1. Intro. This program is an iterative implementation of an interesting recursive algorithm due to Willard L. Eastman, *IEEE Trans.* **IT-11** (1965), 263–267: Given a sequence of nonnegative integers $x = x_0x_1 \ldots x_{n-1}$ of odd length n, where x is not equal to any of its cyclic shifts $x_k \ldots x_{n-1}x_0 \ldots x_{k-1}$ for $1 \le k < n$, we output a cyclic shift σx such that the set of all such σx forms a commafree code of block length n (over an infinite alphabet).

The integers are given as command-line arguments.

The simplest nontrivial example occurs when $n = 3$. If $x = abc$, where a, b, and c aren't all equal, then exactly one of the cyclic shifts $y_0y_1y_2 = abc$, bca, cab will satisfy $y_0 > y_1 \le y_2$, and we choose that one. It's easy to check that the triples chosen in this way are commafree.

Similar constructions are possible when $n = 5$ or $n = 7$. But the case $n = 9$ already gets a bit dicey, and when n is really large it's not at all clear that commafreeness is possible. Eastman's paper resolved a conjecture made by Golomb, Gordon, and Welch in their pioneering paper about comma-free codes (1958).

(Of course, it's not at all clear that we would want to actually *use* a commafree code when n is large; but that's another story, and beside the point. The point is that Eastman discovered a really interesting algorithm.)

```
#define maxn 105

#include <stdio.h>
#include <stdlib.h>
  int x[maxn + maxn + maxn];
  int b[maxn + maxn + maxn];
  int bb[maxn];
  ⟨Subroutines 5⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, n, p, q, t, tt;
    ⟨Process the command line 2⟩;
    ⟨Do Eastman's algorithm 3⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc < 4) {
    fprintf(stderr, "Usage:␣%s␣x1␣x2␣...␣xn\n", argv[0]);
    exit(−1);
  }
  n = argc − 1;
  if ((n & 1) ≡ 0) {
    fprintf(stderr, "The␣number␣of␣items,␣n,␣should␣be␣odd,␣not␣%d!\n", n);
    exit(−2);
  }
  for (j = 1; j < argc; j++) {
    if (sscanf(argv[j], "%d", &x[j − 1]) ≠ 1 ∨ x[j − 1] < 0) {
      fprintf(stderr, "Argument␣%d␣should␣be␣a␣nonnegative␣integer,␣not␣'%s'!\n", j, argv[j]);
      exit(−3);
    }
  }
```

This code is used in section 1.

3. The algorithm. We think of x as written cyclically, with $x_{n+j} = x_j$ for all $j \ge 0$. The basic idea in the algorithm below is to also think of x as partitioned into $t \le n$ subwords by boundary markers b_j where $0 \le b_0 < b_1 < \cdots < b_{t-1} < n$; then subword y_j is $x_{b_j} x_{b_j+1} \ldots x_{b_{j+1}-1}$, for $0 \le j < t$, where $b_t = b_0$. If $t = 1$, there's just one subword, and it's a cyclic shift of x. The number t of subwords during each phase will be odd.

Eastman's algorithm essentially begins with $b_j = j$ for $0 \le j < n$, so that x is partitioned into n subwords of length 1. It successively *removes* boundary points until only one subword is left; that subword is the answer. It operates in phases, so that all subwords during the jth phase have length 3^{j-1} or more; thus at most $\lfloor \log_3 n \rfloor$ phases are needed. (For example, the case $n = 9$ is "dicey" because it might require two phases.)

The algorithm is based on comparison of adjacent subwords y_{j-1} and y_j. If those subwords have the same length, we use lexicographic comparison; otherwise we declare that the longer subword is bigger.

(After the first phase, all subwords not only have length ≥ 3, they also always begin with a nonzero entry; in other words, $x_{b_j} > 0$ for every boundary marker b_j. However, we won't need to use that fact explicitly.)

The algorithm can be described with terminology based on the topography of Nevada: Say that i is a *basin* if the subwords satisfy $y_{i-1} > y_i \le y_{i+1}$. There must be at least one basin; otherwise all the y_j would be equal, and x would equal one of its cyclic shifts. We look at consecutive basins, i and j; this means that $i < j$ and that i and j are basins, and that $i+1$ through $j-1$ are *not* basins. If there's only one basin we have $j = i + t$. The indices between consecutive basins are called *ranges*.

Since t is odd, there's an odd number of consecutive basins for which $j - i$ is odd. Each phase of Eastman's algorithm retains exactly one boundary point in the range between such basins, and deletes all the others. The retained point is the smallest $k = i + 2l$ such that $y_k > y_{k+1}$.

(For example, suppose $i = 2$ and $j = 9$ are consecutive basins. Then we have $y_1 > y_2 \le y_3 \le \cdots \le y_q > y_{q+1} > \cdots > y_9 \le y_{10}$, for some range element $2 < q < 9$. We choose $k = 4$ if $q = 3$ or $q = 4$, $k = 6$ if $q = 5$ or $q = 6$, and $k = 8$ if $q = 7$ or $q = 8$.)

⟨ Do Eastman's algorithm 3 ⟩ ≡
  ⟨ Initialize 4 ⟩;
  for $(p = 1, t = n;\ t > 1;\ t = tt, p{+}{+})$
    ⟨ Do one phase of Eastman's algorithm, putting tt boundary points into bb 6 ⟩;

This code is used in section 1.

4. ⟨ Initialize 4 ⟩ ≡
  for $(j = n;\ j < n + n + n;\ j{+}{+})\ x[j] = x[j - n]$;
  for $(j = 0;\ j < n + n + n;\ j{+}{+})\ b[j] = j$;
  $t = n$;

This code is used in section 3.

5. Here's a basic subroutine that returns 1 if subword y_{i-1} exceeds subword y_i, otherwise it returns 0.

⟨ Subroutines 5 ⟩ ≡

```
int compare(register int i)
{
  register int j;
  if (b[i] - b[i-1] == b[i+1] - b[i]) {
    for (j = 0; b[i] + j < b[i+1]; j++) {
      if (x[b[i-1] + j] == x[b[i] + j]) continue;
      return (x[b[i-1] + j] > x[b[i] + j]);
    }
    return 0;    /* y_{i-1} = y_i */
  }
  return (b[i] - b[i-1] > b[i+1] - b[i]);
}
```

This code is used in section 1.

6. ⟨ Do one phase of Eastman's algorithm, putting *tt* boundary points into *bb* 6 ⟩ ≡

```
{
  for (tt = 0, i = 1; i <= t; i++)
    if (compare(i)) break;
  if (i > t) {
    fprintf(stderr, "The input is cyclic!\n");
    exit(-666);
  }
  for ( ; compare(i+1); i++) ;    /* advance to the first basin */
  for ( ; i <= t; i = j) {
    for (q = i+1; compare(q+1) == 0; q++) ;    /* climb the range */
    for (j = q+1; compare(j+1); j++) ;    /* advance to the next basin */
    if ((j - i) & 1) ⟨ Choose a boundary point to retain 7 ⟩;
  }
  printf("Phase %d leaves", p);
  for (k = 0; k < tt; k++) b[k] = bb[k], printf(" %d", bb[k]);
  printf("\n");
  for ( ; b[k - tt] < n + n; k++) b[k] = b[k - tt] + n;
}
```

This code is used in section 3.

7. ⟨ Choose a boundary point to retain 7 ⟩ ≡

```
{
  if ((q - i) & 1) q++;
  if (q < t) bb[tt++] = b[q];
  else {
    for (k = tt++; k > 0; k--) bb[k] = bb[k-1];
    bb[0] = b[q - t];
  }
}
```

This code is used in section 6.

8. Index.

COMMAFREE-EASTMAN