Minimal Meandering Strings

July 1, 1970

James F. Gimpel
William Keister

A string over an alphabet of k symbols is said to be n-meandering (or is called an n-meander) if every word of length n is contained in the string (wrap arounds are permitted; i.e., the sequence may be regarded as circular). For example

22212111

is 3-meandering over the alphabet (1,2). Clearly an n-meandering sequence has length at least k^n. The questions that immediately arise are: is this minimum always obtainable and, if so, is there a reasonably efficient algorithm for finding such a sequence (i.e., a non-exhaustive procedure). The answer to both questions is in the affirmative. We present the following algorithm and prove that it meets the above requirements.

AN ALGORITHM

1. Place the characters in some arbitrary hierarchial ordering. Without loss of generality and to avoid subscripts we will denote the characters of our alphabet A as

$$A = (1, 2, \ldots, K)$$

with 1 the lowest and K the highest.

2. Initialize a string S as

$$S = 1^{n-1}$$

(that is (n-1) 1's).

3. Append to S the character K provided this does not cause a duplicity of n-tuples. Otherwise, try the next lower character in the hierarchy, continuing in this way until either a character is found and appended or no character can be added. If no character can be added go to 4. Otherwise repeat 3.

4. Call the resulting string S_1. Remove the initial "starter" substring (i.e., 1^{n-1}). Call the result S_2. Then S_2 is an n-meander of length k^n.

<u>Example</u>: Let k = 3 and n = 2. Then

Starter = 1

$$S_1 = 1332312211$$

$$S_2 = 332312211$$

<u>Example</u>: Let k = 2, n = 4

Starter = 111

$$S_1 = 1112222122112121111$$

$S_2 = 2222122112121111$

In what follows, a <u>suffix</u> of a given string is a substring which occurs at the extreme right and a <u>prefix</u> is a substring which appears at the extreme left.

<u>Lemma 1</u>. 1^n is a suffix of S_1.

<u>Proof</u>:

1^{n-1} is a suffix of S_1 because of the following argument. Let α be the trailing n - 1 character in S_1. Since α cannot be continued it must occur (k+1) times within S_1 counting the last time. But if it appears k + 1 times, one of these appearances must be as a prefix or else there would be k + 1 different n-tuples in the string with α as a suffix and this violates our construction rule that an n-tuple not appear twice.

Since 1^{n-1} is a suffix of S_1 and it could not be continued then the string 1^n must appear somewhere within S_1. But 1^n cannot have a continuation so it must be a suffix of S_1.

<u>Theorem 1</u>.

The string S_2 is an n-meander of length k^n.

<u>Proof</u>:

Because the method of construction guaranteed no repeated n-tuples we need only consider whether S_2 is n-meandering, or what is equivalent, whether S_1 contains all n-tuples.

Notation: If X and Y are sets of strings then XY is the set of all concatenations obtained by a string from X and a string from Y. We will use the notation X^n to denote X concatenated with itself n times. Thus A^3 is the set of all strings of length 3.

We will use

$$s_1 \Rightarrow s_2$$

to mean that if a string (or set of strings) s_1 is in S_1 then the string (or set of strings) s_2 is also in S_1. We will use x and x_i to denote characters and the Greek letters α and β to denote strings. As before A will denote the alphabet. We will prove the theorem by induction.

basis Let α be any string of length n - 1. Then

$$\alpha 1 \Rightarrow \alpha A$$

This is true because before a 1 is added to the end of the string S all other characters are tried first.

induction step Assume that for all α of length n - a

$$\alpha 1^a \Rightarrow \alpha A^a$$

then if α is of length n - a - 1

$$\alpha 1^{a+1} \Rightarrow \alpha A^{a+1}$$

If the induction step can be proved we prove our theorem by setting α to null and using Lemma 1.

We will first show that

$$\alpha 1^{a+1} \Rightarrow A\alpha 1^{a}$$

If $\alpha = 1^{n-a-1}$ then the statement is true because we have seen (Lemma 1) that there were k nonprefix instances of 1^{n-1} in S_1. The statement follows from the nonduplicity of n-tuples in S_1.

Assuming $\alpha \neq 1^{n-a-1}$ we have

$$\alpha 1^{a+1} \Rightarrow \alpha 1^{a} A$$

by using the same reasoning when proving the basis. Moreover, none of these are prefix strings since α contains characters other than 1. Hence

$$\alpha 1^{a+1} \Rightarrow (x_1 \alpha 1^{a} 1, \ldots, x_k \alpha 1^{a} K)$$

where each $x_i \in A$. Now each of these substrings is unique and of length n + 1. It follows that the x_i are all different. So that

$$\alpha 1^{a+1} \Rightarrow A\alpha 1^{a}$$

By the induction hypothesis

$$\Rightarrow A\alpha A^{a}$$

Consider any string β in αA^a. Its length is n - 1 so that if $A\beta$ occurs so also does βA and hence

$$\Rightarrow \alpha A^a A = \alpha A^{a+1}$$

Then by mathematical induction

$$1^n \Rightarrow A^n$$

Using Lemma 1 we deduce that S_1 contains all n-tuples.

ALTERNATE GENERATION METHODS

For alphabets of size p^m, where p is some prime and m is some integer, Galois Field theory provides a method of generating minimal n-meanders for any value n [1]. All that is necessary is to find a primitive polynomial in $GF(p^n)$ of degree n (the existence of which is assured).

Although this generation method works for only certain values of alphabet size and requires a search for a primitive polynomial, it nonetheless has the advantage of producing a meander with an algebraic structure which presumably can be used to advantage in determining indexing schemes (i.e., mappings between an n-tuple and its location within the meander).

APPLICATIONS

1. Drum Synchronization

A minimal meander has the characteristic of minimal redundancy and this property can be used to advantage in certain synchronization problems.

Let a drum consist of t tracks (and t read heads) and in each circular track there are m cells (equi-spaced) and in each cell we can place any one of k symbols.

We assume a program is synchronized with the drum in the sense that it is aware at any given moment which of the m cells are under the read heads. In order to build into our system the capability of regaining lost synchronization we can do one of several things. We can place a special mark in track 1 to indicate position 0. This however requires a half drum revolution (on the average) to detect and recover. We could use P tracks to encode the position where $k^P = m$. This has the advantage that drum synchronization can be obtained immediately but has the disadvantage that P tracks have to be given up. A third scheme is to use a minimal P-meander over the alphabet of k symbols on track 1. This has the advantage that only 1 track is used and that synchronization can be regained in a fraction ($\log_k m/m$) of a rotation.

2. Gear Synchronization

It is possible to design a set of self-synchronizing gears such that one gear will slip with respect to the other until synchronism is achieved. (The authors are unaware of any previous publication of such a device). The method is as follows. Assume that rather than have one gear, we have two gears firmly fixed to a driving shaft. Further assume each gear has some teeth missing in a way that the teeth of one gear complement the set of teeth of the other. Wherever the right gear has a tooth we will say this corresponds to a binary digit 1 and wherever the left gear has a tooth this will correspond to the binary digit 0. The teeth around the gear pair can then be described by a 1-0 sequence. Assume that on the driven shaft we have a similar gear-pair arrangement with the identical 1-0 sequence (in the opposite direction). An example of such a system is depicted in Figure 1a; the associated 1-0 sequence is

1111111100000000

Whenever a bit in the upper gear is aligned with the opposite bit in the lower gear, the upper gear will slip with respect to the lower gear so that a nonslip (stable) situation will not be reached until the sequence on the upper gear matches the sequence on the lower gear. Once the sequences correspond, no more slippage can take place and the gears are in synchronism.

Thus in order to obtain a self-synchronizing set of gears all that is necessary is to have a 1-0 sequence which is different from any cyclic (end-around) shift of itself.

Consider now the operation of the set of gears shown in Figure 1a. After revolving freely for almost 180° the upper gear meshes with teeth on the lower (left) gear. This is shown in Figure 1b. After both gears, enmeshed, move through an angle of almost 180°, the left gears disengage as shown in Figure 1c. Finally, after a relatively small angle, the right gears engage and the system of gears is synchronized (as shown in Figure 1d).

A disadvantage of the system shown in Figure 1 is that a substantial impact of one gear upon another may be felt if the traverse is considerable (as in going from 1a to 1b). Define the <u>clunk index</u> as the longest run of 1's or 0's in the 1-0 sequence describing the gears. Then the clunk index will be the maximum traverse, measured in teeth, over which the driving gears can rotate freely without engaging in teeth of the driven gears. The gear arrangement shown in Figure 1 has a clunk index of 8.

The gear arrangement shown in Figure 2 has a 1-0 sequence

1010101010101100

and has a clunk index of 2. It is always possible to design a clunk index of 2 for any number of teeth and a clunk index of 1 is inconsistent with a unique settling point unless the number of teeth is 1 or 2.

The gear arrangement of Figure 2, while having a low clunk index has a relatively large <u>index of rotation</u> defined as the maximum number of turns required to move the driving gear in order to achieve synchronism. This is a consequence of the fact that the 1-0 sequences are locally similar even though not in synchronism.

A meandering string represents a reasonable compromise between these two cases (one such is shown in Figure 3). On the one hand the clunk index is $\log_2 n$ where n is the number of teeth. On the other hand the index of rotation is bounded above by $\log_2 n$ because slippage is guaranteed to occur at least once every $\log_2 n$ tooth movements. As a practical matter the average slippage rate appears to be closer to 1/2 so that, on the average, synchronism could be restored in 1 turn of the driving wheel.

3. <u>Random String Generation</u>

Consider the problem of constructing a (pseudo-) random string generator where the strings are uniformly distributed over all strings of length n from a given alphabet. The normal procedure is to generate n (pseudo-) random integers to index randomly (n times) into a string denoting the alphabet

One can trade storage space for running time in the following way. The alphabetic string can be considered a 1-meander. If in place of the 1-meander we used an m-meander then only n/m random integers need be generated and n/m random selections need be made.

4. Conversion Algorithms

The notion of a meandering string leads to a rather unique method of programming certain kinds of string conversion problems. Consider, for example, the following program which converts from an alphabetic character to Morse code.

A straightforward approach to the problem would be as follows: obtain somehow a number and use that number to index into an array of character strings which depict the set of Morse Code equivalents. The storage required for this method is several (probably 4) characters of Morse code for each alphabetic character.

Since a meandering string meanders over all possible combinations, a single string can be used to house all the codes and instead of indexing into the array one can index into the meander. A problem arises due to the fact that Morse code is variable length whereas the item obtained by indexing into the meander has, because no other information is given, a fixed length. We, therefore, use the convention that all characters up to and including the first dash of a 5-cnaracter sequence is to be ignored. In this way, the 4-character, variable-length Morse Code is transformed into a fixed-length 5-character code.

Figure 4 shows a PL/I program to convert to Morse code using this meandering string algorithm.

One could eliminate the need for storing Morse codes altogether by simply using the indexing number (which are arbitrarily associated with the alphabetic characters) as bit patterns from which to construct the Morse characters. However, the meandering string does not consume much storage so that the storage-saving advantages of the latter method are probably not worth the increase in computation time.

OPEN QUESTIONS

Several open questions remain. How many minimal n-meanders are there over a given alphabet without permitting the relabeling of characters, or the cycling or reversing of the meander to permit producing a different meander? For example there are exactly 4 minimal 4-meanders over the alphabet 0-1. These are

1111011001010000

1111010010110000

1101111001010000

1100101111010000

The first is the one which would be generated using the algorithm presented above; by seeming coincidence it would also be obtained using either of the 2 primitive polynomials of degree 4 over GF(2). Are there ways short of exhaustion of generating all minimal n-meanders for a given alphabet?

To determine the location of any given n-tuple within a meander (decoding), a search can be made. Can this be reduced to a computation which would presumably speed up the decoding process?

It is possible to speak meaningfully of multi-dimensional meanders. For example, a 2×2 meander cver the alphabet (A, B) is shown below.

A	A	A	B	A
B	B	A	B	A
B	B	A	B	B
A	A	B	A	B

A lower bound on the number of cells in such 2×2 meanders is 16. It is not known whether this minimum is obtainable. In fact, it is not known whether the lower bound of $k^{n_1 n_2 \cdots n_d}$ can be reached for any $(n_1 \times n_2 \times \ldots \times n_d)$-meander for both d and $k > 1$. On the other hand, there are no known applications of multi-dimensional meanders.

ACKNOWLEDGMENT

Technical illustrations were kindly furnished by Charles P. Gimpel of Philadelphia, Pa.

[illegible]

JAMES F. GIMPEL

[illegible]

WILLIAM KEISTER

Att.
Reference
Figures 1-4

REFERENCES

[1] W. W. Peterson, "Error-Correcting Codes," M.I.T. Press, 1961, Chapter 8.3.

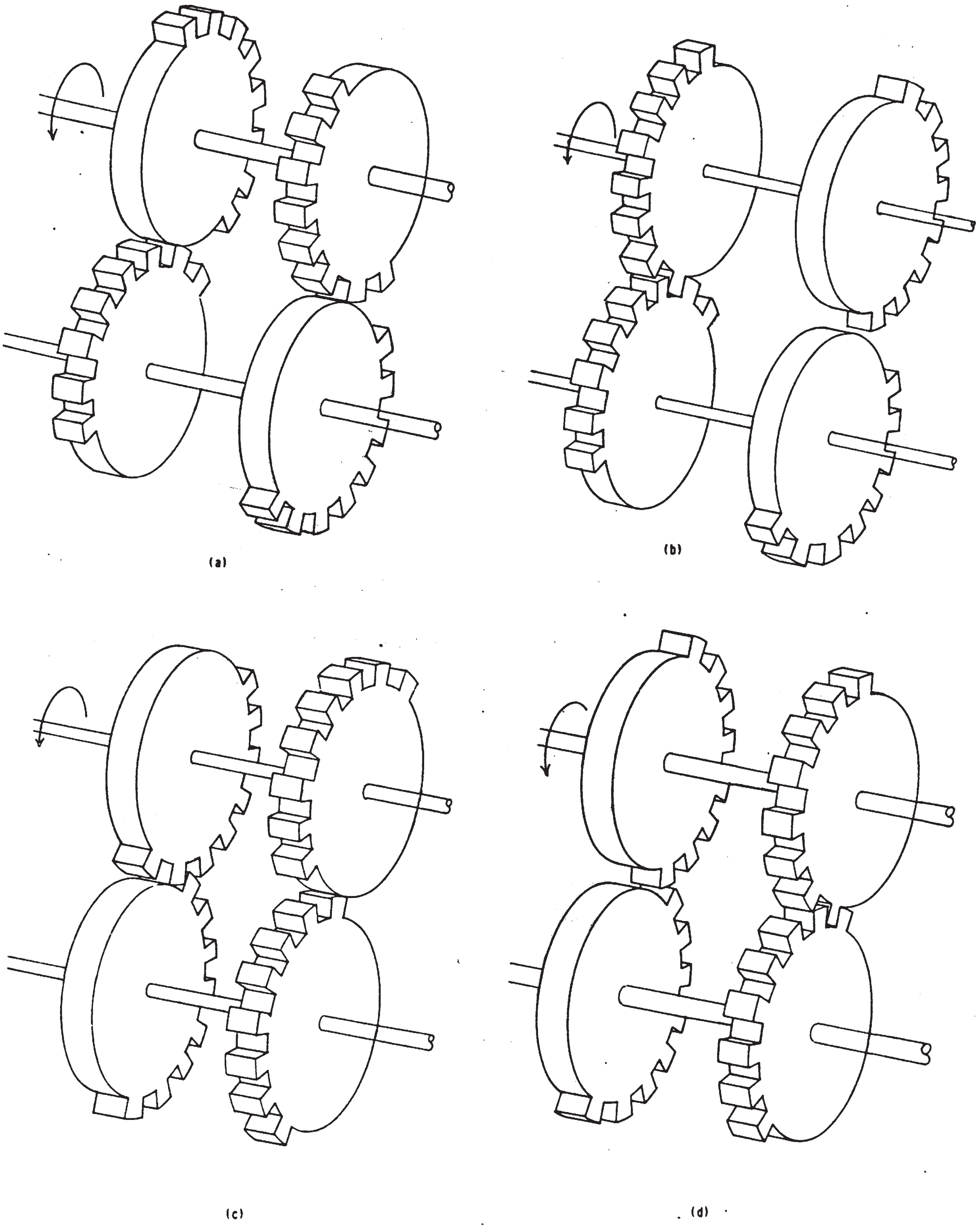

FIGURE 1

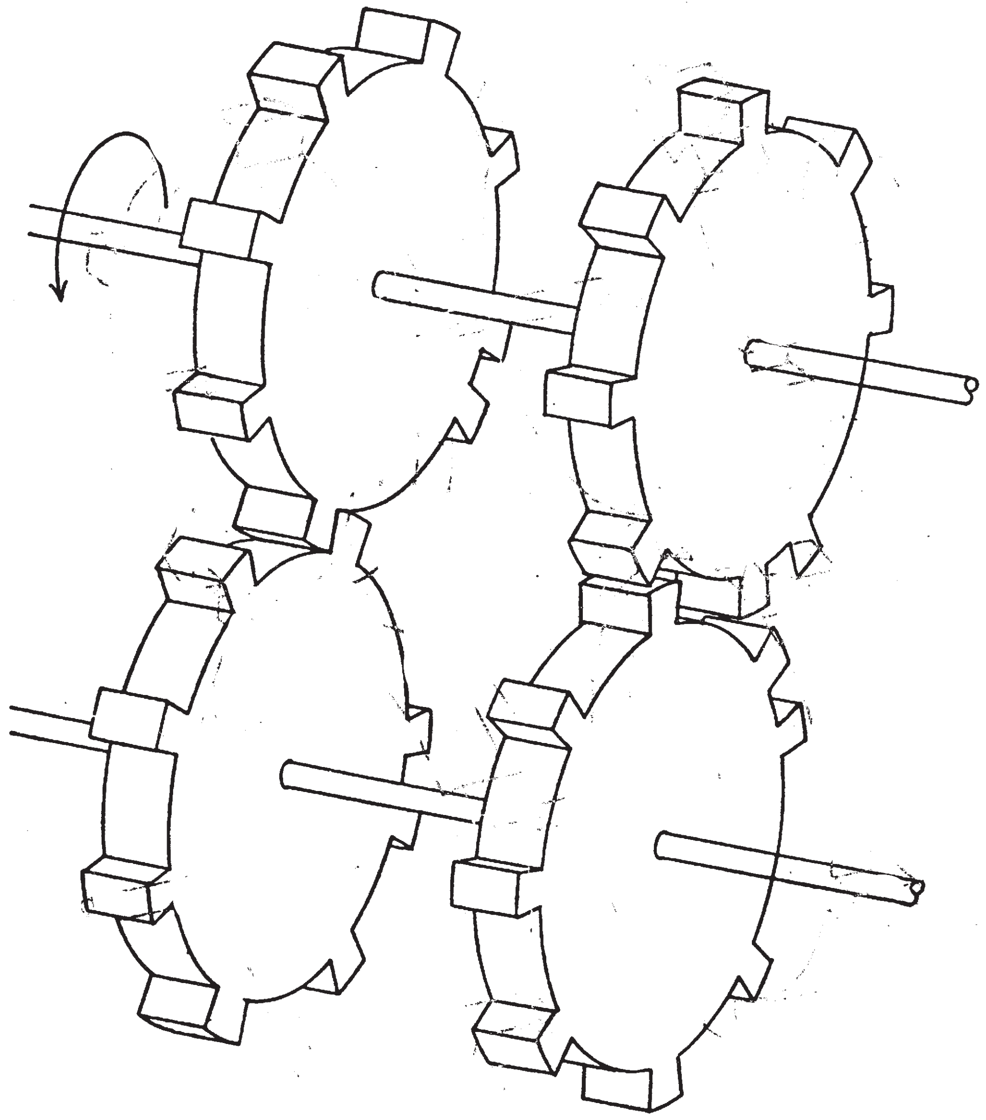

FIGURE 2

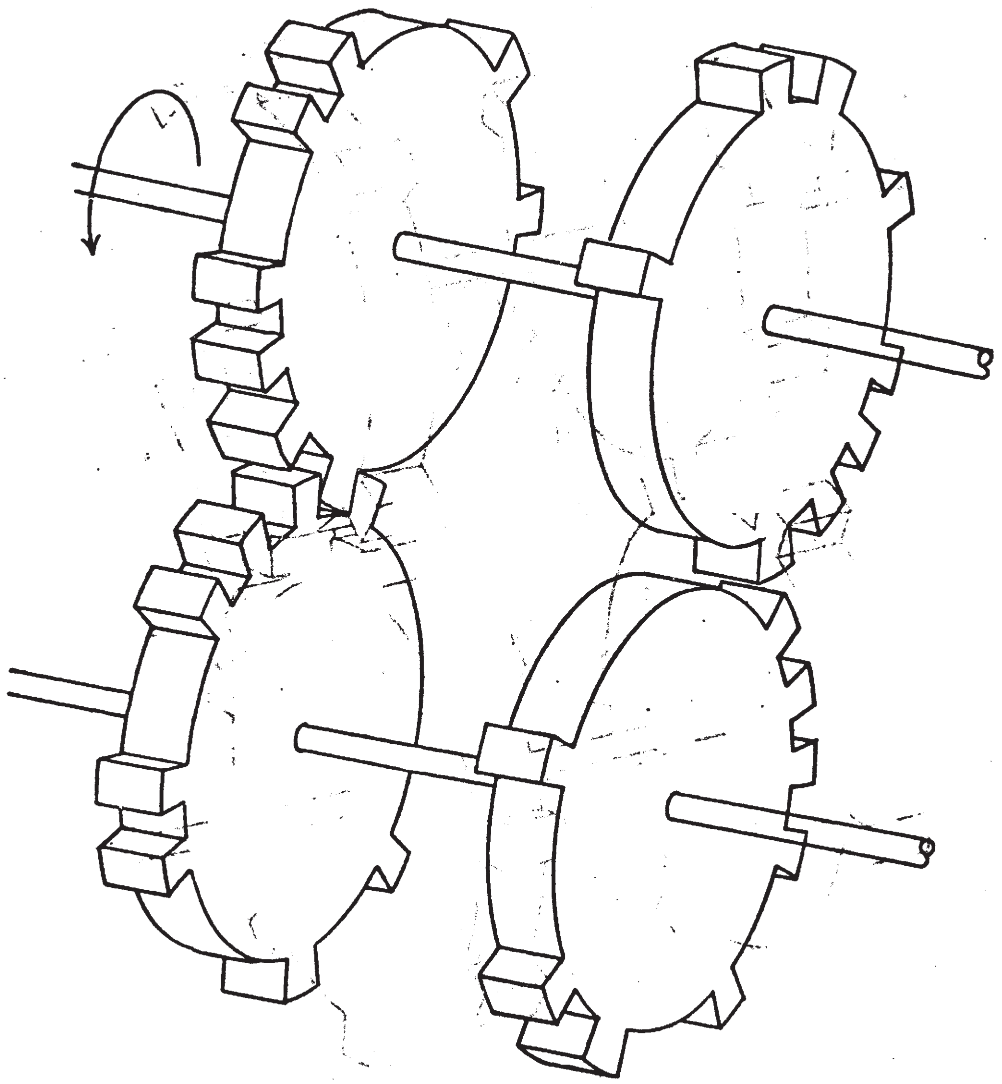

FIGURE 3

```
        DECLARE A CHAR(32);
        DECLARE X CHAR(5);
        DECLARE TXT CHAR(16) VAR;
        DECLARE P CHAR(80) VAR;
        DECLARE C CHAR(36);
           /* assign to A a string containing in the ith
              position the character whose index is i */
        A='**tmo**qj*gzxynkc*wpdbvearlufish';
           /* assign to C the meandering string */
        C='....-----.---..--.-.--...-.-..-....';
           /* read in the text to be converted */
START:  GET LIST(TXT);
           /* null out the result string, P */
        P='';
           /* for each character in the input text... */
        DO K=1 TO LENGTH(TXT);
           /* determine where, within A, the character
              lies and assign to N its position */
        N=INDEX(A,SUBSTR(TXT,K,1));
        IF N=0 THEN GO TO NEXT;
           /* if not found, ignore; otherwise extract the
              nth 5-character string from C */
        X=SUBSTR(C,N,5);
           /* remove everything up to and including the
              first dash of this string and append onto P */
        P=P || SUBSTR(X,INDEX(X,'-')+1);
           /* append on a blank to separate codes */
NEXT:   P=P || ' ';
        END;
           /* print the results and repeat */
        PUT LIST(P);
        GO TO START;
```

Figure 4