Selecting, Ranking, and Unranking Permutations

August 28, 2001

1 Introduction

In the lecture on permutations we developed a recurrence from which we concluded that in generating n-permutations lexicographically we perform roughly 1.54308 swaps per permutation. In the example below we verify this explicitly for 5-permutations. Each of the 120 5-permutations is generated and the number of swaps needed to transform each into its successor is calculated. The mean of these 120 numbers is reported.

```
In[1]:= Mean[Map[(i=4; While[#[[i]] > #[[i+1]],i--]; 1+Floor[(5-i)/2])&,
                  Permutations[5]
              ]
        ] //N

Out[1]= 1.54167
```

Any attempt to verify this explicitly for n-permutations for much larger n, say $n = 100$, would be futile because the number of permutations is so large. However, random sampling provides a way out. In the example below, 1000 random 100-permutations are generated. For each, the number of swaps needed to transform it into its successor is calculated, and the mean of these is reported. So instead of performing the calculation for 100! permutations, we do it for a random sample of 1000 permutations. The result varies from run to run but stays fairly close to 1.54308.

```
In[2]:= Mean[Table[p= RandomPermutation[100];i = 99;
                   While[p[[i]] > p[[i+1]],i--];
                   1+Floor[(100-i)/2],
                   {1000}
              ]
        ] //N

Out[2]= 1.537
```

The success of the above experiment depends on being able to select a random permutation from among 100! permutations uniformly. In other words, the experiment works because each permutation is selected with probability 1/100!. Random sampling is, in general, of crucial importance in performing experiments on many of the large combinatorial families we will encounter. In the next section we study how to select a permutation uniformly, at random.

2 Random Permutations

Selecting a permutation at random turns out be a fairly simple task. However, for several of the combinatorial objects we will encounter later, selecting uniformly at random will turn out to be a somewhat more difficult and subtle. The fastest algorithm for random permutations starts with an arbitrary n-permutation and exchanges the ith element with a randomly selected one from the first i elements, for each i from n to 1. The nth element is therefore equally likely to be anything from 1 to n, The $(n-1)$th element is equally likely to be any element not already in the the nth slot. It follows by induction that any permutation is equally likely and is produced with probability $1/n!$. Since the algorithm consists of $n-1$ iterations, each iteration taking $\Theta(1)$ time, this is an $\Theta(n)$ algorithm and is thus optimal. The code for RandomPermutation is given below.

```
RandomPermutation[n_Integer] :=
        Module[{p = Range[n],i,x},
               Do [x = Random[Integer,{1,i}];
                   {p[[i]], p[[x]]}  = {p[[x]], p[[i]]},
                   {i,n,2,-1}
               ];
               p
        ]
```

In the example below, 300 3-permutations are generated and we observe that each of the six 3-permutations are selected roughly the same number of times. This is evidence that RandomPermutation indeed picks permutation uniformly at random.

```
In[3]:= Distribution[Table[RandomPermutation[3],{300}]]

Out[3]= {58, 32, 55, 48, 57, 50}
```

3 Ranking Permutations

Given an ordering on the set of all n-permutations, the *rank* of a permutation denotes the position of that permutation in the ordered list of all n-permutations. It seems traditional to start ranking at 0 and so n-permutations get assigned distinct ranks in the range 0 through $n!-1$. *Combinatorica* provides a function, RankPermutation that computes the rank of a given permutation in lexicographic order. *Combinatorica* also provides a function, UnrankPermutation that is the inverse of RankPermutation. In other words, UnrankPermutation takes an integer in the range 0 through $n!-1$ and returns the permutation with that rank in lexicographic order. In the example below the rank of a 10-permutation is computed. Then UnrankPermutation is applied to the obtained rank and the original permutation reappears.

```
In[4]:= RankPermutation[{9, 1, 8, 10, 2, 3, 6, 5, 4, 7}]

Out[4]= 2937614

In[5]:= UnrankPermutation[%, 10]

Out[5]= {9, 1, 8, 10, 2, 3, 6, 5, 4, 7}
```

A brute force method to compute the rank of an n-permutation p is: first generate the list of n-permutations in lexicographic order and then find the position of p in this list. This is of course misses the point — we want to be able to rank and unrank in time proportional to the size of each individual combinatorial object, not in time proportional to the size of the entire family. Specifically, we would like to rank and unrank an n-permutation in time close to $\Theta(n)$ opposed to $\Theta(n!)$.

Suppose that $p = (p_1, p_2, \ldots, p_n)$ is an n-permutation whose rank we wish to compute. Observe that in lexicographic order, we first have a block of $(n-1)!$ permutations that start with 1 followed by a block of $(n-1)!$ permutations that start with 2, and so on. The block of permutations that start with p_1 are ranked from $(p_1 - 1)(n-1)!$ through $p_1(n-1)! - 1$. So we know that p has rank in this range. The exact location of p in this block depends on the rest of the elements of p. More precisely, let p' be the $(n-1)$-permutation obtained from p by deleting p_1 and decrementing all elements larger than p_1. Then, where the rank of p lies in the range of ranks $[(p_1 - 1)(n-1)!..p_1(n-1)! - 1]$ is completely determined by p'. This is more precisely expressed recursively as

$$\mathrm{Rank}(p) = (p_1 - 1)(n-1)! + \mathrm{Rank}(p'). \tag{1}$$

We can therefore recurse on p' to compute the rank of p.

For example, $p = (2, 3, 1, 5, 4)$ has rank between $1 \cdot 4! = 24$ and $2 \cdot 4! - 1 = 47$ by virtue of its first element. Then 2 is deleted, the rest of the elements are adjusted from $(3, 1, 5, 4)$ to $(2, 1, 4, 3)$, and rank of $(2, 1, 4, 3)$ is obtained and this is added to 24 to obtain the rank of p. The rank of p, by the way, is 31.

The running time of this algorithm is $\Theta(n^2)$. The is because after each first element is deleted, the rest of the elements have to be checked for possible adjustment. This takes a total of $\Theta(n)$ time per element deleted. The algorithm terminates when all n elements are deleted and therefore the total running time of the algorithm is $\Theta(n^2)$. The code for `RankPermutation` is given below. The code is a faithful implementation of the recurrence given in Equation 1 and the algorithm described above.

```
RankPermutation[{1}] := 0
RankPermutation[{}] := 0

RankPermutation[p_?PermutationQ] := (p[[1]]-1) (Length[Rest[p]]!) +
        RankPermutation[ Map[(If[#>p[[1]], #-1, #])&, Rest[p]] ]
```

Inversion Vectors. Can we devise a more efficient algorithm for `RankPermutation`? It turns out that we can devise a $\Theta(n \log n)$ algorithm, using the notion of *inversion vectors*.

A pair of elements p_i and p_j represent an *inversion* in a permutation p if $i > j$ and $p_i < p_j$. Inversions are pairs which are out of order, and so they play a prominent role in the analysis of sorting algorithms. For any integer i, $1 \leq i \leq n-1$, the ith element of the *inversion vector* v of an n-permutation p is the number of elements in p greater than i to the left of i.

Combinatorica provides a function called `ToInversionVector` that computes the inversion vector of a given permutation. In the example below, the first element of the inversion vector of $(5, 9, 1, 8, 2, 6, 4, 7, 3)$ is 2 because there are two elements, namely 5 and 9, that are larger than 1 and appear to its left.

```
In[6]:= ToInversionVector[{5,9,1,8,2,6,4,7,3}]

Out[6]= {2, 3, 6, 4, 0, 2, 2, 1}
```

The inversion vector contains only $n-1$ elements since there are 0 inversions of the form (i, n), implying that the nth element of an inversion vector is always 0 and hence need not be explicitly specified. The ith element can range from 0 to $n - i$, so there are indeed $n!$ distinct inversion vectors, one for each permutation.

Now how are inversion vectors related to computing the rank of a permutation? Reconsider the $\Theta(n^2)$ algorithm described above for computing the rank of a permutation $p = (p_1, p_2, \ldots, p_n)$.

After we have noted the contribution of p_1 to the rank, we delete it from p and decrement all other elements in p larger than p_1. What if we did not decrement the other elements in each step? When we get to p_i, in order to compute its contribution to the rank of p, we need to determine how many times it would have been decremented, had we been diligently decrementing after each deletion. The number of times p_i would have been decremented is exactly equal to the number of elements in p smaller than p_i, that occur to its left. Let $\ell(i)$ denote the number of elements smaller than i that occur to its left in p. Then, had we decremented p_i, as in the original version of the algorithm, its value would be $p_i - \ell(p_i)$ and its contribution to the rank of p would be $(p_i - \ell(p_i) - 1)(n-i)!$. The rank of p is therefore

$$\sum_{i=1}^{n}(p_i - \ell(p_i) - 1)(n-i)!.$$

Now note that if v is the inversion vector of p, $(i-1) - v[p_i] = \ell(p_i)$. Therefore, $(p_i - \ell(p_i) - 1) = (p_i + v[p_i] - i)$ and the above sum can be rewritten as

$$\sum_{i=1}^{n}(p_i + v[p_i] - i)(n-i)!.$$

This expresses the rank of p in terms of elements in p and the inversion vector of p. Given p and its inversion vector it takes $\Theta(n)$ time to compute the above sum. A *Mathematica* implementation of this algorithm is given below.

```
NewRankPermutation[p_?PermutationQ] :=
        Module[{n =  Length[p], v = Append[ToInversionVector[p], 0]},
               Sum[ (p[[i]] - i + v[[ p[[i]] ]] ) (n-i)!, {i, n-1}]
        ]
```

Just as a sanity check, in the following example we compare the answers produced by `RankPermutation` and `NewRankPermutation` on 10 randomly chosen 20-permutations.

```
In[7]:= Table[p = RandomPermutation[20];
               RankPermutation[p] == NewRankPermutation[p],
               {10}
         ]

Out[7]= {True, True, True, True, True, True, True, True, True, True}
```

The time to compute the rank of an n-permutation p according to the new algorithm is $\Theta(T(n) + n)$ where $T(n)$ is the time required to compute the inversion vector of an n-permutation. The algorithm implemented in *Combinatorica* to compute the inversion vector of a permutation $p = (p_1, p_2, \ldots, p_n)$ is the simple $\Theta(n^2)$ algorithm in which for each p_i, we scan the $(i-1)$ elements prior to p_i and count the number of elements larger than i. The code for `ToInversionVector` is given below. Here we first compute the inverse $q = (q_1, q_2, \ldots, q_n)$ of p because for each $i \in [n]$, q_i gives the position of i in p. So the ith element of the inversion vector of p is computed by taking the length q_i prefix of p and finding the number of elements in this prefix that are larger than p_i.

```
ToInversionVector[p_?PermutationQ] :=
        Module[{i,inverse=InversePermutation[p]},
```

```
            Table[
                    Length[ Select[Take[p,inverse[[i]]], (# > i)&] ],
                    {i,Length[p]-1}
            ]
    ] /; (Length[p] > 0)
```

Computing the rank of an n-permutation using `ToInversionVector` given above still results in a $\Theta(n^2)$ algorithm.

We now describe a $\Theta(n \log n)$ algorithm to compute the inversion vector of an n-permutation. The algorithm is a simple modification of *merge sort*. Suppose that we run the merge sort algorithm with an n-permutation p as input. The algorithm sorts p^L, the first half and p^R, the second half of p separately, by calling itself recursively and then merges the two sorted halves. So the merge step is where all the work takes place. Now suppose that the merge step takes as input, not just two sorted halves, but two corresponding inversion vectors as well. In particular, assume that v^L and v^R are inversion vectors corresponding to p^L and p^R. More precisely, suppose that v^L and v^R are length $n-1$ arrays such that the ith element of v^L is 0 if i is not in p^L; otherwise the ith element of v^L is the number of elements in p^L larger than i that occur to its left. v^R is defined similarly. This means that $v^L + v^R$ is the inversion vector of p, except that inversions due to elements in p^L larger than elements in p^R are not recorded in $v^L + v^R$. The merge step can remedy this as follows. For each pair (i, j), $i \in p^L$, $j \in p^R$, examined by the merge step, if $i > j$ then we need to add x to the jth element in $v^L + v^R$, where x is the number of elements in p^L including and after i. The justification for this is that i and every element following i in p^L is an inversion with respect to j and this needs to be noted in the jth element of the inversion vector.

For example, suppose that $p = (8, 7, 2, 1, 9, 4, 6, 5, 10, 3)$. Then

$$p^L = (8, 7, 2, 1, 9), p^R = (4, 6, 5, 10, 3), v^L = (3, 2, 0, 0, 0, 0, 1, 0, 0), \text{ and } v^R = (0, 0, 4, 0, 1, 0, 0, 0, 0).$$

The sequences $(1, 2, 7, 8, 9)$ and $(3, 4, 5, 6, 10)$ are sorted versions of p^L and p^R respectively and these are passed as input into the merge step along with v^L and v^R. The merge step examines the pairs

$$(1, 3), (2, 3), (7, 3), (7, 4), (7, 5), (7, 6), (7, 10), (8, 10), (9, 10)$$

in that order. The pair $(7, 3)$ represents an inversion and in fact this points to the existence of additional inversions $(8, 3)$ and $(9, 3)$. As a result of this discovery 3 is added to the 3rd element in $v^L + v^R$. Similarly, examination of the pairs $(7, 4)$, $(7, 5)$, and $(7, 6)$ causes 3 to be added to the 4th, 5th, and 6th elements of $v^L + v^R$. As a result of these updates $v^L + v^R$ ends up being:

$$(3, 2, 7, 3, 4, 3, 1, 0, 0).$$

This is the inversion vector of p.

This completes the description of a $\Theta(n \log n)$ algorithm for computing the rank of a permutation. Before we move on to the problem of unranking a permutation, let us linger a little on inversion vectors. We introduced these in the context of computing the rank of a permutation, but they have other uses as well. First of all, no two permutations have the same inversion vector and therefore there is a bijection between the set of n-permutations and their inversion vectors. It is not hard to compute the corresponding permutation given an inversion vector (you should think about how to do this!) and *Combinatorica* has a function called `FromInversionVector` to do this.

```
In[8]:= ToInversionVector[{8, 7, 2, 1, 9, 4, 6, 5, 10, 3}]

Out[8]= {3, 2, 7, 3, 4, 3, 1, 0, 0}
```

```
In[9]:= FromInversionVector[%]

Out[9]= {8, 7, 2, 1, 9, 4, 6, 5, 10, 3}
```

Thus inversion vectors form an alternate representation of permutations and this comes in handy sometimes.

Inversion vectors also provide a measure of the "unsortedness" of permutations. The total number of inversions of a permutation is simply the sum of the entries in the inversion vector. Most sorting algorithms sort by swaps and these swaps can be thought of as ways of decreasing the total number of inversions in the permutation. Algorithms such as *bubble sort* that perform adjacent swaps have to swap exactly as many times as the number of inversions because an adjacent swap decreases the number of inversions by exactly 1. In certain applications, it is useful to enumerate or randomly select "almost sorted" permutations. This motivates the following problems.

(i) For any positive integer n and integer k, $0 \leq k \leq \binom{n}{2}$, generate all permutations with exactly k inversions.

(ii) For any positive integer n and integer k, $0 \leq k \leq \binom{n}{2}$, pick uniformly at random a permutation with exactly k inversions.

Along with these problems, you should also ponder the question of how many n-permutations there are with exactly k inversions. These and other issues pertaining to inversion vectors will be explored in homework problems.

4 Unranking Permutations

As explained in the previous section, the rank of a permutation $p = (p_1, p_2, \ldots, p_n)$ is

$$\sum_{i=1}^{n} (p_i - \ell(p_i) - 1)(n-i)!,$$

where $\ell(p_i)$ is the number of elements in p smaller that p_i occurring to its left. For $j = 1, 2, \ldots, n$ let S_j denote the partial sum

$$S_j = \sum_{i=j}^{n} (p_i - \ell(p_i) - 1)(n-i)!.$$

So $S_1 = (p_1 - \ell(p_1) - 1)(n-1)! + S_2$. As we observed earlier S_2 is the rank of an $(n-1)$-permutation and is therefore in the range 0 through $n! - 1$. Therefore, dividing S_1 by $(n-1)!$ yields a quotient $(p_1 - \ell(p_1) - 1)$ and remainder S_2. Repeatedly dividing the partial sums in this manner, we obtain values $c_1, c_2, \ldots, c_n$ where $c_i = (p_i - \ell(p_i) - 1)$ for each $i = 1, 2, \ldots, n$. It is now simply a matter of obtaining the p_i's from the c_i's. Noting that $\ell(p_1) = 0$ we immediately get $p_1 = c_1 + 1$. In general, having computed the values of $p_1, p_2, \ldots, p_{i-1}$, how do we compute p_i? Let $R_i = [n] - \{p_1, p_2, \ldots, p_{i-1}\}$. We note that there is exactly one element $j \in R_i$ for which $j - \ell(j) = c_i + 1$. We pick this element j as p_i and we are done.

One way to quickly pick the appropriate j from R_i is as follows. Start with R_0 as the sequence $(1, 2, \ldots, n)$ and as elements $p_1, p_2, \ldots$ are computed, delete them to obtain $R_1, R_2, \ldots$. An element $j \in R_i$ starts in the jth position in R_0 and moves to the left as many times as there are elements smaller than it in $\{p_1, p_2, \ldots, p_{i-1}\}$. Thus the position of j in R_i is j minus the number of elements smaller than it in $\{p_1, p_2, \ldots, p_{i-1}\}$. Therefore, the element $j \in R_i$ with $j - \ell(j) = c_i + 1$ is simply the element in position $(c_i + 1)$.

For example, what is the millionth 10-permutation in lexicographic order? Dividing 999999 by 9! yields quotient 2 and remainder 274239. Thus $c_1 = 2$, $p_1 = 3$ and $R_1 = (1, 2, 4, 5, 6, 7, 8, 9, 10)$. Dividing 274239 by 8! yields quotient 6 and remainder 32319. So $c_2 = 6$, $p_2 = 8$ and $R_2 = (1, 3, 4, 5, 6, 8, 9)$. This process continues until we have computed all 10 entries of the permutation.

```
In[10]:= UnrankPermutation[999999, 10]

Out[10]= {3, 8, 9, 4, 10, 2, 6, 5, 7, 1}
```

Given below is the code for `UnrankPermutation`. `UP` is a helper function that computes the values $c_1 + 1, c_2 + 1, \ldots, c_n + 1$. From these values the p_i's are computed in the main body of the function `UnrankPermutation`. Computing the c_i's takes $\Theta(n)$ time. The sequence R_i is maintained in the variable `s`. At each step an element is deleted from `s` and this takes time proportional to the size of `s`. Therefore the running time of the algorithm is $\Theta(n^2)$. You should think about how to improve this to $\Theta(n \log n)$.

```
UP[r_Integer, n_Integer] :=
        Module[{r1 = r, q = n!},
               Table[r1 = Mod[r1, q]; q = q/(n - i + 1); Quotient[r1, q] + 1,
                     {i, n}
               ]
        ]

UnrankPermutation[r_Integer, l_List] :=
        Module[{s = l, k, t, p = UP[Mod[r, Length[l]!], Length[l]]},
               Table[k = s[[t = p[[i]] ]]; s = Delete[s, t]; k,
                     {i, Length[ p ]}
               ]
        ]
```
