(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. This is a somewhat nontrivial example of backtrack programming, intended to exhibit basic ideas as well as perhaps to solve an open problem.

The problem is to find the largest possible comma-free code of length 4 on an m-letter alphabet, where $m = 5$. It was solved for $m \le 4$ in the 60s, so I have some hope. On the other hand, the running time might also be worse than exponential in m, so I must keep my fingers crossed.

The theory was set out by Golomb, Gordon, and Welch in *Canadian Journal of Mathematics* **10** (1958), 202–209. There are m^4 four-letter words, and m^2 of them are "cyclic" (like `dodo`). The others are divided into $(m^4-m^2)/4$ equivalence classes, where a word is considered equivalent to its cyclic shifts. Our problem is to choose as many codewords as possible such that, if $x_1x_2x_3x_4$ and $x_5x_6x_7x_8$ are codewords, then $x_2x_3x_4x_5$ and $x_3x_4x_5x_5$ and $x_4x_5x_6x_7$ are *not* codewords. In particular, when $x_1x_2x_3x_4 = x_5x_6x_7x_8$ this condition tells us that we can choose at most one codeword from each equivalence class.

The number of equivalence classes is 3, 18, 60, and 150 when $m = 2$, 3, 4, and 5. This maximum code size is achievable for $m = 2$ and $m = 3$; but a computer program by Lee Laxdal in 1963 showed that the largest sets for $m = 4$ have 57 codewords, not 60.

I think there's an interesting way to reproduce that result (namely by the methods below), and perhaps to go further.

Note (written after completing this program): I haven't time to follow up on an afterthought that might be able to help crack the case $m = 5$. Please see the detailed remarks near the beginning of ⟨ Find x that maximizes the "kill" potential s 40 ⟩.

```
#define maxm 7 /* unfortunately however it turns out that m = 5 is already too slow */
#define maxmmmm (maxm * maxm * maxm * maxm)
#define maxclasses ((maxmmmm - maxm * maxm)/4)
#define ustacksize 1000000
#define o mems++
#define oo mems += 2
#include <stdio.h>
#include <stdlib.h>
  int m, goal; /* command-line parameters */
  typedef unsigned long long ull;
  ⟨ Type definitions 28 ⟩;
  ull mems; /* 8-byte memory accesses */
  ull nodes; /* size of search tree */
  ull thresh = 10000000000; /* reporting time */
  ull profile[maxclasses + 3]; /* width of search tree levels */
  int count; /* number of solutions found */
  int maxstack, maxpoison; /* how big have these lists gotten? */
  int vbose; /* level of verbosity */
  ⟨ Global variables 6 ⟩;
  ⟨ Subroutines 4 ⟩;
  main(int argc, char *argv[])
  {
    register int a, b, c, d, i, j, k, l, p, q, r, s, pp, x, y, yy, z, zz, alpha, ploc, cls, slack;
    ⟨ Process the command line 3 ⟩;
    ⟨ Initialize the tables 10 ⟩;
    ⟨ Backtrack through all solutions 32 ⟩;
    fprintf(stderr, "Altogether %d solutions (%lld mems, %lld nodes)\n", count, mems, nodes);
    fprintf(stderr, "(maxstack=%d,maxpoison=%d)\n", maxstack, (maxpoison - poison)/2);
    if (vbose) ⟨ Print the profile 2 ⟩;
  }
```

2. ⟨ Print the profile 2 ⟩ ≡

```
{
  fprintf(stderr, "Profile:␣␣␣␣␣␣␣␣␣␣1\n");
  for (k = 2; k ≤ maxl + 1; k++) fprintf(stderr, "%19lld\n", profile[k]);
}
```

This code is used in section 1.

3. ⟨ Process the command line 3 ⟩ ≡

```
if (argc < 3 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &goal) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣m␣goal␣[profile]␣[verbose]␣[extraverbose]\n", argv[0]);
  exit(−1);
}
if (m < 2 ∨ m > maxm) {
  fprintf(stderr, "m␣should␣be␣between␣2␣and␣%d,␣not␣%d!\n", maxm, m);
  exit(−2);
}
vbose = argc − 3;
```

This code is used in section 1.

4. ⟨ Subroutines 4 ⟩ ≡

```
void confusion(char *s)
{
  fprintf(stderr, "I'm␣confused␣(%s)!\n", s);
  exit(−666);
}
```

See also sections 5, 18, 19, 20, 21, 31, and 42.

This code is used in section 1.

5. Infrastructure. This program keeps its state information in an array *mem* of 16-bit quantities. Everything in *mem* also has a 16-bit address. Thus I can undo the changes made at one level if I maintain a stack that holds 32-bit items, (address, oldvalue).

In this implementation I use a "stamp" to see whether the former *mem* value has already been saved. This makes the cost $2r+5$ mems when we store $r>0$ times into a location, rather than $5r$ mems; so it saves time if r is usually greater than 1.

(With a change file I can try removing the stamps, to see what happens in practice.)

```
⟨ Subroutines 4 ⟩ +≡
  inline void store(register short a, register short x)
  {
    if (o, stamp[a] ≠ curstamp) {
      o, stamp[a] = curstamp;
      oo, undo[uptr++] = (a ≪ 16) + mem[a];
      if (uptr > maxstack) {
        maxstack = uptr;
        if (maxstack ≥ ustacksize) {
          fprintf(stderr, "Stack overflow (ustacksize=%d)!\n", ustacksize);
          exit(−9);
        }
      }
    }
    o, mem[a] = x;
  }
  inline void unstore(register int ptr)
  {
    register unsigned int x;
    while (uptr > ptr) {
      o, x = undo[−−uptr];
      o, mem[x ≫ 16] = x & #ffff;      /* everything snaps back to its former value */
    }
  }
```

6. ⟨ Global variables 6 ⟩ ≡

```
  unsigned mem[1 ≪ 16];
  unsigned int undo[ustacksize];
  int uptr;      /* this many entries are currently in undo */
  unsigned int stamp[1 ≪ 16];
  unsigned int curstamp;
```

See also sections 9, 14, 24, and 29.

This code is used in section 1.

7. Here's what we do when we need a new current stamp.

Incidentally, my first draft of this step had a noteworthy bug: I knew that *curstamp* won't actually overflow in this program; still, this program is intended as a model for more complicated ones in which overflow really might happen. So I somewhat grudgingly added code to cover the case of overflow. But I noticed that I could save one (1) line, aha, by leaving $curstamp = 0$ after overflow, and setting all the remaining stamps to $^{\#}$`ffffffff`. Reason: The purpose is to have $stamp[x] \neq curstamp$ for all x after bumping. Fallacy: Later on, when *curstamp* rises for the second time to $^{\#}$`ffffffff`, that invariant relation might no longer hold. A better invariant is to say that $stamp[x] < curstamp$ for all x after bumping.

```
⟨ Bump curstamp 7 ⟩ ≡
  if (++curstamp ≡ 0) {      /* whoops, curstamp has overflowed! cancel all stamps */
    for (k = 0; k < (1 ≪ 16); k++) o, stamp[k] = 0;
    curstamp = 1;
  }
```

This code is used in section 32.

8. This program also makes use of numerous sequential lists, kept in *mem*, for which the basic operations are to append a new item, or to remove an item given its name.

Suppose p points to the first empty spot at the current end of a list. To append a new item x, knowing that it isn't already present, the operation is $*p = x$ followed by $i[x] = p{+}{+}$, where i is an inverse list. To delete item x, knowing that it is present, we set $q = i[x]$ and test if $-{-}p \equiv q$. If so, we're done. If not, we set $y = *p$ and $*q = y$ and $i[y] = q$.

9. Memory layout. Let α be a four-letter word $abcd$. We will think of α as the radix-m integer $(abcd)_m$, although it will also be represented in memory as the hexadecimal integer $(abcd)_{16}$ since that's convenient for programming and debugging.

```
⟨ Global variables 6 ⟩ +≡
  int hexconv[1 ≪ 16];      /* table to convert from hex to radix m */
  int mconv[maxmmmm];       /* and vice versa (used only in diagnostics) */
```

10.
```
⟨ Initialize the tables 10 ⟩ ≡
  for (a = 0; a < m; a++)
    for (b = 0; b < m; b++)
      for (c = 0; c < m; c++)
        for (d = 0; d < m; d++)
          o, hexconv[(a ≪ 12) + (b ≪ 8) + (c ≪ 4) + d] = ((a * m + b) * m + c) * m + d,
              mconv[((a * m + b) * m + c) * m + d] = (a ≪ 12) + (b ≪ 8) + (c ≪ 4) + d;
```

See also sections 17, 25, and 30.

This code is used in section 1.

11. During the backtrack process, every word α will be in one of three states, which I'll call green, red, and blue. (i) Green words have been chosen (tentatively) to be in the code. (ii) Red words have been ruled out, either because they're incompatible with the words that are currently green or because all solutions in which they're green have already been considered. (iii) The remaining words, feeling blue, are still in limbo. The current state of word α is in $mem[\alpha]$.

```
#define green  2
#define red  0
#define blue  1
```

12. Let $p_1(\alpha) = (a000)_m$, $p_2(\alpha) = (ab00)_m$, $p_3(\alpha) = (abc0)_m$ be the three proper prefixes of $\alpha = abcd$, and let $s_1(\alpha) = (d000)_m$, $s_2(\alpha) = (cd00)_m$, $s_3(\alpha) = (bcd0)_m$ be the (shifted) proper suffixes.

Every blue word $\alpha = abcd$ appears in six lists, corresponding to those suffixes. For example, $p_1(\alpha)$ corresponds to the list of all blue words that begin with a, and $s_2(\alpha)$ corresponds to the list of all blue words that end with cd. List $p_1(\alpha)$ begins at *mem* location $p1offset + p_1(\alpha)$; its current pointer is at *mem* location $p1offset + m^4 + p_1(\alpha)$; and α's location in the list is maintained in *mem* location $p1offset - m^4 + \alpha$. Similar conventions apply to the other five lists, whose offsets are $p2offset$, ..., $s3offset$.

```
#define p1(x)  (o, hexconv[(x) & #f000])
#define p2(x)  (o, hexconv[(x) & #ff00])
#define p3(x)  (o, hexconv[(x) & #fff0])
#define s1(x)  (o, hexconv[((x) & #000f) << 12])
#define s2(x)  (o, hexconv[((x) & #00ff) << 8])
#define s3(x)  (o, hexconv[((x) & #0fff) << 4])
```

⟨ Create empty prefix and suffix lists 12 ⟩ ≡

```
for (a = 0; a < m; a++) {
  o, mem[p1offset + mmmm + a*m*m*m] = p1offset + a*m*m*m;
  o, mem[s1offset + mmmm + a*m*m*m] = s1offset + a*m*m*m;
  for (b = 0; b < m; b++) {
    o, mem[p2offset + mmmm + (a*m+b)*m*m] = p2offset + (a*m+b)*m*m;
    o, mem[s2offset + mmmm + (a*m+b)*m*m] = s2offset + (a*m+b)*m*m;
    for (c = 0; c < m; c++) {
      o, mem[p3offset + mmmm + ((a*m+b)*m+c)*m] = p3offset + ((a*m+b)*m+c)*m;
      o, mem[s3offset + mmmm + ((a*m+b)*m+c)*m] = s3offset + ((a*m+b)*m+c)*m;
    }
  }
}
```

This code is used in section 17.

13. The following instructions serve to populate the lists at the beginning. Here *alpha* is the m-ary number, and x is its hexadecimal representation; *del* is a truncated version of *alpha*, identifying where the list begins within its offset region.

```
#define insert(x, alpha, del, off)
        {
          ploc = (del) + (off);      /* the list begins here */
          o, p = mem[ploc + mmmm];   /* p is its current pointer */
          o, mem[p] = x;             /* we put the hex form into the list */
          o, mem[(off) - mmmm + alpha] = p;   /* and remember it in the inverse list */
          o, mem[ploc + mmmm] = p + 1;        /* and update the list size */
        }
```

⟨ Put *alpha* into all of its prefix and suffix lists 13 ⟩ ≡

```
insert(x, alpha, p1(x), p1offset);
insert(x, alpha, p2(x), p2offset);
insert(x, alpha, p3(x), p3offset);
insert(x, alpha, s1(x), s1offset);
insert(x, alpha, s2(x), s2offset);
insert(x, alpha, s3(x), s3offset);
```

This code is used in section 16.

14. The cyclic equivalence classes of words are given identification numbers from 0 to $(m^4 - m^2)/4 - 1$. The number of α's class is precomputed in the *cl* table.

Every blue word also appears in the list of all currently blue words in its class. That list, for class k, begins at *mem* location $cloffset + 4k$. Its pointer is in *mem* location $cloffset + m^4 + 4k$, and α's position in that list is maintained in *mem* location $cloffset - m^4 + \alpha$.

```
⟨ Global variables 6 ⟩ +≡
  int mmmm;      /* m^4 */
  int maxl;      /* (m^4 - m^2)/4 */
  int p1offset, p2offset, p3offset, s1offset, s2offset, s3offset, cloffset;
  char aa[5];      /* the working array for Algorithm 7.2.1.1F */
  int cl[maxmmmm];      /* class identification numbers */
  int clrep[maxclasses];      /* class representatives (as hexadecimal numbers) */
  int classes;      /* the number of classes found so far */
```

15. Here I use Algorithm 7.2.1.1F to find the prime binary strings.

```
⟨ Compute the classes and initialize their lists 15 ⟩ ≡
f1: o, aa[0] = -1, j = 1, classes = 0;
f2: if (j ≡ 4) ⟨ Visit the prime string a_1 ... a_4 16 ⟩;
f3: for (j = 4; o, aa[j] ≡ m - 1; j--) ;
f4: if (j) {
    o, aa[j]++;
  f5: for (k = j + 1; k ≤ 4; k++) oo, aa[k] = aa[k - j];
    goto f2;
  }
  if (classes ≠ maxl) confusion("classes");
```

This code is used in section 17.

16.

```
⟨ Visit the prime string a_1 ... a_4 16 ⟩ ≡
  {
    for (i = 0; i < 4; i++) {
      for (x = 0, k = 0; k < 4; k++) x = (x ≪ 4) + aa[1 + ((i + k) % 4)];
      o, alpha = hexconv[x];
      if (i ≡ 0) clrep[classes] = x;
      if (x ≠ #0100 ∧ x ≠ #1000) {      /* see below */
        o, mem[alpha] = blue;
        o, mem[i + 4 * classes + cloffset] = x;
        o, mem[cloffset - mmmm + alpha] = i + 4 * classes + cloffset;
        ⟨ Put alpha into all of its prefix and suffix lists 13 ⟩;
      }
      o, cl[alpha] = classes;
    }
    o, mem[cloffset + mmmm + 4 * classes] = cloffset + 4 * classes + (classes ? 4 : 2);
    classes++;
  }
```

This code is used in section 15.

17. Notice that we've already allocated $22m^4$ cells of *mem*. Fortunately that's only 52822 when $m = 7$, comfortably less than 65536.

```
⟨ Initialize the tables 10 ⟩ +≡
  mmmm = m * m * m * m;
  for (k = 0; k < mmmm; k++) o, mem[k] = red;
  maxl = (mmmm - m * m) ≫ 2;
  if (goal < maxl - m * (m - 1) ∨ goal > maxl) {
    fprintf(stderr, "The␣goal␣should␣be␣between␣%d␣and␣%d,␣not␣%d!\n", maxl - m * (m - 1), maxl,
        goal);
    exit(-3);
  }
  p1offset = 2 * mmmm, p2offset = 5 * mmmm, p3offset = 8 * mmmm;
  s1offset = 11 * mmmm, s2offset = 14 * mmmm, s3offset = 17 * mmmm;
  cloffset = 20 * mmmm;
  ⟨ Create empty prefix and suffix lists 12 ⟩;
  ⟨ Compute the classes and initialize their lists 15 ⟩;
```

18. Diagnostic information is helped by a symbolic indication of each list name.

```
⟨ Subroutines 4 ⟩ +≡
  void decode_list(int p)
  {
    register int x, alpha;
    alpha = p % mmmm, x = mconv[alpha];
    switch (p/mmmm) {
    case 2: fprintf(stderr, "%x...", x ≫ 12); break;
    case 5: fprintf(stderr, "%02x..", x ≫ 8); break;
    case 8: fprintf(stderr, "%03x.", x ≫ 4); break;
    case 11: fprintf(stderr, "...%x", x ≫ 12); break;
    case 14: fprintf(stderr, "..%02x", x ≫ 8); break;
    case 17: fprintf(stderr, ".%03x", x ≫ 4); break;
    case 20: fprintf(stderr, "[%04x]", clrep[alpha/4]); break;
    default: fprintf(stderr, "???");
    }
  }
```

19. We will actually maintain all of these lists only when none of the list elements is green. As soon as a green element appears in a list that begins at location p, we set the corresponding pointer to $p + m^4 - 1$, as if the list length were -1, and call the list "closed." That pointer value serves as a sentinel to tell us that we needn't update the list any further.

```
⟨ Subroutines 4 ⟩ +≡
  void print_list(int p)
  {
    register int q, r;
    q = mem[p + mmmm];
    fprintf(stderr, "List␣");
    decode_list(p);
    fprintf(stderr, ":");
    if (q < p) fprintf(stderr, "␣.");
    else
      for (r = p; r < q; r++) fprintf(stderr, "␣%04x", mem[r]);
    fprintf(stderr, "\n");
  }
```

20.

```
⟨ Subroutines 4 ⟩ +≡
  void print_all_lists(void)
  {
    register int a, b, c, k;
    for (a = 0; a < m; a++) {
      print_list(p1offset + a * m * m * m);
      print_list(s1offset + a * m * m * m);
      for (b = 0; b < m; b++) {
        print_list(p2offset + (a * m + b) * m * m);
        print_list(s2offset + (a * m + b) * m * m);
        for (c = 0; c < m; c++) {
          print_list(p3offset + ((a * m + b) * m + c) * m);
          print_list(s3offset + ((a * m + b) * m + c) * m);
        }
      }
    }
    for (k = 0; k < maxl; k++) print_list(cloffset + 4 * k);
  }
```

21. Sanity checking. Whenever data structures are highly redundant, we want to monitor them carefully until we're confident that our program isn't messing them up. The *sanity* routine is intended to nip bugs in the bud.

```
#define sanity_checking 0 /* set this to 1 if you suspect a bug */
⟨ Subroutines 4 ⟩ +≡
  void sanity(void)
  {
    register int a, b, c, j, k, p, q, r, x, alpha;
    ⟨ Check that the relevant lists contain only blue words 22 ⟩;
    ⟨ Check that all blue words appear in all relevant lists 23 ⟩;
  }
```

22. Here we check also the inverse property of the inverse lists.

```
#define bluecheck(loc, off)
        {
          for (p = loc + off, q = mem[p + mmmm]; p < q; p++) {
            x = mem[p], alpha = hexconv[x];
            if (mem[alpha] ≠ blue) fprintf(stderr, "␣%04x␣in␣list␣%d␣is␣%s!\n", x, loc + off,
                mem[alpha] ≡ green ? "green" : "red");
            if (mem[off − mmmm + alpha] ≠ p)
              fprintf(stderr, "␣inverse␣index␣for␣%04x␣in␣list␣%d␣should␣be␣%d,␣not␣%d!\n", x,
                  loc + off, p, mem[off − mmmm + alpha]);
          }
        }
⟨ Check that the relevant lists contain only blue words 22 ⟩ ≡
  for (a = 0; a < m; a++) {
    bluecheck(a * m * m * m, p1offset);
    bluecheck(a * m * m * m, s1offset);
    for (b = 0; b < m; b++) {
      bluecheck((a * m + b) * m * m, p2offset);
      bluecheck((a * m + b) * m * m, s2offset);
      for (c = 0; c < m; c++) {
        bluecheck(((a * m + b) * m + c) * m, p3offset);
        bluecheck(((a * m + b) * m + c) * m, s3offset);
      }
    }
  }
  for (k = 0; k < maxl; k++) bluecheck(4 * k, cloffset);
```

This code is used in section 21.

23. Here we check green words also, to make sure that their lists have been "closed."

```
#define appcheck(x, alpha, del, off)
        {
          p = (del) + (off);      /* the list starts here */
          q = mem[p + mmmm];      /* the list currently ends here */
          if (q ≠ p − 1) {      /* it's not closed */
            if (mem[alpha] ≡ green) {
              fprintf(stderr, "␣list␣"); decode_list(p);
              fprintf(stderr, "␣for␣%x␣isn't␣closed!\n", x);
            } else {
              r = mem[off − mmmm + alpha];      /* where x supposedly is */
              if (r < p ∨ r ≥ q) {
                fprintf(stderr, "␣%x␣isn't␣in␣list␣", x);
                decode_list(p); fprintf(stderr, "!\n");
              }
            }
          }
        }
```

⟨ Check that all blue words appear in all relevant lists 23 ⟩ ≡

```
  for (alpha = 0; alpha < mmmm; alpha++)
    if (mem[alpha] ≠ red) {
      x = mconv[alpha];
      appcheck(x, alpha, p1(x), p1offset);
      appcheck(x, alpha, p2(x), p2offset);
      appcheck(x, alpha, p3(x), p3offset);
      appcheck(x, alpha, s1(x), s1offset);
      appcheck(x, alpha, s2(x), s2offset);
      appcheck(x, alpha, s3(x), s3offset);
      appcheck(x, alpha, 4 ∗ cl[alpha], cloffset);
    }
```

This code is used in section 21.

24. The poison list. We also need to maintain another interesting data structure, which remembers forbidden suffix-prefix pairs. Whenever we've chosen *abcd* to be a codeword, we want to remember that no future codeword that ends with *a* can be simultaneously present with a codeword that begins with *bcd*. The first time either of those events happens, we want to redden all of the blue words on the other side. So we'll add the item '*a*/*bcd*' to the current poison list when *abcd* becomes green.

The suffix-prefix pairs *ab*/*cd* and *abc*/*d* are forbidden in the same way.

The poison list is kept within *mem*, beginning at location *poison*; its pointer is in *mem* location *poison* − 1. Each item on this list occupies two consecutive *mem* positions, one pointing a the suffix and the other to the corresponding prefix.

An item will be removed from the poison list whenever we discover that one or both of the lists that it points to has become empty.

```
⟨ Global variables 6 ⟩ +≡
  int poison;
```

25.
```
⟨ Initialize the tables 10 ⟩ +≡
  poison = 22 * mmmm + 1;
  mem[poison − 1] = poison;      /* empty list */
```

26. When the following code is performed, *pp* will point to the current end of the poison list.

```
⟨ Delete entry p from the poison list 26 ⟩ ≡
  pp −= 2;
  if (p ≠ pp) {
    o, store(p, mem[pp]);
    store(p + 1, mem[pp + 1]);      /* mem[pp] and mem[pp + 1] are in the same octabyte */
  }
```

This code is used in sections 39 and 40.

27.
```
⟨ Append y and z to the poison list 27 ⟩ ≡
  store(pp, y), store(pp + 1, z), mems−−;
  pp += 2;
  if (pp > maxpoison) maxpoison = pp;      /* the poison list can't overflow */
```

This code is used in section 38.

28. Lists outside of *mem*. Some of the state information is sufficiently static and well-behaved that we needn't keep it in *mem*. For example, at every level we have a "frame" of vital data, telling for instance what class was involved, what word was chosen in that class, and how big the undo stack was when our data structures were stable.

```
⟨ Type definitions 28 ⟩ ≡
  typedef struct frame_struct {
    int cls;      /* the class number involved at this level */
    int x;      /* the word of that class, or −1 if this class not included */
    int slack;      /* how many classes are we still allowed to omit? */
    int uptr;      /* how many items should we retain on undo after backtracking? */
  } frame;
```

This code is used in section 1.

29. We also maintain a list called *clfree*, of the classes not yet chosen. It will contain $maxl - l$ items when we're on level l; but I maintain the redundant variable *freeptr*, so that I don't have to worry about transient moments as l is changing. There's also an inverse list *ifree*.

```
⟨ Global variables 6 ⟩ +≡
  frame move[maxclasses + 1];      /* basic info at each level */
  int clfree[maxclasses];      /* which classes are free? */
  int freeptr;      /* the number of free classes */
  int ifree[maxclasses];      /* where is this class in free? */
```

30.
```
⟨ Initialize the tables 10 ⟩ +≡
  for (k = 0, freeptr = maxl; k < maxl; k++) oo, clfree[k] = ifree[k] = k;
```

31.
```
⟨ Subroutines 4 ⟩ +≡
  void print_moves(int lev)
  {
    register int l;
    for (l = 1; l ≤ lev; l++)
      if (move[l].x ≥ 0) printf("␣%04x", move[l].x);
      else printf("␣(%04x)", clrep[move[l].cls]);
    printf("\n");
  }
```

32. Doing it. OK, we're (hopefully) ready now for the big climax.

```
⟨ Backtrack through all solutions 32 ⟩ ≡
  l = 1;
  cls = 0, x = #0001, slack = maxl − goal;
  nodes = profile[1] = 1;
  uptr = 0;
  goto tryit;
enter: profile[l]++, nodes++;
  ⟨ Report the current profile, if mems ≥ thresh 33 ⟩;
  if (l > maxl) ⟨ Report a solution and goto backup 46 ⟩;
choose: if (sanity_checking) sanity( );
  ⟨ Choose a potential move x 34 ⟩;
tryit: if (vbose > 1) {
    if (x ≥ 0) fprintf(stderr, "Level␣%d,␣trying␣%04x␣(%lld␣mems)\n", l, x, mems);
    else fprintf(stderr, "Level␣%d,␣trying␣(%04x)␣(%lld␣mems)\n", l, clrep[cls], mems);
  }
  o, move[l].uptr = uptr; ⟨ Bump curstamp 7 ⟩;
  if (x ≥ 0) ⟨ Update the data structures for new codeword x 36 ⟩
  else if (slack ≡ 0 ∨ l ≡ 1) goto backup;      /* disallow slack move at root level */
  else slack−−;
  o, move[l].x = x, move[l].cls = cls;      /* both in same octabyte, so 1 mem */
  o, move[l].slack = slack;
  ⟨ Update the free list 44 ⟩;
  l++; goto enter;
try_again: o, unstore(move[l].uptr);
  if (vbose > 1) fprintf(stderr, "Level␣%d,␣forbidding␣%04x␣(%lld␣mems)\n", l, x, mems);
  ⟨ Bump curstamp 7 ⟩;
  ⟨ Update the data structures to forbid x 43 ⟩;
  goto choose;
backup: if (−−l) {
    o, x = move[l].x, cls = move[l].cls;
    ⟨ Downdate the free list 45 ⟩;
    if (x < 0) goto backup;      /* omitting a class was the last resort */
    slack = move[l].slack;      /* this mem is covered at try_again */
    goto try_again;
  }
```

This code is used in section 1.

33. ⟨ Report the current profile, if *mems* ≥ *thresh* 33 ⟩ ≡

```
  if (mems ≥ thresh) {
    thresh += 10000000000;
    fprintf(stderr, "After␣%lld␣mems:", mems);
    for (k = 2; k ≤ l; k++) fprintf(stderr, "␣%lld", profile[k]);
    fprintf(stderr, "\n");
  }
```

This code is used in section 32.

34. Two strategies are used to decide which possibility to explore next. First we try to find a free class that has the smallest remaining number of blue words. If that number is 0 or 1, we choose it.

Otherwise we look at the poison list, and look for a choice that is guaranteed to redden as many blue words as possible. If such a choice exceeds a certain threshold, we make it, under the assumption that this choice will soon be proved unsatisfactory, thus giving us a new red word at comparatively little cost.

(In a sense, our goal is to make words red, because we are done when enough of them have become red.)

The threshold is a compile-time parameter, because I'm going to be running this program only a few times.

```
#define sthresh 1 /* must be positive */
⟨ Choose a potential move x 34 ⟩ ≡
  ⟨ Find a class cls with fewest blue words r 35 ⟩;
  if (r > 1) {
    ⟨ Find x that maximizes the "kill" potential s 40 ⟩;
    if (s ≥ sthresh) {
      oo, cls = cl[hexconv[x]];
      goto done;
    }
  }
  if (r ≡ 0) x = −1;
  else o, x = mem[4 ∗ cls + cloffset];
  done:
```

This code is used in section 32.

35.

```
⟨ Find a class cls with fewest blue words r 35 ⟩ ≡
  if (vbose > 3) fprintf(stderr, "␣class␣sizes");
  for (r = 5, k = 0; k < freeptr; k++) {
    c = clfree[k];
    o, j = mem[4 ∗ c + cloffset + mmmm] − (4 ∗ c + cloffset);
    if (vbose > 3) fprintf(stderr, "␣%04x:%d", clrep[c], j);
    if (j < r) {
      r = j, cls = c;
      if (r ≡ 0) break;
    }
  }
  if (vbose > 3) fprintf(stderr, "\n");
```

This code is used in section 34.

36. I should perhaps have added the phrase 'or **goto** *try_again*' to the title of this module, because we might find a reason to reject x.

```
⟨ Update the data structures for new codeword x 36 ⟩ ≡
  {
    ⟨ Make x green and close the lists it's in 37 ⟩;
    o, pp = mem[poison − 1]; /* temporarily keep poison list size in a register */
    ⟨ Append x's innards to the poison list 38 ⟩;
    ⟨ Update the poison list, or goto try_again 39 ⟩;
    store(poison − 1, pp);
  }
```

This code is used in section 32.

37. #**define** *closelist*(*del*, *off*)
{
p = (*del*) + (*off*);
o, *q* = *mem*[*p* + *mmmm*];
if (*q* ≠ *p* − 1) *store*(*p* + *mmmm*, *p* − 1);
}

⟨ Make *x* green and close the lists it's in 37 ⟩ ≡
o, *alpha* = *hexconv*[*x*];
store(*alpha*, *green*);
closelist(*p1*(*x*), *p1offset*);
closelist(*p2*(*x*), *p2offset*);
closelist(*p3*(*x*), *p3offset*);
closelist(*s1*(*x*), *s1offset*);
closelist(*s2*(*x*), *s2offset*);
closelist(*s3*(*x*), *s3offset*);
o, *k* = *cl*[*alpha*];
closelist(4 ∗ *k*, *cloffset*);
for (*r* = *cloffset* + 4 ∗ *k*; *r* < *q*; *r*++)
if (*o*, *mem*[*r*] ≠ *x*) *redden*(*mem*[*r*]);
This code is used in section 36.

38. ⟨ Append *x*'s innards to the poison list 38 ⟩ ≡
y = *p1*(*x*) + *s1offset*;
z = *s3*(*x*) + *p3offset*;
⟨ Append *y* and *z* to the poison list 27 ⟩;
y = *p2*(*x*) + *s2offset*;
z = *s2*(*x*) + *p2offset*;
⟨ Append *y* and *z* to the poison list 27 ⟩;
y = *p3*(*x*) + *s3offset*;
z = *s1*(*x*) + *p1offset*;
⟨ Append *y* and *z* to the poison list 27 ⟩;
This code is used in section 36.

39. This section is more or less the heart of the entire program. Or maybe it's better described as the "nerve center." Anyway, it enforces the comma-free condition, when you translate all of this esoteric logic into common sense terms.

After an entry on the poison list causes words to become red, previous entries of that list might become deletable. We don't go back and delete them, however; we can do that later, when we traverse the poison list again.

When I first wrote this part, I thought the case $yy < y \wedge zz < z$ would be impossible. But I first saw it occur legitimately when codewords `0001` and `1011` had been chosen at levels 1 and 2, then `0011` was tried at level 3. In that case, lists `.001` and `1...` were both closed.

```
⟨ Update the poison list, or goto try_again 39 ⟩ ≡
  for (p = poison; p < pp; ) {
    o, y = mem[p], z = mem[p + 1];      /* y and z are pointers to lists */
    o, yy = mem[y + mmmm];
    if (yy ≡ y) goto delete;      /* delete poison list entry that cites an empty list */
    o, zz = mem[z + mmmm];
    if (zz ≡ z) goto delete;      /* delete poison list entry that cites an empty list */
    if (yy < y ∧ zz < z) goto try_again;
    if (yy > y ∧ zz > z) {
      p += 2; continue;
    }
    if (yy < y) {
      if (vbose > 3) {
        fprintf(stderr, "␣killing␣"); decode_list(z); fprintf(stderr, "\n");
      }
      store(z + mmmm, z);      /* make list z empty */
      for (r = z; r < zz; r++) o, redden(mem[r]);
    } else {
      if (vbose > 3) {
        fprintf(stderr, "␣killing␣"); decode_list(y); fprintf(stderr, "\n");
      }
      store(y + mmmm, y);      /* make list y empty */
      for (r = y; r < yy; r++) o, redden(mem[r]);
    }
  delete: ⟨ Delete entry p from the poison list 26 ⟩;
  }
```

This code is used in section 36.

40. Items in the poison list pair a suffix with a prefix, where the suffix list has $yy - y$ elements and the prefix class has $zz - z$ elements. The larger of these lists can be killed off by choosing x in the other list.

(Here's an unimplemented afterthought, which potentially could be more powerful; unfortunately I have no time to explore it: Suppose we've chosen a word *abcd*. Hence we will have made an entry for $***a$ and $bcd*$ in the poison list, and it will be nice if the $bcd*$ list is nonempty because any such word will wipe out all of $***a$. However, that latter list may *already* be empty, in which case we'll drop this item from the poison list. The new idea is that any item *bcdz* on list $bcd*$ will *also* wipe out the entire list $z***$, which might well be huge for at least one z. Similarly, any item *cdyz* on list $cd**$ will wipe out $yz**$; any item *zabc* on list $*abc$ will wipe out $***z$; etc. These six cases are not currently considered, because they aren't easily detected within the present data structures.)

```
⟨ Find x that maximizes the "kill" potential s 40 ⟩ ≡
  s = 0;
  if (vbose > 3) fprintf(stderr, "␣kills");
  o, pp = mem[poison − 1];      /* temporarily keep poison list size in a register */
  for (p = poison; p < pp; ) {
    o, y = mem[p], z = mem[p + 1];
    o, yy = mem[y + mmmm];
    if (yy ≡ y) goto deleet;
    o, zz = mem[z + mmmm];
    if (zz ≡ z) goto deleet;
    if (yy < y ∨ zz < z) confusion("poison␣remnant");
    if (vbose > 3) {
      fprintf(stderr, "␣%d", yy − y);
      decode_list(y);
      fprintf(stderr, "|");
      decode_list(z);
      fprintf(stderr, "%d", zz − z);
    }
    if (yy − y ≥ zz − z) {
      if (yy − y > s) o, s = yy − y, x = mem[z];
    } else if (zz − z > s) o, s = zz − z, x = mem[y];
    p += 2;
    continue;
  deleet: ⟨ Delete entry p from the poison list 26 ⟩;
  }
  if (vbose > 3) fprintf(stderr, "\n");
  store(poison − 1, pp);
```

This code is cited in section 1.

This code is used in section 34.

41. Here I want to mention some code that was deleted from an earlier version, because it introduced a sneaky bug. I'd once had a mechanism in the **frame** structure designed to break symmetry by confining solutions to "restricted growth sequences."

Namely, I had originally said this:

> One of the frame elements, called *rgs*, is used to avoid repeating choices that are equivalent under symmetry. The term '*rgs*' stands for "restricted growth string"; it names the largest digit seen so far among the chosen codewords. We shall insist that the codeword for level 1 is either `0001` or `0010`. Henceforth the next digit greater than `1` must be `2`, and the next digit greater than `2` must be `3`, etc.
>
> (Subtle(?) point: If the class is, say, `0312` and $rgs = 1$, we will allow only the representatives `1203` and `2031`, not `0312` or `3120`, because `2` must precede `3`.)

And the code below was activated just before making x green, if *rgs* was $m - 2$ or less.

However, I'd missed an even more subtle point. This *rgs* logic must be done earlier, in the move-choosing sections: Those sections should not choose an x that will be rejected on *rgs* grounds. For if they do, x will be permanently blacklisted (actually redlisted) even after the *rgs* condition is no longer relevant. Instead, when choosing an x on the basis of a killing strategy, one must find an x that is *rgs*-legitimate. And when choosing an element of a class with few blue elements, one must be sure that the element is *rgs*-safe.

I could have corrected the logic. But the program would have been quite a lot more complicated, and I already knew from experience that I wouldn't be able to solve the case $m = 5$ even with the faulty strategy that eliminated too much.

Furthermore, the best way to break the symmetry in hard cases is probably to prespecify the first few levels of the search tree.

So I've opted to record here the lesson that I learned, but *not* to use any form of the following rejected-rejection code:

```
⟨ Reject x if it violates the restricted growth condition 41 ⟩ ≡
  {
    for (i = 12; i ≥ 0; i -= 4) {
      y = (x ≫ i) & #f;
      if (y > rgs) {
        if (y > rgs + 1) goto try_again;
        rgs = y;
      }
    }
  }
```

42. **#define** *rem*(*alpha*, *del*, *off*)
```
{
  p = (del) + (off);
  o, q = mem[p + mmmm] − 1;
  if (q ≥ p) {      /* list is nonempty */
    store(p + mmmm, q);      /* shorten the list */
    o, t = mem[(off) − mmmm + alpha];      /* where does x appear? */
    if (t ≠ q) {
      o, y = mem[q];      /* get item to move */
      store(t, y);      /* put y in place of x */
      o, store((off) − mmmm + hexconv[y], t);      /* update the inverse index */
    }
  }
}
```
⟨ Subroutines 4 ⟩ +≡
```
inline void redden(register int x)
{
  register int alpha, p, q, t, y;
  if (vbose > 2) fprintf(stderr, "␣red␣%04x\n", x);
  o, alpha = hexconv[x];
  if (mem[alpha] ≠ blue) confusion("redden");
  store(alpha, red);
  rem(alpha, p1(x), p1offset);      /* remove x from its p1 list */
  rem(alpha, p2(x), p2offset);      /* remove x from its p2 list */
  rem(alpha, p3(x), p3offset);      /* remove x from its p3 list */
  rem(alpha, s1(x), s1offset);      /* remove x from its s1 list */
  rem(alpha, s2(x), s2offset);      /* remove x from its s2 list */
  rem(alpha, s3(x), s3offset);      /* remove x from its s3 list */
  o, t = cl[alpha];
  rem(alpha, 4 ∗ t, cloffset);      /* remove x from its class list */
}
```

43. ⟨ Update the data structures to forbid *x* 43 ⟩ ≡
```
redden(x);
```
This code is used in section 32.

44. ⟨ Update the free list 44 ⟩ ≡
```
freeptr−−;
o, p = ifree[cls];
if (p ≠ freeptr) {
  o, y = clfree[freeptr];
  o, clfree[p] = y;
  o, ifree[y] = p;
  o, clfree[freeptr] = cls;
  o, ifree[cls] = freeptr;
}
```
This code is used in section 32.

45. ⟨ Downdate the free list 45 ⟩ ≡
```
freeptr++;
```
This code is used in section 32.

46. And we close with a happy moment.

```
⟨ Report a solution and goto backup 46 ⟩ ≡
  {
    count++;
    printf("%d:", count);
    print_moves(maxl);
    goto backup;
  }
```

This code is used in section 32.

47. Index.

BACK-COMMAFREE4