**1. Introduction.** This program finds all nonisomorphic sets of SET cards that contain no SETs.

In case you don't know what that means, a SET card is a vector $(c_1, c_2, c_3, c_4)$ where each $c_i$ is 0, 1, or 2. Thus there are 81 possible SET cards. A SET is a set of three SET cards that sums to $(0, 0, 0, 0)$ modulo 3. Equivalently, the numbers in each coordinate position of the three vectors in a SET are either all the same or all different. (It's kind of a 4-dimensional tic-tac-toe with wraparound.)

My previous SETSET program considered only isomorphisms that apply directly to the semantics of the game of SET, namely permutations of coordinates and permutations of individual coordinate positions; there are $4! \times 3!^4 = 31104$ of those. But now I want to consider the much larger collection of all isomorphisms that preserve SETs. There are $81 \cdot (81-1) \cdot (81-3) \cdot (81-9) \cdot (81-27) = 1{,}965{,}150{,}720$ of these; thus the method used in the previous program, which had complexity $\Omega$(number of isomorphisms) in both time and space, would be quite inappropriate. Here I'm using a possibly new method, with space requirement only $O(\text{number of elements})^2 D$, where $D$ bounds the partial transitivity of the isomorphism group: The image of the first $D$ elements is sufficient to determine the images of all. In our case, $D = 5$.

A web page of David Van Brink states that you can't have more than 20 SET cards without having a SET. He says that he proved this in 1997 with a computer program that took about one week to run on a 90MHz Pentium machine. I'm hoping to get the result faster by using ideas of isomorph rejection, meanwhile also discovering all of the $k$-element SET-less solutions for $k \le 20$.

The theorem about at most 20 SET-free cards was actually proved in much stronger form by G. Pellegrino, *Matematiche* **25** (1971), 149–157, without using computers. Pellegrino showed that any set of 21 points in the projective space of $81 + 27 + 9 + 3 + 1$ elements, represented by nonzero 5-tuples in which $x$ and $-x$ are considered equivalent, has three collinear points; this would correspond to sets of three distinct points in which the third is the sum or difference of the first two.

Incidentally, I've written this program for my own instruction, not for publication. I still haven't had time to read the highly relevant papers by Adalbert Kerber, Reinhard Laue, and their colleagues at Bayreuth, although I've had those works in my files for many years. Members of that group probably are quite familiar with equivalent or better methods. Perhaps I'm being foolish, but I thought it would be most educational to try my own hand before looking at other people's solutions. I seem to learn a new subject best when I try to write code for it, because the computer is such a demanding, unbluffable taskmaster.

[SET is a registered trademark of SET Enterprises, Inc.]

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
  jmp_buf restart_point;
  ⟨ Type definitions 5 ⟩
  ⟨ Global variables 6 ⟩
  ⟨ Subroutines 22 ⟩
  main( )
  {
    ⟨ Local variables 40 ⟩
    ⟨ Initialize 8 ⟩;
    ⟨ Enumerate and print all solutions 39 ⟩;
    ⟨ Print the totals 47 ⟩;
  }
```

**2.** Our basic approach is to define a linear ordering on solutions, and to look only for solutions that are smallest in their isomorphism class. In other words, we will count the sets $S$ such that $S \le \alpha S$ for all automorphisms $\alpha$. We'll also count the number $t$ of cases where $S = \alpha S$; then the number of distinct solutions isomorphic to $S$ is $1965150720/t$, so we will essentially have also enumerated the distinct solutions.

The ordering we use is almost standard: Vectors are ordered by weight, and vectors of equal weight are ordered lexicographically; thus the sequence is $(0,0,0,0)$, $(0,0,0,1)$, $(0,0,1,0)$, $(0,1,0,0)$, $(1,0,0,0)$, $(0,0,0,2)$, $(0,0,1,1)$, $\ldots$, $(2,2,2,2)$. Also, when $S$ and $T$ are both sets of $k$ SET cards, we define $S \le T$ by first sorting the vectors into order so that $s_1 < \cdots < s_k$ and $t_1 < \cdots < t_k$, then we compare $(s_1, \ldots, s_k)$ lexicographically to $(t_1, \ldots, t_k)$. (Equivalently, we compare the smallest elements of $S$ and $T$; if they are equal, we compare the second-smallest elements, and so on, until we've either found inequality or established that $S = T$.)

**3.** The automorphisms can be thought of as the collection of all linear mappings that take $x \mapsto Ax + b$, where $x$ is the column vector $(c_1, c_2, c_3, c_4)^T$, $A$ is a nonsingular $4 \times 4$ matrix (mod 3), and $b$ is an arbitrary vector. Alternatively we can think of the mapping $x \mapsto Ax$, where each SET card $x$ is a column vector of the form $(c_1, c_2, c_3, c_4, 1)^T$ and $A$ is a nonsingular $5 \times 5$ matrix whose bottom row is $(0,0,0,0,1)$. In either case we have the so-called "affine linear group" in 4-space (mod 3).

For example, any set of five points that are *independent*, in the sense that none is in the subspace spanned by the other four, can be mapped into the smallest five cards

$$\{(0,0,0,0),\ (0,0,0,1),\ (0,0,1,0),\ (0,1,0,0), (1,0,0,0)\}.$$

The set consisting of the smallest four cards has $24 \cdot 54 = 1296$ automorphisms, hence there are exactly $1965150720/1296 = 1516320$ ways to choose four SET cards that are independent in this sense. All other SET-less sets of four cards are isomorphic to the dependent set

$$\{(0,0,0,0),\ (0,0,0,1),\ (0,0,1,0),\ (0,0,1,1)\};$$

and this set has 31104 automorphisms, leading to an additional $1965150720/31104 = 63180$ ways to choose a set of four SET-less cards. This explains why the total number of such sets is $1516320 + 63180 = 1579500$, a number that the SETSET program was able to compute only after it had laboriously considered 128 cases that were nonisomorphic in the previous setup.

**4.** We will generate the elements of a $k$-set in order. If we have $s_1 < \cdots < s_k$ and $\{s_1, \ldots, s_k\} \le \{\alpha s_1, \ldots, \alpha s_k\}$ for all $\alpha$, it is not hard to prove that $\{s_1, \ldots, s_j\} \le \{\alpha s_1, \ldots, \alpha s_j\}$ for all $\alpha$ and $1 \le j \le k$. (The reason is that $S < T$ and $t \ge \max T$ implies $S \cup \{s\} < S \cup \{\infty\} < T \cup \{t\}$, for all $s$.) Therefore every canonical $k$-set is obtained by extending a unique canonical $(k-1)$-set.

**5. Permutations.** It's convenient to represent SET card vectors in a compact code, as an integer between 0 and 80.

```
⟨ Type definitions 5 ⟩ ≡
  typedef char SETcard;      /* a SET card (c1, c2, c3, c4) represented as encode[(c1c2c3c4)3] */
```

See also section 20.

This code is used in section 1.

**6.** Lexicographic order would correspond to ternary notation, but our weight-first ordering is slightly different. Here we specify the card ranks in the desired order.

```
#define nn 81      /* the total number of elements permuted by automorphisms */
#define nnn 128    /* the value of nn rounded up to a power of 2, for efficiency */
⟨ Global variables 6 ⟩ ≡
  SETcard encode[nn] = {0, 1, 5, 2, 6, 15, 7, 16, 31,
      3, 8, 17, 9, 18, 32, 19, 33, 50,
      10, 20, 34, 21, 35, 51, 36, 52, 66,
      4, 11, 22, 12, 23, 37, 24, 38, 53,
      13, 25, 39, 26, 40, 54, 41, 55, 67,
      27, 42, 56, 43, 57, 68, 58, 69, 76,
      14, 28, 44, 29, 45, 59, 46, 60, 70,
      30, 47, 61, 48, 62, 71, 63, 72, 77,
      49, 64, 73, 65, 74, 78, 75, 79, 80};
```

See also sections 7, 9, 11, 15, 16, 21, 23, 28, 41, and 46.

This code is used in section 1.

**7.** When we output a SET card, however, we prefer a decimal code.

```
⟨ Global variables 6 ⟩ +≡
  int decimalform[nn] = {0, 1, 2, 10, 11, 12, 20, 21, 22,
      100, 101, 102, 110, 111, 112, 120, 121, 122,
      200, 201, 202, 210, 211, 212, 220, 221, 222,
      1000, 1001, 1002, 1010, 1011, 1012, 1020, 1021, 1022,
      1100, 1101, 1102, 1110, 1111, 1112, 1120, 1121, 1122,
      1200, 1201, 1202, 1210, 1211, 1212, 1220, 1221, 1222,
      2000, 2001, 2002, 2010, 2011, 2012, 2020, 2021, 2022,
      2100, 2101, 2102, 2110, 2111, 2112, 2120, 2121, 2122,
      2200, 2201, 2202, 2210, 2211, 2212, 2220, 2221, 2222};
  int decode[nn];
```

**8.**

```
⟨ Initialize 8 ⟩ ≡
  for (k = 0; k < nn; k++) decode[encode[k]] = decimalform[k];
```

See also sections 10, 12, and 17.

This code is used in section 1.

**9.** We will frequently need to find the third card of a SET, given any two distinct cards $x$ and $y$, so we store the answers in a precomputed table.

```
⟨ Global variables 6 ⟩ +≡
  char z[3][3] = {{0, 2, 1}, {2, 1, 0}, {1, 0, 2}};      /* x + y + z ≡ 0 (mod 3) */
  char third[nn][nnn];
```

**10.** #**define** *pack*$(a,b,c,d)$ *encode*$[(((a)*3+(b))*3+(c))*3+(d)]$

⟨ Initialize 8 ⟩ +≡

```
{
  int a, b, c, d, e, f, g, h;
  for (a = 0; a < 3; a++)
    for (b = 0; b < 3; b++)
      for (c = 0; c < 3; c++)
        for (d = 0; d < 3; d++)
          for (e = 0; e < 3; e++)
            for (f = 0; f < 3; f++)
              for (g = 0; g < 3; g++)
                for (h = 0; h < 3; h++)
                  third[pack(a,b,c,d)][pack(e,f,g,h)] = pack(z[a][e], z[b][f], z[c][g], z[d][h]);
}
```

**11.** The set of automorphisms is conveniently represented by a mapping table, as in the author's paper "Efficient representation of perm groups," *Combinatorica* **11** (1991), 33–43. If there is an $\alpha$ such that $\alpha k = j$ and $\alpha$ fixes all elements $< j$, we let $perm[j][k]$ be one such permutation $\alpha$, represented as an array of 81 elements. In particular, $perm[j][j]$ always is the identity permutation. If no such $\alpha$ exists, however, we set $perm[j][k][0] = -1$.

Our algorithm for finding nonisomorphic SET-less sets is based entirely on the group of isomorphisms defined by a *perm* table. If *perm* is initialized to the definition of some other group, the rest of this program should need no further modification except for counting the total number of automorphisms. (Of course, not every *perm* table defines a group of permutations; the set of all possible products $\pi_0\pi_1\ldots\pi_{80}$, where each $\pi_j$ is one of the permutations $perm[j][k]$ for some $k \ge j$, must be closed under multiplication. We assume that this condition is satisfied.)

There is an integer $D$ such that $perm[j][k] = -1$ for all $D \le j < k$; in other words, each $\alpha$ that fixes 0, 1, $\ldots$, $D-1$ is the identity map. We needn't bother representing $perm[j][k]$ for $j \ge D$.

*Important Note [30 April 2001]:* No, the algorithm does *not* work for arbitrary permutation groups. I thank Prof. Reinhard Laue for pointing out a serious error. However, I do think the special group dealt with here is handled satisfactorily because of its highly transitive nature.

```
#define dd 5      /* in our case D = 5 */
```

⟨ Global variables 6 ⟩ +≡

```
char perm[dd][nnn][nnn];      /* mapping table */
```

**12.** Now let's set up the mapping table for the affine transformations we need. The basic idea is simple. For example, the group of all $5 \times 5$ matrices that fix $(0,0,0,0) = 0$ and $(0,0,0,1) = 1$ is the set of all nonsingular $A$ of the form

$$\begin{pmatrix} * & * & * & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix};$$

and the image of $(0,0,1,0)$ is the third column. For every possible third column $k$ (which must not be zero in the top three rows, lest the matrix be singular), we need to choose an appropriate setting of the first two columns. Then we get an affine mapping that takes $(0,0,1,0) = 2$ into $k$, and the inverse of this mapping can be stored in $perm[2][k]$ because it maps $k \mapsto 2$.

```
⟨ Initialize 8 ⟩ +≡
  aa[4][4] = 1;
  for (j = 0; j < 5; j++) {      /* we want to set up perm[4 − j] */
    int a, b, c, d, e, f, g, h, jj, kk;
    for (jj = j + 1; jj < 5; jj++)
      for (k = 0; k < 4; k++) aa[jj][k] = (k ≡ jj ? 1 : 0);
    for (a = 0; a < 3; a++)
      for (b = 0; b < 3; b++)
        for (c = 0; c < 3; c++)
          for (d = 0; d < 3; d++) {
            aa[j][0] = a, aa[j][1] = b, aa[j][2] = c, aa[j][3] = d;
            for (kk = j; kk ≥ 0; kk−−)
              if (aa[j][kk]) break;
            if (kk < 0) perm[4 − j][pack(a, b, c, d)][0] = −1;
            else {
              ⟨ Complete aa to a nonsingular matrix 13 ⟩;
              ⟨ Use aa to define the mapping perm[4 − j][pack(a, b, c, d)] 14 ⟩;
            }
          }
  }
```

**13.** ⟨ Complete $aa$ to a nonsingular matrix 13 ⟩ ≡

```
  for (jj = 0; jj < j; jj++) {
    for (k = 0; k < 4; k++) aa[jj][k] = 0;
    aa[jj][(jj + kk + 1) % (j + 1)] = 1;
  }
```

This code is used in section 12.

**14.** ⟨ Use $aa$ to define the mapping $perm[4-j][pack(a,b,c,d)]$ 14 ⟩ ≡

```
  kk = pack(a, b, c, d);
  for (e = 0; e < 3; e++)
    for (f = 0; f < 3; f++)
      for (g = 0; g < 3; g++)
        for (h = 0; h < 3; h++) {
          for (k = 0; k < 4; k++)
            trit[k] = (e ∗ aa[0][k] + f ∗ aa[1][k] + g ∗ aa[2][k] + h ∗ aa[3][k] + aa[4][k]) % 3;
          perm[4 − j][kk][pack(trit[0], trit[1], trit[2], trit[3])] = pack(e, f, g, h);
        }
```

This code is used in section 12.

**15.** ⟨ Global variables 6 ⟩ +≡
```
char trit[4];      /* four ternary digits */
char aa[5][8];     /* matrix A, stored columnwise */
```

**16.** The algorithm below also needs another table of permutations: For $j < D$ and $k \ge j$, we let $minp[j][k]$ be a permutation that takes $k$ into the smallest possible value, among all permutations that fix all elements less than $j$.

⟨ Global variables 6 ⟩ +≡
```
char minp[dd][nnn][nnn + nnn];
```

**17.** The *minp* table is readily built from the *perm* table, working from bottom to top. (In the affine linear group under the ordering we have chosen, *minp* actually is the same as *perm* if we plug the identity in place of empty permutations. But this code tries to be general.)

We store the inverse permutation too: If $p = minp[j][k]$, then $p$ maps $v$ to $p[v]$ and $p[v + nnn]$ to $v$.

⟨ Initialize 8 ⟩ +≡
```
for (j = dd − 1; j ≥ 0; j−−)
  for (k = j; k < nn; k++) {
    if (perm[j][k][0] ≠ −1) {      /* the best we can do is obviously k ↦ j */
      for (l = 0; l < nn; l++) minp[j][k][l] = perm[j][k][l];
    } else if (j ≡ dd − 1) {
      for (l = 0; l < nn; l++) minp[j][k][l] = l;
      for (i = j + 1; i < nn; i++)
        if (perm[j][i][0] ≠ −1 ∧ perm[j][i][k] < minp[j][k][k])
          for (l = 0; l < nn; l++) minp[j][k][l] = perm[j][i][l];
    } else {
      register int kk;
      for (l = 0; l < nn; l++) minp[j][k][l] = minp[j + 1][k][l];
      for (i = j + 1; i < nn; i++)
        if (perm[j][i][0] ≠ −1) {
          kk = perm[j][i][k];
          if (minp[j + 1][kk][kk] < minp[j][k][k])
            for (l = 0; l < nn; l++) minp[j][k][l] = minp[j + 1][kk][perm[j][i][l]];
        }
    }
    for (l = 0; l < nn; l++) minp[j][k][minp[j][k][l] + nnn] = l;
  }
```

**18.** The mapping table of a permutation group has many magical properties. For example, consider the digraph with arcs from $(j, x)$ to $(j+1, y)$ whenever there is a $k$ with $perm[j][k]$ mapping $x$ to $y$. This digraph has a path from $(j, x)$ to $(D, y)$ if and only if the group has a permutation that maps $x$ to $y$ and fixes all elements less than $j$; hence such a path exists if and only if there is also a path from $(j, y)$ to $(D, x)$.

Furthermore, if we let $S$ be the set of all elements $k$ such that $perm[0][k][0]$ is $-1$, namely the set of all elements that are not permutable into 0 (not in the "orbit" of 0), then the group never maps an element of $S$ to an element not in $S$. In other words, the group also induces a permutation group on the elements of $S$. We will make use of this property in the algorithm below.

**19. Data structures.** I'm going to try to describe the algorithm simultaneously as I explain its main data structures, because the two are somewhat intertwined.

Our main task is to check that a set $S = \{x_1, \ldots, x_l\}$, where $x_1 < \cdots < x_l$, cannot be mapped into a smaller set $\alpha S$, for any automorphism $\alpha$. This breaks down into subtasks where we consider cases in which $\alpha x_i$ is the smallest element of $\alpha S$. If we can map $x_i$ into a number less than $x_1$, we reject the set $S$. If we cannot map $x_i$ into any number $\le x_1$, we need not consider this subtask any further. But if there is an $\alpha$ such that $\alpha x_i = x_1$, we fix our attention on one such $\alpha$ and we reduce the problem to showing that the set $\alpha x_1, \ldots, \alpha x_{i-1}, \alpha x_{i+1}, \ldots, \alpha x_l\}$ cannot be mapped into a set smaller than $\{x_2, \ldots, x_l\}$ in the subgroup of automorphisms that fix all elements $\le x_1$. For this subproblem we first rule out all elements that could map into anything *between* $x_1$ and $x_2$; then we consider only the subgroup of automorphisms on the *remaining* elements. (Only one $\alpha$ is needed for each $x_i$; this is a key point, which is proved below.)

**20.** The data structure we use to support such an approach has one node $p$ for each subtask. If we are currently trying to see whether $\{y_1, \ldots, y_r\}$ can be mapped into a set smaller than $\{x_d, \ldots, x_l\}$, by automorphisms that fix all elements $\le x_{d-1}$, then $p$ will be on level $d$ of the current tree of tasks and subtasks, and $p\rightarrow val$ will be $y_i$ for some $i$.

The task tree is triply linked in the conventional way, with links *par*, *kid*, and *sib* for parent, child, and sibling, respectively. In other words, if $p$ is a subtask of $q$, we have $p\rightarrow par = q$, and the subtasks of $q$ (including $p$ itself) are respectively $q\rightarrow kid$, $q\rightarrow kid\rightarrow sib$, $q\rightarrow kid\rightarrow sib\rightarrow sib$, etc. The youngest child of a family, namely the child added most recently to the structure, is $q\rightarrow kid$, and the next youngest is $q\rightarrow kid\rightarrow sib$.

An additional field $p\rightarrow trans$, if non-null, is the automorphism $\alpha$ by which the values of $p$'s subtasks $\{\alpha y_1, \ldots, \alpha y_{i-1}, \alpha y_{i+1}, \ldots, \alpha y_r\}$ have been transformed. If $p\rightarrow trans$ is null it means that subtask $p$ is "dead" because $y_i$ cannot be transformed into anything $\le x_d$.

⟨ Type definitions 5 ⟩ +≡

```
typedef struct node_struct {
  SETcard val;      /* value that is assumed to be smallest after subsequent mapping */
  char level;       /* state information for terminal nodes, see below */
  struct node_struct *par, *kid, *sib;      /* pointers in triply linked tree */
  char *trans;      /* a permutation, or Λ */
} node;
```

**21.** Part of the algorithm is somewhat subtle and requires proof, so I shall try to state the inductive assumptions carefully. But I will state them only in a particular case, in order to keep the notation simple. The general case can easily be inferred from the special case considered here.

Suppose $q$ is a node at level 2 that corresponds to the subtask for permutations that map $x_3 \mapsto x_1$ and $x_5 \mapsto x_2$. Then we have found particular automorphisms $\alpha_1$ and $\alpha_2$ such that $\alpha_1 x_3 = x_1$ and $\alpha_2\alpha_1 x_5 = x_2$, where $\alpha_2$ fixes all elements $\le x_1$. (Automorphism $\alpha_2$ will be stored in $q$→*trans* and $\alpha_1$ will be in $q$→*par*→*trans*; $x_3$ will be $q$→*par*→*val*, and $q$→*val* will be $\alpha_1 x_5$. Also $q$→*sib*→*val* $= \alpha_1 x_4$, $q$→*sib*→*sib*→*val* $= \alpha_1 x_2$, $q$→*sib*→*sib*→*sib*→*val* $= \alpha_1 x_1$, and $q$→*sib*→*sib*→*sib*→*sib* $= \Lambda$.)

We assume that there is a set $X_q$ of "legal" elements that might be used to extend the current set $\{x_1, \ldots, x_l\}$. This set has the property that, if $\alpha$ is any automorphism with $\alpha x_3 = x_1$ and $\alpha x_5 = x_2$ and $\alpha x = y$ for some $y$, where $x \in X_q$, then there is an automorphism $\alpha'$ that fixes all elements $\le x_2$ and satisfies $\alpha'\alpha_2\alpha_1 x = y$. In other words, we assume that we can obtain the images of all legal elements that belong to our subtask by restricting consideration to automorphisms of the form $\alpha\alpha_2\alpha_1$, where $\alpha$ fixes all elements $\le x_2$ and where $\alpha_1$ and $\alpha_2$ are the particular automorphisms we have chosen..

Now one of the subtasks below $q$ will be node $p$, having $p$→*val* $= \alpha_2\alpha_1 x_4$ and $p$→*trans* $= \alpha_3$, where $\alpha_3$ is some automorphism that fixes everything $\le x_2$ and satisfies $\alpha_3\alpha_2\alpha_1 x_4 = x_3$. The set of legal elements $X_p$ is obtained by deleting from $X_q$ all elements $z$ such that $\alpha\alpha_2\alpha_1 z < x_3$ for some $\alpha$ fixing $\le x_2$.

To prove that $X_p$ satisfies the required inductive assumption, suppose $\alpha$ is any automorphism with $\alpha x_3 = x_1$, $\alpha x_5 = x_2$, $\alpha x_4 = x_3$, and $\alpha x = y$, where $x \in X_p$. Then since $x \in X_q$, there is $\alpha'$ fixing $\le x_2$ such that $\alpha'\alpha_2\alpha_1 x = y$. And by definition of $X_p$ we know that $y \ge x_3$; we can assume $x \ne x_4$, hence $y > x_3$. We can write $\alpha' = \beta\alpha''$, where $\beta$ is a product of $x_3 - x_2$ elements from the *perm* table (one from each row $j$ for $x_2 \le j < x_3$) and $\alpha''$ fixes all elements $\le x_3$.

Consider the group $G$ of all $\alpha$ that fix the elements $\le x_2$. Every permutation $\pi$ in $G$ is a product $\pi'\pi''$, where $\pi'$ permutes the elements that can map into $\{x_2, \ldots, x_3 - 1\}$ and $\pi''$ permutes those that can't. Thus we have $\beta = \beta'\beta''$ and $\alpha_3 = \alpha_3'\alpha_3''$. We also have $\pi\alpha_2\alpha_1 x = \pi''\alpha_2\alpha_1 x$ for any $\pi$ in $G$.

Now the permutation $\alpha''\beta''\alpha_3''^{-}$ fixes all elements $\le x_3$, and we have $\alpha''\beta''\alpha_3''^{-}\alpha_3\alpha_2\alpha_1 x = \alpha''\beta''\alpha_3'\alpha_2\alpha_1 x = \alpha''\beta''\alpha_2\alpha_1 x = \alpha''\beta''\beta'\alpha_2\alpha_1 x = \alpha'\alpha_2\alpha_1 x = y$.

*Important Note [30 April 2001]:* Oops no, that permutations does *not* necessarily fix $x_3$. I don't know at present how to repair this error without spoiling the efficiency of the algorithm.

Suppose $\{x_1, \ldots, x_l\}$ is non canonical because some $\alpha$ has $\alpha x_3 = x_1$, $\alpha x_5 = x_2$, $\alpha x_4 = x_3$, and $\alpha x_1 < x_4$. If $x_1$ is in $X_p$, we've proved that there must be $\alpha'$ fixing $\le x_3$ such that $\alpha\alpha_3\alpha_2\alpha_1 x_1 < x_4$; and $\alpha_3\alpha_2\alpha_1 x_1$ will be the value of one of $p$'s children, so we will discover the existence of $\alpha'$ by looking at the *minp* table. If $x_1$ is in $X_q$ but not $X_p$, there must be $\alpha'$ fixing $\le x_2$ such that $\alpha\alpha_2\alpha_1 x_1 < x_3$; the fact that $\{x_1, \ldots, x_l\}$ is noncanonical will be discovered on a sibling task $r$ of $p$, having $r$→*val* $= \alpha_2\alpha_1 x_1$. Similarly, if $x_1$ is not in $X_q$ there must be $\alpha'$ fixing $\le x_1$ such that $\alpha'\alpha_1 x_1 < x_2$, and a sibling task of $q$ will discover this.

That, for me, completes the proof. Readers who do not believe that I lived up to my promise of "keeping the notation simple" are encouraged to supply a nicer argument; I decided to use brute force here in order to familiarize myself with the underlying structure.

(Speaking of notation, I must admit to being unhappy today with my former choice, in SETSET, of writing $\alpha x$ instead of $x\alpha$ for the image of $x$ under a permutation $\alpha$. This has compelled me to write $\alpha_2\alpha_1$ for the permutation in which $\alpha_1$ is applied before $\alpha_2$, against my normal custom and preference. Certainly I'll use the other order if I ever write this up.)

⟨ Global variables 6 ⟩ +≡

```
char legal[nn + 1];      /* nonzero when a card is legal in all subtasks */
```

**22.** Nodes come and go in a last-in-first-out fashion, so we can allocate them sequentially.

```
#define max_node_count 22000000
⟨ Subroutines 22 ⟩ ≡
  node *new_node( )
  {
    register node *p = node_ptr++;
    if (p ≥ &nodes[max_node_count]) {
      fprintf(stderr, "Node␣memory␣overflow!\n");
      exit(−3);
    }
    p→kid = Λ;
    return p;
  }
```

See also sections 24, 25, 30, 34, 36, 37, 38, and 44.

This code is used in section 1.

**23.**

```
#define root &nodes[0]
⟨ Global variables 6 ⟩ +≡
  node nodes[max_node_count];
  node *node_ptr = &nodes[1];
```

**24.** When we're processing a set $\{x_1, \ldots, x_l\}$, every active node on level $d$ of the tree has $l - d$ children. Thus when $l$ increases by 1, every active node gains a child; this leads to an interesting recursive algorithm.

The *new_kid* procedure creates a new child of $p$ at level $d + 1$ with a given value $v$.

If $d$ is sufficiently deep, we switch to another strategy described below.

```
⟨ Subroutines 22 ⟩ +≡
  void launch(node *, int, node *);      /* prototype for a subroutine declared below */
  void new_terminal_kid(node *, SETcard);      /* and another */
  void new_kid(node *p, int d, SETcard v)
  {
    register node *q;
    ⟨ Use special strategy for new_kid if d is sufficiently deep 29 ⟩;
    q = new_node( );
    q→val = v, q→par = p, q→level = 0;
    q→sib = p→kid, p→kid = q;
    launch(p, d + 1, q);
    for (q = q→sib; q; q = q→sib)
      if (q→trans) new_kid(q, d + 1, q→trans[v]);
  }
```

**25.** At this point we've reached the heart of the algorithm, the *launch* subroutine that initializes each new node created by *new_kid* (or by *launch* itself). When *launch*$(p, d, q)$ is called, node $q$ is a child of $p$ on level $d$ whose *val* field has been set but the *trans* field has not.

```
⟨ Subroutines 22 ⟩ +≡
  void launch(node *p, int d, node *q)
  {
    register int v, w;
    register node *r, *s, *t;
    v = q→val;
    w = minp[x[d − 1] + 1][v][v];
    if (w < x[d]) ⟨ Reject the current set {x_1, ..., x_l} 42 ⟩;
    if (w > x[d]) q→trans = Λ;      /* this branch is dormant */
    else ⟨ Launch a new node that maps v to x_d 26 ⟩;
  }
```

**26.** We can exclude values that will map between $x_{d-1}$ and $x_{d+1}$, at least when $d < l$. By setting $x_{l+1} = x_l$ below, we make this work also when $d = l$.

```
⟨ Launch a new node that maps v to x_d 26 ⟩ ≡
  {
    w = x[d − 1] + 1;
    q→trans = minp[w][v];
    for ( ; w < x[d + 1]; w++)
      if (w ≠ x[d]) ⟨ Make sure w is forbidden 27 ⟩;
    ⟨ Use special strategy for launch if d is sufficiently deep 33 ⟩;
    for (r = p→kid, s = Λ; r; r = r→sib)
      if (r ≠ q) {
        t = new_node( );
        t→par = q;
        t→val = q→trans[r→val], t→level = 0;
        if (s) s→sib = t; else q→kid = t;
        s = t;
      }
    if (s) s→sib = Λ;
    else auts++;      /* when we've reached level l, we've found an automorphism */
    for (r = q→kid; r; r = r→sib) launch(q, d + 1, r);
  }
```

This code is used in section 25.

**27.** The *forbidden* table is used in the algorithm below to ensure that no SET occurs among the elements $\{x_1, \ldots, x_l\}$. We also use it here to forbid card values that will be transformed into $w$ by a sequence of permutations ending with $q\rightarrow trans$. (Since all such cases will ultimately lead to rejection, we can presumably save time by ruling them out in advance. If I myself had more time to spend, I'd check this to see just how much it helps.)

The reader should not confuse "forbidden" elements with elements that are "illegal" in the sense of the proof above. The two concepts are related, but the algorithm would work even if the present step were omitted.

```
⟨ Make sure w is forbidden 27 ⟩ ≡
  {
    for (r = q, v = w; r ≠ root; r = r→par) v = r→trans[v + nnn];
    forbidden[v] = 1;
  }
```

This code is used in section 26.

**28.**
```
⟨ Global variables 6 ⟩ +≡
  SETcard x[22];      /* here's where we remember x_1, x_2, etc. */
  char forbidden[nn + 1];      /* nonzero for noncanonical choices */
  int auts;      /* automorphisms of {x_1, ..., x_l} found */
  int l;      /* the current level */
```

**29.** Eventually we get to a level so deep that only the identity mapping fixes all elements $\le x_d$. Then the algorithm we have described so far, although correct, begins to spin its wheels as it laboriously finds at most one active child of each node.

Therefore we streamline the data structures for all such nodes, which we call "terminal," and we go into a different mode of operation when we reach a terminal node. Such a node $q$ is identified by the condition $q\rightarrow level > 0$; if $q\rightarrow level = k$ it means that all nodes $\{x_1, \ldots, x_{k-1}\}$ have been matched among $q$ and its ancestors and older siblings. The next younger siblings of $q$ will therefore try to match $x_k$.

The *new_kid* procedure will add a new terminal node to the descendants of $p$, if $d$ is sufficiently deep as described above; this will be true if and only if $x_d \ge D-1$, where $D$ is the *perm* table depth. If $p$ already has at least one child, all of its children are terminal, and we use the *new_terminal_kid* procedure to extend $p$'s family. Otherwise we initiate the family, using the fact that $l$ must equal $d+1$ if $p$ was previously childless.

```
⟨ Use special strategy for new_kid if d is sufficiently deep 29 ⟩ ≡
  if (x[d] ≥ dd − 1) {
    if (p→kid) new_terminal_kid(p, v);
    else {      /* l = d + 1 */
      if (v < x[l]) ⟨ Reject the current set {x_1, ..., x_l} 42 ⟩;
      q = new_node( );
      p→kid = q, q→val = v, q→sib = Λ, q→par = p;
      if (v ≡ x[l]) q→level = l + 1, auts ++;
      else q→level = l;
    }
    return;
  }
```

This code is used in section 24.

**30.** The *kid* links in terminal nodes are *not* used for parenting; they jump across siblings known to be irrelevant in future searches. (This is just a heuristic, designed to ameliorate the fact that we don't want to complicate the backtracking process by updating any existing links in a family when a new child is born.)

```
⟨ Subroutines 22 ⟩ +≡
  void new_terminal_kid(node *q, SETcard v)
  {
    register node *r, *p;
    register int k, w;
    r = new_node( );
    r→val = v, r→sib = q→kid, q→kid = r;
    ⟨ Find the index k such that we've matched {x_1, ..., x_{k-1}} and such that all other values exceed x_k 31 ⟩;
    r→level = k;
    for (p = r→sib; p; p = p→kid)
      if (p→val > x[k]) break;
    r→kid = p;
  }
```

**31.**

```
⟨ Find the index k such that we've matched {x_1, ..., x_{k-1}} and such that all other values
    exceed x_k 31 ⟩ ≡
  k = r→sib→level;
  while (1) {      /* v is the smallest value greater than x_{k-1} */
    if (v < x[k]) ⟨ Reject the current set {x_1, ..., x_l} 42 ⟩;
    if (v > x[k]) break;
    ⟨ Forbid values that will cause rejection if they propagate this far 32 ⟩;
    k++;
    if (k > l) {
      auts++;      /* hey, we've matched everything */
      break;
    }
    for (p = r→sib, v = nn; p; p = p→kid)
      if (p→val > x[k-1] ∧ p→val < v) v = p→val;
  }
```

This code is used in section 30.

**32.** We can exclude values that will map between $x_{k-1}$ and $x_{k+1}$, at least when $k < l$. By setting $x_{l+1} = x_l$ below, we make this work also when $k = l$.

```
⟨ Forbid values that will cause rejection if they propagate this far 32 ⟩ ≡
  for (w = x[k-1] + 1; w < x[k+1]; w++)
    if (w ≠ x[k]) {
      for (p = q, v = w; p ≠ root; p = p→par) v = p→trans[v + nnn];
      forbidden[v] = 1;
    }
```

This code is used in section 31.

**33.** When all children of a newly launched node $q$ are terminal, we append them in reverse order. This is done only for convenience, because the order is unimportant in newly launched nodes. (Such nodes will disappear completely when we backtrack.)

```
⟨ Use special strategy for launch if d is sufficiently deep 33 ⟩ ≡
  if (x[d] ≥ dd − 1 ∧ d < l) {
    s = p→kid; if (s ≡ q) s = s→sib;
    v = q→trans[s→val];
    if (v < x[d + 1]) ⟨ Reject the current set {x_1, ..., x_l} 42 ⟩;
    r = new_node( );
    q→kid = r, r→par = q;
    r→val = v, r→sib = Λ;
    if (v ≡ x[d + 1]) {
      r→level = d + 2;
      if (d + 1 ≡ l) auts++;
    } else r→level = d + 1;
    for (s = s→sib; s; s = s→sib)
      if (s ≠ q) new_terminal_kid(q, q→trans[s→val]);
    return;
  }
```

This code is used in section 26.

**34.** Here's a subroutine that I expect will be useful during the debugging process.

```
⟨ Subroutines 22 ⟩ +≡
  void print_subtree(node *p, int d)
  {
    register node *r;
    register int k;
    for (k = 0; k < d; k++) printf(p→level ? " " : ".");
    printf("%04d", decode[p→val]);
    if (p→level) {
      printf(",%d", p→level);
      if (p→kid) printf("->%04d", decode[p→kid→val]);
      printf("\n");
    }
    else if (p→trans) {
      ⟨ Print the current transform matrix 35 ⟩;
      for (r = p→kid; r; r = r→sib) print_subtree(r, d + 1);
    } else printf("\n");
  }
```

**35.** We print the matrix by giving five vectors $y_i$ such that $(x_1, x_2, x_3, x_4) \mapsto y_0 + x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$.

```
⟨ Print the current transform matrix 35 ⟩ ≡
  k = p→trans[0];
  printf(" [%04d,%04d,%04d,%04d,%04d]\n", decode[k], decode[third[k][third[0][p→trans[4]]]],
      decode[third[k][third[0][p→trans[3]]]], decode[third[k][third[0][p→trans[2]]]],
      decode[third[k][third[0][p→trans[1]]]]);
```

This code is used in section 34.

**36.** ⟨ Subroutines 22 ⟩ +≡

```
void print_trees( )
{
  register node *r;
  for (r = (root)→kid; r; r = r→sib) print_subtree(r, 0);
}
```

**37.** More help for debugging: *nod*("123") gives the third-youngest child of the second-youngest child of the youngest child of the root.

⟨ Subroutines 22 ⟩ +≡

```
node *nod(char *s)
{
  register char *p;
  register int j;
  register node *q = root;
  for (p = s; *p; p++) {
    if (¬q) return Λ;
    for (j = *p - '1', q = q→kid; j; j--) {
      if (¬q) return Λ;
      q = q→sib;
    }
  }
  return q;
}
void dummy( )
{
  malloc(1);      /* loads a routine needed by gdb */
}
```

**38.** And here's a sort of converse routine, *whoami*.

⟨ Subroutines 22 ⟩ +≡

```
void print_id(node *p)
{
  register node *q = p→par, *r;
  register char j;
  if (q) {
    print_id(q);
    for (r = q→kid, j = '1'; r ≠ p; j++)
      if (r) r = r→sib;
      else {
        printf("???"); return;
      }
    printf("%c", j);
  }
}
void whoami(node *p)
{
  print_id(p); printf("\n");
}
```

**39. Backtracking.** Now we're ready to construct the tree of all canonical SET-free sets $\{x_1, \ldots, x_l\}$.

```
⟨ Enumerate and print all solutions 39 ⟩ ≡
  l = 0; j = 0;
  x[0] = −1;
  if (setjmp(restart_point)) goto backup;      /* get ready for longjmp */
moveup: while (forbidden[j]) j++;
  if (j ≡ nn) goto backup;
  for (k = 0; k < nn; k++) forbidden_back[l][k] = forbidden[k];
  node_ptr_back[l] = node_ptr;
  auts = 0;
  l++, x[l] = x[l + 1] = j;
  new_kid(root, 0, x[l]);
  ⟨ Record the current canonical l-set as a solution 45 ⟩;
  ⟨ Forbid all SETs that include {x_k, x_l} for 1 ≤ k < l 43 ⟩;
  j = x[l] + 1; goto moveup;
backup: l−−;
  node_ptr = node_ptr_back[l];
  prune(root);
  for (k = 0; k < nn; k++) forbidden[k] = forbidden_back[l][k];
  j = x[l + 1] + 1;
  if (l) goto moveup;
```

This code is used in section 1.

**40.**

```
⟨ Local variables 40 ⟩ ≡
  register int i, j, k;      /* miscellaneous indices */
```

This code is used in section 1.

**41.**

```
⟨ Global variables 6 ⟩ +≡
  char forbidden_back[22][nnn];      /* brute-force undoing */
  node *node_ptr_back[22];
```

**42.** If the recursive procedures invoked by *new_kid* lead to a non-canonical situation, we leave them and back up by using C's *longjmp* library function. (The code above will then cause control to pass to the label *rejected*.)

```
⟨ Reject the current set {x_1, ..., x_l} 42 ⟩ ≡
  longjmp(restart_point, 1);
```

This code is used in sections 25, 29, 31, and 33.

**43.**

```
⟨ Forbid all SETs that include {x_k, x_l} for 1 ≤ k < l 43 ⟩ ≡
  for (k = 1; k < l; k++) forbidden[third[x[k]][x[l]]] = 1;
```

This code is used in section 39.

**44.** The data structures have been designed so that all changes invoked by *new_kid* and *launch* are easily undone. Indeed, if *new_kid*$(root, 0, x[l])$ terminates normally, it has added precisely one child to each active node (including any terminal nodes that are present), and the newly added child will of course be the youngest. But if *new_kid* terminates abnormally via *longjmp*, some active nodes may not have been reached.

```
⟨ Subroutines 22 ⟩ +≡
  void prune(node *p)
  {
    register node *q = p→kid;
    register node *r;
    if (q) {
      r = q;
      if (q ≥ node_ptr) p→kid = q = q→sib;      /* the youngest should be pruned away */
      if (¬r→level)
        for ( ; q; q = q→sib) prune(q);
    }
  }
```

**45. The totals.** I want to know not only the nonisomorphic solutions but also the exact number of SET-less $k$ sets that are possible. Then I'll know the precise odds of having no SET in a random deal.

When the program reaches this point, *auts* will have been set to the number of permutations of $\{x_1, \ldots, x_l\}$ that are achievable by automorphisms. The true number of automorphisms of $\{x_1, \ldots, x_l\}$ will therefore be *auts* times the number of automorphisms that fix each of $\{x_1, \ldots, x_l\}$.

I don't know how to compute the latter quantity easily from the *perm* table of a general permutation group. But in the affine linear group of interest here, we need only determine the number of independent elements. This is the smallest index $k$ such that $x_{k+1} \neq k$.

```
⟨ Record the current canonical l-set as a solution 45 ⟩ ≡
  for (j = 1; j < l; j++) printf(".");
  non_iso_count[l]++;
  for (k = 0; x[k + 1] ≡ k; k++) ;
  total_count[l] += multiplier[k − 1]/(double) auts;
  printf("%04d␣(%d:%d)␣%d\n", decode[x[l]], auts, k, node_ptr − nodes);
```

This code is used in section 39.

**46.** Integers of 32 bits are insufficient to hold the numbers we're counting, but double precision floating point turns out to be good enough for exact values in this problem.

```
⟨ Global variables 6 ⟩ +≡
  int non_iso_count[30];      /* number of canonical solutions */
  double total_count[30];     /* total number of solutions */
  double multiplier[5] = {81.0, 6480.0, 505440.0, 36391680.0, 1965150720.0};
```

**47.**

```
⟨ Print the totals 47 ⟩ ≡
  for (j = 1; j ≤ 21; j++)
    printf("%20.20g␣SETless␣%d-sets␣(%d␣cases)\n", total_count[j], j, non_iso_count[j]);
```

This code is used in section 1.

## 48. Index.

⟨ Complete $aa$ to a nonsingular matrix 13 ⟩ Used in section 12.
⟨ Enumerate and print all solutions 39 ⟩ Used in section 1.
⟨ Find the index $k$ such that we've matched $\{x_1, \ldots, x_{k-1}\}$ and such that all other values exceed $x_k$ 31 ⟩ Used in section 30.
⟨ Forbid all SETs that include $\{x_k, x_l\}$ for $1 \le k < l$ 43 ⟩ Used in section 39.
⟨ Forbid values that will cause rejection if they propagate this far 32 ⟩ Used in section 31.
⟨ Global variables 6, 7, 9, 11, 15, 16, 21, 23, 28, 41, 46 ⟩ Used in section 1.
⟨ Initialize 8, 10, 12, 17 ⟩ Used in section 1.
⟨ Launch a new node that maps $v$ to $x_d$ 26 ⟩ Used in section 25.
⟨ Local variables 40 ⟩ Used in section 1.
⟨ Make sure $w$ is forbidden 27 ⟩ Used in section 26.
⟨ Print the current transform matrix 35 ⟩ Used in section 34.
⟨ Print the totals 47 ⟩ Used in section 1.
⟨ Record the current canonical $l$-set as a solution 45 ⟩ Used in section 39.
⟨ Reject the current set $\{x_1, \ldots, x_l\}$ 42 ⟩ Used in sections 25, 29, 31, and 33.
⟨ Subroutines 22, 24, 25, 30, 34, 36, 37, 38, 44 ⟩ Used in section 1.
⟨ Type definitions 5, 20 ⟩ Used in section 1.
⟨ Use special strategy for *launch* if $d$ is sufficiently deep 33 ⟩ Used in section 26.
⟨ Use special strategy for *new_kid* if $d$ is sufficiently deep 29 ⟩ Used in section 24.
⟨ Use $aa$ to define the mapping $perm[4-j][pack(a,b,c,d)]$ 14 ⟩ Used in section 12.

# SETSET-ALL