SEPnkA: Simple, Efficient P(n,k) Algorithm

Alistair A. Israel
aisrael@gmail.com

Abstract

Drawing heavily from "SEPA: A Simple, Efficient Permutation Algorithm" by Jeffrey A. Johnson, this paper adapts and extends Johnson's algorithm to generate all permutations of *n* items taken *k* at a time, in lexicographic order.

Introduction

Jeffrey A. Johnson's SEPA: A Simple, Efficient Permutation Algorithm is indeed a fast and simple way to generate all the permutations of *n* items or P(*n*) in lexicographic order. In this article, we describe a similar algorithm for generating permutations of *n* items taken *k* at a time, which we denote as P(*n*, *k*).

SEPA Explained

Johnson discovered that a set of logical steps could be taken from one permutation to the next. Repeatedly calling the algorithm on the last output obtained generates all permutations in sorted order.

It works by first scanning for the rightmost ascent or pair of numbers where the first number is less than the one immediately after it.

Table 1: $P(5)_{17..24}$ with rightmost ascents highlighted.

P_i	a_0	a_1	a_2	a_3	a_4
⋮			⋮		
17	0	3	4	2	1
18	0	4	1	2	3
19	0	4	1	3	2
20	0	4	2	1	3
21	0	4	2	3	1
22	0	4	3	1	2
23	0	4	3	2	1
24	1	0	2	3	4
⋮			⋮		

If no ascent is found, then all numbers are in descending order and this is the last, lexicographic permutation.

Otherwise, the number at the start of the ascent is swapped with the smallest higher number to its right.

Table 2: P'(5)$_{17..24}$ after swaps (in red). Highlighted cells are about to be reversed.

P'_i	a_0	a_1	a_2	a_3	a_4
⋮			⋮		
17'	0	4	3	2	1
18'	0	4	1	3	2
19'	0	4	2	3	1
20'	0	4	2	3	1
21'	0	4	2	3	1
22'	0	4	3	2	1
23'	1	4	3	2	0
24'	1	0	2	4	3
⋮			⋮		

Finally, all the numbers to the right (which will be in descending order) are 'flipped' or reversed (to ascending order). The resulting array is the next permutation.

P(n) to P(n,k)

To begin generating P(n,k), let's take a look at the first few permutations P(5,3) and compare this with P(5):

Table 3: P(5, 3)$_{0...5}$

P_i	a_0	a_1	a_2
0	0	1	2
1	0	1	3
2	0	1	4
3	0	2	1
4	0	2	3
5	0	2	4
⋮		⋮	

Table 4: P(5)$_{0..10}$

P_i	a_0	a_1	a_2	a_3	a_4
0	0	1	2	3	4
1	0	1	2	4	3
2	0	1	3	2	4
3	0	1	3	4	2
4	0	1	4	2	3
5	0	1	4	3	2
6	0	2	1	3	4
7	0	2	1	4	3
8	0	2	3	1	4
9	0	2	3	4	1
10	0	2	4	1	3
⋮			⋮		

If we look carefully at the first 3 values of P(5), a_0, a_1 and a_2, we actually find P(5, 3). Except, to get P(5, 3) from P(5) we need to 'skip' a few permutations.

Table 5: P(5) with P(5, 3) highlighted

$P(5)_i$	a_0	a_1	a_2	a_3	a_4
0	0	1	2	3	4
1	0	1	2	4	3
2	0	1	3	2	4
3	0	1	3	4	2
4	0	1	4	2	3
5	0	1	4	3	2
6	0	2	1	3	4
7	0	2	1	4	3
8	0	2	3	1	4
9	0	2	3	4	1
10	0	2	4	1	3
⋮			⋮		

Basically, we can safely 'skip' all permutations in P(5) that don't permute the first 3 positions. For purposes of discussion, we designate the k^{th} or 3rd position, a_2 as the 'edge'.

Table 6: P(5) where a_0, a_1 and a_2 change. The 3rd position, or a_2 is the 'edge'

P_i	a_0	a_1	a_2	a_3	a_4
0	0	1	2	3	4
2	0	1	3	2	4
4	0	1	4	2	3
6	0	2	1	3	4
8	0	2	3	1	4
10	0	2	4	1	3
12	0	3	1	2	4
14	0	3	2	1	4
16	0	3	4	1	2
18	0	4	1	2	3
20	0	4	2	1	3
⋮			⋮		

Another way to look at this is that in generating P(n) using Johnson's SEPA, we look for the rightmost ascent. In P(n, k) generation, we're only interested in ascents at the 'edge' or to the left of the 'edge'.

We first assume that the edge contains our 'ascent' and look for the next higher number to the right of the edge.

Table 7: P(5) → P(5, 3), showing the edge about to be swapped with the next higher number

P_i	a_0	a_1	a_2	a_3	a_4
0	0	1	2	3	4
	0	1	3	2	4
	0	1	4	2	3
	0	2	1	3	4
	0	2	3	1	4
	0	2	4	1	3
⋮	0	3	1	2	4
	0	3	2	1	4
	0	3	4	1	2
	0	4	1	2	3
	0	4	2	1	3
	0	4	3	1	2
			⋮		

If all numbers to the right of the edge are smaller, then we know that the ascent must be to the left of the edge.

However, unlike with regular P(n) generation, at this point all numbers to the right of the edge will be in ascending order. We know this since we began with all numbers in ascending order, and all our operations at this point simply swap the value at the edge with the next higher number.

Fortunately, by simply reversing everything to the right of the edge we return to a state similar to P(n) where all values to the right of the actual ascent are in descending order. We can now proceed as with the regular SEPA.

Table 8: P(5, 3), showing the edge about to be swapped, or, after reversal, the actual ascent and values to swap (as in SEPA)

P_i	a_0	a_1	a_2	a_3	a_4
0	0	1	2	3	4
	0	1	3	2	4
	0	1	4	3	2
	0	2	1	3	4
	0	2	3	1	4
	0	2	4	3	1
⋮	0	3	1	2	4
	0	3	2	1	4
	0	3	4	2	1
	0	4	1	2	3
	0	4	2	1	3
	0	4	3	2	1
			⋮		

A Simple, Efficient Algorithm for Generating P(n, k)

This leads us to the adapted algorithm for generating permutations of *n* items taken *k* at a time, in lexicographic order.

```
def a = { 0, 1, 2 … n - 1 }
def edge = k - 1

// find j in (k…n-1) where a_j > a_edge
j = k
while j < n and a_edge >= a_j,
  ++j

if j < n {
  swap a_edge, a_j
} else {
  reverse a_k to a_n-1

  // find rightmost ascent to left of edge
  i = edge - 1
  while i > 0 and a_i >= a_i+1,
    --i

  if i < 0,
    // no more permutations
    return 0

  // find j in (n-1…i+1) where a_j > a_i
  j = n - 1
  while j > i and a_j < a_i
    --j

  swap a_i, a_j
  reverse a_i+1 to a_n-1
}

output a_0, a_1 … a_k-1
```

SEP(n,k) Algorithm Illustrated

To show how the algorithm works, let's step through the major operations showing the state of the array *a* at each step.

Table 9: P(5, 3). a_2 is the 'edge'. Red values are about to be swapped, yellow cells are about to be reversed.

P_i	a_0	a_1	a_2	a_3	a_4	*Step*
0	0	1	2	3	4	swap a_2, a_3
1	0	1	3	2	4	swap a_2, a_4
	0	1	4	2	3	reverse $a_{3..4}$
2	0	1	4	3	2	swap a_1, a_4
	0	2	4	3	1	reverse $a_{2..4}$
3	0	2	1	3	4	swap a_2, a_3
4	0	2	3	1	4	swap a_2, a_4
	0	2	4	1	3	reverse $a_{3..4}$
5	0	2	4	3	1	swap a_1, a_3
	0	3	4	2	1	reverse $a_{2..4}$
6	0	3	1	2	4	swap a_2, a_3
7	0	3	2	1	4	swap a_2, a_4
	0	3	4	1	2	reverse $a_{3..4}$
8	0	3	4	2	1	swap a_1, a_2
	0	4	3	2	1	reverse $a_{2..4}$
9	0	4	1	2	3	swap a_2, a_3
10	0	4	2	1	3	swap a_2, a_4
	0	4	3	1	2	reverse $a_{3..4}$
11	0	4	3	2	1	swap a_0, a_4
	1	4	3	2	0	reverse $a_{1..4}$
12	1	0	2	3	4	swap a_2, a_3
⋮			⋮			