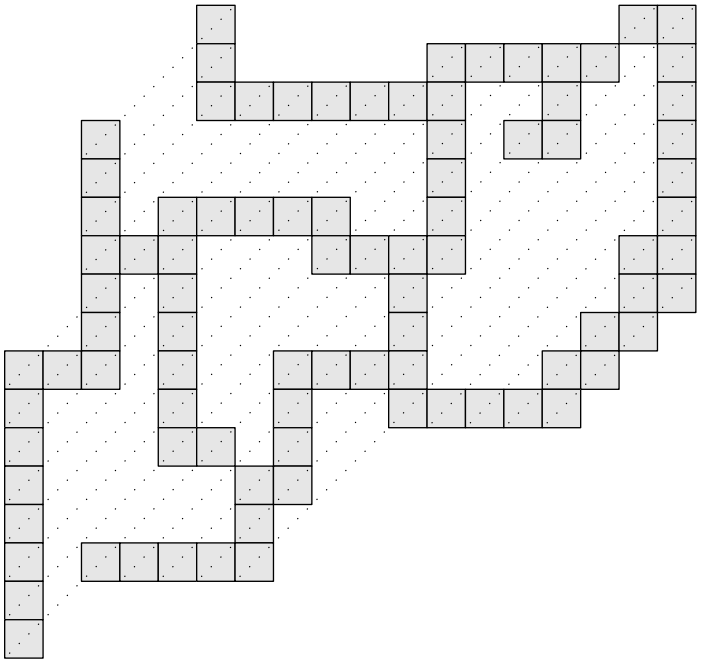
1. Introduction. The purpose of this program is to enumerate polyominoes of up to 30 cells. The method—possibly new?—is based on a sequence of diagonal slices through the shape, proceeding from the upper left to the lower right. For example, the curious polyomino



has 20 diagonal slices, which we will call

$$\begin{aligned}
&1002_{2},\\
&1002_{4},\\
&1002_{6},\\
&10002_{8},\\
&10223004_{13},\\
&112022003_{19},\\
&1010101002_{24},\\
&100010010023_{29},\\
&1000100010022_{34},\\
&10001000100202_{39},\\
&100010000102002_{44},\\
&10200103001405446_{54},\\
&100022033003004_{61},\\
&10230340000005_{67},\\
&1220020000003_{72},\\
&100010000002_{75},\\
&1000002_{77},\\
&100023_{80},\\
&12344_{85},\\
&1111_{89},
\end{aligned}$$

respectively. (This polyomino obviously has more than 30 cells, but large examples will help clarify the concepts needed in the program below.)

Each slice name consists of a string of hexadecimal digits, beginning and ending with a nonzero digit; it also has a numeric subscript (in decimal). The subscript counts the cells that lie on and above this diagonal slice. The nonzero hexadecimal digits represent cells in the current diagonal; such cells have the same digit if and only if they are rookwise connected as a consequence of the cells seen so far.

The main virtue of such an approach is that many polyominoes have identical slice names, hence they are essentially equivalent to each other with respect to the lower part of the diagram. For example, 2^{29} of the fixed 30-ominoes have the simple slice sequence $\texttt{1}_1$, $\texttt{1}_2$, ..., $\texttt{1}_{30}$. The total number of possible slices will therefore be substantially smaller than the total number of possible polyominoes.

This program enumerates polyominoes in fixed position, because the task of correcting such counts to take account of symmetries takes much less time.

Actually this program doesn't do the whole job of enumeration; it only outputs the edges of a certain directed acyclic graph. Another program reads that graph and computes the number of paths through it.

```
#define nmax 30    /* the size of polyominoes being counted, must not exceed 30 */
#include <stdio.h>
  ⟨ Type definitions 10 ⟩
  ⟨ Global variables 6 ⟩
  ⟨ Subroutines 2 ⟩
  main(int argc, char *argv[])
  {
    ⟨ Local variables 20 ⟩;
    ⟨ Scan the command line 4 ⟩;
    ⟨ Initialize 19 ⟩;
    ⟨ Compute 49 ⟩;
    ⟨ Print the results 62 ⟩;
  }
```

2. ⟨ Subroutines 2 ⟩ ≡

```
void panic(char *mess)
{
  fprintf(stderr, "%s!\n", mess);
  exit(-1);
}
```

See also sections 5, 15, 35, 52, 53, 54, and 59.

This code is used in section 1.

3. The base name of the output file should be given as a command-line argument. This name will actually be extended by `.0`, `.1`, ..., as explained below, because there might be an enormous amount of output.

4. ⟨ Scan the command line 4 ⟩ ≡

```
if (argc ≠ 2) {
  fprintf(stderr, "Usage: %s outfilename\n", argv[0]);
}
base_name = argv[1];
```

This code is used in section 1.

5. Connectivity. By definition, a polyomino is a rookwise connected set of cells. We'll want to restrict consideration to slices that can actually lead to a 30-omino; this means that we reject any slice for which one cannot connect all the so-far-unconnected pieces with $30 - m$ cells that lie below the m cells already accounted for.

Fortunately a simple algorithm is available to compute the minimum number of cells needed for connection. For example, a hexadecimal pattern like `1002` needs 5 such cells, and `10203` needs 6. A more complex pattern like `1002000301` also needs 6; and in general the subpattern $\texttt{10}^{g_1}\texttt{20}^{g_2}\ldots\texttt{k0}^{g_k}\texttt{1}$ needs $\sum_{j=1}^{k}(2g_j+1) - \max_{j=1}^{k}(2g_j+1)$, after which that entire subpattern can effectively be replaced by the single digit `1`.

Equal digits are always nested, in the sense that we cannot have a pattern like '`...1...2...1...2...`'. Therefore a simple stack-like approach suffices for the computation, given the hexadecimal digits $a[0]$, $a[1]$, $\ldots$, $a[len-1]$.

```
⟨ Subroutines 2 ⟩ +≡
  int conn_distance(int len, char *a)
  {
    register int j, k, m, acc = 0;
    stk[0] = a[0], m = 0;
    for (j = 1; j < len; j++) {
      for (k = 1; a[j] ≡ 0; j++) k += 2;
      acc += k;
      if (a[j] > stk[m]) {
        dst[m] = k;
        stk[++m] = a[j];
      } else {
        while (a[j] < stk[m]) {
          m--;
          if (dst[m] > k) k = dst[m];
        }
        if (a[j] ≠ stk[m]) panic("Oops, the program logic is screwed up");
        acc -= k;
      }
    }
    return acc;
  }
```

6. ⟨ Global variables 6 ⟩ ≡

```
  char stk[16], dst[16];      /* stacks for component numbers and distances */
```

See also sections 13, 14, 18, 21, 31, 34, 37, 40, 48, 51, 58, and 63.

This code is used in section 1.

7. The *conn_distance* of an initial, totally disconnected slice like $\texttt{102300400056}_6$, having k nonzero digits and length l, is $2l-1-k$. Thus we see that the set of all 30-ominoes has exactly 2^{14} possible initial states, corresponding to the bits of the odd numbers less than 2^{15}.

(Also, if we ignore connectivity but consider only the on-off pattern of cells, the same set of 2^{14} patterns accounts for all interior slices. The reason is that *conn_distance* also computes the minimum number of cells either above or below *or both* that will connect up a given pattern: Folding does not decrease connectivity.

```
#define maxlen ((nmax + 1) ≫ 1)      /* usable patterns won't be longer than this */
⟨ Place the initial slices 7 ⟩ ≡
  for (k = 1; k < (1 ≪ maxlen); k += 2)
    ⟨ Place the initial slice corresponding to the binary number k 47 ⟩;
```

This code is used in section 49.

8. Successive slices. Before we choose data structures for the main part of the computation, let's look at the key problem that faces us, namely the determination of all feasible slices that can follow a given slice. Ignoring the subscripts for the moment, what are the possible successors of a slice like, say, `1020032014`?

That hypothetical slice has 6 cells, namely 6 nonzero hexadecimal digits, and they are adjacent to 10 cells in the slice that comes immediately below. We must occupy at least one cell that is adjacent to each of the component classes `1`, `2`, `3`, and `4`, lest the whole polyomino be disconnected. Choosing the cell between the `3` and the `2`, and/or the cell between the `1` and the `4`, will kill two birds with one stone; so we can get by with as few as 2 cells in the slice that follows `1020032014`, and in such a case its pattern will be `1002`. The principle of inclusion and exclusion tells us how many ways there are to fulfill the connectivity constraint, namely

$$2^{10}-2^6-2^6-2^8-2^8+2^2+2^4+2^5+2^5+2^4+2^6-2^1-2^1-2^3-2^3+2^0=529.$$

For each of these ways to occupy the 10 adjacent cells, we can also add new cells that are not connected to any of the previous ones. For example, we could put a cell midway between the `2` and the `3` in `2003`; we could also occupy cells that lie off to the left or the right.

The calculations for outlying cells are the same for all slices: A pattern of k cells that extends l positions left of the cells adjacent to a given slice adds $2l-k$ to the *conn_distance*, plus a constant that depends only on the position of the leftmost occupied adjacent cell. A similar situation applies at the right.

It's important to notice that some successors of a slice will occur more than once. For example, there are five ways to go from $\texttt{1011}_8$ to $\texttt{1}_9$. This is one of the reasons I have such high hopes for the slice method.

9. Several slices will have the same pattern of digits but different subscripts. In such cases both slices have the same successors, except for cutoffs based on the subscripts.

Closer study of this situation reveals, in fact, that each pattern in a slice with subscript m occurs also with subscript $m+1$, unless its *conn_distance* equals $nmax-m$. The reason is that we could have added one more cell above the topmost slice.

Therefore each pattern has a definite "lifetime": It is born at a certain level m, and it dies after level $nmax-d$, where d is its *conn_distance*.

Our job then is to consider every pattern that is born at level m and to compute all successor patterns of cost at most $nmax-m$, where "cost" is the cell-count w plus the connection distance d. Such a successor will be born at level $m+w$. We output this information to a file, so that a postprocessor can rapidly count the number of paths through the network of possible slices.

If *nmax* is not too large, we could easily build the network ourselves and avoid any postprocessing stage. For example, the author's first attempt at such a program enumerated all fixed polynomials of size at most $nmax=15$ in less than half a second. And even when *nmax* is 25, the number of patterns turns out to be less than 300,000 and the number of slices less than 600,000. But the number of arcs between slices is 19 million, and these numbers grow exponentially as *nmax* increases.

10. Data structures. If you've been reading this commentary sequentially instead of hypertextually, you will now understand that the task of designing efficient data structures for our network of slices is quite interesting, although elementary.

Our program will go through the list of all slices with subscript m and successively generate their successors. I have a hunch that the total number of different slices will fit comfortably in memory. (In theory, a pattern of k cells can appear with $\binom{2k}{k}\frac{1}{k+1}$ different sequences of connection numbers, but in practice most of those sequences never arise. For example, a pattern like `1213` or `10213` is impossible.)

Therefore each pattern like `100201` has its own **pattern** node in the program below; that node can be addressed via a hash table.

The first digit of a pattern is always `1`, so we can omit it. We won't need patterns of length more than 15, so each pattern can be represented as a hexadecimal number with 14 digits (and trailing zeros). An additional byte is prepended, containing the pattern length; thus `100201` is actually represented by the 64-bit hexadecimal number `#0600201000000000`. To make this program work on 32-bit machines, a special type is declared in which we use 8 bytes instead of 16 nybbles.

```
#define length(pk) pk.bytes[0]
⟨ Type definitions 10 ⟩ ≡
  typedef struct {
    unsigned char bytes[8];
  } patkey;
```

See also sections 11 and 12.

This code is used in section 1.

11. In the present program I'm also putting an *aux* field into each pattern node, with the aim of eliminating a potentially long search when deciding whether a pattern has appeared before as a successor. This field takes up space, but it probably saves enough time to make it worthwhile.

```
⟨ Type definitions 10 ⟩ +≡
  typedef struct patt_struct {
    patkey key;      /* the hexadecimal pattern of connection digits */
    struct succ_struct *aux;      /* reference from a predecessor */
    struct patt_struct *link;      /* chain pointer used by lookup */
    struct patt_struct *next;      /* previous pattern with the same birthdate */
  } patt;
```

12. Each successor to the current pattern has a table entry telling its weight, date of death, and the number of ways in which it succeeds.

```
⟨ Type definitions 10 ⟩ +≡
  typedef struct succ_struct {
    patt *pat;      /* the successor pattern */
    char weight;      /* the number of cells it contains */
    char degree;      /* the replication number */
    char death;      /* level at which it last appears */
  } succ;
```

13. Here I use a generous upper bound on the number of possible successors of any pattern that might arise.

```
#define succ_size (maxlen * (1 ≪ maxlen))
⟨ Global variables 6 ⟩ +≡
  succ succ_table[succ_size];      /* successors of the current pattern */
  succ *succ_ptr;      /* the first unused slot in succ_table */
```

14. The pattern table is the big memory hog; when $nmax = 30$, more than three million patterns are involved.

```
#define patt_size 3002000  /* must exceed the number of patterns generated */
⟨ Global variables 6 ⟩ +≡
  patt patt_table[patt_size];  /* the patterns */
  patt *patt_list[nmax + 1];  /* lists of patterns sorted by birthdate */
  int patt_count[nmax + 1];  /* lengths of those lists */
  patt *patt_ptr = patt_table;  /* the first unused slot in patt_table */
  patt *bad_patt = patt_table + patt_size − 1;  /* the first unusable slot */
```

15. The *lookup* routine finds a node given its pattern. I'm using separate chains because I want patterns to be encoded as consecutive numbers in the output.

Parameters *len* and *a* are the pattern length and digits, as in the *conn_distance* routine. Parameter m is the birthdate of the pattern, if it happens to be new.

```
⟨ Subroutines 2 ⟩ +≡
  patt *lookup(int len, char *a, int m)
  {
    patkey key;
    register int j, l;
    register unsigned int h;
    register patt *p;
    register unsigned char *q;
    ⟨ Pack and hash the key from a and len 17 ⟩;
    p = hash_table[h];
    if (¬p) hash_table[h] = patt_ptr;
    else while (1) {
        for (j = 0, q = &(p→key.bytes[0]); j < l; j++, q++)
          if (*q ≠ key.bytes[j]) goto mismatch;
        return p;  /* successful search, the key matches */
      mismatch: if (¬p→link) {
          p→link = patt_ptr; break;
        }
        p = p→link;
      }
    ⟨ Insert a new pattern into patt_table and return 16 ⟩;
  }
```

16. ⟨ Insert a new pattern into *patt_table* and return 16 ⟩ ≡

```
  if (patt_ptr ≡ bad_patt) panic("Pattern␣memory␣overflow");
  patt_ptr→key = key;
  patt_ptr→next = patt_list[m];
  patt_list[m] = patt_ptr;
  patt_count[m]++;
  return patt_ptr++;
```

This code is used in section 15.

17. "Universal hashing" (TAOCP exercise 6.4–72) is used to get a good hash function, because most of the key bits are zero.

```
#define hash_width 20 /* lg of hash table size */
#define hash_mask ((1 ≪ hash_width) − 1)
⟨ Pack and hash the key from a and len 17 ⟩ ≡
  a[len] = 0, h = len ≪ (hash_width − 4);
  for (l = 1; l + l ≤ len; l++) {
    key.bytes[l] = (a[l + l − 1] ≪ 4) + a[l + l];
    h += hash[l][key.bytes[l]];
  }
  length(key) = len;
  h &= hash_mask;
```

This code is used in section 15.

18.
```
⟨ Global variables 6 ⟩ +≡
  patt *hash_table[hash_mask + 1]; /* heads of the chains */
  unsigned int hash[8][256]; /* random bits for universal hashing */
```

19. The random number generator used here doesn't have to be of sensational quality.

```
⟨ Initialize 19 ⟩ ≡
  m = 314159265;
  for (j = 1; j < 8; j++)
    for (k = 0; k < 256; k++) {
      m = 69069 * m + 1;
      hash[j][k] = m ≫ (32 − hash_width);
    }
```

See also sections 24 and 55.

This code is used in section 1.

20.
```
⟨ Local variables 20 ⟩ ≡
  register int j, k, l, m;
```

See also sections 23 and 45.

This code is used in section 1.

21. Computing the successors. Now let's turn to the details of the procedure sketched earlier. We will want some special data structures for that, in addition to the major structures used for patterns.

The cells of a possible successor slice will be numbered from 0 to 49, with cell 16 being adjacent-to-and-left-of the initial `1` in the slice whose successors are being found. (Any number exceeding $3 \times nmax/2$ will do in place of 50; we will have fewer than 15 new elements to the left of the pattern and fewer than 15 to the right, hence we have plenty of elbow room.) Cell j will be occupied if and only if $occ[j]$ is nonzero. Cell 0 is permanently unoccupied.

The cells adjacent to the previous pattern are $adjcell[0]$, $adjcell[1]$, ..., terminating with 0. The cells interior to but not adjacent to the previous pattern are $intcell[0]$, $intcell[1]$, ..., again terminating with 0. The nonadjacent cells to the left of the pattern are 15, 14, ..., and the nonadjacent cells to the right are $rightend$, $rightend + 1$, ... The array *dig* contains the hexadecimal pattern digits. The array *touched* tells how many occupied cells are adjacent to a given connected component. Finally, there's an array *first* with a slightly tricky meaning: $first[j] = k$ if $dig[j]$ was the leftmost appearance of component k.

For example, with pattern `1020032014`, we have $dig[16] = 1$, $dig[17] = 0$, $dig[18] = 2$, ..., $dig[25] = 4$; also $adjcell[0] = 16$, $adjcell[1] = 17$, ..., $adjcell[9] = 26$, $adjcell[10] = 0$; and $intcell[0] = 20$, $intcell[1] = 0$, $rightend = 27$. The values of *first* are zero except that $first[16] = 1$, $first[18] = 2$, $first[21] = 3$, and $first[25] = 4$. If $occ[14] = occ[16] = occ[19] = occ[20] = occ[25] = 1$ and other entries of *occ* are zero, we will have $touched[1] = 2$, $touched[2] = 1$, $touched[3] = 0$, and $touched[4] = 1$.

```
⟨ Global variables 6 ⟩ +≡
  char dig[50];      /* component numbers in previous slice */
  char adjcell[17];      /* cells adjacent to the previous slice */
  char intcell[13];      /* nonadjacent cells between adjacent ones */
  char first[50];      /* initial appearances of components that mustn't die */
  char occ[50];      /* is this cell occupied? */
  char touched[16];      /* occupied cells adjacent to components */
  char appeared[16];      /* auxiliary record of component appearances */
  char rightend;      /* the smallest nonadjacent cell at the right of the pattern */
  char leftbound, rightbound;      /* first and last occupied cells */
```

22. Given a pattern p whose successors need to be found, we begin by initializing the structures just mentioned.

Later we will mention a *leader* table, which might as well be initialized while we're setting up the other things. Any cell that is not adjacent to the previous slice should have $leader[j] = j$.

The program in this step does not clear $dig[rightend]$ to zero. No harm is done, because subsequent steps never look at $dig[j]$ for $j \ge rightend$.

```
⟨ Unpack and massage the pattern p→key 22 ⟩ ≡
  l = length(p→key);
  for (j = 2; j ≤ l; j += 2) {
    k = p→key.bytes[(j ≫ 1)];
    dig[j + 15] = k ≫ 4, dig[j + 16] = k & #f;
  }
  dig[l + 16] = 0;
  rightend = l + 17;
  for (j = rightend; j ≤ 31; j++) leader[j] = j;
  for (j = 1; j ≤ l; j++) touched[j] = appeared[j] = 0;
  for (j = 16, k = l = 0; j < rightend; j++) {
    first[j] = 0;
    if (dig[j]) {
      if (dig[j − 1] ≡ 0) adjcell[k++] = j;
      adjcell[k++] = j + 1;
      if (¬appeared[dig[j]]) first[j] = dig[j], appeared[dig[j]] = 1;
    }
    else if (dig[j − 1] ≡ 0) intcell[l++] = j, leader[j] = j;
  }
  adjcell[k] = intcell[l] = 0;
```

This code is used in section 27.

23.
```
⟨ Local variables 20 ⟩ +≡
  register patt *p;
```

24.
```
⟨ Initialize 19 ⟩ +≡
  dig[16] = 1;
  for (j = 1; j < 16; j++) leader[j] = j;
  for (j = 31; j < 50; j++) leader[j] = j;
```

25. A setting of the *occ* array for adjacent cells is valid if and only if each component is adjacent to at least one occupied cell. The simple algorithm in this step moves from one valid setting to the colexicographically next one, or does a **break** if the last valid setting has been considered.

It is convenient to set $touched[0]$ to such a large value that it cannot become zero.

```
⟨ Move to the next valid pattern of adjacent cells, or break 25 ⟩ ≡
  touched[0] = 128;
  for (k = 0; ; k++) {
    j = adjcell[k];
    if (occ[j]) touched[dig[j − 1]]−−, touched[dig[j]]−−, occ[j] = 0;
    else break;
  }
  if (¬j) break;      /* all were occupied, but now occ is entirely zero */
  touched[dig[j − 1]]++, touched[dig[j]]++, occ[j] = 1;
  ⟨ Move up to the next valid setting 26 ⟩;
```

This code is used in section 27.

26. We have essentially added 1 in binary notation, clearing *occ* bits to zero when "carrying." Now we might have to reset some of them in order to keep components alive.

(This computation is done also when we're getting started. Then *occ* is identically zero and k is at the end of the list of adjacent cells. In that case it finds the colexicographically smallest valid configuration.)

```
⟨ Move up to the next valid setting 26 ⟩ ≡
  for (k−−; k ≥ 0; k−−) {
    j = adjcell[k];
    if (¬touched[first[j]]) touched[dig[j − 1]]++, touched[dig[j]]++, occ[j] = 1;
  }
```

This code is used in sections 25 and 27.

27. Fans of top-down programming will have noticed that we've recently been working bottom-up. Now let's get back in balance by giving an outline of the successor generation process.

When this code is performed, m will be the current slice's subscript, namely the number of cells on and above the slice whose successor is being found.

The canonization process below will set l to the cost of the new pattern.

```
⟨ Generate all successors to p 27 ⟩ ≡
  {
    succ_ptr = succ_table;
    ⟨ Unpack and massage the pattern p¬key 22 ⟩;
    touched[0] = 128;
    ⟨ Move up to the next valid setting 26 ⟩;
    while (1) {
      ⟨ Canonize the new pattern based on adjacent occupied cells 41 ⟩;
      if (m + l > nmax) goto move;      /* prune it away, it makes only big polyominoes */
      ⟨ Insert the new pattern into the successor list 44 ⟩;
      ⟨ Run through all patterns of nonadjacent cells that might be relevant 28 ⟩;
    move: ⟨ Move to the next valid pattern of adjacent cells, or break 25 ⟩;
    }
    ⟨ List also the null successor, if appropriate 46 ⟩;
  }
```

This code is used in section 49.

28.

```
⟨ Run through all patterns of nonadjacent cells that might be relevant 28 ⟩ ≡
  while (1) {
    ⟨ Run through all patterns of nonadjacent cells at the left 29 ⟩;
  advance: for (k = 0; ; k++) {
      j = intcell[k];
      if (occ[j]) occ[j] = 0;
      else break;
    }
    if (¬j) break;
    occ[j] = 1;
    ⟨ Canonize the new pattern based on adjacent and interior occupied cells 42 ⟩;
    if (m + l > nmax) goto advance;
    ⟨ Insert the new pattern into the successor list 44 ⟩;
  }
```

This code is used in section 27.

29. Here I make use of Mathematics, although it saves only a little computation: When the **while** loop in this section ends, $occ[j]$ will be zero for $leftbound < j < 16$, because of a property of the *conn_distance* function that was mentioned earlier.

```
⟨ Run through all patterns of nonadjacent cells at the left 29 ⟩ ≡
  save_leftbound = leftbound;
  while (1) {
    ⟨ Run through all patterns of nonadjacent cells at the right 30 ⟩;
    for (j = 15; ; j--) {
      if (occ[j]) occ[j] = 0;
      else break;
    }
    occ[j] = 1;
    if (j < leftbound) leftbound = j;
    if (rightbound - leftbound ≥ maxlen) break;
    ⟨ Canonize the new pattern based on all occupied cells 43 ⟩;
    if (m + l > nmax) break;
    ⟨ Insert the new pattern into the successor list 44 ⟩;
  }
  occ[leftbound] = 0;      /* this clears out the whole left end */
  leftbound = save_leftbound;
```

This code is used in section 28.

30.

```
⟨ Run through all patterns of nonadjacent cells at the right 30 ⟩ ≡
  save_rightbound = rightbound;
  rightbound = rightend;
  occ[rightend] = 1;
  while (1) {
    if (rightbound - leftbound ≥ maxlen) break;
    ⟨ Canonize the new pattern based on all occupied cells 43 ⟩;
    if (m + l > nmax) break;
    ⟨ Insert the new pattern into the successor list 44 ⟩;
    for (j = rightend; ; j++) {
      if (occ[j]) occ[j] = 0;
      else break;
    }
    occ[j] = 1;
    if (j > rightbound) rightbound = j;
  }
  occ[rightbound] = 0;      /* this clears out the whole right end */
  rightbound = save_rightbound;
```

This code is used in section 29.

31.

```
⟨ Global variables 6 ⟩ +≡
  char save_leftbound, save_rightbound;      /* boundaries within the previous pattern */
```

32. Canonization. Once a sequence of occupied cells has been proposed, we need to represent it in canonical form as a sequence of component digits. For example, if we occupy the four cells marked `x` in

```
1020032014
x00000x00xx
```

then the components `23` and `14` are merged, so the new digits are `10000020011`. (A canonical sequence always numbers the components `1`, `2`, ... in order as they appear from left to right.)

Our first task is trivial: We occasionally need to locate the leftmost and rightmost occupied cells.

```
⟨ Find the proper leftbound and rightbound 32 ⟩ ≡
  for (j = 16; ¬occ[j]; j++) ;
  leftbound = j;
  for (j = rightend − 1; ¬occ[j]; j−−) ;
  rightbound = j;
```

This code is used in sections 41 and 42.

33. Our next job is more interesting: After changing the status of adjacent cells, we need to merge components that are being joined.

A simple "union–find" algorithm is appropriate for this task. Each occupied cell will point to the smallest cell in its class, and the cells of a class are also circularly linked.

However, we make a quick exit to *done* if the cost is obviously so high that the rest of the calculation cannot succeed.

```
⟨ Find the new classes, or goto done 33 ⟩ ≡
  for (j = leftbound − 1, l = m; j ≤ rightbound; j++) {
    if (occ[j]) l++, circ[j] = leader[j] = j;
    if (dig[j]) {
      if (occ[j]) appeared[dig[j]] = j;
      else if (occ[j + 1]) appeared[dig[j]] = j + 1;
    }
  }
  if (l > nmax) goto done;      /* pointless to continue */
  for (j = leftbound − 1; j ≤ rightbound; j++)
    if ((k = dig[j])) {
      if (occ[j] ∧ leader[j] ≠ leader[appeared[k]]) merge(leader[j], leader[appeared[k]]);
      if (occ[j + 1] ∧ leader[j + 1] ≠ leader[appeared[k]]) merge(leader[j + 1], leader[appeared[k]]);
    }
```

This code is used in section 41.

34.

```
⟨ Global variables 6 ⟩ +≡
  char circ[50];      /* circular links for component classes */
  char leader[50];      /* class representatives */
```

35. ⟨ Subroutines 2 ⟩ +≡

```
void merge(int j, int k)
{
  register int p, t;
  if (j < k) {
    for (p = t = circ[k]; p ≠ k; p = circ[p]) leader[p] = j;
    leader[p] = j;
    circ[p] = circ[j], circ[j] = t;
  } else {
    for (p = t = circ[j]; p ≠ j; p = circ[p]) leader[p] = k;
    leader[p] = k;
    circ[p] = circ[k], circ[k] = t;
  }
}
```

36. The *newpat* array, which contains the canonical component numbers starting with `1`, can now be written down without further ado.

⟨ Establish the *newpat* array 36 ⟩ ≡

```
for (j = leftbound, k = l = 0; j ≤ rightbound; j++, k++)
  if (¬occ[j]) newpat[k] = 0;
  else if (leader[j] < j) newpat[k] = class[leader[j]];
  else newpat[k] = class[j] = ++l;
```

This code is used in section 39.

37. ⟨ Global variables 6 ⟩ +≡

```
char class[50];      /* canonical component number */
```

38. We aren't done yet, however. The reverse of a pattern is computationally equivalent to the pattern itself; so we gain a factor of roughly two (in both time and space) by switching to the reverse pattern when it is lexicographically smaller.

⟨ Establish the *backpat* array 38 ⟩ ≡

```
for (j = rightbound, k = l = 0; j ≥ leftbound; j--, k++)
  if (¬occ[j]) backpat[k] = 0;
  else if (class[leader[j]] & #10) backpat[k] = class[leader[j]] & #f;
  else backpat[k] = ++l, class[leader[j]] = l + #10;
```

This code is used in section 39.

39. Here then is how we compute the cost of a purported successor. (Again we bypass computation when a detailed calculation would be fruitless.)

```
⟨ Determine the cost of the new pattern 39 ⟩ ≡
  for (j = leftbound, l = 0; j ≤ rightbound; j++)
    if (occ[j]) l++;
  if (m + l ≤ nmax) {
    weight = l;
    ⟨ Establish the newpat array 36 ⟩;
    ⟨ Establish the backpat array 38 ⟩;
    len = k;      /* at this point k is the pattern length */
    bestpat = newpat;
    for (j = 1; j < k; j++)
      if (newpat[j] < backpat[j]) break;
      else if (newpat[j] > backpat[j]) {
        bestpat = backpat;
        break;
      }
    k = conn_distance(k, bestpat);
    l = weight + k;
  }
```

This code is used in sections 41, 42, and 43.

40.

```
⟨ Global variables 6 ⟩ +≡
  char newpat[16];      /* canonical sequence for the new pattern */
  char backpat[16];     /* canonical sequence for its reversal */
  char *bestpat;        /* the lexicographically smaller */
  int weight;           /* the number of occupied cells */
```

41.

```
⟨ Canonize the new pattern based on adjacent occupied cells 41 ⟩ ≡
  ⟨ Find the proper leftbound and rightbound 32 ⟩;
  if (rightbound − leftbound ≥ maxlen) {
    l = nmax; goto done;      /* too long, so we make the cost huge */
  }
  ⟨ Find the new classes, or goto done 33 ⟩;
  ⟨ Determine the cost of the new pattern 39 ⟩;
done:
```

This code is used in section 27.

42. At this point the new classes of adjacent cells have already been determined. Interior cells cannot make $rightbound - leftbound \geq maxlen$.

```
⟨ Canonize the new pattern based on adjacent and interior occupied cells 42 ⟩ ≡
  ⟨ Find the proper leftbound and rightbound 32 ⟩;
  ⟨ Determine the cost of the new pattern 39 ⟩;
```

This code is used in section 28.

43. And at this point the *leftbound* and *rightbound* are already known.

```
⟨ Canonize the new pattern based on all occupied cells 43 ⟩ ≡
  ⟨ Determine the cost of the new pattern 39 ⟩;
```

This code is used in sections 29 and 30.

44. Loose ends. We have finished the complicated decision-making that goes into listing all successors of a given slice, but we still haven't actually generated any successors. Now we're ready to do that simple task, thereby unmasking the mystery of *aux*.

```
⟨ Insert the new pattern into the successor list 44 ⟩ ≡
  q = lookup(len, bestpat, m + weight);
  if (q→aux) q→aux→degree++;      /* been there, done that */
  else {
    s = succ_ptr++;
    s→pat = q;
    q→aux = s;
    s→degree = 1;
    s→weight = weight;
    s→death = nmax − l + weight;
  }
```

This code is used in sections 27, 28, 29, and 30.

45.
```
⟨ Local variables 20 ⟩ +≡
  register patt *q;
  register succ *s;
```

46. The special successor pattern Λ, of cost 0, is added to the list if slice p had only one component. This will be true if and only if *appeared*[2] is zero. It means, "We can stop now if we like, having generated a polyomino of weight m."

```
⟨ List also the null successor, if appropriate 46 ⟩ ≡
  if (¬appeared[2]) {
    s = succ_ptr++;
    s→pat = Λ, s→degree = 1, s→weight = 0, s→death = m;
  }
```

This code is used in section 27.

47. Placing initial slices is complicated by the fact that we want to gain a factor of two by symmetry. Thus if k is palindromic, we start with pattern k itself; otherwise we consider k and its reflection but with double weight. The latter case is essentially the same as a degree-2 transition from the null state.

The total number of initial transitions, which is also the total number of slices that will appear at level $nmax$, is

$$2^{t-2} + \begin{cases} 2^{t/2} - 1, & \text{if } t \text{ is even,} \\ 2^{(t-1)/2} + 2^{(t-3)/2} - 1, & \text{if } t \text{ is odd,} \end{cases}$$

where $t = maxlen = \lceil nmax/2 \rceil$. For example, when $nmax$ is 30 this number is 8383.

```
⟨ Place the initial slice corresponding to the binary number k 47 ⟩ ≡
  {
    m = k;
    for (j = l = 0; m; j++, m >>= 1)
      if (m & 1) newpat[j] = ++l;
      else newpat[j] = 0;
    len = j;
    weight = l;
    for (j--; j ≥ 0; j--, m++)
      if (newpat[j]) backpat[m] = l + 1 − newpat[j];
      else backpat[m] = 0;
    mult = 1;
    for (j = 1; j < len; j++)
      if (newpat[j] < backpat[j]) {
        mult = 2;
        break;
      } else if (newpat[j] > backpat[j]) goto bypass;
    ⟨ Record an initial transition to newpat with degree mult 57 ⟩;
  bypass: ;
  }
```

This code is used in section 7.

48.

```
⟨ Global variables 6 ⟩ +≡
  int len;      /* the pattern length */
  int mult;     /* its multiplicity */
```

49. The heart of the computation is, of course, the process of generating the non-initial slices.

```
⟨ Compute 49 ⟩ ≡
  ⟨ Place the initial slices 7 ⟩;
  for (m = 1; ; m++) {
    printf("␣%d␣new␣patterns␣on␣level␣%d␣(%d,%d)\n", patt_count[m], m, patt_ptr − patt_table, arcs);
    ⟨ Record the arrival of a new m 56 ⟩;
    if (m ≡ nmax) break;
    for (p = patt_list[m]; p; p = p→next) {
      ⟨ Generate all successors to p 27 ⟩;
      ⟨ Output all transitions from p 50 ⟩;
    }
  }
```

This code is used in section 1.

50. ⟨ Output all transitions from p 50 ⟩ ≡

```
⟨ Record p as the current predecessor 60 ⟩;
for (s = succ_table; s < succ_ptr; s++) {
  ⟨ Record a transition to s 61 ⟩;
  if (s→pat) s→pat→aux = Λ;
}
```

This code is used in section 49.

51. Output. Finally we must deal with transition records, which are sent to a file for subsequent processing. That file might be huge, so it is generated in a compact binary format. Each sequence of transitions from a pattern is specified by one word identifying that pattern followed by one word for each successor pattern.

In fact several gigabytes of output are generated when $nmax = 30$, and my Linux system frowns on files of length greater than $2^{31} - 1 = 2147483647$. Therefore this program breaks the output up into a sequence of files called `foo.0`, `foo.1`, ..., each at most one large gigabyte in size. (That's one GGbyte $= 2^{30}$ bytes.)

If the special variable *verbose* is nonzero, transitions are also displayed in symbolic form on standard output.

```
#define filelength_threshold 10000000      /* in tetrabytes */
⟨ Global variables 6 ⟩ +≡
  int verbose = 0;      /* set nonzero for debugging */
  FILE *out_file;      /* the output file */
  unsigned int buf;      /* place for binary output */
  int words_out;      /* the number of tetrabytes output in current output file */
  int file_extension;      /* the number of GGbytes output */
  char *base_name, filename[100];
```

52.
```
⟨ Subroutines 2 ⟩ +≡
  void open_it()
  {
    sprintf(filename, "%.90s.%d", base_name, file_extension);
    out_file = fopen(filename, "wb");
    if (¬out_file) {
      fprintf(stderr, "I␣can't␣open␣file␣%s", filename);
      panic("␣for␣output");
    }
    words_out = 0;
  }
```

53.
```
⟨ Subroutines 2 ⟩ +≡
  void close_it()
  {
    if (fclose(out_file) ≠ 0) panic("I␣couldn't␣close␣the␣output␣file");
    printf("[%d␣bytes␣written␣on␣file␣%s.]\n", 4 * words_out, filename);
  }
```

54.
```
⟨ Subroutines 2 ⟩ +≡
  int out_it()
  {
    if (words_out ≡ filelength_threshold) {
      close_it();
      file_extension++;
      open_it();
    }
    words_out++;
    return fwrite(&buf, sizeof(unsigned int), 1, out_file) ≡ 1;
  }
```

55.
```
⟨ Initialize 19 ⟩ +≡
  open_it();
```

56. How should we encode the binary output? Each transition has a multiplicity, and the multiplicity can get as large as 30. Therefore we will devote 5 bits to that piece of information. We can also give special meaning to the code numbers 0 and 31 if they happen to appear in the high-order 5 bits of a 32-bit word.

(At first I thought the maximum multiplicity was 16, because of examples like $\texttt{1020304050607}_7 \to \texttt{1}_{30}$ or $\texttt{11223344556677}_{21} \to \texttt{1}_{30}$ or $\texttt{101010101010101}_{29} \to \texttt{1}_{30}$. But then I realized that there are 28 ways to go from $\texttt{1}_1$ to $\texttt{10203040506007}_8$ or to $\texttt{10203040506077}_9$, because of the way we collapse symmetric slices together. Still later I encountered the examples $\texttt{11}_3 \to \texttt{10002003004005}_8$, $\texttt{11}_3 \to \texttt{10020003004005}_8$, $\texttt{1011}_7 \to \texttt{100000200003}_{10}$, $\texttt{1111}_7 \to \texttt{100000200003}_{10}$. I believe these are the only four cases of degree ≥ 30, but the program now checks explicitly to make sure that I haven't miscalculated again.)

For the main body of information, we can take advantage of the fact that new patterns arise consecutively. Thus if bit 6 is 0, it means, "The successor is the next new pattern; here are its birth and death dates." But if bit 6 is 1 it means, "The low-order 26 bits are the serial number of the successor pattern, whose birth and death dates you already know."

```
#define new_pred_code 0      /* high 5 when new slice is the predecessor */
#define new_level_code 31    /* high 5 when m increases */
⟨ Record the arrival of a new m 56 ⟩ ≡
  buf = (new_level_code ≪ 27) + m;
  if (¬out_it()) panic("Bad write of newlevel message");
```

This code is used in section 49.

57. The first pattern has serial number 1, not 0, because we let 0 stand for the sink vertex.

```
#define patt_code(q) (((q) − patt_table) + 1)
⟨ Record an initial transition to newpat with degree mult 57 ⟩ ≡
  l = nmax − conn_distance(len, newpat);
  q = lookup(len, newpat, weight);
  if (patt_code(q) ≠ ++prev_pat) panic("Out of sync");
  if (verbose) {
    printf("-%s>", mult ≡ 2 ? "2" : mult ≠ 1 ? "?" : "");
    print_slice(q, weight, l);
  }
  buf = (mult ≪ 27) + (weight ≪ 8) + l;      /* multiplicity, birth, death */
  if (¬out_it()) panic("Bad write of initial transition");
  slices += l − weight + 1;
```

This code is used in section 47.

58. ⟨ Global variables 6 ⟩ +≡

```
int prev_pat;      /* the number of patterns encountered so far */
```

59. The verbose output marks the first appearance of a slice by printing both birth and death dates as a range of subscripts. Later it will use a single subscript within that interval. (For example, if $nmax = 30$ the simple pattern `1` will be shown first as '`1:1..30`', and the pattern `12` will be shown first as '`12:2..29`'. But later when `1` occurs as a successor of pattern `12`, it will show up as '`1:3`', meaning that $\texttt{1}_3$ is a successor of $\texttt{12}_2$, $\texttt{1}_4$ is a successor of $\texttt{12}_3$, etc.)

```
⟨ Subroutines 2 ⟩ +≡
  void print_slice(patt *p, int m, int death)
  {
    register int j;
    for (j = 0; j < length(p→key); j++)
      printf("%x", j ≡ 0 ? 1 : j & 1 ? p→key.bytes[(j + 1) ≫ 1] ≫ 4 : p→key.bytes[j ≫ 1] & #f);
    if (death) printf(":%d..%d\n", m, death);
    else printf(":%d\n", m);
  }
```

60. ⟨ Record p as the current predecessor 60 ⟩ ≡

```
  {
    if (verbose) print_slice(p, m, 0);
    buf = (new_pred_code ≪ 27) + patt_code(p);
    if (¬out_it()) panic("Bad write of predecessor pattern");
  }
```

This code is used in section 50.

61. ⟨ Record a transition to s 61 ⟩ ≡

```
  {
    if (verbose) {
      if (s→degree ≡ 1) printf("->");
      else printf("-%d>", s→degree);
      if (¬s→pat) printf("0:%d\n", m);
      else print_slice(s→pat, m + s→weight, patt_code(s→pat) > prev_pat ? s→death : 0);
    }
    if (s→degree > 30) panic("Surprisingly large arc multiplicity");
    if (¬s→pat) buf = (1 ≪ 27) + (1 ≪ 26);
    else if (patt_code(s→pat) ≤ prev_pat) buf = (s→degree ≪ 27) + (1 ≪ 26) + patt_code(s→pat);
    else {
      prev_pat++, buf = (s→degree ≪ 27) + ((s→weight + m) ≪ 8) + s→death;
      slices += s→death − (s→weight + m) + 1;
    }
    if (¬out_it()) panic("Bad write of transition");
    arcs++;
  }
```

This code is used in section 50.

62. Hooray, we are done.

```
⟨ Print the results 62 ⟩ ≡
  if (patt_ptr ≠ patt_table + prev_pat) panic("Output␣out␣of␣sync");
  printf("All␣done!\n");
  printf("␣%d␣patterns␣generated,", prev_pat);
  printf("␣%d␣slices,", slices);
  printf("␣%d␣arcs.\n", arcs);
  close_it();
```

This code is used in section 1.

63. Multiplicity of arcs is not taken into account.

```
⟨ Global variables 6 ⟩ +≡
  int slices;      /* total number of slices */
  int arcs;      /* total number of arcs (double precision) */
```

64. Note added two weeks later: This program, though interesting, is obsolete. Please see POLYNUM, which runs hundreds of times faster when $n = 30$ and faster yet for larger values of n.

65. Index.

POLYENUM