

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

**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  $(x_1, x_2, x_3, x_4)$  where each  $x_i$  is 1, 2, or 3. 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.)

There are  $4! \times 3!^4 = 31104$  isomorphisms, since we can permute the coordinates in  $4!$  ways and we can permute the individual values of each coordinate position in  $3!$  ways.

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  $k \leq 20$ .

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.

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

```
#define maps (6 * 6 * 6 * 6)    /* this many ways to permute individual coordinates */
#define isos (24 * maps)        /* this many automorphisms altogether */
#include <stdio.h>
<Type definitions 3>
<Global variables 4>
<Subroutines 15>
main()
{
    <Local variables 18>
    <Initialize 6>;
    <Enumerate and print all solutions 16>;
    <Print the totals 36>;
}
```

**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  $S \leq \alpha S$  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  $31104/t$ , so we will essentially have also enumerated the distinct solutions.

The ordering we use is standard: Vectors are ordered lexicographically, so that  $(1, 1, 1, 1)$  is the smallest SET card and  $(3, 3, 3, 3)$  is the largest. Also, when  $S$  and  $T$  both are sets of  $k$  SET cards, we define  $S \leq T$  by first sorting the vectors into order so that  $s_1 < \dots < s_k$  and  $t_1 < \dots < t_k$ , then we compare  $(s_1, \dots, s_k)$  lexicographically to  $(t_1, \dots, t_k)$ . (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  $S = T$ .)

For example, the set  $\{(1, 2, 2, 3), (2, 2, 3, 3)\}$  is isomorphic to the set  $\{(1, 1, 1, 1), (1, 1, 2, 2)\}$ , because we can interchange coordinates 1 and 4, then map  $3 \mapsto 1$  in coordinate 1,  $2 \mapsto 1$  in coordinate 2, and  $(2, 3) \mapsto (1, 2)$  in coordinate 3. The set  $\{(1, 1, 1, 1), (1, 1, 2, 2)\}$  has 32 automorphisms, hence  $31104/32 = 972$  sets are isomorphic to it.

We will generate the elements of a  $k$ -set in order. If we have  $s_1 < \dots < s_k$  and  $\{s_1, \dots, s_k\} \leq \{\alpha s_1, \dots, \alpha s_k\}$  for all  $\alpha$ , it is not hard to prove that  $\{s_1, \dots, s_j\} \leq \{\alpha s_1, \dots, \alpha s_j\}$  for all  $\alpha$  and  $1 \leq j \leq k$ . (The reason is that  $S < T$  and  $t \geq \max T$  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.

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

⟨ Type definitions 3 ⟩ ≡

```
typedef char SETcard;    /* a SET card  $(x_1 + 1, x_2 + 1, x_3 + 1, x_4 + 1)$  represented as  $((x_1 x_2 x_3 x_4)_3$  */
```

See also section 9.

This code is used in section 1.

**4.** When we output a SET card, however, we prefer a hexadecimal code.

⟨ Global variables 4 ⟩ ≡

```
int hexform[81] = {#1111, #1112, #1113, #1121, #1122, #1123, #1131, #1132, #1133,
#1211, #1212, #1213, #1221, #1222, #1223, #1231, #1232, #1233,
#1311, #1312, #1313, #1321, #1322, #1323, #1331, #1332, #1333,
#2111, #2112, #2113, #2121, #2122, #2123, #2131, #2132, #2133,
#2211, #2212, #2213, #2221, #2222, #2223, #2231, #2232, #2233,
#2311, #2312, #2313, #2321, #2322, #2323, #2331, #2332, #2333,
#3111, #3112, #3113, #3121, #3122, #3123, #3131, #3132, #3133,
#3211, #3212, #3213, #3221, #3222, #3223, #3231, #3232, #3233,
#3311, #3312, #3313, #3321, #3322, #3323, #3331, #3332, #3333};
```

See also sections 5, 8, 10, 12, 13, 17, and 35.

This code is used in section 1.

**5.** 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 4 ⟩ +≡

```
char z[3][3] = {{0, 2, 1}, {2, 1, 0}, {1, 0, 2}};    /*  $x + y + z \equiv 0 \pmod{3}$  */
char third[81][81];
```

**6. #define** pack( $a, b, c, d$ ) ((( $(a) * 3 + (b) * 3 + (c) * 3 + (d)$ ))

⟨ Initialize 6 ⟩ ≡

```
{
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]);
}
```

See also sections 7, 11, and 14.

This code is used in section 1.

7. An even bigger table comes next: We precompute the permutation of SET cards for each of the 31104 potential automorphisms.

And, what the heck, we compute the inverse permutation too; it's only another 2.5 megabytes.

```
#define pmap(d) trit[perm[p][d]]
#define ppack(p, a, b, c, d) (((((p) * 6 + (a)) * 6 + (b)) * 6 + (c)) * 6 + (d))
⟨ Initialize 6 ⟩ +=
{
    int a, b, c, d, e, f, g, h, p, s, t;
    for (p = 0; p < 24; p++)
        for (a = 0; a < 6; a++)
            for (b = 0; b < 6; b++)
                for (c = 0; c < 6; c++)
                    for (d = 0; d < 6; 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++)
                                        trit[0] = perm[a][e], trit[1] = perm[b][f],
                                        trit[2] = perm[c][g], trit[3] = perm[d][h],
                                        alf = ppack(p, a, b, c, d),
                                        s = pack(e, f, g, h), t = pack(pmap(0), pmap(1), pmap(2), pmap(3)),
                                        aut[alf][s] = t, tua[alf][t] = s;
}
```

8. ⟨ Global variables 4 ⟩ +=

```
char trit[4]; /* four ternary digits */
char perm[24][4] = {{0, 1, 2, 3}, {0, 2, 1, 3}, {1, 0, 2, 3}, {1, 2, 0, 3}, {2, 0, 1, 3}, {2, 1, 0, 3},
{0, 1, 3, 2}, {0, 3, 1, 2}, {1, 0, 3, 2}, {1, 3, 0, 2}, {3, 0, 1, 2}, {3, 1, 0, 2},
{0, 2, 3, 1}, {0, 3, 2, 1}, {2, 0, 3, 1}, {2, 3, 0, 1}, {3, 0, 2, 1}, {3, 2, 0, 1},
{1, 2, 3, 0}, {1, 3, 2, 0}, {2, 1, 3, 0}, {2, 3, 1, 0}, {3, 1, 2, 0}, {3, 2, 1, 0}};
char aut[31104][81], tua[31104][81]; /* basic permutation tables */
```

9. Cards of a set are linked together cyclically in order of their values, with an “infinite” card at the head.

We also maintain an array of 31104 elements, one for each automorphism of a given element  $s_l$  of the canonical set  $\{s_1, \dots, s_l\}$  that we’re working with. Such an array is called a “node.” In essence, the nodes for  $(s_1, \dots, s_l)$  represent an array of 31104 sets  $\{\alpha s_1, \dots, \alpha s_l\}$ , each isomorphic to  $\{s_1, \dots, s_l\}$ .

Each element  $\alpha s_k$  at level  $k$  also has a threshold level  $tlevel$ , which can be understood as follows: Suppose  $S = \{s_1, \dots, s_l\}$  is the current canonical  $l$ -set of interest, so that  $\alpha S = \{\alpha s_1, \dots, \alpha s_l\} \geq S$  for all  $\alpha$ . If  $\alpha S > S$ , there is a smallest index  $i$  such that  $t_i > s_i$ , where  $t_i$  is the  $i$ th smallest element of  $\alpha S$ ; in that case we say that the threshold value of  $\alpha s_k$  is  $s_i$ , and the threshold level is  $i$ . A tentative value of  $s_{l+1}$  can be immediately rejected if  $\alpha s_{l+1}$  is less than  $s_i$ , because such a set  $\{s_1, \dots, s_{l+1}\}$  would not be canonical. On the other hand, if  $\alpha s_{l+1}$  is greater than  $s_i$ , no action needs to be taken since the threshold stays the same in this case.

The threshold level is considered to be  $l + 1$  if  $\alpha S = S$ . In that case, we say by convention that the threshold value is unknown.

⟨Type definitions 3⟩ +≡

```
typedef struct elt_struct {
    SETcard val; /* value of this element */
    char tlevel; /* the level of the threshold value */
    char level; /* the level when the threshold was set */
    struct elt_struct *link; /* next larger element of a set */
    struct elt_struct *next; /* next element waiting for the same threshold */
    struct elt_struct *fixer; /* the link to change when the threshold is hit */
} element;

typedef struct {
    SETcard v; /*  $s_l$  */
    element image[isos]; /*  $\alpha s_l$  for each automorphism  $\alpha$  */
} node;
```

10. The node for  $s_l$  is called  $current[l]$ , and  $current[0]$  contains the header nodes of circular lists.

```
#define head current[0]
#define curval(i) current[i].v /*  $s_i$  */

⟨Global variables 4⟩ +≡
    node current[22]; /* the nodes for  $s_1, s_2$ , etc. */
```

11. #define infty 81 /\* larger than any SETcard value \*/

⟨Initialize 6⟩ +≡

```
    for (j = 0; j < isos; j++) head.image[j].val = infty, head.image[j].tlevel = 1,
        head.image[j].link = head.image[j].fixer = &head.image[j];
```

12. Each pair  $(s_i, s_j)$  for  $1 \leq i < j \leq l$  defines a third SET card  $t$  that must not be appended to the set  $\{s_1, \dots, s_l\}$ . The auxiliary table  $tab[t]$  tells how many such pairs exist for a given  $t$ . This table also counts cards that are forbidden because they would produce values  $\alpha s_{l+1}$  less than the threshold for some  $\alpha$ .

Another auxiliary table, called *here*, records the cards that are present in the current set.

⟨Global variables 4⟩ +≡

```
    unsigned int tab[82]; /* nonzero for forbidden cards */
    char here[81]; /* nonzero for cards in  $\{s_1, \dots, s_l\}$  */
```

**13.** We keep lists of all elements that need to be updated when a particular value  $s$  is appended to the current set. Such a list begins at  $top[s]$ . The list beginning at  $top[infty]$  is the one for unknown thresholds, namely for all elements such that  $\alpha$  is an automorphism of  $\{s_1, \dots, s_l\}$ .

When an element is removed from a list as part of the updating at level  $l$ , it is placed on list  $back[l]$ , so that everything can be downdated when we backtrack. A separate list  $aback[l]$  is for elements removed from  $top[infty]$ .

```

⟨ Global variables 4 ⟩ +=
  element *top[82];      /* elements waiting for a particular card */
  element *oldtop[22][81]; /* saved values of top */
  element *back[22], *aback[22]; /* lists for undoing */

```

**14.** Automorphism 0 is the identity, and we need not bother updating its entries.

```

⟨ Initialize 6 ⟩ +=
  head.v = -1;
  for (k = 1; k < isos - 1; k++) head.image[k].next = &head.image[k + 1];
  top[infty] = &head.image[1];

```

**15.** Here's a subroutine that might facilitate debugging: It simply counts the elements of a list.

```

⟨ Subroutines 15 ⟩ =
  int count(element *p)
  {
    register int c;
    register element *q;
    for (q = p, c = 0; q; q = q->next) c++;
    return c;
  }

```

This code is used in section 1.

**16. Backtracking.** Now we're ready to construct the tree of all canonical SET-free sets  $\{s_1, \dots, s_l\}$ .

```

⟨Enumerate and print all solutions 16⟩ ≡
  l = 0; j = 0;
moveup: while (tab[j]) j++;
  if (j ≡ infty) goto big_backup;
  l++, curval(l) = j, here[j] = 1;
  for (k = 0; k < infty; k++) oldtop[l][k] = top[k];
  auts = 1, newauts = Λ;
  ⟨Update the data structures for all elements whose threshold is j, or backup 21⟩;
  ⟨Update the data structures for all elements whose threshold is unknown, or backup 29⟩;
  ⟨Record the current canonical l-set as a solution 34⟩;
  ⟨Update tab 19⟩;
  j = curval(l) + 1; goto moveup;
big_backup: ⟨Downdate tab 20⟩;
  j = curval(l);
  ⟨Downdate the data structures for all elements whose threshold was unknown 30⟩;
  ⟨Downdate the data structures for all elements whose threshold was j 28⟩;
  for (k = 0; k < infty; k++) top[k] = oldtop[l][k];
  here[j] = 0;
  j++, l--;
  if (l) goto moveup;

```

This code is used in section 1.

**17.** ⟨Global variables 4⟩ +≡

```

int auts; /* automorphisms of the current l-set */
element *newauts; /* the list of nontrivial automorphisms at level l */

```

**18.** ⟨Local variables 18⟩ ≡

```

int l; /* the current level */
register int j, k; /* miscellaneous indices; usually j = sl */

```

See also section 22.

This code is used in section 1.

**19.** ⟨Update tab 19⟩ ≡

```

for (j = 1; j < l; j++) tab[third[curval(j)][curval(l)]]++;

```

This code is used in section 16.

**20.** ⟨Downdate tab 20⟩ ≡

```

for (j = 1; j < l; j++) tab[third[curval(j)][curval(l)]]--;

```

This code is used in section 16.

**21.** Now we come to the main point of this program, the part where elements  $\alpha s$  are incorporated into the data structures because their threshold value has occurred.

```

⟨ Update the data structures for all elements whose threshold is  $j$ , or backup 21 ⟩ ≡
  for ( $pp = \Lambda, p = top[j]$ ;  $p; r = p\text{-next}, p\text{-next} = pp, pp = p, p = r$ ) {
     $ll = p\text{-level}$ ;
     $alf = p - \&current[ll].image[0]$ ;
    ⟨ Make quick check for easy cases that become dormant 23 ⟩;
    ⟨ Bring  $current[k].image[alf]$  up to date for  $ll < k \leq l$  24 ⟩;
    ⟨ Compute the new threshold for  $\alpha$ , or backup 25 ⟩;
  }
   $top[j] = \Lambda, back[l] = pp$ ;

```

This code is used in section 16.

**22.** ⟨ Local variables 18 ⟩ +=

```

  element * $p, *pp$ ;    /* element of list and its predecessor */
  int  $ll$ ;             /* a previous or future level number */
  int  $alf$ ;            /* the current automorphism of interest */
  register element * $q, *r$ ; /* registers for list manipulations */
  int  $jj$ ;             /* another convenient integer variable */

```

**23.** The list of elements waiting for  $j$  to occur will, I believe, consist mostly of the 384 elements inserted on level 1, namely those  $\alpha$  for which  $\alpha j = 0$ . Once we have set  $s_l = j$ , the next question is almost always, “What is the value of  $j'$  for which  $\alpha j' = 1$ ?” because we usually have  $s_0 = 0$  and  $s_1 = 0$ . More generally, if we are waiting for  $j$  because  $\alpha j = s_i$ , we will next be interested in the value  $j'$  for which  $\alpha j' = s_{i+1}$ . If that value of  $j'$  is less than  $j$  (which equals  $s_l$ ) but not already present, or if  $tab[j']$  is nonzero, we know that  $j'$  will never be added to the current set, so we need not consider  $\alpha$  any further.

We can save a significant amount of work in such cases, especially when  $l$  is rather large, so the following code is useful even though not strictly necessary.

```

⟨ Make quick check for easy cases that become dormant 23 ⟩ ≡
   $jj = tua[alf][curval(p\text{-tlevel} + 1)]$ ;
  if ( $tab[jj] \vee (jj < j \wedge \neg here[jj])$ ) {
    for ( $jj = curval(p\text{-tlevel}) + 1; jj < curval(p\text{-tlevel} + 1); jj++$ ) {
       $k = tua[alf][jj]$ ;
      if ( $k > j$ )  $tab[k]++$ ;
      else if ( $here[k]$ ) ⟨ Begin backing up in Case A 33 ⟩; /* ( $s_1, \dots, s_l$ ) isn't canonical */
    }
    continue; /* no need to update since  $jj$  won't occur */
  }

```

This code is used in section 21.

**24.** #define  $succ(p)$  (element \*)((char \*)  $p + sizeof(node)$ )

```

⟨ Bring  $current[k].image[alf]$  up to date for  $ll < k \leq l$  24 ⟩ ≡
  for ( $ll++, q = succ(p)$ ;  $q < \&current[l].image[0]$ ;  $ll++, q = succ(q)$ ) {
     $q\text{-val} = aut[alf][curval(ll)]$ ;
    for ( $r = p\text{-fixer}$ ;  $r\text{-link-val} < q\text{-val}$ ;  $r = r\text{-link}$ ) ;
     $q\text{-link} = r\text{-link}$ ;
     $r\text{-link} = q$ ; /* we have inserted  $q\text{-val}$  into the sorted list for  $\alpha$  */
  }
   $q\text{-val} = curval(p\text{-tlevel}), q\text{-link} = p\text{-fixer-link}, p\text{-fixer-link} = q$ ;

```

This code is used in section 21.

**25.**  $\langle$  Compute the new threshold for  $\alpha$ , or backup 25  $\rangle \equiv$   
**for** ( $r = q, ll = p\text{-}tlevel + 1$ ;  $r\text{-}link\text{-}val \equiv curval(ll)$ ;  $r = r\text{-}link, ll++$ ) ;  
**if** ( $r\text{-}link\text{-}val < curval(ll)$ ) /\* oops,  $(s_1, \dots, s_l)$  isn't canonical \*/  
 $\langle$  Begin backing up in Case B 32  $\rangle$ ;  
 $q\text{-}tlevel = ll, q\text{-}fixer = r$ ;  
 $\langle$  Tabulate newly forbidden values 26  $\rangle$ ;  
**if** ( $ll > l$ )  $auts++$ ,  $q\text{-}next = newauts$ ,  $newauts = q$ ;  
**else**  $jj = tua[alf][curval(ll)]$ ,  $q\text{-}level = l$ ,  $q\text{-}next = top[jj]$ ,  $top[jj] = q$ ;

This code is used in section 21.

**26.** If  $p\text{-}tlevel = i$ , we have already used  $tab$  to forbid all  $s$  values such that  $\alpha s < s_i$  and  $\alpha s \notin \{s_1, \dots, s_i\}$ . At this point we essentially want to increase  $i$  to the new threshold level  $ll$ . If  $ll > l$ , however, we forbid values only up to  $s_l$ , because  $\alpha$  is an automorphism of the full set  $\{s_1, \dots, s_l\}$  in this case.

$\langle$  Tabulate newly forbidden values 26  $\rangle \equiv$   
**for** ( $jj = (ll > l ? j : curval(ll)) - 1$ ;  $jj > curval(p\text{-}tlevel)$ ;  $jj--$ ) {  
 $k = tua[alf][jj]$ ;  
**if** ( $k > j$ )  $tab[k]++$ ;  
}

This code is used in section 25.

**27.** Later we'll want to undo that last step.

$\langle$  Untabulate values that were considered newly forbidden 27  $\rangle \equiv$   
**for** ( $jj = (ll > l ? j : curval(ll)) - 1$ ;  $jj > curval(p\text{-}tlevel)$ ;  $jj--$ ) {  
 $k = tua[alf][jj]$ ;  
**if** ( $k > j$ )  $tab[k]--$ ;  
}

This code is used in section 28.

**28.** Indeed, in a backtrack program, everything we do that affects subsequent decisions must eventually be undone.

The main thing we must undo at this point is to remove the  $l - ll$  elements that were sorted in to the list  $\{s_1, \dots, s_l\}$ .

$\langle$  Downdate the data structures for all elements whose threshold was  $j$  28  $\rangle \equiv$   
 $pp = \Lambda, p = back[l]$ ;  
 $backup\_a$ : **while** ( $p$ ) {  
 $alf = p - \&current[p\text{-}level].image[0]$ ;  
**if** ( $p\text{-}fixer\text{-}link < \&current[l].image[0]$ ) { /\* the "quick check" worked \*/  
**for** ( $jj = curval(p\text{-}tlevel) + 1$ ;  $jj < curval(p\text{-}tlevel + 1)$ ;  $jj++$ ) {  
 $k = tua[alf][jj]$ ;  
**if** ( $k > j$ )  $tab[k]--$ ;  
}  
}  
**else** {  
 $ll = current[l].image[alf].tlevel$ ;  
 $\langle$  Untabulate values that were considered newly forbidden 27  $\rangle$ ;  
 $backup\_b$ :  $ll = p\text{-}level$ ;  
**for** ( $r = p\text{-}fixer, jj = l - ll$ ;  $jj$ ;  $r = r\text{-}link$ )  
**if** ( $r\text{-}link > p$ )  $jj--$ ,  $r\text{-}link = r\text{-}link\text{-}link$ ;  
}  
 $r = p\text{-}next, p\text{-}next = pp, pp = p, p = r$ ;  
}

This code is used in section 16.



**29.**  $\langle$  Update the data structures for all elements whose threshold is unknown, or backup 29  $\rangle \equiv$

```

for ( $pp = \Lambda, p = \text{top}[\text{infty}]; p; r = p\text{-next}, p\text{-next} = pp, pp = p, p = r$ ) {
   $\text{alf} = p - \&\text{current}[l - 1].\text{image}[0];$ 
   $jj = \text{aut}[\text{alf}][j];$ 
  if ( $jj < j$ )  $\langle$  Begin backing up in Case C 31  $\rangle$ ;
   $q = \text{succ}(p);$ 
   $q\text{-link} = p\text{-fixer-link}, p\text{-fixer-link} = q;$ 
  if ( $jj > j$ ) {
     $q\text{-val} = jj, q\text{-level} = l, q\text{-tlevel} = l, q\text{-fixer} = p\text{-fixer};$ 
     $jj = \text{tua}[\text{alf}][j], q\text{-next} = \text{top}[jj], \text{top}[jj] = q;$ 
  } else {
     $q\text{-val} = jj, q\text{-tlevel} = l + 1, q\text{-fixer} = q;$ 
     $\text{auts}++, q\text{-next} = \text{newauts}, \text{newauts} = q;$ 
  }
  for ( $jj = \text{curval}(l - 1) + 1; jj < j; jj++$ ) {
     $k = \text{tua}[\text{alf}][jj];$ 
    if ( $k > j$ )  $\text{tab}[k]++;$ 
  }
}
 $\text{top}[\text{infty}] = \text{newauts}, \text{aback}[l] = pp;$ 

```

This code is used in section 16.

**30.**  $\langle$  Downdate the data structures for all elements whose threshold was unknown 30  $\rangle \equiv$

```

 $pp = \Lambda, p = \text{aback}[l];$ 
 $\text{backup\_c: while } (p) \{$ 
   $\text{alf} = p - \&\text{current}[l - 1].\text{image}[0];$ 
   $q = \text{succ}(p);$ 
   $p\text{-fixer-link} = q\text{-link};$ 
  for ( $jj = \text{curval}(l - 1) + 1; jj < j; jj++$ ) {
     $k = \text{tua}[\text{alf}][jj];$ 
    if ( $k > j$ )  $\text{tab}[k]--;$ 
  }
   $r = p\text{-next}, p\text{-next} = pp, pp = p, p = r;$ 
}
 $\text{top}[\text{infty}] = pp;$ 

```

This code is used in section 16.

**31.** It's slightly tricky to begin backing up when we're in the middle of updating a data structure.

$\langle$  Begin backing up in Case C 31  $\rangle \equiv$

```

{
   $r = p, p = pp, pp = r;$ 
  goto  $\text{backup\_c};$ 
}

```

This code is used in section 29.

**32.** This is one of those fairly rare occasions when it's OK to jump into the middle of a loop.

⟨Begin backing up in Case B 32⟩ ≡

```
{
   $r = p\text{-next}, p\text{-next} = pp, pp = r;$ 
  goto backup_b;
}
```

This code is used in section 25.

**33.** ⟨Begin backing up in Case A 33⟩ ≡

```
{
  for ( $jj \text{--}; jj > \text{curval}(p\text{-tlevel}); jj \text{--}$ ) {
     $k = \text{tua}[\text{alf}][jj];$ 
    if ( $k > j$ )  $\text{tab}[k] \text{--};$ 
  }
   $r = p, p = pp, pp = r;$ 
  goto backup_a;
}
```

This code is used in section 23.

**34. The totals.** While we're at it, we might as well determine exactly how many SET-less  $k$  sets are possible. Then we'll know the precise odds of having no SET in a random deal.

```

⟨ Record the current canonical  $l$ -set as a solution 34 ⟩ ≡
  if (verbose ∨  $l \leq 8$ ) {
    for ( $j = 1$ ;  $j < l$ ;  $j++$ ) printf(".");
    printf("%04x□(%d)\n", hexform[curval( $l$ )], auts);
  } else if ( $l \geq 20$ ) {
    for ( $j = 1$ ;  $j \leq l$ ;  $j++$ ) printf("□%x", hexform[curval( $j$ )]);
    printf("□(%d)\n", auts);
  }
  non_iso_count[ $l$ ]++;
  total_count[ $l$ ] += 31104.0/(double) auts;

```

This code is used in section 16.

**35.** 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 4 ⟩ +=
  int non_iso_count[30]; /* number of canonical solutions */
  double total_count[30]; /* total number of solutions */
  int verbose = 0; /* set nonzero for debugging */

```

```

36. ⟨ Print the totals 36 ⟩ ≡
  for ( $j = 1$ ;  $j \leq 21$ ;  $j++$ )
    printf("%20.20g□SETless□%d-sets□(%d□cases)\n", total_count[ $j$ ],  $j$ , non_iso_count[ $j$ ]);

```

This code is used in section 1.

**37. Index.**

*a*: [6](#), [7](#).  
*aback*: [13](#), [29](#), [30](#).  
*alf*: [7](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [33](#).  
*aut*: [7](#), [8](#), [24](#), [29](#).  
*auts*: [16](#), [17](#), [25](#), [29](#), [34](#).  
*b*: [6](#), [7](#).  
*back*: [13](#), [21](#), [28](#).  
*backup\_a*: [28](#), [33](#).  
*backup\_b*: [28](#), [32](#).  
*backup\_c*: [30](#), [31](#).  
*big\_backup*: [16](#).  
*c*: [6](#), [7](#), [15](#).  
*count*: [15](#).  
*current*: [10](#), [21](#), [24](#), [28](#), [29](#), [30](#).  
*curval*: [10](#), [16](#), [19](#), [20](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#),  
[29](#), [30](#), [33](#), [34](#).  
*d*: [6](#), [7](#).  
*e*: [6](#), [7](#).  
*element*: [9](#), [13](#), [15](#), [17](#), [22](#), [24](#).  
*elt\_struct*: [9](#).  
*f*: [6](#), [7](#).  
*fixer*: [9](#), [11](#), [24](#), [25](#), [28](#), [29](#), [30](#).  
*g*: [6](#), [7](#).  
*h*: [6](#), [7](#).  
*head*: [10](#), [11](#), [14](#).  
*here*: [12](#), [16](#), [23](#).  
*hexform*: [4](#), [34](#).  
*image*: [9](#), [11](#), [14](#), [21](#), [24](#), [28](#), [29](#), [30](#).  
*infty*: [11](#), [13](#), [14](#), [16](#), [29](#), [30](#).  
*isos*: [1](#), [9](#), [11](#), [14](#).  
*j*: [18](#).  
*jj*: [22](#), [23](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [33](#).  
*k*: [18](#).  
*l*: [18](#).  
*level*: [9](#), [21](#), [25](#), [28](#), [29](#).  
*link*: [9](#), [11](#), [24](#), [25](#), [28](#), [29](#), [30](#).  
*ll*: [21](#), [22](#), [24](#), [25](#), [26](#), [27](#), [28](#).  
*main*: [1](#).  
*maps*: [1](#).  
*moveup*: [16](#).  
*newauts*: [16](#), [17](#), [25](#), [29](#).  
*next*: [9](#), [14](#), [15](#), [21](#), [25](#), [28](#), [29](#), [30](#), [32](#).  
*node*: [9](#), [10](#), [24](#).  
*non\_iso\_count*: [34](#), [35](#), [36](#).  
*oldtop*: [13](#), [16](#).  
*p*: [7](#), [15](#), [22](#).  
*pack*: [6](#), [7](#).  
*perm*: [7](#), [8](#).  
*pmap*: [7](#).  
*pp*: [21](#), [22](#), [28](#), [29](#), [30](#), [31](#), [32](#), [33](#).  
*ppack*: [7](#).

*printf*: [34](#), [36](#).  
*q*: [15](#), [22](#).  
*r*: [22](#).  
*s*: [7](#).  
**SETcard**: [3](#), [9](#), [11](#).  
*succ*: [24](#), [29](#), [30](#).  
*t*: [7](#).  
*tab*: [12](#), [16](#), [19](#), [20](#), [23](#), [26](#), [27](#), [28](#), [29](#), [30](#), [33](#).  
*third*: [5](#), [6](#), [19](#), [20](#).  
*tlevel*: [9](#), [11](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [33](#).  
*top*: [13](#), [14](#), [16](#), [21](#), [25](#), [29](#), [30](#).  
*total\_count*: [34](#), [35](#), [36](#).  
*trit*: [7](#), [8](#).  
*tua*: [7](#), [8](#), [23](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [33](#).  
*v*: [9](#).  
*val*: [9](#), [11](#), [24](#), [25](#), [29](#).  
*verbose*: [34](#), [35](#).  
*z*: [5](#).

- ⟨Begin backing up in Case A 33⟩ Used in section 23.
- ⟨Begin backing up in Case B 32⟩ Used in section 25.
- ⟨Begin backing up in Case C 31⟩ Used in section 29.
- ⟨Bring  $current[k].image[alf]$  up to date for  $ll < k \leq l$  24⟩ Used in section 21.
- ⟨Compute the new threshold for  $\alpha$ , or backup 25⟩ Used in section 21.
- ⟨Downdate the data structures for all elements whose threshold was  $j$  28⟩ Used in section 16.
- ⟨Downdate the data structures for all elements whose threshold was unknown 30⟩ Used in section 16.
- ⟨Downdate  $tab$  20⟩ Used in section 16.
- ⟨Enumerate and print all solutions 16⟩ Used in section 1.
- ⟨Global variables 4, 5, 8, 10, 12, 13, 17, 35⟩ Used in section 1.
- ⟨Initialize 6, 7, 11, 14⟩ Used in section 1.
- ⟨Local variables 18, 22⟩ Used in section 1.
- ⟨Make quick check for easy cases that become dormant 23⟩ Used in section 21.
- ⟨Print the totals 36⟩ Used in section 1.
- ⟨Record the current canonical  $l$ -set as a solution 34⟩ Used in section 16.
- ⟨Subroutines 15⟩ Used in section 1.
- ⟨Tabulate newly forbidden values 26⟩ Used in section 25.
- ⟨Type definitions 3, 9⟩ Used in section 1.
- ⟨Untabulate values that were considered newly forbidden 27⟩ Used in section 28.
- ⟨Update the data structures for all elements whose threshold is  $j$ , or backup 21⟩ Used in section 16.
- ⟨Update the data structures for all elements whose threshold is unknown, or backup 29⟩ Used in section 16.
- ⟨Update  $tab$  19⟩ Used in section 16.

# SETSET

	Section	Page
Introduction .....	<a href="#">1</a>	1
Data structures .....	<a href="#">3</a>	2
Backtracking .....	<a href="#">16</a>	6
The totals .....	<a href="#">34</a>	11
Index .....	<a href="#">37</a>	12