(Downloaded from https://cs.stanford.edu/~knuth/programs.html and typeset on May 28, 2023)

1. This is a quick program to find all canonical forms of reflection networks for small n.

Well, when I wrote that paragraph I believed it, but subsequently I have added lots of bells and whistles because I wanted to compute more stuff. At present this code determines the number B_n of equivalence classes of reflection networks (i.e., irredundant primitive sorting networks); also the number of weak equivalence classes, either with (C_{n+1}) or without (D_{n+1}) anti-isomorphism; and the number of preweak equivalence classes (E_{n+1}), which is the number of simple arrangements of $n+1$ pseudolines in a projective plane. For each representative of D_{n+1} it also computes the "score," which is the number of ways to add another pseudoline crossing the network.

If compiled without the `NOPRINT` switch, each member of B_n is printed as a string of transposition numbers, generated in lexicographic order. This is followed by `*` if the string is also a representative of C_{n+1} when prefixed by $01\ldots n$. And if the string is also a representative of D_{n+1}, you also get the score in brackets, followed by `#` if it is a representative of E_{n+1}. If not a representative of D_{n+1}, the symbol `>` is printed followed by the string of an anti-equivalent network.

If compiled with the `DEBUG` switch, you also get intermediate output about the backtrack tree and the networks generated while searching for anti-equivalence and preweak equivalence.

I wrote this program to allow n up to 10; but integer overflow will surely occur in $B_{10} \approx 2 \times 10^{10}$, if I ever get a computer fast enough to run that case. When $n = 7$, this program took 48 seconds to run, on January 12, 1991; the running time for $n = 6$ was 1 second, and for $n = 8$ it was 57 minutes. Therefore I made a stripped-down version to enumerate only B_n when $n = 9$.

```
#include <stdio.h>
```

2. There's an array $a[1\,..\,n]$ containing k inversions; an index j showing where we are going to try to reduce the inversions by swapping $a[j]$ with $a[j+1]$; and two arrays for backtracking. At choice-level l we set $t[l]$ to the current j value, and we also set $c[l]$ to 1 if we swapped, 0 if we didn't.

```
#define swap(j)
        { int tmp = a[j]; a[j] = a[j + 1]; a[j + 1] = tmp; }
#define npairs 120    /* should be greater than 2(n+1 choose 2) */
#define ncycle 240    /* should be greater than 4(n+1 choose 2) */
```

⟨ Global variables 2 ⟩ ≡

```
  int n;    /* number of elements to be reflected */
  int a[10];    /* array that shows progress */
  int k;    /* number of inversions yet to be removed */
  int j;    /* current place in array */
  int l;    /* current choice level */
  int c[npairs];    /* code for choices made */
  int t[npairs];    /* j values where choices were made */
  int i, ii, iii;    /* general-purpose indices */
  int bn, cn, dn, en;    /* counters for B_n, C_{n+1}, D_{n+1}, E_{n+1} */
  int smin, smax;    /* counters for "scores" */
  float stot;    /* grand total of scores */
```

See also sections 8 and 13.

This code is used in section 3.

3. The value of n is supposed to be an argument.

```
#define abort(s)
        { fprintf(stderr, s); exit(1); }
  ⟨ Global variables 2 ⟩
  main(argc, argv)
      int argc;    /* number of args */
      char **argv;    /* the args */
  {
    if (argc ≠ 2) abort("Usage:␣reflect␣n\n");
    if (sscanf(argv[1], "%d", &n) ≠ 1 ∨ n < 2 ∨ n > 10) abort("n␣should␣be␣in␣the␣range␣2..10!\n");
    ⟨ Initialize 4 ⟩;
    ⟨ Run through all canonical reflection networks 5 ⟩;
    printf("B=%d,␣C=%d,␣D=%d,␣E=%d\n", bn, cn, dn, en);
    printf("scores␣min=%d,␣max=%d,␣mean=%.1f\n", smin, smax, stot/(float) dn);
  }
```

4. ⟨ Initialize 4 ⟩ ≡

```
  for (j = 1; j ≤ n; j++) a[j] = n + 1 − j;
  k = n * (n − 1); k /= 2;
  c[0] = 0;    /* a convenient sentinel */
  l = 1;
  j = n;
  bn = cn = dn = en = smax = 0;
  stot = 0.0;
  smin = 1000000000;
```

This code is used in section 3.

5. ⟨ Run through all canonical reflection networks 5 ⟩ ≡

```
moveleft: j−−;
loop:
  if (j ≡ 0) {
    if (k ≡ 0) ⟨ Print a solution 7 ⟩;
    ⟨ Backtrack, either going to loop or to finished when all possibilities are exhausted 6 ⟩;
  }
  if (a[j] < a[j + 1]) goto moveleft;
  t[l] = j;
  c[l++] = 0;
  goto moveleft;
finished: ;
```

This code is used in section 3.

6. ⟨ Backtrack, either going to *loop* or to *finished* when all possibilities are exhausted 6 ⟩ ≡

```
while (c[--l]) {
  j = t[l];
  swap(j);
  k++;
}
if (l ≡ 0) goto finished;
j = t[l];
c[l++] = 1;
swap(j);
k--;
if (++j ≡ n) j--;
goto loop;
```

This code is used in section 5.

7. ⟨ Print a solution 7 ⟩ ≡

```
{
#ifdef DEBUG
  for (i = 1; i < l; i++) putchar('0' + c[i]);
  putchar(':');
#endif
#ifndef NOPRINT
  for (i = 1; i < l; i++)
    if (c[i]) putchar('0' - 1 + t[i]);
#endif
  ⟨ Check if it gives a new CC system on n + 1 elements 9 ⟩;
#ifndef NOPRINT
  putchar('\n');
#endif
  bn++;
}
```

This code is used in section 5.

8. Here's part of the program I wrote after getting the above to work. The idea is to see if the almost-canonical form for an $(n+1)$-element network is weakly equivalent to any lexicographically smaller almost-canonical forms. If not, we print an asterisk, because it represents a new weak equivalence class.

The forms are kept in locations r through $r+n(n+1)/2-1$ of array b, which starts out like t but with the transpositions 1, 2, . . . , n prefaced. End-around shifts are performed (advancing r by 1 each time) until the original form appears again.

⟨ Global variables 2 ⟩ +≡

```
int b[ncycle];   /* larger array used for testing weak equivalence */
int r, rr;   /* the first and last active locations in b */
int d[npairs];   /* copy of the present network */
int rrr;   /* (n+1 choose 2) */
```

9. ⟨ Check if it gives a new CC system on $n+1$ elements 9 ⟩ ≡

```
for (rr = 0; rr < n; rr++) b[rr] = rr + 1;
for (i = 1; i < l; i++)
  if (c[i]) {
    b[rr] = d[rr] = t[i];
    rr++;
  }
d[rr] = 1;   /* sentinel */
rrr = rr;
r = 0;
while (1) {
  ⟨ Shift the first transposition to the other end 10 ⟩;
  if (b[r] ≡ 1) ⟨ Test lexicographic order; break if equal or less 11 ⟩;
}
```

This code is used in section 7.

10. ⟨ Shift the first transposition to the other end 10 ⟩ ≡

```
j = n − b[r++];
for (i = rr++; b[i − 1] < j; i−−) b[i] = b[i − 1];
b[i] = j + 1;
```

This code is used in section 9.

11. ⟨ Test lexicographic order; **break** if equal or less 11 ⟩ ≡

```
{
  b[rr] = 0;   /* sentinel, is less than the 1 we put in d */
  for (i = r + n; b[i] ≡ d[i − r]; i++) ;
  if (b[i] < d[i − r]) {
    if (i ≡ rr) {   /* total equality */
#ifndef NOPRINT
      putchar('*');
#endif
      cn++;
      ⟨ Make the big test for pre-weak equivalence 12 ⟩;
    }
    break;
  }
}
```

This code is used in section 9.

12. Well, after I got that going I couldn't resist continuing until I had all simple arrangements of pseudolines enumerated. That requires looking at another $\binom{n+1}{2}$ cases to see if they are weakly equivalent to anything seen before.

And, surprise, it also meant testing for anti-isomorphism.

```
⟨ Make the big test for pre-weak equivalence 12 ⟩ ≡
  ⟨ Reset b to a double cycle 14 ⟩;
  ⟨ Test the reverse of b for weak equivalence; goto done if weakly equivalent to a previous case 15 ⟩;
  ⟨ Compute the score for this weak equivalence/antiequivalence class rep 22 ⟩;
  for (r = 0; r < rrr; r++) {
    ⟨ Move the "pole" into the cell preceding the first transposition module 20 ⟩;
    for (ref = 0; ref < 2; ref++) {
      if (ref ≡ 0)
        for (i = 0; i < rrr; i++) y[i] = x[i];
      else ⟨ Replace the present x by the reverse of y 16 ⟩;
      ⟨ If the new network is weakly equivalent to a lexicographically smaller one, goto done 17 ⟩;
    }
  }
#ifndef NOPRINT
  putchar('#');    /* a new preweak class, not related to anything earlier */
#endif
  en++;
done: ;
```

This code is used in section 11.

13. For this part of the program we use an array x analogous to b; also variables s and ss analogous to r and rr; also an array e analogous to d.

```
⟨ Global variables 2 ⟩ +≡
  int x[ncycle];    /* network to be tested for weak equivalence */
  int m;    /* largest element in x so far */
  int y[npairs];    /* elements to be carried around to the right as x is formed */
  int jj;    /* the number of elements in y */
  int s, ss;    /* the active region of x */
  int e[npairs];    /* starting point */
  int rep;    /* number of repetitions */
  int ref;    /* number of reflections */
```

14. At this point $i - r$ points just past the end of the d data, and the first n entries of b are still equal to 1, 2, . . . , n. The network we construct here is not necessarily in canonical form.

```
⟨ Reset b to a double cycle 14 ⟩ ≡
  rr = i − r;
  for (i = n; i < rr; i++) b[i] = d[i];
  for ( ; i < rr + rr; i++) b[i] = n + 1 − b[i − rr];
```

This code is used in section 12.

15. One nice thing is that reflection and turning upside down preserve canonicity when we do both simultaneously.

```
⟨ Test the reverse of b for weak equivalence; goto done if weakly equivalent to a previous case 15 ⟩ ≡
  for (i = 0; i < rrr; i++) x[rrr − 1 − i] = n + 1 − b[i];
  s = 0; ss = rrr;
  while (x[s] > 1) ⟨ End-around shift x 19 ⟩;
  for (i = s + n; i < ss; i++) e[i − s] = x[i];
  e[rrr] = 1;   /* another sentinel */
  while (1) { x[ss] = 0;   /* sentinel */
  for (i = s + n; x[i] ≡ d[i − s]; i++) ;
  if (i ≡ ss) break;   /* anti-isomorphic to itself */
  if (x[i] < d[i − s]) {   /* anti-isomorphic to previous guy */
#ifndef NOPRINT
    putchar('>');
    for (i = s + n; i < ss; i++) putchar(x[i] + '0' − 1);
#endif
    goto done;
  }
  do ⟨ End-around shift x 19 ⟩
  while (x[s] > 1) ;
  x[ss] = 0;
  for (i = s + n; x[i] ≡ e[i − s]; i++) ;
  if (i ≡ ss) break;   /* anti-isomorphic to some future guy */
  }
```

This code is used in section 12.

16.

```
⟨ Replace the present x by the reverse of y 16 ⟩ ≡
  {
    for (i = 0; i < rrr; i++) x[rrr − 1 − i] = n + 1 − y[i];
    s = 0; ss = rrr;
    while (x[s] > 1) ⟨ End-around shift x 19 ⟩;
#ifdef DEBUG
    putchar('/');
    ⟨ If debugging, print the active region of x 25 ⟩;
#endif
  }
```

This code is used in section 12.

17.

```
⟨ If the new network is weakly equivalent to a lexicographically smaller one, goto done 17 ⟩ ≡
  for (i = s + n; i < ss; i++) e[i − s] = x[i];
  while (1) { ⟨ If the x network is weakly equivalent to an earlier one, goto done; if weakly equivalent to
      the present one, goto okay 18 ⟩;
  do ⟨ End-around shift x 19 ⟩
  while (x[s] > 1) ;
  ⟨ If debugging, print the active region of x 25 ⟩;
  x[ss] = 0;   /* sentinel */
  for (i = s + n; x[i] ≡ e[i − s]; i++) ;
  if (i ≡ ss) break;   /* now x is back to its original state and we found nothing */
  }
okay: ;
```

This code is used in section 12.

18. ⟨If the x network is weakly equivalent to an earlier one, **goto** *done*; if weakly equivalent to the present one, **goto** *okay* 18⟩ ≡

```
x[ss] = 0;    /* sentinel */
for (i = s + n; x[i] ≡ d[i − s]; i++) ;
if (i ≡ ss) goto okay;
if (x[i] < d[i − s]) goto done;
```

This code is used in section 17.

19. ⟨End-around shift x 19⟩ ≡

```
{
  j = n − x[s++];
  for (i = ss++; x[i − 1] < j; i−−) x[i] = x[i − 1];
  x[i] = j + 1;
}
```

This code is used in sections 15, 16, and 17.

20. The only somewhat tricky operation comes in here. We use the fact that the first '1' in a canonical network is always immediately followed by 2, ..., n; reversing these, decreasing the previous by 1, and increasing the remaining by 1 takes that line around the pole. This operation might require carrying some transpositions around from left to right.

```
⟨ Move the "pole" into the cell preceding the first transposition module 20 ⟩ ≡
  ⟨ If debugging, print the active region of b 24 ⟩;
  s = 0; ss = rrr;
  iii = jj = 0;
  x[0] = m = rep = b[r];
  rr = r + rrr;
  for (i = r + 1; i < rr; i++) {
    j = b[i] − 1;
    ⟨ Insert the value j + 1 canonically into x 21 ⟩;
  }
  for (i = 0; iii < rrr − 1; i++) {
    j = n − 1 − y[i];
    ⟨ Insert the value j + 1 canonically into x 21 ⟩;
  }
  ⟨ If debugging, print the active region of x 25 ⟩;
  while (rep −−) {
    m = 0;
    for (i = 0; x[i] ≠ 1; i++) {
      x[i]−−;
      if (x[i] > m) m = x[i];
    }
    iii = i − 1;
    jj = 0;
    for (j = n − 1; j ≥ 0; j−−)
      if (j ≡ 0 ∧ i ≡ 0) {
        x[0] = m = 1;
        iii = 0;
      }
      else ⟨ Insert the value j + 1 canonically into x 21 ⟩;
    for (i += n; i < rrr; i++) {
      j = x[i];
      ⟨ Insert the value j + 1 canonically into x 21 ⟩;
    }
    for (i = 0; iii < ss − 1; i++) {
      j = n − 1 − y[i];
      ⟨ Insert the value j + 1 canonically into x 21 ⟩;
    }
    ⟨ If debugging, print the active region of x 25 ⟩;
  }
```

This code is used in section 12.

21. We must carry over items that exceed m, which denotes the maximum value stored so far, because we want the first element of $x[0]$ to remain in place.

⟨ Insert the value $j+1$ canonically into x 21 ⟩ ≡

```
if (j > m) y[jj++] = j;
else {
  if (j ≡ m) m++;
  for (ii = ++iii; x[ii − 1] < j; ii−−) x[ii] = x[ii − 1];
  x[ii] = j + 1;
}
```

This code is used in section 20.

22. The score is computed in several passes, although I do know how to do it in linear time. Since the x array is currently unused, I store in $x[i]$ the score for the cell following transposition i.

⟨ Compute the score for this weak equivalence/antiequivalence class rep 22 ⟩ ≡

```
dn++;
rr = rrr + rrr;
for (i = 0; i < rr; i++) x[i] = 1;
for (j = 2; j ≤ n; j++) ⟨ Fill in the cell counts x[i] for cases when b[i] = j 23 ⟩;
{ register int score = 0;
  for (i = 0; i < rr; i++)
    if (b[i] ≡ n) score += x[i];
  stot += (float) score;
  if (score > smax) smax = score;
  if (score < smin) smin = score;
#ifndef NOPRINT
  printf("␣[%d]", score);
#endif
}
```

This code is used in section 12.

23. As we fill the cell counts, we assume that $x[ii]$ is the previous cell having $b[i]=j$. We assume that $b[i]\equiv i+1$ for $0\le i<n$.

⟨ Fill in the cell counts $x[i]$ for cases when $b[i]=j$ 23 ⟩ ≡

```
{ int acc = 0;
  int p;    /* most recent x[i] when b[i] = j − 1 */
  ii = rr;
  for (i = 0; i < rr; i++) { register int delta = j − b[i];
    if (delta ≡ 0) {
      x[ii] = acc;
      ii = i;
      acc = p;
    }
    else if (delta ≡ 1) {
      p = x[i];
      acc += p;
    }
  }
  x[ii] = acc + x[rr];
}
```

This code is used in section 22.