } x_1 \ldots x_n. \text{ Otherwise, for } x_l \leftarrow \min D_l, \min D_l + 1, \ldots, \max D_l, \text{ if } P_l(x_1, \ldots, x_l) \text{ call } try(l + 1).\}$

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 whose 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 $k = l$ from (3) yields respectively (312, 396, 430, 438, 458, 430, 396, 312) solutions. (With column $r$ also omitted there are just (40, 46, 42, 80, 80, 46, 42, 40, 46).)

6. Yes, probably for all $n \geq 17$. One such is $x_1, x_2, \ldots, x_{17} = 2, 1, 7, 12, 10, 7, 11, 14, 3, 5, 9, 13, 15, 4, 11, 6, 1, 16$. [See Proc. Edinburgh Math. Soc. 8 (1890). 43 and Fig. 52.]

7. Yes: (42736815, 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 $7, 10, 13, 20, 17, 24, 3, 6, 23, 11, 16, 21, 4, 9, 14, 2, 19, 21, 1, 8, 5, 12, 15$ 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 $\sum_{i=1}^{n} (x_i + k)$ is even.

9. Let bit vectors $a_i, b_i, c_i$ represent the “useful” elements of the sets in (6), with $a_i = \sum \{2^{i-1} \mid x \in A_i\}, b_i = \sum \{2^{i-1} \mid x \in B_i \cap [1 \ldots n]\}, c_i = \sum \{2^{i-1} \mid x \in C_i \cap [1 \ldots n]\}$. Then step W2 sets $a_i \leftarrow \mu \& \tilde{a}_i \& \tilde{b}_i$, where $\mu$ is the mask $2^n - 1$. In step W3 we can set $t \leftarrow s_0 \& \neg \eta_i, a_i \leftarrow a_i - t, b_i \leftarrow (b_i - t) \gg 1, c_i \leftarrow (\eta_0 + \mu \& \& \neg \mu)$; and it’s also convenient to set $s_i \leftarrow s_i - t$ at this time, instead of deferring this change to step W4.

(There’s no need to store $x_t$ in memory, or even to compute $x_t$ in step W3 as an integer in $[1 \ldots n]$, because $x_t$ can be deduced from $a_t - a_{t-1}$ 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 $c_0$, we can independently tilt (or not) each of the $[n/4]$ groups of four. [Mathematische Unterhaltungen und Spiele 1, second edition (Leipzig: Teubner, 1910), 249–258.]
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 $\sum_{i=1}^{n} (x_i + k) = \sum_{i=1}^{n} (x_i - k)^2 \equiv \sum_{i=1}^{n} k^2 = n(n+1)(2n+1)/6$. And we also have $\sum_{i=1}^{n} (x_i + k)^2 + \sum_{i=1}^{n} (x_i - k)^2 = 4n(n+1)(2n+1)/6 \equiv 2n/3$, 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 $x_i = n$ and $x_i = (2k) \mod n$ for $1 \leq k < n$. (The rule $x_i = (3k) \mod n$ also works. See Édouard Lucas, *Récréations Mathématiques* 1 (1882), 84–86.)

Another nice solution was found by J. Franel [L’*Intérimaire 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 $(1 \text{ or } 3) (k - j) + (0 \text{ or } 3) \equiv 0 \pmod{n}$; hence $k - j = n - 1 \text{ or } 3$. 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.

[A more complicated construction for all $n > 3$ had been given earlier by E. Pauls, 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 $X_{i=(i-1)/n+j} = (x_i^{(j)} - 1) n + y_j$ (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 *converging*.]

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 just $x_i - x_j$ in (3); so we can ignore the $b$ vector in Algorithm B* or in exercise 9. We get

\[
\begin{align*}
n &= 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 \\
H(n) &= 1 1 1 3 7 23 83 405 2113 12657 82927 596483 4698685 40071743 367854835 \\
\end{align*}
\]


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

17. $x_4 \ldots x_8 = \bar{2} \bar{1} 0 \bar{1} 0, p_1 \ldots p_5 \equiv 33300$, 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 $y_4 \ldots y_6$ 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 $D_t = \{-n, -n-1, -n-2, \ldots, n\}$, or $\{k \mid k \neq 0 \text{ and } 2 \leq l \leq k \leq 2n - l - 1\}$, 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 \"k has appeared,\" as in Algorithm B*. It's 0...0 in step L1; we set $a_i \leftarrow 1$ in step L3, $a_i \leftarrow 0$ in step L5.

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

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

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

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

22. The solutions \(1\) and \(2112\) for $n = 1$ and $n = 2$ are self-dual; the solutions for $n = 4$ and $n = 5$ are 4312134, 241351435, 451324353, 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, 75678864, \ldots; there are none when $n \mod 4 = 3$, by a parity argument.

Algorithm 1 needs only obvious changes. To compute solutions by a streamlined method like exercise 21, use $n' = n - [n \text{ is even}]$ and substitute $l \leq [n/2] + (0, 1, 2, 1)'$ for $l \leq [n/2]'$, when $n \mod 4 = (0, 1, 2, 3)$; also replace $l \geq n - 1$ and $a_{2n-l-1} = 0$' by $l \geq [n/2]'$ and $a_{1(2n-2-x)/3} = 0'$. The case $n = 15$ is proved impossible with 397 million nodes and 9.59 gigabytes.

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

24. Build the list of 5-letter words and the trie of 6-letter words in step B1; also set $a_{03}a_{030403040} = 00000$. Use $D_5 = 1$ in step B2 and max $D_5 = 5577$ in step B4. Testing $P_5$ in step B3, if word $x_5$ is $c_4c_5c_6c_7c_8$, consists of forming $a_{11} \ldots a_{15}$, where 
\[a_{15} = \text{trie}[a_{2n-13}, c_8] \text{ for } 1 \leq k \leq 5; \text{ but jump to B4 if any } a_{15k} \text{ is zero.}
\]

25. There are $5 \times 26$ singly linked lists, accessed from pointers $b_{15k}$, all initially zero.

The zth word $c_2c_3c_4c_5c_6c_7c_8$, for $1 \leq z \leq 5577$, belongs to 5 lists and has five pointers $L_{15}c_2L_{15}c_3L_{15}c_4L_{15}c_5L_{15}c_6L_{15}c_7L_{15}c_8$. To insert it, set $L_{15k} \leftarrow b_{15k}$, $b_{15k} \leftarrow x$, and $a_{15z} \leftarrow a_{15z} + 1$, for $1 \leq k \leq 5$. (Thus $a_{15}$ will be the length of the list accessed from $b_{15k}$.)

We can store a \"signature\" $\sum_{i=1}^{26} \; x^{i-1} \text{[trie}[a, c] \neq 0]\) with each node $a$ of the trie. For example, the signature for node 390 is $2^6 + 2^5 + 2^3 + 2^0$, according to (11); here $a = 1, \ldots, z = 26$.

The process of running through all $x$ that match a given signature $y$ with respect to position $z$, as needed in steps B2 and B4, now takes the following form: (i) Set $i \leftarrow 0$. (ii) While $2^i \times y = 0$, set $i \leftarrow i + 1$. (iii) Set $x \leftarrow b_{15(i+1)}$; go to (iv) if $x = 0$. (iv) Visit $x$. (v) Set $x \leftarrow L_{15x}$; go to (iv) if $x \neq 0$. (vi) Set $i \leftarrow i + 1$; go to (ii) if $2^i \leq y$.

Let $\text{trie}[a, 0]$ be the signature of node $a$. We choose $z$ and $y = \text{trie}[a_{2n-13}, 0]$ in step B2 so that the number of nodes to visit, $\sum_{i=1}^{26} a_{15k}[2^{i-1} \times y = 0]$, is minimum for $1 \leq z \leq 5$. For example, when $l = 3, x_1 = 1446$, and $x_2 = 185$, as in (23), that sum for $z = 1$ is $s_{15} = 512 + 819 + 819 + 819 + 819 + 819 + 1286 + 1286 + 1286 = 997$ and the sums for $z = 2, 3, 4, 5$ are 4722, 1370, 5057, and 1646. Hence we choose $z = 1$ and $y = *11241111; only 997 words, not 5577, need be tested for $x_3$.

The values $y_{15}$ and $z_{15}$ are maintained for use in backtracking. (In practice we keep $x$, $y$, and $z$ in registers during most of the computation. Then we set $x_{15} \leftarrow x, y_{15} \leftarrow y, z_{15} \leftarrow z$ before increasing $l \leftarrow l + 1$ in step B3; and we set $x \leftarrow x_{15}, y \leftarrow y_{15}, z \leftarrow z_{15}$ in
26. Here are the author’s favorite 5 × 7 and 5 × 8, and the only 5 × 9’s:

- smashes
- grandest
- pastelist
- varisters
- partial
- renounce
- accident
- gentivial
- immense
- episodes
- mortgagor
- coelomate
- emerged
- basement
- preoreform
- undeleted
- sadness
- eyesores
- andesynes
- oysterers

No 5 × 10 word rectangles exist, according to our ground rules.

27. (1.15727, 8072679, 630067290, 90062081, 625415) and (15727.0, 43216, 1749.7, 450.4, 286.0). Total time ≈ 18.3 teramemps. (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 compressed representation that omits the zeros. For example, the compressed representation of the node for prefix ‘corn’ in (22) consists of five consecutively stored pairs of entries (‘a’, 3879), (‘d’, 3878), (‘t’, 9602), (‘x’, 171), (‘z’, 013), 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 l corresponds to row \( i = 1 + (l - 1) \mod m \) and column \( j = 1 + \lfloor (l - 1)/m \rfloor \).

For backtracking we store the n-trie pointer \( a_{i,j} \), as before, together with an index \( x \) into the compressed m-trie.

This method was suggested by Bernard Gattengo in 1996 (unpublished). It finds all 5 × 6 word rectangles in just 400 gigamemps; and its running time for “transposed” 6 × 5 rectangles turns out to be slightly less (380 gigamemps). Notice that only one men is needed to access each (character, link) pair in the compressed trie.

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

30. Keep tables \( p_i, p_{i,j}, p_{i,j,k}, s_i, s_{i,j}, s_{i,j,k} \), 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 \( w \) = abcd, where \( 0 < a, b, c, d < m \), set \( x_g \leftarrow abcd \) and also do the following: Set \( p_a \leftarrow p_a + 1 \), \( p_{a,b} \leftarrow p_{a,b} + 1 \), \( p_{a,b,c} \leftarrow p_{a,b,c} + 1 \), \( s_a \leftarrow s_a + 1 \), \( s_{a,b} \leftarrow s_{a,b} + 1 \), \( s_{a,b,c} \leftarrow s_{a,b,c} + 1 \), where \( x + y = \min(2, x + y) \) denotes saturating ternary addition. Then if \( s_{a,b,c} + p_{a,b,c} = s_{a,b} + p_{a,b} + s_{a} + p_{a} = 0 \) for all \( x \in a'B'c'd' \), where \( 0 \leq k \leq g \), set \( g \leftarrow g + 1 \). Otherwise reject \( w \) and set \( p_a \leftarrow p_a - 1 \), \( p_{a,b} \leftarrow p_{a,b} - 1 \), \( p_{a,b,c} \leftarrow p_{a,b,c} - 1 \), \( s_a \leftarrow s_a - 1 \), \( s_{a,b} \leftarrow s_{a,b} - 1 \), \( s_{a,b,c} \leftarrow s_{a,b,c} - 1 \).

31. (a) The word be appears in message abed if and only if \( a \to b, b \to c, \) and \( c \to d \).

(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 \( \sum_{0 \leq i,j,c,r} p_i p_j p_k \) subject to \( p_0 + p_1 + \ldots + p_{r-1} = m \). The values \( p_j = \left[ (m+j)/r \right] \) achieve this (see exercise 7.2.1.4-61(a)). When \( r = 3 \) the maximum simplifies to \( [m^2/3] \).

32. (a) The factors of the period, 15926, 535 897 923 275 13414, begin at the respective boundary points 3, 5, 8, 11, 13, 15, 18 (and then \( 3 + 19 = 22 \), etc.). Thus round 1 retains boundaries 5, 8, and 15. The second-round substrings \( y_0 = 926, y_1 = 5358979, y_2 = 323831415 \) have different lengths, so lexicographic comparison is unnecessary; the answer is \( y_2 y_0 y_1 = x_{15} \ldots x_{33} \).

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

December 22, 2015
(c) Let \( a_0 = 0, b_0 = 1, a_{n+1} = a_n a_c b_{n+1}, b_{n+1} = a_n b_c b_n; \) use \( a_c \) or \( b_c \) when \( n = \text{odd}. \)

(d) We use an auxiliary subroutine ‘less(i)’, which returns \([y_{i+1} < y_i]\), given \( i > 0; \)

If \( b_{i} - b_{i+1} = b_{i+1} - b_i \), return \([b_i - b_{i+1} < b_{i+1} - b_i]\). Otherwise, for \( j = 0, 1, \ldots \), while

\[b_i + j < b_{i+1}, \text{ if } x_{b_{i+1}} \neq x_{b_i + j} \text{ return } [x_{b_{i+1}} + j < x_{b_i + j}]. \]

Otherwise return 0.

The tricky part of the algorithm is to discard initial factors that aren’t periodic.

The secret is to let \( i_0 \) be the smallest index such that \( y_{i-3} \geq y_{i-2} < y_{i-1}; \) then we can be sure that a factor begins with \( y_i. \)

1. [Initialize.] Set \( x_j \leftarrow x_{j-n} \) for \( n \leq j < 2n, \) \( b_j \leftarrow j \) for \( 0 \leq j < 2n, \) and \( t \leftarrow n. \)

2. [Begin a round.] Set \( t' \leftarrow 0. \) Find the smallest \( i > 0 \) such that \( \text{less}(i) = 0. \)

Then find the smallest \( j \geq i + 2 \) such that \( \text{less}(j - 1) = 1 \) and \( j \leq t + 2. \) (If no such \( j \) exists, report an error: The input \( x \) was equal to one of its cyclic shifts.) Set \( i \leftarrow i_0 \leftarrow j \mod t. \) (Now a dip of the period begins at \( i_0. \))

3. [Find the next factor.] Find the smallest \( j \geq i + 2 \) such that \( \text{less}(j - 1) = 1. \)

If \( j = i \) is even, go to 05.

4. [Retain a boundary.] If \( j < t, \) set \( b'_j \leftarrow b_j; \) otherwise set \( b'_j \leftarrow b'_{j-1} \) for \( t' \geq k > 0 > b'_0 \leftarrow b'_{t-1}. \) Finally set \( t' \leftarrow t' + 1. \)

5. [Done with round?] If \( j < i_0 + t, \) set \( i \leftarrow j \) and return to 03. Otherwise, if \( t' = 1, \) terminate; \( \sigma x \) begins at item \( x_{b'_0}. \) Otherwise set \( t \leftarrow t', b_i \leftarrow b'_i \) for \( 0 \leq k < t, \) and \( b_i \leftarrow b_{i-k} + n \) for \( k \geq t \) while \( b_{i-t} < n. \) Return to 02.

(e) Say that a “super dip” is a dip of odd length by zero or more dips of even length. Any infinite sequence \( y \) that begins with an odd-length dip has a unique factorization into superdips. Those superdips can, in turn, be regarded as atomic elements of a higher-level string that can be factored into dips. The result \( \sigma x \) of Algorithm O is an infinite periodic sequence that allows repeated factorization into infinite periodic sequences of superdips at higher and higher levels, until constant.

Notice that the first dip of \( \sigma x \) ends at position \( i_0 \) in the algorithm, because its length isn’t 2. Therefore we can prove the comma-free property by observing that, if codeword \( \sigma \sigma' \) appears within the concatenation \( \sigma x \sigma x' \) of two codewords, its super dip factors are also super dip factors of those codewords. This yields a contradiction if any of \( \sigma x, \sigma x', \) or \( \sigma x'' \) is a super dip. Otherwise the same observation applies to the super dip factors at the next level. [Eastman’s original algorithm was essentially the same, but presented in a more complicated way; see IEEE Trans. IT-11 (1965), 263–267. R. A. Scholtz subsequently discovered an interesting and totally different way to define the set of codewords produced by Algorithm O, in IEEE Trans. IT-15 (1965), 300–306.]

33. Let \( f_i(m) \) be the number of dips of length \( k \) for which \( m > z_1 \) and \( z_2 < m. \) The number of such sequences with \( z_2 = j \) is \( (m - 1) (m - 2) \cdots (m - j + 3) \) \( (k - 1) (k - 2) \cdots (k - j + 3) \); summing for \( 0 \leq j < m \) gives \( f_i(m) = k^{(m - j + 3) \cdots (k - j + 3)} \). Thus \( F_n(z) = \sum_{i=0}^{n-1} f_i(m) z^i = (m - 1)/(1 - z)^m. \) (The fact that \( f_0(m) = -1 \) in these formulas turns out to be useful!)

Algorithm O finishes in one round if and only if some cyclic shift of \( x \) is a super dip.

The number of aperiodic \( x \) that finish in one round is therefore \( n [z^n] G_n(z), \) where

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

To get the stated probability, divide by \( \sum_{i=0}^{n-1} \mu(d) n^{i/d}, \) the number of aperiodic \( x. \) (See Eq. 7.2.11—(6a). For \( n = 3, 5, 7, 9 \) these probabilities are 1, 1, 1, and 1–3/f \( (m^3 - 1). \))

34. If so, it couldn’t have 0011, 0110, 1000, or 0011.

December 22, 2015
35. That section considered such representations of stacks and queues, but not of unordered sets, because large blocks of sequential data were either nonexistent or ultra-expensive in olden days. Linked lists were the only decent option for families of variable-size sets, because they could more readily fit in a limited high-speed memory.

36. (a) The blue word \( x \) with \( c = d \) (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 0101 in its S1 list, when 0101 became red. (Previously 1011 had moved to the former position of 0011.)

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 leaving \( \sigma = 0 \). That might fail when \( \sigma \) reaches the value \(-1\).)

39. (a) Set \( r \leftarrow m + 1 \). Then for \( k \leftarrow 0, 1, \ldots, f - 1 \), set \( t \leftarrow \text{FREE}[k] \), \( j \leftarrow \text{MEM} \{ \text{CLOFF} + 4t + m^4 \} \), \( j \leftarrow \text{MEM} \{ \text{CLOFF} + 4t + m^4 \} \), 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 \( q \leftarrow 0 \), \( p' \leftarrow \text{MEM}[p] \), and \( p \leftarrow \text{POISON} \). While \( p < p' \) do the following steps: Set \( y \leftarrow \text{MEM}[p] \), \( z \leftarrow \text{MEM}[p + 1] \), \( y' \leftarrow \text{MEM}[y + m^4] \), and \( z' \leftarrow \text{MEM}[z + m^4] \). (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 (38), to set \( p' \leftarrow y' - 2 \), and if \( p \neq p' \) to store \( p, \text{MEM}[y'] \) and \( p + 1, \text{MEM}[p' + 1] \). Otherwise set \( p \leftarrow p + 2 \); if \( y' - y \geq z' - z \) and \( y' - y > q \), set \( q \leftarrow y' - y \) and \( x \leftarrow \text{MEM}[z] \); if \( y' - y < z' - z \) and \( z' - z > q \), set \( q \leftarrow z' - z \) and \( x \leftarrow \text{MEM}[y] \). Finally, after \( p \) has become equal to \( p' \), store(PP, \( p' \)) and set \( c \leftarrow c' \text{ALF}[x] \). (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 \( \text{rem}(\alpha, \delta, \alpha) \) that removes an item from a list, following the protocol (23): Set \( p \leftarrow \delta + o \) and \( q \leftarrow \text{MEM}\{p + m^4\} - 1 \). If \( q \geq p \) (meaning that list \( p \) isn’t closed or being killed), store \( p, \text{MEM}[p + m^4] \), set \( t \leftarrow \text{MEM}[\alpha + o - m^3] \), and if \( t \neq q \) also set \( y \leftarrow \text{MEM}[t] \), \( \text{store}(t, y) \), and \( \text{store}(\text{ALF}[y] + \alpha - m^3, \delta) \).

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

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

   Now, to green \( x \) we set \( \alpha \leftarrow \text{ALF}[x] \), \( \text{store}(\alpha, \text{GREEN}) \); then \( \text{close}(p_1(\alpha), \text{POFF}) \), \( \text{close}(p_2(\alpha), \text{P2OFF}) \), \( \ldots \), \( \text{close}(s_3(\alpha), \text{S3OFF}) \), 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 \( p' \leftarrow \text{MEM}[p] + 6 \), and store \( p' - 6, p_1(\alpha) + 50000 \), store \( p' - 5, s_3(\alpha) + 30000 \), store \( p' - 4, p_2(\alpha) + 20000 \), store \( p' - 3, s_2(\alpha) + 30000 \), store \( p' - 2, s_1(\alpha) + 30000 \), store \( p' - 1, s_1(\alpha) + 10000 \); 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 39(c), and delete poison entry \( p \) if \( y = y' \) or \( z = z' \). Otherwise if \( y < y' \) and \( z' < z \), go to 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 \( z' > z \), store \( x \mathord{<} x \), \( y + m^4, z, \mathord{<} \), reden \( \text{MEM}[r] \) for \( z \leq r < z' \), and delete poison entry \( p \). Otherwise (namely if \( y' > y \) and \( z' < z \), store \( y + m^4, y, \mathord{<} \), reden \( \text{MEM}[r] \) for \( y < r < y' \), and delete poison entry \( p \).

   Finally, after \( p \) has become equal to \( p' \), store(PP, \( p' \)).
Exercise 32 exhibits such codes explicitly for all odd \( n \). The earliest papers on the subject gave solutions for \( n = 2, 4, 6, 8 \). Yoji Niho subsequently found a code for \( n = 10 \) but was unable to resolve the case \( n = 12 \) [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 7.2.2.2C in 10.5 gigamens. 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 teramens of computation.

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

(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 or 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, 0011, 1101\}. The other six are represented by \{0001, 0110, 0111 or 1110\}, \{0001, 1001, 1011 or 1101\}, \{0001, 1001, 1101\}, \{0010, 0011, 1001\}.

(b) Algorithm C produces half of the 144 solutions, which form twelve equivalence classes. Eight are represented by \{0001, 0002, 1001, 1002, 2001, 2002, 2011, 2012, 2102, 2112, 2122 or 2212\} and \{0102, 0111, 1012\} or \{1020, 1101, 2101\}\} and \{1202, 2022, 1112\}; four are represented by \{0001, 0020, 0021, 0022, 1001, 1020, 1021, 1022, 1121 or 1211, 1201, 1202, 1221, 2001, 2021, 2202\} and \{1011, 1102, 1221\} or \{1101, 2101, 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\}, \{0132\}, \{1213\}, one such being the code \{0030, 0020, 0050, 0130, 0131, 0120, 0121, 0132, 0210, 0212, 0231, 0232, 0302, 0320, 0330, 0331, 1110, 1112, 1113, 1130, 1131, 1132, 1133, 1210, 1211, 1212, 1213, 2210, 2212, 2213, 2230, 2310, 2311, 2312, 2313, 2320, 2321, 2330, 2331, 3102, 3112, 3113, 3120, 3121, 3122, 3210, 3211, 3212, 3213, 3230, 3231, 3232, 3310, 3311, 3312, 3313\}. The others each have two representatives that omit cycle classes \{0123\}, \{0132\}, \{1213\}, one such being the code \{0001, 0002, 0003, 0020, 0021, 0023, 1001, 1002, 1003, 1011, 1013, 1021, 1022, 1023, 1031, 1032, 1033, 1201, 1202, 1203, 1211, 1212, 1213, 1221, 1223, 1231, 1232, 1311, 1312, 1313, 1321, 1322, 1331, 1332, 1333, 2001, 2002, 2003, 2021, 2022, 2032, 2201, 2203, 2211, 2221, 2231, 3001, 3002, 3003, 3011, 3013, 3021, 3022, 3023, 3031, 3032, 3033, 3021, 3030, 3221, 3222, 3321, 3322, 3331, and its isomorphic image under reflection and \{01\}\{23\}.

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; a SAT solver shows quickly that a full set of \((3^6 - 3^3 - 3^2 + 3^1)/6 = 116\) codewords cannot be achieved.

The 3-bit sequences 101, 111, 110 were rejected before seeing 000. In general, to make a uniformly random choice from \( q \) possibilities, the text suggests looking at the next \( t = \lfloor \log_2 q \rfloor \) bits \( b_1 \ldots b_t \). If \( (b_1 \ldots b_t)_2 < q \), we use choice \( b_1 \ldots b_t \); otherwise we reject \( b_1 \ldots b_t \) and try again. This simple method is optimum when \( q \leq 4 \), and for the best possible running time for other values of \( q \) uses more than half as many bits. But a better scheme is available for \( q = 5 \), using only \( 3 \frac{1}{4} \) bits per choice instead of \( 4 \frac{1}{4} \); and for \( q = 6 \), one random bit reduces to the case \( q = 3 \). See D. E. Knuth and A. C. Yao, Algorithms and Complexity, edited by J. F. Traub (Academic Press, 1976), 357–428, §2.
48. It's the number of nodes on level \( l + 1 \) (depth \( l \)) of the search tree. (Hence we can estimate the profile. Notice that \( D = D_1 \ldots D_{l+1} \) in step E2 of Algorithm E.)

49. \( Z_0 = C(1), \ Z_{i+1} = c(i) + D_1 c(X_1) + D_2 c(X_1, X_2) + \ldots + D_i c(X_1, \ldots X_i) + D_{i+1} C(X_1, \ldots X_{i+1}) \).

50. (a) True: The generating function is \( z(z + 1) \ldots (z + n - 1)/n! \); see Eq. 1.2.10-(g).

(b) For instance, suppose \( Y_1Y_2 \ldots Y_l = 1457 \) and \( n = 9 \). Alice's probability is \( 1457 1457 1457 1457 1457 1457 1457 1457 1457 = 111111111111111 \). Elmo obtains \( X_1X_2 \ldots X_l = 7541 \) with probability \( 111 \).

(c) The upper tail inequality (see exercise 1.2.10-22 with \( \mu = H_n \)) tells us that \( \Pr(l \geq (\ln n / \ln \ln n)) \leq \exp(-n/\ln n) \approx \exp(-n/(\ln n)) \).

(d) If \( k \leq n/3 \) we have \( \sum_{i=0}^k \binom{n}{i} \leq 2^n \). By exercise 1.2.6-7, the number of nodes on the first \( (\ln n / \ln \ln n) \) levels is therefore at most \( 2^{(\ln n / \ln \ln n) / \ln n} \).

51. The key idea is to introduce recursive formulas analogous to (29):

\[
m(x_1 \ldots x_l) = c(x_1 \ldots x_l) + \min(m(x_1 \ldots x_i x_{i+1}^{(1)} \ldots x_{i+1}^{(d)} \ldots ) m(x_1 \ldots x_i x_{i+1}^{(d)} \ldots ) \ldots m(x_1 x_i x_{i+1}^{(d)} \ldots ));
M(x_1 \ldots x_l) = c(x_1 \ldots x_l) + \max(M(x_1 \ldots x_i x_{i+1}^{(1)} \ldots x_{i+1}^{(d)} \ldots ), M(x_1 \ldots x_i x_{i+1}^{(d)} \ldots ) \ldots , M(x_1 \ldots x_i x_{i+1}^{(d)} \ldots ));
\]

\[
\hat{C}(x_1 \ldots x_l) = c(x_1 \ldots x_l)^2 + \sum_{i=1}^{d} (\hat{C}(x_1 \ldots x_i x_{i+1}^{(1)} \ldots x_{i+1}^{(d)} \ldots ) + 2c(x_1 \ldots x_i) C(x_1 \ldots x_i x_{i+1}^{(i)} \ldots ));
\]

They can be computed via auxiliary arrays \( \text{MIN}, \text{MAX}, \text{KIDS}, \text{COST}, \) and \( \text{CHAT} \) as follows:

At the beginning of step B3, set \( \text{MIN}[l] \gets \infty, \text{MAX}[l] \gets \text{KIDS}[l] \gets \text{COST}[l] \gets \text{CHAT}[l] \gets 0. \) Set \( \text{KIDS}[l] \gets \text{KIDS}[l] + 1 \) just before \( l \gets l + 1 \) in step B3.

At the beginning of step B3, set \( m \gets c(x_1 \ldots x_{i-1}) + \text{KIDS}[l] \times \text{MIN}[l], \ M \gets c(x_1 \ldots x_{i-1}) + \text{KIDS}[l] \times \text{MAX}[l], \ C \gets c(x_1 \ldots x_{i-1}) + \text{COST}[l], \ \hat{C} \gets c(x_1 \ldots x_{i-1})^2 + \text{KIDS}[l] \times \text{CHAT}[l] + 2 \times \text{COST}[l]. \) Then, after \( l \gets l + 1 \) is positive, set \( \text{MIN}[l] \gets \min(m, \text{MIN}[l]), \ \text{MAX}[l] \gets \max(M, \text{MAX}[l]), \ \text{COST}[l] \gets \text{COST}[l] + C, \ \text{CHAT}[l] \gets \text{CHAT}[l] + \hat{C}. \) But when \( l \) reaches zero in step B5, return the values \( m, \ M, \ C, \ \hat{C} = C^2. \)

52. Let \( p(i) = p(x_1 \ldots x_{i-1} \cdot y_i) \), and simply set \( D \rightarrow D / p(I) \) instead of \( D \rightarrow Dd \). Then node \( x_1 \ldots x_i \) is reached with probability \( \Pi(x_1 \ldots x_i) = p(x_1)p(x_2 \ldots p(x_{i-1} \cdot x_i) \ldots ) \), and \( c(x_1 \ldots x_i) \) has weight \( 1/\Pi(x_1 \ldots x_i) \) in \( S \); the proof of Theorem E goes through as before. Notice that \( \mu(I) \) is the \( c \)-a posteriori probability of having taken branch \( I \).

(The formulas of answer 51 should now use \( p'(i) \) instead of \( d' \), and that algorithm should be modified appropriately, no longer needing the KIDS array.)

53. Let \( p(x_1 \ldots x_{i-1} \cdot y_i) = C(x_1 \ldots x_{i-1} \cdot y_i) / C(x_1 \ldots x_{i-1} - c(x_1 \ldots x_{i-1}) \). (Of course we generally need to know the cost of the tree before we know the exact values of these ideal probabilities, so we cannot achieve zero variance in practice. But the form of this solution shows what kinds of bias are likely to reduce the variance.)

55. The effects of lookahead, dynamic ordering, and reversible memory are all captured easily by a well-designed cost function at each node. But there's a fundamental difference in step C2, because different codeword classes can be selected for branching at the same node (that is, with the same ancestors \( x_1 \ldots x_{i-1} \) after C5 has undone the effects of a prior choice. The level \( l \) never surpasses \( L + 1 \), but in fact the search tree involves hidden levels of branching that are implicitly combined into single nodes.

Thus it's best to view Algorithm C's search tree as a sequence of binary branches. Should \( x \) be one of the codewords or not? (At least this is true when the "max kill" strategy of answer 39 has selected the branching variable \( x \). But if \( r > 1 \) and the poison list is empty, an \( r \)-way branch is reasonable (or an \( (r + 1) \)-way branch when the slack is positive), because \( r \) will be reduced by 1 and the same class \( c \) will be chosen after \( x \) has been explored.)
40 ANSWERS TO EXERCISES 7.2.2

If \( x \) has been selected because it kills many other potential codewords, we probably should bias the branch probability as in exercise 52, giving smaller weight to the “yes” branch because the branch that includes \( x \) is less likely to lead to a large subtree.

57. Let \( p_k = 1/D^{(k)} \) be the probability that Algorithm E terminates at the \( k \)th leaf. Then \( \sum_{k=1}^{M} (1/M) \log(1/(Mp_k)) \) is the Kullback-Leibler divergence \( D(q||p) \), where \( q \) is the uniform distribution (see exercise MPR-121). Hence \( \frac{1}{M} \sum_{k=1}^{M} \log D^{(k)} \geq \log M \). (The result of this exercise is essentially true in any probability distribution.)

58. Let \( \varepsilon \) be any convenient value \( \geq n \). When vertex \( v \) becomes part of the path we will perform a two-phase algorithm. The first phase identifies all “tarnished” vertices, whose DIST must change; these are the vertices \( u \) from which every path to \( t \) passes through \( u \). It also forms a queue of “resource” vertices, which are untarnished but adjacent to tarnished ones. The second phase updates the DIST of all tarnished vertices that are still connected to \( t \). Each vertex has LINK and STAMP fields in addition to DIST.

For the first phase, set \( d \leftarrow \text{DIST}(v) \), \( \text{DIST}(v) \leftarrow \infty + 1 \), \( R \leftarrow \Lambda \), \( T \leftarrow v \), \( \text{LINK}(v) \leftarrow \Lambda \), then do the following while \( T \neq \Lambda \): (a) Set \( u \leftarrow T \), \( T \leftarrow S \leftarrow \Lambda \). For each \( w \leftarrow u \), if \( \text{DIST}(w) < d \) do nothing (this happens only when \( u = v \)); if \( \text{DIST}(w) \geq d \) do nothing (\( w \) is gone or already known to be tarnished); if \( \text{DIST}(w) = d \), make \( w \) a resource (see below); otherwise \( \text{DIST}(w) = d + 1 \). If \( w \) has no neighbor at distance \( d \), \( w \) is tarnished: Set \( \text{LINK}(w) \leftarrow T \), \( \text{DIST}(w) \leftarrow \infty \), \( T \leftarrow w \). Otherwise make \( w \) a resource (see below). Then set \( u \leftarrow \text{LINK}(u) \), and return to (a) if \( u \neq \Lambda \).

The queue of resources will start at \( R \). We will stamp each resource with \( v \) so that nothing is added twice to that queue. To make \( w \) a resource when \( \text{DIST}(w) = d \), do the following (unless \( u = v \) or \( \text{STAMP}(w) = v \)): Set \( \text{STAMP}(w) \leftarrow v \), if \( R = \Lambda \), set \( R \leftarrow RT \leftarrow w \); otherwise set \( \text{LINK}(RT) \leftarrow w \) and \( RT \leftarrow w \). To make \( w \) a resource when \( \text{DIST}(w) = d + 1 \) and \( u \neq v \) and \( \text{STAMP}(w) \neq v \), put it first on stack \( S \) as follows: Set \( \text{STAMP}(w) \leftarrow v \), if \( S = \Lambda \), let \( S \leftarrow SB \leftarrow w \); otherwise set \( \text{LINK}(w) \leftarrow S \), \( S \leftarrow w \).

Finally, when \( u = \Lambda \), we append \( S \) to \( R \). Nothing needs to be done if \( S = \Lambda \). Otherwise, if \( R = \Lambda \), set \( R \leftarrow S \) and \( RT \leftarrow SB \); but if \( R \neq \Lambda \), set \( \text{LINK}(RT) \leftarrow S \) and \( RT \leftarrow SB \). (These shenanigans keep the resource queue in order by DIST.)

Phase 2 operates as follows: Nothing needs to be done if \( R = \Lambda \). Otherwise we set \( \text{LINK}(RT) \leftarrow \Lambda \), \( S \leftarrow \Lambda \), and do the following while \( R \neq \Lambda \) or \( S \neq \Lambda \): (i) If \( S = \Lambda \), set \( d \leftarrow \text{DIST}(R) \). Otherwise set \( u \leftarrow S \), \( d \leftarrow \text{DIST}(u) \), \( S \leftarrow \Lambda \); while \( u \neq \Lambda \), update the neighbors of \( u \) and set \( u \leftarrow \text{LINK}(u) \). (ii) While \( R \neq \Lambda \) and \( \text{DIST}(R) = d \), set \( u \leftarrow R \), \( R \leftarrow \text{LINK}(u) \), and update the neighbors of \( u \). In both cases “update the neighbors of \( u \)” means to look at all \( w \to u \), and if \( \text{DIST}(w) = \infty \), set \( \text{DIST}(w) \leftarrow d + 1 \), \( \text{STAMP}(w) \leftarrow v \), \( \text{LINK}(w) \leftarrow S \), and \( S \leftarrow w \). (It works!)

59. (a) Compute the generating function \( g(z) \) (see exercise 7.1.4–209) and then \( g'(1) \).

(b) Let \( (A, B, C) \) denote paths that touch (center, NE corner, SW corner). Recursively compute eight counts \((c_0, \ldots, c_7)\) at each node, where \( c_j \) counts paths \( \pi \) with \( j = 4[\pi \in A] + 2[\pi \in B] + [\pi \in C] \). At the sink node \( \square \) we have \( c_0 = 1 \), \( c_1 = \cdots = c_7 = 0 \). Other nodes have the form \( x = (\xi', x_0; x_i) \) where \( x_i \) is an edge. Two edges go across the center and affect \( A \); three edges affect each of \( B \) and \( C \). Say that these edges have types 4, 2, 1, respectively; other edges have type 0. Suppose the counts for \( x_0 \) and \( x_i \) are \((c_0', \ldots, c_7')\) and \((c_0'', \ldots, c_7'')\), and \( c \) has type \( t \). Then count \( c_j \) for node \( x \) is \( c_j + [t=0]c_j' + [t \neq 0][c_j' + c_j''] \).

(This procedure yields the following exact “Venn diagram” set counts at the root: \( c_0 = (A \cap B \cap C) = 76535838488901984991064; c_1 = c_2 = (A \cap B \cap C) = (A \cap B \cap C) = 77550190537790917830134; c_3 = (A \cap B \cap C) = 78570697050366819944024; December 22, 2015

bias Kullback-Leibler divergence generating function Venn diagram
60. Yes, the paths are less chaotic and the estimates are better:

![Graph showing less chaotic paths]

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

\[
F(\alpha, \beta) = F(\alpha_1, \ldots, \alpha_n, x_1 \ldots x_n) = \bigwedge_{j=1}^n \bigwedge_{s \in S} ([a_j = s] \Rightarrow x_j \equiv f_{j\alpha}(\alpha, \beta)).
\]

The maximum score is the largest value of \( x_1 + \cdots + x_n \) over all graded answer lists \((\alpha, \beta)\) that are valid. A perfect score is achieved if and only if \( F(\alpha, 1 \ldots 1) \) holds.

Thus, in the warmup problem we have \( n = 2, S = \{A, B\} \); \( f_{1A} = [a_2 = B] \); \( f_{2A} = [a_1 = A]; f_{1B} = [a_1 = A]; f_{2B} = [a_1 = B] \). The four possible answer lists are:

- AA: \( F = (a_1 = [A = B]) \land (a_2 = x_1) \)
- AB: \( F = (a_1 = [B = B]) \land (a_2 = x_1 \oplus [A = A]) \)
- BA: \( F = (a_1 = [B = A]) \land (a_2 = x_1) \)
- BB: \( F = (a_1 = [B = A]) \land (a_2 = x_1 \oplus [B = B]) \)

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 606 we have, for example, \( f_{1C} = [a_2 \neq A] \land [a_3 = A]; f_{1D} = [a_1 = D] \land [a_{15} = D]; f_{1E} = [\Sigma_A - 1 = \Sigma_B], \) where \( \Sigma_s = \sum_{1 \leq j \leq 20}[a_j = s] \). It’s amusing to note that \( f_{1E} = \{[\Sigma_A, \ldots, \Sigma_B] = \{2, 3, 4, 5, 6\}\} \).

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 ‘\( x_{20} \equiv \bar{x}_{11} \)’ while 17E says ‘\( x_{11} \equiv x_{20} \)’.

By suitable backtrack programming, we can prove first 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 \( a_1 \neq a_2 \neq \cdots \neq a_{16} \neq a_{17} = a_{18} \neq a_{19} \neq a_{20} \), 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 makes 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 52830 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 kilomems. But then, ta da, three solutions were found: One for \(j = 19\) (185 kilomems, 11 megamems) and two for \(j = 20\) (131 kilomems, 8 megamems), namely

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

(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 \(f_{ij}\). Let’s explore a bit of recursive territory by considering the following two-question, two-letter example:

1. (A) Answer 1 is incorrect.
2. (A) Some answers can’t be graded consistently. (B) No answers achieve a perfect score.

Here we have \(f_{1A} = \bar{x}_1\); \(f_{1B} = \bar{x}_2\); \(f_{2A} = \exists x_1 \forall x_2 \neg F(a_1, a_2, x_1, x_2)\); \(f_{2B} = x_1 \forall x_2 \neg F(a_1, a_2, 11)\). (Formulas quantified by \(\exists x\) or \(\forall x\) expand into \(|S|\) terms, while \(\exists x\) or \(\forall x\) expand into two; for example, \(\exists x \forall x g(a, x) = (g(a, 0) \lor g(a, 1)) \lor (g(a, 0) \land g(a, 1))\) 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 \(x_1 \equiv \bar{x}_1\). Also \(f_{2B}\) is true, because both BA and BB imply \(x_1 \equiv \bar{x}_2\). 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 \(x_{30}\) false. And we can conclude that \(m = 18\), because of the following two solutions (which are the only possibilities with 20C false):

\[
\begin{array}{cccccccccccccc}
\end{array}
\]
7.2.2 ANSWERS TO EXERCISES

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

\[
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 \\
D e D A B E D e C A E D A E D B D C C A
\]
or 47 other answer lists. This 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

\[
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 \\
D C E A B E D C E A E B A E D B D A d D
\]
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.

Question 20 was designed to create difficulties. \([-\ldots]\)

— DONALD R. WOODS (2001)

95. (a) Let \( x_k \) be the number of nodes at distance \( k \) from the root.

(b) Let \( Q_n^{(m)} = P_n^{(1)} + \ldots + P_n^{(m)} \). Then we have the joint recurrence

\[
P_n^{(m)} = Q_n^{(2m)} \quad \text{in particular}, \quad Q_1^{(m)} = m
\]

for certain constants \( a_n \) that can be computed as follows: Set \( t_k \leftarrow P_n^{(k)} \) for \( 1 \leq k \leq n \). Then for \( k = 2, \ldots, n \) set \( t_n \leftarrow t_n - t_{n-1}, \ldots, t_k \leftarrow t_k - t_{k-1} \). Finally \( a_{n,k} \leftarrow t_k \) for \( 1 \leq k \leq n \). For example, \( a_{21} = a_{22} = 2 \); \( a_{31} = 6 \); \( a_{32} = 14 \); \( a_{33} = 8 \). The numbers \( P_n^{(m)} \) have \( O(n^2 + n \log m) \) bits, so this method needs \( O(n^3) \) bit operations to compute \( P_n \).

(c) \( P_n^{(m)} \) corresponds to random paths with \( X_1 = m \), \( D_0 = 2X_k \), \( X_{k+1} = [2U_kX_k] \), where each \( U_k \) is an independent uniform deviate. Therefore \( P_n^{(m)} = E(D_1 \ldots D_n) \) is the number of nodes on level \( n \) of an infinite tree. We have \( X_{k+1} \geq 2U_1 \ldots U_k m \), by induction; hence \( P_n^{(m)} \geq E(2U_1^2 \ldots U_k^m - 2U_2^3 \ldots U_{n-2}^m m) = 2(2^m m^{m-1}/(n-1)!) \).

[M. Cook and M. Kleber have discussed similar sequences in Electronic Journal of Combinatorics 7 (2000), #R44. See also K. Mahler’s asymptotic formula for binary partitions, in J. London Math. Society 15 (1940), 115–123, which shows that \( \log P_n = \binom{\log n}{2} - \log(n-1)! + \left[ \log \frac{n}{2} \right] + O(1) \).]

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.

2-letter block codes, 27.
4-letter codewords, 9–18, 27–29.
5-letter words of English, 8–9, 27.
6-letter and k-letter words of English, 8–9.
\( \gamma \) (Euler’s constant), as source of “random” data, 19.
\( \pi \) (circle ratio), as source of “random” data, 19, 22, 28.
\( \phi \) (golden ratio), as source of “random” data, 19.

A posteriori probability, 30.
Active elements of a list, 12.
Ahrens, Joachim Heinrich Luddecke, 6, 26, 33.
Analysis of algorithms, 28, 29.
Aperiodic words, 10, 27, 36.

Backjumping, 24.
Backmarking, 24.
Backtrack programming, 2–30.

Backtrack tree, 3, 4, 7, 9–11, 18–20, 38, 39, 41.

estimating the size of, 20, 38.
Baumert, Leonard Daniel, 24, 43.
Bees, queen, 26.
Bernoulli, Jacques (= Jakob = James), 24.
Bezzel, Max Friedrich Wilhelm, 24.
Biased random walks, 29–30, 40.
Binary partitions, 31.
Binary trees, 21.
Biter, James Richard, 6, 24.
Bitwise operations, 5, 26, 34.

Block codes, 9, 27.
Boundary markers, 28.
Breath-first search, 33.
Breaking symmetry, 8, 14.

British National Corpus, 8.
Broken diagonals, see Wraparound.
Bumping the current stamp, 16, 28.
Bunch, Steve Raymond, 6.

Cache-friendly data structures, 11.
Carroll, Lewis (= Dodgson, Charles Lutwidge), 10.
Cavenagh, Nicholas John, 33.

INDEX AND GLOSSARY  GOLDSMITH

December 22, 2015
Dynamic shortest distances, 29.

e, as source of “random” data, 19.
Eastman, Willard Lawrence, 27, 28, 36.
Empty lists, 12, 17.
English words, 8–9, 27.
Error bars, 22.
Estimates of run time, 18–21.
Estimating the number of solutions, 21–23.
Fallback points, 16.
Five-letter words, 8–9, 27.
Fixed point of recursive formula, 42.
Floyd, Robert W., 15.
FPGA devices: Field-programmable
gate arrays, 6.
Frames, 11.
Fränkel, Jérôme, 33.
Gattegno, Bernard, 35.
Gauss (= Gauss), Johann Friderich Carl
(= Carl Friedrich), 24, 32.
Generating functions, 28, 39, 40.
Gigamem (Mi): One billion memory
accesses, 5.
Global variables, 26.
Goldsmith, Oliver, 44.
Gödel, Solomon Wolf, 9, 24, 25.
Godfrey, Basil, 9, 45.
Graduate, 41.
Griffith, John Stanley, 9.
Hales, Alfred Washington, 45.
Hall, Marshall, Jr., 24.
Hamilton, William Rowan, paths, 23.
Hammondsley, John Michael, 24.
Handcomb, David Christopher, 24.
Height of binary trees, 31.
Hexagons, 36.
Historical notes, 2, 5–6, 24–25.
Honeycomb, 26.
Hurewitz, Adolf, 33.
IBM 1620 computer, 6.
IBM System 360-75 computer, 6.
Importance sampling, 24.
Indeterminate statements, 42.
Insertion loops, 32.
Insertion operation, 12.
Integer partitions, 26, 31.
Internet, ii, iii.
Inverse lists, 12–15, 17.
Inverse permutations, 12–13.
Iteration versus recursion, 26, 32.
Jaenisch, Carl Friedrich Andrejevitch de
(Mueller, Kaputs Andrejevitsch), 32.
Jewett, Robert Isadell, 45.
Jigga, B. H. (pen name of Baumert, Hales,
Jewett, Imaginary, Goeland, Gordon,
and Selfridge), 17.
Kennedy, Michael David, 6.
Kilometer (Km): One thousand memory
accesses, 41.
King, Benjamin Franklin, Jr., 2.
Kleiber, Michael Steven, 43.
Knight moves, 23.
Knuist, Donald Ervin (高德纳), i, iv,
18, 24, 35, 38, 42.
Knuth, Donald Ervin (高德纳), 24, 33.
Langford, Charles Dudley, pairs, 6–8, 26–27.
Landau, Albert Lee, 17.
Leibler, Richard Arthur, 40.
Lennon, John Winston Ono, 2.
Lexicographic order, 2, 7, 24, 28.
Linked lists, 6–7, 37.
Locakhead, 10, 16, 24, 29.
Looser Langford pairs, 27.
Lucas, François Édouard Anatole, 24, 33.
Mahler, Kurt, 43.
Marginals, 29.
Masks, 32.
Megaemem (Mi): One million memory
accesses, 18.
MEM, an array of “cells,” 11–18, 28.
Memory constraints, historic, 37.
Mem (μ): One 64-bit memory access, 4.
Minimum remaining values heuristic, 24.
MIX computer, ii.
Monte Carlo estimates, 18–25, 29.
Moves, 11.
MPT: Mathematical Preliminaries
Redex, v, i.
n-letter words of English, 8.
n queens problem, 2–6, 18–20, 24, 26.
n-tuples, 26.
Nauck, Franz, 24.
Nested parentheses, 26.
Niho, Yoichi Goff (二尾�), 38.
Nonisomorphic solutions, 29.
Onnen, Hendrick, Sr., 6.
Optimization, 24.
Ongel, Leslie Eleazer, 9.
Orthogonal lists, 27.
OSPD4: Official SCRABBLE® Players
Dictionary, 8.
Overflow of memory, 12, 16.

December 22, 2015
Index and Glossary

Paradox, 31.
Parentheses, 26.
Parity argument, 34.
Partitions, 26, 31.
Paths, simple, 22, 25, 29.
Pauli, Emil, 33.
Periodic sequences, 27.
Periodic words, 10, 13.
Permutations, 6, 26.
Pheaton, John Lewis, 45.
Polya, György (= George), 33.
Preufer, Thomas Bond, 6.
Prime strings, 10.
Probabilities, 1.
Profile of a tree, 3, 9, 27, 31, 39.
Propagation algorithm, 42.
Properties: Logical propositions (relations), 2, 20.
q.s., 29.
Quantified Boolean formulas, 42.
Queen bees, 26.
Questionnaires, 30.
Questions, 37, 40.
Quick, Jonathan Horatio, 26.
Radix m representation, 13.
Random bits, 29.
Random sampling, 18.
Random variables, 19.
Recurrence relations, 30, 43.
In a Boolean equation, 42.
Recursion versus iteration, 25, 32.
Recursive algorithms, 26, 39.
Reflection symmetry, 14, 20, 32, see also Dual solutions.
Register, 5, 34–45.
Reingold, Edward Martin (רנקלדע, ר"א רנגלד), 24.
Rejection method, 19, 38.
Restricted growth strings, 32.
Reversible memory technique, 16, 29.
Rivin, Igor (Ривин, Игорь Евгеньевич), 33.
Root node, 19.
Rosentrath, Arianna Wright, 25.
Rosentrath, Marshall Nicholas, 25.
Rotation by 90°, 26.
Running time estimates, 18–21.
Sample variance, 22.
SAT solvers, 38.
Saturating ternary addition, 35.
Schötz, Robert Arno, 36.
Schumacher, Heinrich Christian, 24, 32.
Search rearrangement, see Dynamic ordering.
Search trees, 3, 4, 7, 9–11, 18–20, 33, 38, 39, 42.
estimating the size, 20, 38.
Self-avoiding walks, 22, 25, 29.
Self-reference, 30, 46.
Self-synchronizing block codes, 9.
Selfridge, John Lewis, 45.
Semi-queens, 33.
Sequential allocation, 28.
Sequential lists, 11–15.
Set partitions, 26.
Shortest distances, dynamic, 29.
Signature of a trie node, 34.
Simple paths, 22, 25, 29.
Slack, 18, 30.
Sprague, Thomas Bond, 5, 6, 26.
Stacks, 16, 37.
Stamping, iv, 16–19, 28.
Standard deviation, 20, 22, 29.
Stanford GraphBase, ii, 8.
Statistics, 22.
Stirling, James, cycle numbers, 29.
Students, 41.
Substrings, 28.
Subtrees, 20.
Superdiagonals, 36.
SWAC computer, 6.
Symmetries, 29, 32.
breaking, 8, 14.
Tail inequality, 39.
Teramem (Tb): One trillion memory accesses, 9.
Torus, 26.
Toti tibi . . . , 24.
Trah, Joseph Frederick, 38.
Trémaux, Charles Pierre, 24.
Tries, 8–9, 27, 34.
compressed, 35.
Tuples, 26.
Twenty Questions, 30.
Two-letter block codes, 27.

UCLA: The University of California at Los Angeles, 6.
Undod, 16.
Uniformly random numbers, 38.
Unit clauses, 35.
University of California, 6.
University of Dresden, 6.
University of Illinois, 6.
University of Tennessee, 6.
Unordered sequential lists, 11.
Unordered sets, 37.

December 22, 2015
Valid grading, 41.
Vardi, Ilan, 33.
Variance of a random variable, 22, 29.
Venn, John, diagram, 40.
Visiting an object, 2, 4, 5, 7, 18.

Walker, Robert John, 2, 5, 6, 24.
Wanless, Ian Murray, 33.
Welsh, Lloyd Richard, 9.
Welsh, Mark Brimhall, 24.
White squares, 26.
Woods, Donald Roy, 30, 43.

Word rectangles, 8–9, 27.
WORDS(n), the n most common five-letter words of English, 8.
Worst-case bounds, 28.
Wraparound, 26.

Yao, Andrew Chi-Chih (姚期智), 38.
ZDD: A zero-suppressed decision diagram, 25, 30.
Zimmermann, Paul Vincent Marie, 33.