1. Introduction. The purpose of this program is to implement a pretty algorithm that has a very pleasant theory. But I apologize at the outset that the algorithm seems to be rather subtle, and I have not been able to think of any way to explain it to dummies. Readers who like discrete mathematics and computer science are encouraged to persevere nonetheless.

An overview of the relevant theory appears in a paper called "Deconstructing coroutines," by D. E. Knuth and F. Ruskey, but this program tries to be self-contained. Earlier versions of the ideas were embedded in now-obsolete programs called KODA-RUSKEY and LI-RUSKEY, written in June, 2001.

```
#include <stdio.h>
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 13 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ Local variables 5 ⟩;
    ⟨ Parse the command line 3 ⟩;
    ⟨ Initialize the data structures 7 ⟩;
    ⟨ Generate the answers 24 ⟩;
    return 0;
  }
```

2. Given a digraph that is *totally acyclic*, in the sense that it has no cycles when we ignore the arc directions, we want to find all ways to label its vertices with 0s and 1s in such a way that $x \to y$ implies $bit[x] \le bit[y]$. Moreover, we want to list all such labelings as a Gray path, changing only one bit at a time. The algorithm below does this, with an extra proviso: Given a designated "root" vertex v, $bit[v]$ begins at 0 and changes exactly once.

For brevity, a totally acyclic digraph is called a *tad*, and a connected tad is called a *spider*.

The simple three-vertex spider with $x \to y \leftarrow z$ has only five such labelings, and they form a Gray path in essentially only one way, namely $(000, 010, 011, 111, 110)$. This example shows that we cannot require the Gray path to end at a convenient prespecified labeling like $11\ldots1$; and the dual graph, obtained by reversing all the arrows and complementing all the bits, shows that we can't require the path to start at $00\ldots0$. [Generalizations of this example, in which the vertices are $\{x_1, x_2, \ldots, x_n\}$ and each arc is either $x_{k-1} \to x_k$ or $x_{k-1} \leftarrow x_k$, have solutions related to continued fractions. Interested readers will enjoy working out the details.]

It is convenient to describe a tad by using a variant of right-Polish notation, where a dot means "put a new node on the stack" and where a `+` or `-` sign means "draw an arc $x \leftarrow y$ (`+`) or $x \to y$ (`-`) and remove y from the stack," if x and y are the top stack elements. For example, a digraph with four vertices and no arcs is represented by '`....`'; the digraph $1 \to 2 \leftarrow 3$ is '`...+-`', and $2 \to 1 \leftarrow 3$ is '`..+.+`', numbering the dots from left to right. This numbering corresponds to *preorder* of the forest that is obtained if we ignore arc directions.

The Polish notation implicitly specifies a hierarchical order, if the `+` and `-` operations make y a child of x. Using this tree structure, each node of the digraph defines a subtree, and that subtree is a spider. The Gray path we construct is obtained by judiciously combining the Gray paths obtained from the spiders.

3. In the following program, the parent of node k is $par[k]$, and the arc between k and its parent goes toward $par[k]$ if $sign[k] = 1$, toward k if $sign[k] = 0$. The spider corresponding to k consists of nodes k through $scope[k]$, inclusive. The Gray path corresponding to this spider will be called G_k.

While we build the data structures, we might as well compute also $rchild[k]$ and $lsib[k]$, the rightmost child and left sibling of node k. Then we have a triply linked tree.

```
#define maxn 100 /* limit on number of vertices */
#define abort(f,d,n) { fprintf(stderr,f,d); exit(-n); }
⟨ Parse the command line 3 ⟩ ≡
  {
    register char *c;
    if (argc < 2 ∨ argc > 3 ∨ (argc ≡ 3 ∧ sscanf(argv[2],"%d",&verbose) ≠ 1))
      abort("Usage: %s graphspecification [verbosity]\n",argv[0],1);
    for (c = argv[1], j = n = 0; *c; c++)
      switch (*c) {
      case '.': if (n ≡ maxn - 1) abort("Sorry, I can only handle %d vertices!\n",maxn - 1,2);
        stack[j++] = ++n; break;
      case '+': case '-': if (j < 2) abort("Parsing error: '%s' should start with '.'!\n",c,3);
        j--, k = stack[j], l = stack[j - 1];
        sign[k] = (*c ≡ '+' ? 1 : 0);
        par[k] = l, lsib[k] = rchild[l], rchild[l] = k;
        scope[k] = n;
        break;
      default: abort("Parsing error: '%s' should start with '.' or '+' or '-'!\n",c,4);
      }
    scope[0] = n, sign[0] = 1, rchild[0] = stack[--j];
    for (k = n; j ≥ 0; j--) {
      l = stack[j], scope[l] = k, k = l - 1;
      if (j > 0) lsib[l] = stack[j - 1];
    }
  }
```

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
int par[maxn]; /* the parent of k */
int sign[maxn]; /* 0 if par[k] → k, 1 if par[k] ← k */
int scope[maxn]; /* rightmost element of the spider k */
int stack[maxn]; /* vertices whose scope is not yet set */
int rchild[maxn], lsib[maxn]; /* tree links for traversal */
int verbose; /* controls the amount of output */
```

See also sections 9, 12, 18, 25, and 32.

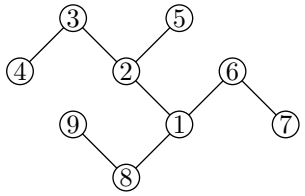
This code is used in section 1.

5. ⟨ Local variables 5 ⟩ ≡

```
register int j, k, l = 0; /* heavily-used miscellaneous indices */
int n; /* size of the input graph */
```

This code is used in section 1.

6. Consider the example spider



in which all arcs are directed upward; it could be written `....+-.--..+-..-+` in Polish notation. Vertex 1 is the root. A nonroot vertex k is called *positive* if $par[k] \to k$ and *negative* if $par[k] \leftarrow k$; thus $\{2,3,5,6,9\}$ are positive in this example, and $\{4,7,8\}$ are negative.

We write $j \to^* k$ if there is a directed path from j to k. Removing all vertices j such that $j \to^* k$ disconnects spider k into a number of pieces having positive roots; in our example, removing $\{1,8\}$ leaves three components rooted at $\{2,6,9\}$. We call these roots the set of *positive vertices near* k, and denote that set by U_k. Similarly, the *negative vertices near* k are obtained when we remove all j such that $k \to^* j$; the set of resulting roots, denoted by V_k, is $\{4,7,8\}$ in our example.

Why are the sets U_k and V_k so important? Because the labelings of the kth spider for which $bit[k] = 0$ are precisely those that we obtain by setting $bit[j] = 0$ for all $j \to^* k$ and then labeling each spider u for $u \in U_k$. Similarly, all labelings for which $bit[k] = 1$ are obtained by setting $bit[j] = 1$ for all $k \to^* j$ and labeling each spider v for $v \in V_k$.

Thus if n_k denotes the number of labelings of spider k, we have $n_k = \prod_{u\in U_k} n_u + \prod_{v\in V_k} n_v$.

Every positive child of k appears in U_k, and every negative child appears in V_k. These are called the *principal* elements of U_k and V_k. Every nonprincipal member of U_k is a member of U_v for some unique principal vertex $v \in V_k$. Similarly, every nonprincipal member of V_k is a member of V_u for some unique principal vertex $u \in U_k$. For example, 9 is a nonprincipal member of U_1 and it also belongs to U_8; 4 is a nonprincipal member of V_1 and it also belongs to V_2.

If k is a root of the given digraph, we say that k's parent is 0. This dummy vertex 0 is assumed to have arcs to all such k, and it follows that U_0 is the collection of those root vertices; the total number of labelings is therefore $\prod_{u\in U_0} n_u$. According to this convention, the root vertices are considered to be positive. We also regard 0 as negative.

For example, the sample spider above has the following characteristics:

k	$sign[k]$	$scope[k]$	$par[k]$	$rchild[k]$	$lsib[k]$	U_k	V_k	n_k
0	1	9	0	1		$\{1\}$	$\{4,7,8\}$	
1	0	9	0	8	0	$\{2,6,9\}$	$\{4,7,8\}$	$48+12=60$
2	0	5	1	5	0	$\{3,5\}$	$\{4\}$	$6+2=8$
3	0	4	2	4	0	$\emptyset$	$\{4\}$	$1+2=3$
4	1	4	3	0	0	$\emptyset$	$\emptyset$	$1+1=2$
5	0	5	2	0	3	$\emptyset$	$\emptyset$	$1+1=2$
6	0	7	1	7	2	$\emptyset$	$\{7\}$	$1+2=3$
7	1	7	6	0	0	$\emptyset$	$\emptyset$	$1+1=2$
8	1	9	1	9	6	$\{9\}$	$\emptyset$	$2+1=3$
9	0	9	8	0	0	$\emptyset$	$\emptyset$	$1+1=2$

7. We don't want to compute the sets U_1, ..., U_n explicitly, because the total number of elements $|U_1| + \cdots + |U_n|$ can be $\Omega(n^2)$ in cases like $.^{n/2}(.+)^{n/2}+^{n/2-1}$. But luckily for us, there is a nice way to represent all of those sets implicitly, computing the representation in linear time.

Suppose u is a positive vertex, not a root, so that $u \leftarrow v_1$ where v_1 is u's parent and $v_1 \neq 0$. If v_1 is negative, let v_2 be the parent of v_1, and continue until reaching a positive vertex v_j. We call v_j the *positive progenitor* of v_1; it is also the positive progenitor of v_2, ..., v_{j-1}, and itself. By definition, $u \in U_k$ if and only if $k \in \{v_1, \ldots, v_j\}$. It follows that $U_k = U_{k'} \cap [k \,..\, scope[k]]$ when k' is the positive progenitor of k.

We can therefore represent all the sets U_k by linking their elements together explicitly whenever k is a positive vertex; such sets U_k are disjoint. Then if we compute $umax[k]$ for *every* vertex k, namely the index of the largest element of U_k, the set U_k will consist of $umax[k]$, $prev[umax[k]]$, $prev[prev[umax[k]]]$, etc., proceeding until reaching an element less than k.

One pass through the forest in preorder suffices to compute the *prev* values. A second pass in reverse postorder suffices to compute each *umax*, because postorder traverses nodes in order of their scopes.

A similar idea works, of course, for V_1, ..., V_n, using *negative* progenitors.

```
⟨ Initialize the data structures 7 ⟩ ≡
  for (j = 1; j ≤ n; j++) {
    k = par[j];
    if (sign[j] ≡ 0) {
      ppro[j] = j, npro[j] = npro[k];
      if (k) prev[j] = umax[ppro[k]], umax[ppro[k]] = j;
      else prev[j] = lsib[j];      /* special case when j is a root */
    } else {
      npro[j] = j, ppro[j] = ppro[k];
      prev[j] = vmax[npro[k]], vmax[npro[k]] = j;
    }
  }
  ⟨ Fill in all umax and vmax links, traversing in reverse postorder 8 ⟩;
```

See also sections 10, 11, and 16.

This code is used in section 1.

8. Tree traversal is great fun, when it works.

```
⟨ Fill in all umax and vmax links, traversing in reverse postorder 8 ⟩ ≡
  lsib[0] = −1;      /* sentinel */
  ptr[0] = vmax[0];
  umax[0] = rchild[0];
  for (j = rchild[0]; ; ) {
    if (sign[j] ≡ 0) {
      ptr[j] = umax[j];      /* this pointer will run through U_j */
      k = npro[j], l = ptr[k];
      while (l > scope[j]) l = prev[l];
      ptr[k] = l;
      if (l > j) vmax[j] = l;
    } else {
      ptr[j] = vmax[j];      /* this pointer will run through V_j */
      k = ppro[j], l = ptr[k];
      while (l > scope[j]) l = prev[l];
      ptr[k] = l;
      if (l > j) umax[j] = l;
    }
    if (rchild[j]) j = rchild[j];      /* now we move to the next node */
    else {
      while (¬lsib[j]) j = par[j];
      j = lsib[j];
      if (j < 0) break;
    }
  }
```

This code is used in section 7.

9. The sample spider leads, for example, to the following values:

k	$ppro[k]$	$npro[k]$	$prev[k]$	$umax[k]$	$vmax[k]$
0	0	0	0	1	8
1	1	0	0	9	8
2	2	0	0	5	4
3	3	0	0	0	4
4	3	4	0	0	0
5	5	0	3	0	0
6	6	0	2	0	7
7	6	7	4	0	0
8	1	8	7	9	0
9	9	8	6	0	0

```
⟨ Global variables 4 ⟩ +≡
  int ppro[maxn], npro[maxn];      /* progenitors */
  int prev[maxn];      /* previous element in the same progenitorial list */
  int ptr[maxn];      /* current element in such a list */
  int umax[maxn], vmax[maxn];      /* rightmost elements in U_k, V_k */
```

10. Aha! We can begin to see how to get the desired Gray path. The well-known reflected Gray code for mixed-radix number systems tells us how to obtain a path P_k of length $\prod_{u\in U_k} n_u$ for the labelings of spider k with $bit[k]=0$, as well as a path Q_k of length $\prod_{v\in V_k} n_v$ for the labelings with $bit[k]=1$. All we have to do is figure out a way to end P_k with a labeling that differs only in $bit[k]$ from the starting point of Q_k; then we can let G_k be 'P_k, Q_k'. And indeed, such a labeling is fairly obvious: It consists of the last labeling of spider u for each positive child u of k, and the first labeling of spider v for each negative child v.

If $j\in U_k$, the reflected code in P_k involves traversing G_j a total of

$$\delta_{jk}=\prod_{\substack{u<j\\ u\in U_k}} n_u$$

times, in alternating directions. Similarly, if $j\in V_k$, the path G_j is traversed

$$\delta_{jk}=\prod_{\substack{v<j\\ v\in V_k}} n_v$$

times in Q_k. We need to know whether these numbers are even or odd, in order to figure out how P_k should begin and Q_k should end, because we know how P_k ends and Q_k begins.

The numbers δ_{jk} can get as large as 2^n, and we don't want to mess with n-bit arithmetic if we don't have to. The trick is to compute two more tables, $ueven[k]$ and $veven[k]$, which point to the smallest elements $u\in U_k$ and $v\in V_k$ such that n_u and n_v are even. (If no such elements exist, $ueven[k]$ and/or $veven[k]$ are set to $maxn$, representing ∞.) These tables give us the information we need about δ_{jk}.

We set $ueven[0]$ artificially to ∞. This has the effect of keeping each component of the digraph independent.

While we're computing the *ueven* and *veven* tables, we might as well also compute *umin* and *vmin*, a counterpart of *umax* and *vmax* that will prove useful later.

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = 0; k ≤ n; k++) ueven[k] = veven[k] = umin[k] = vmin[k] = maxn;
  for (j = n; j > 0; j--) {
    k = ppro[j];
    if (umin[k] ≤ scope[j]) umin[j] = umin[k];
    if (ueven[k] ≤ scope[j]) ueven[j] = ueven[k];
    k = npro[j];
    if (vmin[k] ≤ scope[j]) vmin[j] = vmin[k];
    if (veven[k] ≤ scope[j]) veven[j] = veven[k];
    l = (ueven[j] < maxn) ⊕ (veven[j] < maxn);      /* l = n_j mod 2 */
    k = par[j];
    if (sign[j] ≡ 0) {
      umin[ppro[k]] = j;
      if (l ≡ 0) ueven[ppro[k]] = j;
    } else {
      vmin[npro[k]] = j;
      if (l ≡ 0) veven[npro[k]] = j;
    }
  }
  ueven[0] = maxn;
```

11. Another thing we'll need to know is $umaxbit[k]$, the value of $bit[umax[k]]$ when $bit[k]$ changes from 0 to 1. And of course the dual value $vmaxbit[k]$ will be equally important.

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = n; k > 0; k--) {
    l = par[k];
    if (k ≡ umax[l]) umaxbit[l] = 1;
    else {
      j = umax[k];
      if (j ∧ umax[l] ≡ j) {
        if (ueven[k] < j) umaxbit[l] = umaxbit[k];      /* δ_jk is even */
        else umaxbit[l] = 1 ⊕ umaxbit[k];
      }
    }
    if (k ≡ vmax[l]) vmaxbit[l] = 0;
    else {
      j = vmax[k];
      if (j ∧ vmax[l] ≡ j) {
        if (veven[k] < j) vmaxbit[l] = vmaxbit[k];      /* δ_jk is even */
        else vmaxbit[l] = 1 ⊕ vmaxbit[k];
      }
    }
  }
```

12. For the record, our example spider has the following additional characteristics (including some that we'll introduce later):

k	$bstart[k]$	$umin[k]$	$ueven[k]$	$umaxbit[k]$	$umaxscope[k]$	$vmin[k]$	$veven[k]$	$vmaxbit[k]$	$vmaxscope[k]$
0		1	∞			4	4		
1	1	2	2	0	9	4	4	0	9
2	2	3	5	1	5	4	4	1	4
3	3	∞	∞	0	3	4	4	0	4
4	4	∞	∞	0	4	∞	∞	0	4
5	5	∞	∞	0	5	∞	∞	0	5
6	6	∞	∞	0	6	7	7	0	7
7	7	∞	∞	0	7	∞	∞	0	7
8	8	9	9	1	9	∞	∞	0	8
9	9	∞	∞	0	9	∞	∞	0	9

```
⟨ Global variables 4 ⟩ +≡
  int umin[maxn], vmin[maxn];      /* the smallest guys in U_k, V_k */
  int ueven[maxn], veven[maxn];    /* the smallest even guys in U_k, V_k */
  int umaxbit[maxn], vmaxbit[maxn];    /* significant transition bits */
  int bit[maxn];      /* the current labeling */
```

13. A somewhat subtle point arises here, and it provides an important simplification: Suppose j is a negative child of k, and $ueven[k] \ge j$. Then the initial bits of spider j in the sequence for spider k are the same as the transition bits of spider j. The reason is that $\delta_{ij} + \delta_{ik}$ is even for all $i \in U_j$.

Using this principle, we can write recursive procedures so that *setfirst*(0) computes the very first setting the bit table, in $O(n)$ steps. (This bound on the running time comes from the fact that each procedure sets the bits of a subspider using a number of steps bounded by a constant times the number of bits being set. Formally, if $T_n \le a + (b + T_{n_1}) + \cdots + (b + T_{n_t})$ where $n_1 + \cdots + n_t = n - 1$, then it follows by induction that $T_n \le (a + b)n - b$.)

The first labeling of our example spider uses the first labeling of subspider 2, the last labeling of subspider 6, and the first labeling of subspider 8, so it is $bit[1] \ldots bit[9] = 000001100$.

Recursion is lots of fun too. Why do I sometimes prefer traversal?

⟨ Subroutines 13 ⟩ ≡

```
void setlast(register int k);      /* see below */
void setmid(register int k, int b);      /* ditto */
void setfirst(register int k)
{
  register int j;
  bit[k] = 0;
  for (j = rchild[k]; j; j = lsib[j])
    if (sign[j] ≡ 0) {
      if (ueven[k] ≥ j) setfirst(j);      /* δ_jk is odd */
      else setlast(j);
    } else if (ueven[k] ≥ j) setmid(j, 0);      /* by the subtle point */
    else setfirst(j);      /* δ_ik is even for all i ∈ U_j */
}
void setlast(register int k)
{
  register int j;
  bit[k] = 1;
  for (j = rchild[k]; j; j = lsib[j])
    if (sign[j] ≡ 1) {
      if (veven[k] ≥ j) setlast(j);      /* δ_jk is odd */
      else setfirst(j);
    } else if (veven[k] ≥ j) setmid(j, 1);      /* by the subtle point */
    else setlast(j);      /* δ_ik is even for all i ∈ V_j */
}
void setmid(register int k, int b)
{
  register int j;
  bit[k] = b;
  for (j = rchild[k]; j; j = lsib[j])
    if (sign[j] ≡ 0) setlast(j); else setfirst(j);
}
```

See also section 19.

This code is used in section 1.

14. The active list. Reflected Gray code is nicely generated by a process based on a list of elements that are alternately active and passive. (See, for example, Algorithm 7.2.1.1L in *The Art of Computer Programming*.) A slight generalization of that notion works admirably for the problem faced here: We maintain a so-called *active list* L, whose elements are alternately awake and asleep. Elements are occasionally inserted into L and/or deleted from L according to the following protocol:

1) Find the largest node $k \in L$ that is awake, and wake up all elements of L that exceed k.
2) If $bit[k] = 0$, set $bit[k] \leftarrow 1$ and $L \leftarrow (L \setminus U'_k) \cup V'_k$; otherwise set $bit[k] \leftarrow 0$ and $L \leftarrow (L \setminus V'_k) \cup U'_k$. Here U'_k and V'_k denote the *principal elements* of U_k and V_k, namely the positive and negative children of k.
3) Put k to sleep.

The process stops when all elements of L are asleep in step 1. In such a case, waking them up and repeating the process will run through the bit labelings again, but in reverse order.

The elements of L are the positive vertices k for which $bit[par[k]] = 0$ and the negative vertices k for which $bit[par[k]] = 1$. For example, the initial active list for the example spider is $L = \{1, 2, 3, 5, 6, 7, 9\}$. All elements are awake at the beginning.

We can conveniently represent L as a list of elements k, with subscripts to indicate the current setting of $bit[k]$. Then k_0 is always followed in the list by sublists for the spiders of U_k, and k_1 is always followed by sublists for the spiders of V_k. With these conventions, the initial active list is

$$1_0 \quad 2_0 \quad 3_0 \quad 5_0 \quad 6_1 \quad 7_1 \quad 9_0\,.$$

Since 9_0 is awake, we complement $bit[9]$, and L becomes

$$1_0 \quad 2_0 \quad 3_0 \quad 5_0 \quad 6_1 \quad 7_1 \quad \overline{9}_1\,.$$

The bar over 9 indicates that this node is now asleep.

The next step complements $bit[7]$ and wakes up 9; thus the first few steps take place as follows:

$$\begin{array}{lllllll}
1_0 & 2_0 & 3_0 & 5_0 & 6_1 & 7_1 & 9_0 \\
1_0 & 2_0 & 3_0 & 5_0 & 6_1 & 7_1 & \overline{9}_1 \\
1_0 & 2_0 & 3_0 & 5_0 & 6_1 & \overline{7}_0 & 9_1 \\
1_0 & 2_0 & 3_0 & 5_0 & 6_1 & \overline{7}_0 & \overline{9}_0 \\
1_0 & 2_0 & 3_0 & 5_0 & \overline{6}_0 & 9_0 & \\
1_0 & 2_0 & 3_0 & 5_0 & \overline{6}_0 & \overline{9}_1 & \\
1_0 & 2_0 & 3_0 & \overline{5}_1 & 6_0 & 9_1 & \\
1_0 & 2_0 & 3_0 & \overline{5}_1 & 6_0 & \overline{9}_0 & \\
1_0 & 2_0 & 3_0 & \overline{5}_1 & \overline{6}_1 & 7_0 & 9_0
\end{array}
\qquad
\begin{array}{l}
\cdots \text{ complement } bit[9] \\
\cdots \text{ complement } bit[7] \\
\cdots \text{ complement } bit[9] \\
\cdots \text{ complement } bit[6] \\
\cdots \text{ complement } bit[9] \\
\cdots \text{ complement } bit[5] \\
\cdots \text{ complement } bit[9] \\
\cdots \text{ complement } bit[6]
\end{array}$$

Notice that 7 disappears from L when $bit[6]$ becomes 0, but it comes back again when $bit[6]$ reverts to 1. Soon $bit[3]$ will change to 1, and 4_0 will enter the fray.

The most dramatic change will occur after the first $n_2n_6n_9 = 48$ labelings, when $bit[1]$ changes:

$$\begin{array}{llllll}
1_0 & \overline{2}_1 & \overline{4}_0 & \overline{6}_1 & \overline{7}_1 & \overline{9}_0 \\
\overline{1}_1 & 4_0 & 7_1 & 8_0 & 9_0 & \\
\overline{1}_1 & 4_0 & 7_1 & 8_0 & \overline{9}_1 & \\
\overline{1}_1 & 4_0 & 7_1 & \overline{8}_1 & & \\
 & \vdots & & & & \\
\overline{1}_1 & \overline{4}_1 & \overline{7}_1 & \overline{8}_0 & 9_1 & \\
\overline{1}_1 & \overline{4}_1 & \overline{7}_1 & \overline{8}_0 & \overline{9}_0 &
\end{array}
\qquad
\begin{array}{l}
\cdots \text{ complement } bit[1] \\
\cdots \text{ complement } bit[9] \\
\cdots \text{ complement } bit[8] \\
\cdots \text{ complement } bit[7] \\
\\
\cdots \text{ complement } bit[8] \\
\cdots \text{ complement } bit[9]
\end{array}$$

And finally the whole list is asleep; all 60 labelings have been generated.

15. Using the active list protocol, the average amount of work per bit change is only $O(1)$ when amortized over the entire computation, even if we do a sequential search for k in step (1) and recopy all elements greater than k in step (2). Our implementation goes beyond the notion of amortization, however; after $O(n)$ steps of initialization, the program below does at most a bounded number of operations between bit changes. Thus it is actually *loopless*, in the sense defined by Gideon Ehrlich [*Journal of the ACM* **20** (1973), 500–513].

The extra contortions that we need to go through in order to achieve looplessness are usually ill-advised, because they actually cause the total execution time to be longer than it would be with a more straightforward algorithm. But hey, looplessness carries an academic cachet. So we might as well treat this task as a challenging exercise that might help us to sharpen our algorithmic wits.

(There may actually be a loopless algorithm for this problem that does not slow down the total execution time. For example, the loopless implementation in the program LI-RUSKEY is quite fast, but it sometimes needs $\Omega(n^2)$ steps for initialization and $\Omega(n^2)$ space for tables. The existence of such a fast implementation suggests that totally acyclic digraphs might well have additional properties that yield improvements over the approach taken here; readers are encouraged to find a better way.)

The first step we shall take toward a loopless algorithm is to introduce "focus pointers," as in Ehrlich's Algorithm 7.2.1.1L. Usually $\mathit{focus}[k] = k$, except when k is asleep and the successor of k is awake. In the latter case, $\mathit{focus}[k]$ is the largest $j < k$ such that j is awake.

The active list will be doubly linked, with k preceded by $\mathit{left}[k]$ and followed by $\mathit{right}[k]$. We make the list circular by letting $\mathit{left}[0]$ be its rightmost element and $\mathit{right}[0]$ the leftmost. Then, for example, $\mathit{focus}[\mathit{left}[0]]$ will be the element k needed in step (1) of the protocol. We can wake up all elements to k's right by setting $\mathit{focus}[\mathit{left}[0]] = \mathit{left}[0]$, and we can put k to sleep by setting $\mathit{focus}[k] = \mathit{focus}[\mathit{left}[k]]$, $\mathit{focus}[\mathit{left}[k]] = \mathit{left}[k]$.

16. Now let's focus on the implementation of step (2), which is the heart of the computation.

A positive child j of k is called "simple" if V_j is empty; a negative child is called simple if U_j is empty. Same-sign siblings always enter or leave the active list as a unit. Therefore if j and j' have the same sign and if $j = \mathit{lsib}[j']$ is simple, we will have $\mathit{right}[j] = j'$ and $\mathit{left}[j'] = j$ whenever they are inserted or deleted. These links can be established as part of the initialization. On the other hand when j cannot be combined with its right neighbor, we compute $\mathit{bstart}[j]$, the leftmost sibling that forms a block with j. (Possibly $\mathit{bstart}[j] = j$.)

The following preprocessing steps establish the initial values of *bstart*, *left*, and *right*. They also compute two further quantities that sometimes turn out to be indispensable: $\mathit{umaxscope}[k]$ is the largest node that is forced to be in the active list at a transition point when $\mathit{bit}[k] = 0$, and $\mathit{vmaxscope}[k]$ is the corresponding quantity when $\mathit{bit}[k] = 1$.

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = n; k; k--) {
    j = lsib[k];
    if (j) left[k] = j, right[j] = k;
    else ⟨ Compute the bstart links for k's family 17 ⟩;
    j = umax[k];
    if (¬j) umaxscope[k] = k;
    else umaxscope[k] = (umaxbit[k] ≡ 1 ? (vmax[j] ? vmax[j] : j) : umaxscope[j]);
    j = vmax[k];
    if (¬j) vmaxscope[k] = k;
    else vmaxscope[k] = (vmaxbit[k] ≡ 0 ? (umax[j] ? umax[j] : j) : vmaxscope[j]);
  }
```

17. ⟨ Compute the *bstart* links for k's family 17 ⟩ ≡

```
for (j = l = k; j; j = right[j]) {
  if (right[j] ∧ sign[j] ≡ sign[right[j]] ∧
        ((sign[j] ≡ 0 ∧ ¬vmax[j]) ∨ (sign[j] ≡ 1 ∧ ¬umax[j]))) continue;
  bstart[j] = l, l = right[j];
}
```

This code is used in section 16.

18. ⟨ Global variables 4 ⟩ +≡

```
int left[maxn], right[maxn];      /* neighbors in the active list */
int bstart[maxn];      /* start of a block */
int umaxscope[maxn], vmaxscope[maxn];      /* extreme nodes when bit[k] changes */
int flag[maxn];      /* nonzero when an insertion or deletion is needed */
int focus[maxn];      /* pointers that encode wakefulness */
```

19. When $bit[k]$ changes from 0 to 1, we want to delete k's positive blocks of children from the active list and insert the negative ones. The rightmost block is addressed by $rchild[k]$, and we get to the others by following *bstart* and *lsib* links. Our algorithm is supposed to be loopless, so we can't do all this updating at once. Therefore we do only the rightmost step, and we plant a warning in the data structure so that subsequent steps will be performed before the missing information is needed. All nodes are awake while waiting to be inserted or deleted, so the focus pointers are unaffected by these delayed actions.

The *fixup* subroutine is the basic mechanism by which nodes enter or leave the active list. This subroutine not only inserts or deletes a block of children, it also inserts a flag so that the previous block will be fixed in due time.

⟨ Subroutines 13 ⟩ +≡

```
void fixup(register int k, register int l)
{
  register int i, j;
  flag[l] = 0;
  if (k > 0) ⟨ Insert block k before l and return 20 ⟩
  ⟨ Delete block k before l 21 ⟩;
}
```

20. Once the process has gotten started, $left[j]$ and $right[k]$ will already have the correct values, unchanged from the time block k was previously deleted. But we don't make use of this fact, because we don't want to worry about presetting $left[j]$ and $right[k]$ when the action list is initialized.

```
⟨ Insert block k before l and return 20 ⟩ ≡
  {
    j = bstart[k], i = lsib[j];
    left[j] = left[l], right[left[l]] = j;
    left[l] = k, right[k] = l;
    if (i) {
      if (sign[k] ≡ 1) {
        if (sign[i] ≡ 0) {
          if (vmin[i] < maxn) j = vmin[i];
          i = −i;      /* the next fix will be a deletion */
        } else j = umin[i];
      } else {
        if (sign[i] ≡ 1) {
          if (umin[i] < maxn) j = umin[i];
          i = −i;      /* the next fix will be a deletion */
        } else j = vmin[i];
      }
      flag[j] = i;
    }
    return;
  }
```

This code is used in section 19.

21. A block being deleted might be preceded by a simple block of the other sign that wants to be inserted. In that case we insert the latter in place of the former.

```
⟨ Delete block k before l 21 ⟩ ≡
  k = −k, j = bstart[k], i = lsib[j];
  if (left[l] ≠ k) printf("Oops, fixup(%d,%d) is confused!\n", −k, l);      /* can't happen */
  if (i ∧ sign[i] ≠ sign[k]) {
    if ((sign[i] ≡ 0 ∧ vmax[i] ≡ 0) ∨ (sign[i] ≡ 1 ∧ umax[i] ≡ 0))
      ⟨ Replace block k by block i and return 22 ⟩;
  }
  left[l] = left[j], right[left[j]] = l;
  if (i) {
    if (sign[k] ≡ 0) {
      if (sign[i] ≡ 1) j = umin[i];
      else j = vmin[i], i = −i;      /* the next fix will be another deletion */
    } else {
      if (sign[i] ≡ 0) j = vmin[i];
      else j = umin[i], i = −i;      /* the next fix will be another deletion */
    }
    flag[j] = i;
  }
```

This code is used in section 19.

22. ⟨ Replace block k by block i and **return** 22 ⟩ ≡

```
{
  left[l] = i, right[i] = l;
  k = bstart[i], left[k] = left[j], right[left[k]] = k;
  i = lsib[k];
  if (i) {
    if (sign[k] ≡ 0) {
      if (sign[i] ≡ 1) {
        if (umin[i] < maxn) k = umin[i];
        i = −i;      /* the next fix will be another deletion */
      } else k = vmin[i];
    } else {
      if (sign[i] ≡ 0) {
        if (vmin[i] < maxn) k = vmin[i];
        i = −i;      /* the next fix will be another deletion */
      } else k = umin[i];
    }
    flag[k] = i;
  }
  return;
}
```

This code is used in section 21.

23. How does the active list get there in the first place? We compute it from the *bit* table, as follows.

⟨ Launch the active list 23 ⟩ ≡

```
setfirst(0);      /* compute the initial setting of bit[1] ... bit[n] */
for (l = k = 0; k ≤ n; k++) {
  focus[k] = k;
  if (sign[k] ≡ bit[par[k]]) right[l] = k, left[k] = l, l = k;
}
right[l] = 0, left[0] = l;      /* link in the rightmost node of the active list */
```

This code is used in section 24.

24. Doing it. The time has come to construct the loopless implementation in practice, as we have been doing so far in theory.

Of course the printout in each step does involve a loop. This printout is suppressed if $verbose < 0$.

```
⟨ Generate the answers 24 ⟩ ≡
  ⟨ Launch the active list 23 ⟩;
  if (verbose > 1) ⟨ Print out the results of initialization 33 ⟩;
  while (1) {
    count++;
    if (verbose ≥ 0) ⟨ Print out all the current bits 30 ⟩;
    ⟨ Set k to the rightmost nonsleeping node of the active list 26 ⟩;
    if (k) {
      if (flag[k]) fixup(flag[k], k);
      if (bit[k] ≡ 0) ⟨ Move forward, setting bit[k] = 1 28 ⟩
      else ⟨ Move backward, setting bit[k] = 0 29 ⟩;
    } else if (been_there_and_done_that) break;
    else {
      printf("...%d so far; now we generate in reverse:\n", count);
      been_there_and_done_that = 1;
      continue;
    }
    ⟨ Put k to sleep 27 ⟩;
  }
  printf("Altogether %d/2 labelings.\n", count);
```

This code is used in section 1.

25.
```
⟨ Global variables 4 ⟩ +≡
  int count;      /* the number of labelings found so far */
  int been_there_and_done_that;      /* have we reached all-asleep state before? */
```

26.
```
⟨ Set k to the rightmost nonsleeping node of the active list 26 ⟩ ≡
  j = left[0], k = focus[j], focus[j] = j;
```

This code is used in section 24.

27. At this point we know that all nodes greater than k are awake and that $flag[k] = 0$.

```
⟨ Put k to sleep 27 ⟩ ≡
  j = left[k], focus[k] = focus[j], focus[j] = j;
```

This code is used in section 24.

28. ⟨ Move forward, setting $bit[k] = 1$ 28 ⟩ ≡

```
{
  bit[k] = 1, j = rchild[k];
  if (j) {
    if (sign[j] ≡ 0) {      /* we want to delete j = umax[k] */
      l = vmin[j];
      if (l < maxn) fixup(−j, l);
      else fixup(−j, right[j]);      /* j ends a simple block */
    } else {      /* we want to insert j = vmax[k] */
      l = umin[j];
      if (l < maxn) fixup(j, l);
      else fixup(j, right[umaxscope[k]]);      /* j ends a simple block */
    }
  }
}
```

This code is used in section 24.

29. ⟨ Move backward, setting $bit[k] = 0$ 29 ⟩ ≡

```
{
  bit[k] = 0, j = rchild[k];
  if (j) {
    if (sign[j] ≡ 1) {      /* we want to delete j = vmax[k] */
      l = umin[j];
      if (l < maxn) fixup(−j, l);
      else fixup(−j, right[j]);      /* j ends a simple block */
    } else {      /* we want to insert j = umax[k] */
      l = vmin[j];
      if (l < maxn) fixup(j, l);
      else fixup(j, right[vmaxscope[k]]);      /* j ends a simple block */
    }
  }
}
```

This code is used in section 24.

30. ⟨ Print out all the current bits 30 ⟩ ≡

```
{
  for (k = 1; k ≤ n; k++) putchar('0' + bit[k]);
  if (verbose > 0) ⟨ Print the active list in symbolic form 31 ⟩;
  putchar('\n');
}
```

This code is used in section 24.

31. Here I recompute what the active list should be, and compare it to the current links. Discrepancies are noted only if no flagged nodes follow.

Sleeping nodes are enclosed in parentheses; an exclamation point is printed before a node that is flagged.

```
⟨ Print the active list in symbolic form 31 ⟩ ≡
  {
    for (k = left[0]; ; k--) {
      for (j = k, k = focus[k]; j > k; j--) {
        asleep[j] = 1;
        if (flag[j]) printf("\n0ops, flag[%d] is wrong!\n", j);
      }
      if (k ≡ 0) break;
      asleep[k] = 0;
    }
    for (k = 1, j = 0; k ≤ left[0]; k++)
      if (sign[k] ≡ bit[par[k]]) {
        if (asleep[k]) printf(" (%d)", k);
        else if (flag[k]) printf(" !%d", k);
        else printf(" %d", k);
        if ((k ≠ right[j] ∨ left[k] ≠ j) ∧ k > l) printf("[oops]");
        j = k;
      }
  }
```

This code is used in section 30.

32.

```
⟨ Global variables 4 ⟩ +≡
  int asleep[maxn];      /* sleeping (or inactive) nodes */
```

33. Finally, we print even more stuff when the user calls for an exceptional level of absolute verbosity.

```
⟨ Print out the results of initialization 33 ⟩ ≡
  {
    for (k = 0; k ≤ n; k++) {
      printf("%d(%c): scope=%d, par=%d, rchild=%d, lsib=%d,", k, sign[k] ? '-' : '+', scope[k],
          par[k], rchild[k], lsib[k]);
      printf(" ppro=%d, npro=%d, prev=%d, bstart=%d\n", ppro[k], npro[k], prev[k], bstart[k]);
      printf(" umin=%d, ueven=%d, umax=%d, umaxbit=%d, umaxscope=%d\n", umin[k], ueven[k],
          umax[k], umaxbit[k], umaxscope[k]);
      printf(" vmin=%d, veven=%d, vmax=%d, vmaxbit=%d, vmaxscope=%d\n", vmin[k], veven[k],
          vmax[k], vmaxbit[k], vmaxscope[k]);
    }
  }
```

This code is used in section 24.

34. Index.

SPIDERS