A GRAY CODE FOR SET PARTITIONS

Richard KAYE
Wharton School, University of Pennsylvania, USA

Received 1 September 1976

Partition, set, Gray Code, computational complexity.

1. Introduction

We consider here the collection of all partitions of an n-set, for fixed n, and we ask: is it possible to arrange these partitions in a list (Gray Code) so that each partition is obtained from its immediate predecessor by changing the class of exactly one element? The answer is affirmative, and we give descriptions of two algorithms which implement the method, the first recursive, the second non-recursive. The question was raised by Nijenhuis and WiIf in [1]. The answer in recursive form was given first by Knuth [unpublished]. Our non-recursive algorithm has uniformly bounded average labor per partition as a function of n.

2. The recursive construction

In this section we construct the list of partitions as described above.

Let π be any partition of {l, 2,…,n–l}. By the "children" of π we mean those partitions of {l, …,n} which can be obtained from π by either inserting the letter n into one of the classes of π or adjoining n to π as a singleton class.

Suppose, inductively, that we have the list L_{n-1} of partitions of {l, 2,…,n–l} in Gray Code sequence. Take the first partition from the n–l list and list its children in their natural order *(n* in the first class, n in the second class, etc.), then take the children of the second element in the n–l list and list them in reverse order, then the children of the third partition in their normal order and so on reversing the order with each group of siblings. Within each group of siblings n is the only letter that changes class. At the junction point between families, n stays in either the first or last class and only one element moves; for if we delete n, we are left looking at two consecutive members of the n − 1 list, and only one letter moves, by induction.

The list for n = 3
1. (1 2 3)
2. (l 2) (3)
3. (l) (2) (3)
4. (l) (2 3)
5. (1 3) (2)

3. The non-recursive algorithm

By introducing suitable auxiliary arrays we can convert the algorithm to non-recursive form. Observe that within its own universe (all numbers less than or equal to the given number) each number moves one class to the right when it is the active number until it is in a singleton class (in its universe) and then it moves one class to the left each time till it gets back to the first class. (At each end it pauses for one move and allows smaller numbers to move). This process then repeats itself. So we define a velocity array vel(j) which holds 1, if j is moving to the right; –1, if j is moving to the left; 2, if j has moved into a singleton class in its universe and after a pause will start to move back to the left; and –2 if j has moved into the first class and after a pause will start to move to the right again. We initialize this array to all l's. To hold the partition, we set up an array class (j) which tells which class j is in. This is also initialized to all l's.

Now we are ready to look at our arrays and develop the next partition. We start by checking vel (n). If it is 2 (–2), n is pausing, so we change vel (n) to –1(1) and inspect vel(n–1), repeating the same procedure until we find an element (j) whose velocity is 1 or –1. If it is 1, we increase class (j) by 1 and check for one of two cases. If j has moved into a class with elements less than j we are done; but if j has become a singleton in its universe we must adjust the class vector so that all classes to the right of j's original class get renamed with a name one greater than before, since a new class has been created. If vel (j) is –1, we let class (j) decrease by one and we must check to see if j has vacated a singleton class in which case the classes to the right will close ranks by being renamed with a number one less than their present name.

At this point we could write an algorithm which would produce the Gray Code, but it would require a lot of searching and looping to test which case we were in once the active number was found. To eliminate most of this searching we introduce a surprisingly easy-to-maintain array small (k), which contains the smallest element in the kth class. When vel (j) equals 1, we need only test small (class (j)); if it is less than j we are done, if it is greater than j we must rename.

In the second case small (class (j)) is set to j after the other classes have been renamed but in either case, j was not the smallest in its old class so the small array needs no maintenance. When vel (j) is –1, we test small (class (j)) before j moves. If it equals j, then j is a singleton class (no number larger than j could be in its class or else the larger number would be active); if it is greater than j then again no maintenance is required. The formal algorithm follows:

```
Algorithm NXEQU1 [Given $n$. Output next partition
                 of $n$-set in the form class ($j$) ($j$ =
                 1, n). Variable $nc$ is the number
                 of classes.]
(A) [First entry. Put all elements in first class and initialize other arrays.] Set class ($i$) $\leftarrow$ 1 ($j$ = 1, $n$);
    vel ($i$) $\leftarrow$ 1 ($j$ = 2, $n$); small (1) $\leftarrow$ 1; $nc \leftarrow$ 1; exit.
(B) [All later entries.] Set $j \leftarrow n$.
(C) If $j$ = 1, final exit [there are no more partitions];
    otherwise [test if $j$ is active] if vel ($j$) = ±2, go to
    (G); otherwise [find direction $j$ moves] if vel ($j$) =
    1, go to (H); otherwise [$j$ is moving to left] if
    small (class ($j$)) $\neq j$, go to (F); otherwise [$j$ was a
    singleton so rename classes.] set $nc \leftarrow nc$ - 1;
    $p \leftarrow j$, and go to (E).
(D) If class ($p$) = 1, go to (E); otherwise [rename class
    ($p$)] set class ($p$) $\leftarrow$ class ($p$) - 1; set small (class
    ($p$)) $\leftarrow p$.
(E) If $p = n$, go to (F); otherwise set $p \leftarrow p$ + l and go
    to (D).
(F) [Done with renaming]. Set class ($j$) $\leftarrow$ class ($j$) - 1;
    then if class ($j$) = 1, set vel ($j$) $\leftarrow$ -2. Exit.
(G) [Change $j$'s velocity and examine next number]
    vel ($j$) $\leftarrow$ vel ($j$)/(-2); $j \leftarrow j$ - 1; go to (C).
(H) [$j$ is moving to right]. Set class ($j$) $\leftarrow$ class ($j$) + 1;
    if class ($j$) > $nc$, go to (L); otherwise [test for $j$
    becoming singleton in its universe] if small (class
    ($j$)) < $j$, exit; otherwise [$j$ becomes a singleton]
    set $nc \leftarrow nc$ + 1, vel ($j$) $\leftarrow$ 2, and $p \leftarrow j$ + 1.
(I) If class ($p$) = 1, go to (J); otherwise [rename class
    ($p$)] set class ($p$) $\leftarrow$ class ($p$) + 1; set small (class
    ($p$)) $\leftarrow p$.
(J) If $p \neq n$, go to (K); otherwise set small (class ($j$))
    $\leftarrow j$ and exit.
(K) Set $p \leftarrow p$ + 1; go to (I).
(L) [$j$ becomes a singleton] . Set vel ($j$) $\leftarrow$ 2; $nc \leftarrow nc$ +
    1; small (class ($j$)) $\leftarrow j$; exit.
```

4. The complexity of the algorithm

Notice that all loops are of length n - j where j is the active letter, so we must examine the behavior of the average j as n increases. Let a_n be the number of partitions of an n-set (i.e., the familiar Bell number). We see from the inductive argument of section one that n is the active letter $a_n - a_{n-1}$ times, n - 1 is the active letter $a_{n-1} - a_{n-2}$ times, and k is the active letter $a_k - a_{k-1}$ times for every k. The average active letter ($\bar{j}$) is therefore,

$$\bar{j} = \frac{\sum_{k=1}^{n} k\left(a_k - a_{k-1}\right)}{a_n}$$

$$= n - \frac{1}{a_n}\sum_{k=0}^{n-1} a_k \ ,$$

so the average length of the loop is:

$$\frac{1}{a_n}\sum_{k=0}^{n-1} a_k \ .$$

Now by examining our earlier statements about the construction of partitions of an *n*-set from partitions of an $n - 1$ set, we see that each partition of an $n - 1$ set generates at least two partitions of an *n* set, so $a_{k+1} \geq 2a_k$; therefore, $a_n > \sum_{k=0}^{n-1} a_k$ and the average length of the loops is less than 1, independent of *n*. This proves that the average work needed to produce a partition is uniformly bounded.

It is interesting to notice that the algorithm always keeps the classes arranged in ascending order of their least element and not only does just one element change class but it always moves into an adjacent class. Also the storage requirements can be decreased; since the velocity array has only four possible values; it can be packed into the other two arrays in their sign bits.

Acknowledgement

I would like to thank Dr. Herbert Wilf for his suggestions and advice.

References

[1] S. Nijenhuis and H.S. Wilf, Combinatorial Algorithms (Academic Press, New York, 1975).
[2] S. Even, Combinatorial Algorithms (Macmillan, New York, 1973).