Algorithms

ALGORITHM 152

V-Partitions and Permutations by Inversions

J.O. Shallit

Abstract

We present an algorithm for generating, subject to certain restrictions, additive partitions of a non-negative integer. We obtain simple time estimates for this algorithm. The computation of combination vectors is given as an application. Finally, it is shown that a simple transformation permits as a special case the computation of permutations on *N* letters with *K* inversions.

- - - - -

Introduction

A partition of a non-negative integer *K* is a vector of non-negative integers *W* such that *K*=+/*W* (i.e., *W* sums to *K*).

A V-partition of *K* is a partition *W* subject to the restriction that ∧/*W*<*V*, i.e. *W* is elementwise strictly less than the vector of non-negative integers *V*.

For example, let *V*←1 2 3 4. Then all *V*-partitions of 3 are given by the rows of the following matrix:

```
0 1 2 0
0 1 1 1
0 1 0 2
0 0 2 1
0 0 1 2
0 0 0 3
```

Algorithm

The program *PART* computes all *V*-partitions of *K*:

```
     ∇ Z←K PART V;B;R;I;T
       ⍝ THE RESULT Z IS A MATRIX WHOSE ROWS
       ⍝    CONSIST OF ALL INTEGER VECTORS W
       ⍝    WITH K=+/W AND (0≤W)∧W<V .
       ⍝ THE VECTORS ARE PRODUCED IN REVERSE
       ⍝    LEXICOGRAPHICAL ORDER BY A
       ⍝    NON-RECURSIVE ALGORITHM.
       Z←(0,⍴V)⍴0
       →(0∊V)/0
       T←(⍴V)⍴I←0
       L0: B←K-+/I↑T
       R←B BREAK I↓V
       →(B=+/R)/L1
       L2: I←((T>0)∧I>⍳⍴T) RIOTA 1
       →(I=0)/0
       T[I]←T[I]-1
       →L0
       L1: T←(I↑T),R
       Z←Z,[1] T
       I←⍴V
       →L2
     ∇

     ∇ Z←K BREAK V;R;T
       ⍝ THE RESULT Z IS A VECTOR SUCH THAT
       ⍝    (⍴V) = ⍴Z AND K = +/Z (IF
       ⍝    POSSIBLE) AND (0≤Z)∧Z<V AND THIS
       ⍝    IS THE LAST SUCH Z (IN LEXICO-
       ⍝    GRAPHICAL ORDER).
       T←(K≤+\V-1)⍳1
       R←(T-1)↑V-1
       Z←(⍴V)↑R,K-+/R
     ∇

     ∇ Z←V RIOTA K
       ⍝ GIVES INDEX OF FIRST OCCURRENCE OF
       ⍝    K IN THE VECTOR V, SCANNING FROM
       ⍝    THE RIGHT.
       ⍝    IF ~K∊V THEN THE RESULT IS 0.
       Z←1+(⍴V)-(⌽V)⍳K
     ∇
```

Method

Clearly if *V* contains any 0-elements, then we cannot find <u>any</u> partition satisfying the given conditions. This check is performed in Line 6.

The partitions are produced from largest to smallest (in the sense of lexicographic order). *W*, the largest *V*-partition of *K*, is easily computed. The technique is to fill in the positions of *W* successively from left to right with the largest possible element for each position, until the sum equals or exceeds *K*; then the last position is decremented to get a sum that exactly equals *K*. This is done by the subfunction *BREAK*.

Now, given *W*, the problem is to determine the next smallest partition *Y*. Such a partition would have an *I* with `(1≤I)∧I<N`; and `Y[I]<W[I]` but `Y[K]=W[K]` for all `K>I`. We are therefore looking for an element of *W* to decrement. This search is performed in Line 11.

Once a suitable *I* has been found, we decrement `W[I]` by 1 in Line 13. We then replace `W[I,(I+1),...,N]` by the largest possible subvector that will allow *W* to retain the *K*-sum property. If no such subvector exists, we decrement *I* by 1 and continue. If *I* = 0, there are no smaller vectors and we are done.

Worst-Case Analysis

Let `N=ρV`, the number of elements in *V*. Then the time required to compute an individual partition is, in the worst case, proportional to `N*2`. To see this, note that *I* is specified to be `ρV` in Line 17 and is decreased by at least one in Line 11. Hence the loop in Lines 8-14 is performed at most *N* times. Computation time within the loop is easily seen to be proportional to *N*, and the result follows.

Actually, this algorithm usually performs much better than this simple example would indicate. Results of some timings (in milliseconds) on an IBM 4341 were as follows:

```
E←'(⌊N÷2) PART Nρ2'
F←'(N-1) PART ⍳N'
```

N	(TIME E)÷1↑ρ⍎E	(TIME F)÷1↑ρ⍎F
1	20	20
3	17	18
5	17	15
7	17	16
9	19	--
11	23	--
13	38	--

Applications

A. Computation of Combinations

The expression `K PART Nρ2` calls the partition function with *N* repetitions of 2. Hence for `(0≤K)∧K≤N` this expression computes all logical vectors of length *N* that sum to *K*; there are `K!N` such. For example, the function *COMB* returns a matrix of all possible *K*-choices from 1,2,...,*N*:

```
     ∇ Z←K COMB N
[1]    Z←((K!N),K)ρ(,K PART Nρ2)/(N×K!N)ρ⍳N
     ∇

       2 COMB 4
 1 2
 1 3
 1 4
 2 3
 2 4
 3 4

       'ABCD'[2 COMB 4]
AB
AC
AD
BC
BD
CD
```

B. Tabulation of Permutations by Number of Inversions

We define a <u>permutation</u> *V* to be a vector of length *N* that is a rearrangement of `⍳N`. An <u>inversion</u> occurs in *V* when `I<J` but `V[I]>V[J]`.

For each permutation *V* we may consider the associated vector `IFP V` formed by letting the *I*-th element be the number of indices *J* with `J>I` and `V[I]>V[J]`, i.e. the number of inversions occurring at *I*.

```
     ∇ Z←IFP P
[1]    ⍝ INVERSION FROM PERMUTATION
[2]    Z←+/(P∘.>P)×(⍳ρP)∘.<⍳ρP
     ∇
```

Note that:

```
    ∧/(IFP V)<⌽⍳ρV                                  (1)
```

that is, there can be no more than *N*-1 inversions occurring at `V[1]`, *N*-2 inversions occurring at `V[2]`, etc. The following theorem relates permutations to `(⌽⍳N)`-partitions:

<u>Theorem</u>. There exists a 1-to-1 correspondence between permutations of length *N* with *K* inversions and `(⌽⍳N)`-partitions of *K*.

<u>Proof</u>. Let P be a permutation of length *N* with *K* inversions. Then *K* = `+/IFP P` and by

(1) we have ∧/(*IFP P*)<⌽⍳⍴*P*. Thus *IFP P* is a (⌽⍳*N*)-partition of *K*. Now we must show that there is a permutation with an inversion pattern corresponding to any (⌽⍳*N*)-partition *R* of *K*.

If *R*[1] = *J*, then clearly the permutation *P* corresponding to *R* must have *P*[1] = *J*+1, as any other choice for *P*[1] would lead to an incorrect number of inversions in the first position. Now we discard *J*+1 from the set S = {1,2,...,*N*} to get a new set S'. Then it is easy to see that *P*[2] must be the (*R*[2]+1)-st smallest element of S'. We can continue in this fashion to define each element of *P* = *PFI R*.

```
     ∇ P←PFI Z;S;T
[1]    ⍝ PERMUTATION FROM INVERSION
[2]    P←⍳0
[3]    S←⍳⍴Z
[4]    L1:→(0=⍴S)/0
[5]    T←1+Z[1]
[6]    P←P,S[T]
[7]    S←(T≠⍳⍴S)/S
[8]    Z←1↓Z
[9]    →L1
     ∇
```

The permutation we get by this process is uniquely defined, and no other permutation *Q* can have ∧/*R*=*IFP Q*. QED

To tabulate permutations by inversions, we first compute the (⌽⍳*N*)-partitions of *K* using *PART* and then transform them to permutations using the transformation *PFI*.

```
     ∇ Z←K PWKI N;J;T
[1]    ⍝ PERMUTATIONS OF LENGTH N
[2]    ⍝   WITH K INVERSIONS
[3]    T←K PART⌽⍳N
[4]    Z←(0,N)⍴J←0
[5]    L0:→(J≥1↑⍴T)/0
[6]    Z←Z,[1] PFI T[J←J+1;]
[7]    →L0
     ∇

       2 PWKI 5
 3 1 2 4 5
 2 3 1 4 5
 2 1 4 3 5
 2 1 3 5 4
 1 4 2 3 5
 1 3 4 2 5
 1 3 2 5 4
 1 2 5 3 4
 1 2 4 5 3
```

J.O. Shallit
Department of Mathematics
University of California
Berkeley, California
USA 94720

⎕