

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

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 $x = x_0x_1 \dots x_{n-1}$ of odd length n , where x is not equal to any of its cyclic shifts $x_k \dots x_{n-1}x_0 \dots x_{k-1}$ for $1 \leq k < n$, 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 $y_0 \geq y_1 < y_2$, 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.)

Note: This program was written after I presented a lecture about Eastman's algorithm at Stanford on 3 December 2015. While preparing the lecture I realized that some nice structure was present, and a day later it occurred to me that the algorithm could therefore be streamlined. This program significantly simplifies the method of the previous one, which was called COMMAFREE-EASTMAN. It produces essentially the same outputs, but they are reflected left-to-right. (More precisely, if the former program gave the codeword y from the input sequence $x = x_0 \dots x_{n-1}$, this program gives the reverse of y when given the reverse of x .)

```
#define maxn 105
#include <stdio.h>
#include <stdlib.h>
int x[maxn + maxn];
int b[maxn + maxn];
int bb[maxn];
<Subroutines 5>;
main(int argc, char *argv[])
{
    register int i, i0, j, k, n, p, q, t, tt;
    <Process the command line 2>;
    <Do Eastman's algorithm 3>;
    <Print the solution 8>;
}
```

2. \langle Process the command line 2 $\rangle \equiv$

```

if (argc < 4) {
    fprintf(stderr, "Usage: %s x1 x2 . . . xn\n", argv[0]);
    exit(-1);
}
n = argc - 1;
if ((n & 1)  $\equiv$  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])  $\neq$  1  $\vee$  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 $j \geq 0$. The basic idea in the algorithm below is to also think of x as partitioned into $t \leq n$ subwords by boundary markers b_j where $0 \leq b_0 < b_1 < \dots < b_{t-1} < n$; then subword y_j is $x_{b_j}x_{b_j+1} \dots x_{b_{j+1}-1}$, for $0 \leq j < t$, 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 $0 \leq j < n$, 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 j th 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.

The algorithm is based on an interesting factorization of strings into substrings that have the form $z = z_1 \dots z_k$ where $k \geq 2$ and $z_1 \geq \dots \geq z_{k-1} < z_k$. Let's call such a substring a “dip.” It is not difficult to see that any string $y = y_0 y_1 \dots$ in which the condition $y_i < y_{i+1}$ occurs infinitely often can be factored *uniquely* as a sequence of dips, $y = z^{(0)} z^{(1)} \dots$. For example, $3141592653589 \dots = 314\ 15\ 926\ 535\ 89 \dots$.

Furthermore if y is a periodic sequence, its factorization is also ultimately periodic, although some of its initial factors may not occur in the period. Consider, for example, the factorizations

$$\begin{aligned} 1234501234501234501 \dots &= 12\ 34\ 501\ 23\ 45\ 01\ 23\ 45\ 01 \dots ; \\ 1234560123456012345601 \dots &= 12\ 34\ 56\ 01\ 23\ 45\ 601\ 23\ 45\ 601 \dots . \end{aligned}$$

If the period length is t , and if i_0 is the smallest i such that $y_{i-3} \geq y_{i-2} < y_{i-1}$, then one of the factors ends at i_0 and all factors are periodic after that point. The value of i_0 is at most $t + 2$.

Since t is odd, the period contains an odd number of dips of odd length. Each phase of Eastman's algorithm simply retains the boundary points that precede those odd dips.

```

⟨ 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. We might need to refer to $b[n + n - 1]$, but not $b[n + n]$.

```

⟨ Initialize 4 ⟩ ≡
  for (j = n; j < n + n; j++) x[j] = x[j - n];
  for (j = 0; j < 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} is less than subword y_i , otherwise it returns 0.

⟨Subroutines 5⟩ \equiv

```

int less(register int i)
{
    register int j;
    if ( $b[i] - b[i-1] \equiv b[i+1] - b[i]$ ) {
        for ( $j = 0; b[i] + j < b[i+1]; j++$ ) {
            if ( $x[b[i-1] + j] \equiv x[b[i] + j]$ ) 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⟩ \equiv

```

{
    for ( $i = 1; ; i++$ )
        if ( $\neg \text{less}(i)$ ) break;    /* now  $i \leq t$  and  $y[i-1] \geq y[i]$  */
    for ( $i += 2; i \leq t+2; i++$ )
        if ( $\text{less}(i-1)$ ) break;
    if ( $i > t+2$ ) {
        fprintf(stderr, "The input is cyclic!\n");
        exit(-666);
    }
    /* now  $y[i-3] \geq y[i-2] < y[i-1]$  */
    if ( $i < t$ )  $i0 = i$ ; else  $i = i0 = i - t$ ;
    for ( $tt = 0; i < i0 + t; i = j$ ) {
        for ( $j = i + 2; ; j++$ )
            if ( $\text{less}(j-1)$ ) break;    /* advance past the next dip */
            if ( $((j-i) \& 1) \langle \text{Retain } i \text{ as a boundary point 7} \rangle$ );
    }
    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. If $i \geq t$ at this point, we have “wrapped around,” so we actually want to retain the boundary point $i-t$. (This case will arise at most once per phase, because $j \geq i+3$ and we must have $j = i0 + t$. Therefore $i-t$ will be smaller than all of the previously retained points, and we want to shift them one space to the right.)

⟨Retain i as a boundary point 7⟩ \equiv

```

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

```

This code is used in section 6.

8. \langle Print the solution [8](#) $\rangle \equiv$
 for ($j = b[0]; j < b[0] + n; j++$) *printf*("␣%d", $x[j]$);
 printf("␣");

This code is used in section [1](#).

9. Index.

argc: [1](#), [2](#).
argv: [1](#), [2](#).
b: [1](#).
bb: [1](#), [6](#), [7](#).
exit: [2](#), [6](#).
fprintf: [2](#), [6](#).
i: [1](#), [5](#).
i0: [1](#), [6](#), [7](#).
j: [1](#), [5](#).
k: [1](#).
less: [5](#), [6](#).
main: [1](#).
maxn: [1](#).
n: [1](#).
p: [1](#).
printf: [6](#), [8](#).
q: [1](#).
sscanf: [2](#).
stderr: [2](#), [6](#).
t: [1](#).
tt: [1](#), [3](#), [6](#), [7](#).
x: [1](#).

- ⟨ Do Eastman's algorithm [3](#) ⟩ Used in section [1](#).
- ⟨ Do one phase of Eastman's algorithm, putting tt boundary points into bb [6](#) ⟩ Used in section [3](#).
- ⟨ Initialize [4](#) ⟩ Used in section [3](#).
- ⟨ Print the solution [8](#) ⟩ Used in section [1](#).
- ⟨ Process the command line [2](#) ⟩ Used in section [1](#).
- ⟨ Retain i as a boundary point [7](#) ⟩ Used in section [6](#).
- ⟨ Subroutines [5](#) ⟩ Used in section [1](#).

COMMAFREE-EASTMAN-NEW

	Section	Page
Intro	1	1
The algorithm	3	3
Index	9	6