1. Intro. This program enumerates biconnected squaregraphs of small order, more or less by brute force.

Every such graph of perimeter $2p$ is definable by a set partition of $2p$ elements into p blocks of size 2. Equivalently, it's definable by a restricted growth string of a very simple kind, namely a permutation of the multiset $\{0, 0, 1, 1, \ldots, p-1, p-1\}$ in which the first appearance of $j > 0$ is preceded by $j-1$. For example, the restricted growth string 01023132 corresponds to the set partition 13 | 26 | 48 | 57.

However, the set partition is considered to be *circular*, so that it corresponds to taking $2p$ points around a circle and connecting them by chords. Positions $(1, 2, \ldots, 8)$ around a circle are equivalent to positions $(2, \ldots, 8, 1)$; so the set partition above is also equivalent to 24 | 37 | 51 | 68, which has the restricted growth string 01210323. And there are six others, because we consider $2p$ cyclic shifts.

Therefore it turns out that restricted growth strings are not the best representations of the set partitions in this application. Instead we use the cyclic distances or "deltas" to the next occurrence of each symbol; for instance, the growth string 01023132 corresponds to the delta sequence 24642464. Cyclic shifting of the growth sequence is complicated, because the digits need to be renamed; the delta sequence, however, simply shifts cyclically. For instance, the delta sequence of 01210323 is 46424642.

A squaregraph is represented by its *canonical sequence*, which is the lexicographically smallest of the $2p$ associated delta sequences. In our example, the smallest turns out to be 24642464. (And only four of the eight possible shifts actually lead to different results in this case, because the graph has a nontrivial automorphism.)

Two additional restrictions must be satisfied. First, the restricted growth sequence must not contain a subword of the form *abcabc* (not necessarily consecutive). Our example satisfies this restriction, because the relevant subwords of 01023132 are 010212, 010313, 002332, 123132, and because the condition is preserved under cyclic shifting. When this restriction is met, there's a unique way to subdivide the circle into regions by connecting the chords. (The algorithm below demonstrates this fact constructively.)

Each of the regions just mentioned will have two or more vertices, where the paths between consecutive vertices are either segments of the circle or segments of a chord. Furthermore, a region whose boundaries are interior — not segments of the enclosing circle — will always have four or more vertices. (Again, the algorithm below will prove this.)

The squaregraph that corresponds to a canonical string that meets this conditions is defined to be the dual of the region graph, namely the graph whose vertices are the regions, with adjacency defined as sharing the same segment of a chord. For example, 24642464 turns out to be the graph of the so-called "straight tromino," also known as $P_4 \,\square\, K_2$.

A final constraint is that none of the regions may include more than one circle segment. For example, set partitions like 0011 and 01012323 are disallowed. This restriction makes the squaregraph biconnected, because it is equivalent to saying that there are no articulation points.

The reader who tries to draw the resulting graphs will soon understand why the name "squaregraphs" is brilliant. (Also the Wikipedia already has a nice example picture!)

One might argue whether or not the shortest possible sequence, 00, is a biconnected squaregraph. Is the "bridge" K_2 biconnected? In fact, should the empty string perhaps also qualify? Anyway this program starts out with $p = 3$, because the smaller cases are obvious. (For example, the only biconnected squaregraph with $p = 2$ is the square C_4, which corresponds to the canonical sequence 2222.)

For each squaregraph we give its canonical sequence, $\delta_0\delta_1 \ldots \delta_{2p-1}$; the number of interior vertices, q; and the number of squares, s. The graph then has $2p + q$ vertices altogether, and $p + 2s$ edges. We also compute the automorphisms, which are "dihedral": cyclic and/or reflected. In verbose mode, the full graph is listed as well.

If a squaregraph is not isomorphic to its reflection, we list it only once, by taking the minimum of the two canonical sequences.

2. OK, here we go.

One simplification is obvious: The first element δ_0 of the delta sequence can't be 1, because that would produce an articulation point. And because the sequence is canonical, we must have $\delta_j \ge \delta_0$ for all j. In particular, $\delta_0 \le p$; and in fact $\delta_0 = p$ is impossible, when $p > 2$, because of the *abcabc* condition.

When reporting totals, we consider both "unlabeled" delta sequences (reduced by symmetries to canonical form) and "labeled" ones (not reduced by symmetries). For example, 24642464 corresponds to three different labeled solutions, namely 46424642 and 64246424 in addition to itself.

The labeling condition essentially corresponds to assigning a direction to one edge on the periphery of the squarefree graph. That edge and that direction tell us what the delta sequence is; the delta sequence determines all other vertices and edges uniquely, in any squaregraph.

```
#define maxp 10      /* upper bound on p */
#define verbose (argc > 1)      /* a command-line parameter triggers verbosity */
#include <stdio.h>
#include <stdlib.h>
  ⟨ Global variables 3 ⟩;
  int pp;      /* the current perimeter */
  int ptot, pltot;      /* how many found with the current perimeter */
  int vtot[maxp * maxp], vltot[maxp * maxp], stot[maxp * maxp], sltot[maxp * maxp];
    /* how many graphs found with a given number of vertices or squares */
  ⟨ Subroutines 6 ⟩;
  main(int argc)
  {
    register int j, k, l, p, q, s;
    int rfl, rot;
    for (p = 3; p ≤ maxp; p++) {
      pp = p + p;
      ptot = pltot = 0;
      for (j = 2; j < p; j++) {
        ⟨ Initialize the delta sequence, with δ0 = j 4 ⟩;
        ⟨ Generate all answers for (p, j) 5 ⟩;
      }
      printf("%d␣(%d␣labeled)␣with␣perimeter␣%d;␣", ptot, pltot, pp);
      printf("%d␣(%d␣labeled)␣with␣%d␣vertices;␣", vtot[pp], vltot[pp], pp);
      printf("%d␣(%d␣labeled)␣with␣%d␣vertices;␣", vtot[pp + 1], vltot[pp + 1], pp + 1);
      printf("%d␣(%d␣labeled)␣with␣%d␣squares.\n", stot[p - 1], sltot[p - 1], p - 1);
    }
  }
```

3. Generating the deltas. The delta sequence is stored in array *del*, and another array *occ* records the cells of the implicit restricted growth string that are currently known ("occupied"). If the first occurrence of j in the growth string is in cell i_j, we set $back[i_j] = i_{j-1}$ for $0 < j < p$.

```
⟨ Global variables 3 ⟩ ≡
  int del[4 * maxp];
  int occ[3 * maxp + 1];
  int back[2 * maxp];
```

See also section 12.

This code is used in section 2.

4. ⟨ Initialize the delta sequence, with $\delta_0 = j$ 4 ⟩ ≡

```
  del[0] = j, del[j] = pp − j;
  occ[0] = occ[j] = 1;
  k = 1, s = 0;
```

This code is used in section 2.

5. This program is one of those backtrack jobs where I still like to use **goto** statements, forty years after the "structured programming revolution."

The value of s points to the most recent entry that we've entered into the *del* table. It's essentially a stack pointer.

```
⟨ Generate all answers for (p, j) 5 ⟩ ≡
advance: while (occ[k]) k++;
  if (k ≡ pp) ⟨ Print the current del table if it's a solution, then goto next 8 ⟩;
  occ[k] = 1, back[k] = s;
  l = k + j;      /* the first conceivable way to place the mate of cell k */
nextslot: while (occ[l]) l++;
  if (l ≥ pp) goto backtrack;      /* no more places to match cell k */
  if (abcabc(k, l)) {
    l++; goto nextslot;
  }
  del[k] = l − k, del[l] = pp − l + k, occ[l] = 1;
  s = k++;
  goto advance;
next: occ[s + del[s]] = 0;
  k = s;
backtrack: occ[k] = 0;
  k = back[k];
  if (k) {
    l = k + del[k], occ[l] = 0, l++;
    goto nextslot;
  }
  occ[j] = 0;
```

This code is used in section 2.

6. ⟨ Subroutines 6 ⟩ ≡

```
int abcabc(int k, int l)
{
  register int i, j;
  for (j = back[k]; j; j = back[j]) {
    if (l < j + del[j]) continue;
    for (i = back[j]; ; i = back[i]) {
      if (j + del[j] < i + del[i]) goto OK;
      if (i + del[i] < k) goto OK;
      return 1;      /* abcabc failure: i < j < k < i + δ_i < j + δ_j < l */
    OK: if (i ≡ 0) break;
    }
  }
  return 0;      /* no problem */
}
```

See also sections 13, 15, 18, 19, and 20.

This code is used in section 2.

7. Testing canonicity. We want to reject a delta sequence that isn't canonical, namely when it (or its reflection) has a cyclic shift that's smaller. While we're doing this we can also determine any automorphisms (symmetries) that are present.

When a sequence is canonical, we set $rfl = 1$ if it has reflection symmetry, and $rot = q$ if it has q-fold rotation symmetry. (The latter means that shifting by $2p/q$ yields the same result, where q is as large as possible.)

The reflection of a delta sequence $\delta_0\delta_1\ldots\delta_{2p-1}$ is $(2p-\delta_{2p-1})\ldots(2p-\delta_1)(2p-\delta_0)$. The delta sequence 24642464, for example, has reflection 42464246. Hence $rfl = 1$ and $rot = 2$, corresponding to the four symmetries of the straight tromino.

Incidentally, simply connected polyominoes are always squaregraphs. The L-tromino corresponds to delta sequence 23635256; it has two symmetries, exhibited by $rfl = 1$ and $rot = 1$.

8. ⟨ Print the current *del* table if it's a solution, then **goto** *next* 8 ⟩ ≡

```
{
  ⟨ Determine the automorphisms, but goto next if del isn't canonical 9 ⟩;
  ⟨ Construct the graph corresponding to the chords of del 16 ⟩;
  ⟨ Mark the regions, but goto next if there's an articulation problem 22 ⟩;
  ⟨ Print del and its characteristics 23 ⟩;
  ⟨ Update the totals 25 ⟩;
  goto next;
}
```

This code is used in section 5.

9. ⟨ Determine the automorphisms, but **goto** *next* if *del* isn't canonical 9 ⟩ ≡

```
for (k = 0; k < pp; k++) del[pp + k] = del[k];
for (k = 1; k < pp; k++) {      /* try cyclic shifting by k */
  for (q = k; q < k + pp; q++)
    if (del[q] ≠ del[q − k]) break;
  if (del[q] < del[q − k]) goto next;      /* not canonical */
  if (q ≡ k + pp) break;      /* match found */
}
rot = pp/k;      /* pp is a multiple of k */
⟨ Check the reflected delta sequence for canonicity and possible identity 10 ⟩;
```

This code is used in section 8.

10. The reflected sequence is unchanged by a k-shift if and only if the unreflected sequence is.

```
#define rdel(x) pp − del[pp + pp − x]
```

⟨ Check the reflected delta sequence for canonicity and possible identity 10 ⟩ ≡

```
rfl = 0;
for (k = 1; k ≤ pp; k++) {      /* try cyclic shifting by k */
  for (q = k; q < k + pp; q++)
    if (rdel(q) ≠ del[q − k]) break;
  if (q ≡ k + pp) {
    rfl = 1;
    break;
  }
  if (rdel(q) < del[q − k]) goto next;      /* not canonical */
}
```

This code is used in section 9.

11. Constructing the graph. Now comes the fun part, where we actually build the squaregraph vertices and edges, by constructing the regions defined by the chords.

The data structure used here is undoubtedly overkill, but this program was written with an intent to favor transparency over trickery.

Each region is represented as a cyclic sequence of edges, with $e[j].succ$ the index of the successor to edge j. There are two kinds of edges, external and internal. External edge j, for $1 \le j \le 2p$, simply corresponds to a segment of the enclosing circle, from point $j-1$ to point j (modulo $2p$). An internal edge j, on the other hand, can be distinguished from the external edges because $j > 2p$; it represents a segment of the chord that runs between points $e[j].from$ and $e[j].to$. It also has a "mate," $e[j \pm 1]$, which runs the other way on the same chord segment and belongs to an adjacent region.

After the whole graph has been constructed we will mark each edge with the number of the region to which it belongs.

12. ⟨ Global variables 3 ⟩ +≡

```
struct {
  int succ;      /* index of the next edge in this region */
  int from, to;      /* points that define the chord, if internal */
  int reg;      /* region number */
} e[2 * maxp * maxp];
int eptr;      /* address of first unused entry in e */
```

13. New edges are allocated in mated pairs.

The e array, with $2 * maxp * maxp$ entries, should have plenty of room for all the edges we need. But we check it, just to be sure.

⟨ Subroutines 6 ⟩ +≡

```
int newedge(int s, int t)
{
  e[eptr].from = e[eptr + 1].to = s;
  e[eptr].to = e[eptr + 1].from = t;
  e[eptr].reg = e[eptr + 1].reg = 0;
  eptr += 2;
  if (eptr ≥ 2 * maxp * maxp) {
    fprint(stderr, "Memory␣overflow!\n");
    exit(−2);
  }
  return eptr − 2;
}
```

14. For convenience we assume that $eptr$ is always even, so that the mate of edge k is obtained by simply complementing the units bit of k.

```
#define mate(k) ((k) ⊕ 1)
```

15. We could have made the region lists doubly linked, but they tend to be short. Therefore it probably doesn't hurt to search sequentially for the predecessor of an edge.

⟨ Subroutines 6 ⟩ +≡

```
int pred(int k)
{
  register int j;
  for (j = k; e[j].succ ≠ k; j = e[j].succ) ;
  return j;
}
```

16. The task of building the graph consists of starting with just the enclosing circle, then inserting the chords one by one.

```
⟨ Construct the graph corresponding to the chords of del 16 ⟩ ≡
  ⟨ Initialize for chord placement 17 ⟩;
  for (k = s; ; k = back[k]) {
    newchord(k, (k + del[k]) % pp);
    if (k ≡ 0) break;
  }
```

This code is used in section 8.

17.

```
⟨ Initialize for chord placement 17 ⟩ ≡
  for (k = 1; k < pp; k++) e[k].succ = k + 1, e[k].reg = 0;
  e[pp].succ = 1, e[pp].reg = 0;
  eptr = pp + 2;      /* we want eptr to be even, as mentioned earlier */
  e[pp + 1].reg = −1;      /* make the discarded edge unusable */
```

This code is used in section 16.

18. The *newchord* subroutine, which inserts a chord from point s to point t, is the heart of the computation. It subdivides existing regions that are crossed by the new chord.

Consider, for example, the insertion of the very first chord. Then we want to insert a new edge, and its mate, so that there are two regions. External edge $s+1$ will then be succeeded by the new edge from s to t, and external edge $t+1$ will be succeeded by the new edge from t to s.

Every chord insertion process ends in a similar way, with the insertion of a new edge from t to s that succeeds edge $t+1$, and insertion of the mate edge from s to t that succeeds some other edge q.

```
⟨ Subroutines 6 ⟩ +≡
  void finishchord(int q, int s, int t)
  {
    register int m = newedge(s, t);
    e[m].succ = e[t + 1].succ, e[m + 1].succ = e[q].succ;
    e[t + 1].succ = m + 1, e[q].succ = m;
  }
```

19. A more complex operation is needed when we split a region by introducing an interior vertex. Then we need to create two new pairs of mated edges. One of these, from t to s, becomes the successor of some *interior* edge p, which is being cut into two edges; its mate, from s to t, becomes the successor of a given edge q, which doesn't need to be cut. The second mated pair of new edges becomes the other half of edge p (and its mate) after cutting.

The following subroutine returns the index of the internal edge that can be used to continue chord insertion on the next region to be encountered between s and t.

```
⟨ Subroutines 6 ⟩ +≡
  int internalchord(int p, int q, int s, int t)
  {
    register int m = newedge(s, t), mm = newedge(e[p].from, e[p].to);
    register int pp = mate(p), ppp = pred(pp);
    e[m].succ = mm, e[m + 1].succ = e[q].succ;
    e[mm].succ = e[p].succ, e[mm + 1].succ = pp;
    e[p].succ = m + 1, e[q].succ = m, e[ppp].succ = mm + 1;
    return mm + 1;
  }
```

20. Now we're ready to insert a new chord in its entirety.

We use the fact that points (a, b, c) are "cyclically ordered" if and only if $(b-a) \bmod pp < (c-a) \bmod pp$.

```
⟨ Subroutines 6 ⟩ +≡
  void newchord(int s, int t)
  {
    register int p, q;
    for (q = s + 1; ; ) {
      p = e[q].succ;      /* the edge following q will never be cut */
      for (p = e[p].succ; ; p = e[p].succ) {
        if (p ≡ q) {
          fprintf(stderr, "This can't happen (newchord loop)!\n");
          exit(-1);
        }
        if (p ≤ pp) {      /* exterior edge */
          if (p ≡ t + 1) break;
        } else {      /* interior edge */
          if (((t - e[p].from + pp) % pp) < ((e[p].to - e[p].from + pp) % pp)) break;
        }
      }
      if (p ≤ pp) break;
      q = internalchord(p, q, s, t);      /* see the discussion below */
    }
    finishchord(q, s, t);
  }
```

21. The situation that arises when the *internalchord* routine is called in the loop above needs some justification and further explanation.

Edge p is an internal edge that goes from s' to t', say. We also know that the points (s, s', t, t') appear in that order as we traverse the outer circle.

If the predecessor of edge p is also an internal edge, say from s'' to t'', then the points (s, s'', s', t'', t') are in cyclic order. In order to be sure that the new chord from s to t is forced to cut edge p, we need to know that (t'', t, t') are in cyclic order. The alternative is the cyclic order (s, s'', s', t, t'', t'); if that were true, the edge to cut would be ambiguous. Fortunately, however, the *abcabc* condition has ruled out such a possibility.

A similar argument applies if the successor of edge p is internal; again, we conclude that the *newchord* algorithm is justified in cutting edge p.

A fancier argument would set up an invariant relation on each region that arises during the construction process, showing that the endpoints of each internal edge maintain a simple cyclic relationship. Instead of making formal definitions, an example should suffice here: Consider a region with edges $(k_1, k_2, \ldots, k_8)$ in cyclic order, where k_1, k_2, and k_5 are external, while k_3 is internal from s_3 to t_3, etc. Then $k_2 - 1 = k_1$, $s_3 = k_2$, $t_4 = k_5 - 1$, $s_6 = k_5$, and $t_8 = k_1 - 1$; and the points

$$(k_1, k_2, s_4, t_3, k_5 - 1, k_5, s_7, t_6, s_8, t_7, k_1 - 1)$$

are in cyclic order. Think about it.

22. Once we've inserted the chords, we can construct the graph by giving each region an identifying number.

At the same time we can reject cases with articulation points — namely, when a region has more than one exterior edge.

When this computation is finished, we will have discovered the total number of regions, l, which is also the total number of vertices in the corresponding squaregraph.

Note: We execute this code at a time when the local variables p, j, and s must not be changed. (They are part of the backtracking mechanism for delta sequences.) The author apologies for not subroutinizing everything.

⟨ Mark the regions, but **goto** *next* if there's an articulation problem 22 ⟩ ≡

```
for (l = 0, k = 1; k < eptr; k++)
  if (e[k].reg ≡ 0) {
    e[k].reg = ++l;
    for (q = e[k].succ; e[q].reg ≡ 0; q = e[q].succ) {
      if (q ≤ pp) goto next;
      e[q].reg = l;
    }
  }
```

This code is used in section 8.

23. Finishing up. The rest of this program is easy.

When we're ready to print a delta sequence, we know the total number of vertices in the corresponding squarefree graph, l. The total number of edges is also essentially known, because there are $(eptr - pp - 2)/2$ pairs of mates. Furthermore, each square has four sides; hence the number of squares, times 4, is the number of perimeter edges plus twice the number of nonperimeter edges.

```
⟨ Print del and its characteristics 23 ⟩ ≡
  ptot++;
  printf("%d:", ptot);
  for (k = 0; k < pp; k++) printf("␣%d", del[k]);
  printf("␣(%d%s),", rot, rfl ? "R" : "");
  printf("␣%d␣v,␣%d␣e,␣%d␣iv,␣%d␣sq\n", l, (eptr ≫ 1) − 1 − p, l − pp, (eptr ≫ 2) − p);
    /* that's vertices, edges, internal vertices, and squares */
  if (verbose) ⟨ Print the graph 24 ⟩;
```

This code is used in section 8.

```
24. ⟨ Print the graph 24 ⟩ ≡
  for (k = q = 1; k ≤ l; k++) {
    register int r;
    printf("␣%d␣--", k);
    while (e[q].reg ≠ k) q++;
    for (r = e[q].succ; ; r = e[r].succ) {
      if (r ≤ pp) break;
      printf("␣%d", e[mate(r)].reg);
      if (r ≡ q) break;
    }
    printf("\n");
  }
```

This code is used in section 23.

```
25. ⟨ Update the totals 25 ⟩ ≡
  q = pp/rot;
  if (rfl ≡ 0) q ≪= 1;      /* now q is the number of labeled varieties of del */
  pltot += q;
  vtot[l]++;
  vltot[l] += q;
  stot[(eptr ≫ 2) − p]++;
  sltot[(eptr ≫ 2) − p] += q;
```

This code is used in section 8.

26. Index.

⟨ Check the reflected delta sequence for canonicity and possible identity 10 ⟩ Used in section 9.
⟨ Construct the graph corresponding to the chords of *del* 16 ⟩ Used in section 8.
⟨ Determine the automorphisms, but **goto** *next* if *del* isn't canonical 9 ⟩ Used in section 8.
⟨ Generate all answers for (p, j) 5 ⟩ Used in section 2.
⟨ Global variables 3, 12 ⟩ Used in section 2.
⟨ Initialize for chord placement 17 ⟩ Used in section 16.
⟨ Initialize the delta sequence, with $\delta_0 = j$ 4 ⟩ Used in section 2.
⟨ Mark the regions, but **goto** *next* if there's an articulation problem 22 ⟩ Used in section 8.
⟨ Print the current *del* table if it's a solution, then **goto** *next* 8 ⟩ Used in section 5.
⟨ Print the graph 24 ⟩ Used in section 23.
⟨ Print *del* and its characteristics 23 ⟩ Used in section 8.
⟨ Subroutines 6, 13, 15, 18, 19, 20 ⟩ Used in section 2.
⟨ Update the totals 25 ⟩ Used in section 8.

SQUAREGRAPH