Permutations

August 24, 2001

1 Introduction

An *n-permutation* is a bijection from $[n]$ to $[n]$[1]. When n is unimportant or is obvious from the context, we drop the n and simply use "permutation." The two most common ways of writing a permutation p are (i) as a sequence $(p_1, p_2, \ldots, p_n)$, where $p(i) = p_i$ and (ii) as a $2 \times n$ matrix

$$\begin{pmatrix} i_1 & i_2 & i_3 & \ldots & i_n \\ p_1 & p_2 & p_3 & \ldots & p_n \end{pmatrix}$$

where $p(i_j) = p_j$ for $j = 1, 2, \ldots, n$. This form is usually referred to as the *two-line notation.* For example,

$$p = \begin{pmatrix} 3 & 4 & 1 & 2 & 5 \\ 5 & 1 & 3 & 4 & 2 \end{pmatrix}$$

is the permutation $p(1) = 3$, $p(2) = 4$, $p(3) = 5$, $p(4) = 1$, and $p(5) = 2$. Note that changing the order of columns in the two-line notation does not change the permutation.

Multiplication. Permutation *multiplication* is a binary operation, denoted by $\times$, and defined as $p \times q = p \circ q$, where $\circ$ is the standard function composition. In other words, $p \times q$ is a function defined as

$$(p \times q)(i) = p(q(i)).$$

The composition of two bijections whose domains and ranges are all identical is also a bijection (verify this!). So $p \times q$ is also a bijection from $[n]$ to $[n]$ and hence the set of all $[n]$ permutations is closed under permutation multiplication. For example, suppose that $p = (3, 5, 1, 6, 2, 4)$ and $q = (2, 3, 4, 6, 5, 1)$. Then $p \times q = (5, 1, 6, 4, 2, 3)$. To verify this, you need to check that for each $i = 1, 2, \ldots, 6$, the resident of the ith slot in $p \times q$ is $p(q(i))$. Note that q is applied first followed by p. The *Combinatorica* function `Permute` is the permutation multiplication operation.

```
In[1]:= Permute[{3, 5, 1, 6, 2, 4}, {2, 3, 4, 6, 5, 1}]

Out[1]= {5, 1, 6, 4, 2, 3}
```

It is easy to see that permutation multiplication is associative, but not commutative (try to prove the former and produce a counterexample to show the latter!).

Identity. The *identity* permutation is $(1, 2, \ldots, n)$. We denote this by I_n or simply I, if n is not relevant in the context. For any permutation p, $p \times I = I \times p = p$.

[1] $[n]$ denotes the set $\{1, 2, \ldots, n\}$

Inverse. Every permutation p has an *inverse*, denoted p^{-1}, such that $p \times p^{-1} = p^{-1} \times p = I$. It is easy to see that $p(i) = j$ if and only if $p^{-1}(j) = i$. The *Combinatorica* function `InversePermutation` computes the inverse of a permutation.

```
In[2]:= p = InversePermutation[q = {5, 1, 6, 4, 2, 3}]

Out[2]= {2, 5, 6, 4, 1, 3}

In[3]:= Permute[p, q]

Out[3]= {1, 2, 3, 4, 5, 6}
```

2 Generating Permutations Lexicographically

The simplest generation problem we will encounter in this course is the problem of generating permutations. These can be generated in a variety of orders, but the most natural order seems to be the *lexicographic order*. *Mathematica* has a built-in function called `Permutations` that generates permutations in lexicographic order.

```
In[4]:= Permutations[3]

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

Combinatorica has a function called `LexicographicPermutations` that performs the same task.

```
In[5]:= LexicographicPermutations[3]

Out[5]= {{1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}}
```

Many combinatorial objects, including permutations, have a pleasing recursive structure that turns out to be very useful in counting or enumerating them. We use the recursive structure or permutations to devise an algorithm to enumerate them lexicographically. Start by pulling out each element of $[n]$ and then prepend the element pulled out to all the permutations of the remaining $n-1$ elements. The pseudocode for the algorithm is as follows:

```
GenPerms([n]){
      bigList ← {};
      for i ← 1 to n do{
            list ← GenPerms([n] - {i});
            for each permutation p in list do
                  Prepend i to p;
            Append list to bigList;
      }
      return bigList;
}
```

You should try implementing this in *Mathematica* and compare its performance to that of `Permutations` and `LexicographicPermutations`.

More insight is gained into lexicographically ordered permutations by attempting to implement the above algorithm non-recursively. To do this we need to answer the question: how do we transform a permutation into its *lexicographic successor*? Let p be an n-permutation whose largest decreasing suffix is $p[i+1..n]$. Note that this suffix can be as small as a size-1 sequence or as big as a size-n sequence. If $p[i+1..n]$ has length n, it means that $p = (n, n-1, \ldots, 1)$ and this permutation has (strictly speaking) no lexicographic successor. So we assume that $p[i+1..n]$ has length at most $n-1$. Now note that $p[i] < p[i+1]$ because otherwise we would have a larger decreasing suffix. This implies that there is an integer j, $i+1 \le j \le n$ such that $p[j]$ is the smallest element in $p[i+1..n]$ that is larger than $p[i]$. To get the lexicographic successor of p, exchange $p[i]$ and $p[j]$ and then reverse $p[i+1..n]$. *Combinatorica* has a function called `NextPermutation` that computes the lexicographic successor of a given permutation.

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

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

In this example, $(8,7,6,1)$ is the largest decreasing suffix, 5 is the element just before this, and 6 is the smallest element in the suffix that is larger than 5. The function `NextPermutation` first exchanges 5 and 6 and then reverses the suffix $(8,7,5,1)$. The *Mathematica* code for `NextPermutation` is given below.

```
NextPermutation[l_List] :=
        Module[{n = Length[l], i, j, t, p = l},
                i = n-1; While[ p[[i]] > p[[i+1]], i--];
                j = n; While[ p[[j]] < p[[i]], j--];
                {p[[i]], p[[j]]} = {p[[j]], p[[i]]};
                Join[ Take[p, i], Reverse[Drop[p, i]] ]
        ]
```

In the first line after `Module`, the largest decreasing suffix of the given permutation is found. In the next line, $p[j]$, the smallest element in the suffix larger than $p[i]$ is found. In the next line, $p[i]$ and $p[j]$ are exchanged. In the last line, the suffix is reversed.

`LexicographicPermutations` works by simply calling `NextPermutation` repeatedly.

3 Minimum Change Permutations

The work we do in generating permutations lexicographically can be measured by the total number of exchanges or swaps the algorithm performs. This is simply the sum of the number of swaps needed in going from a permutation to its successor, summed over all permutations. The successor of $(1,2,\ldots,n-1,n)$ is $(1,2,\ldots,n,n-1)$ and so one swap is what we need in this case. The successor of $(1,n,n-1,\ldots,2)$ is $(2,1,3,\ldots,n-1,n)$. This transformation requires one swap to exchange 1 and 2 and $\lfloor (n-1)/2 \rfloor$ swaps to reverse $(n,n-1,\ldots,3,1)$ for a total of $\lfloor (n+1)/2 \rfloor$ swaps. These two cases illustrate the cases that need the fewest and the most swaps respectively. However, this exercise, by itself, gives us rather trivial bounds on the sum: a lower bound of $\Omega(n!)$ and an upper bound of $O(n \cdot n!)$.

Tight Bounds. To get tight bounds on this quantity, let us first give it a name. Let I_n denote the total number of swaps needed to generate the the lexicographic sequence of n-permutations. The lexicographic sequence of n-permutations can be partitioned into blocks — the first block containing permutations that start with 1, the second block containing permutations that start with 2, and so on. So there are n blocks and we can view I_n as the number of swaps that are performed within each block plus the number of swaps that are performed at the boundary of consecutive blocks. Within each block the first element is fixed and deleting this element from all permutations in a block gives us a lexicographic sequence of permutations of the remaining $n-1$ elements. Since the first element is fixed, the total number of swaps that occur within a block is I_{n-1}. Thus the total number of swaps that occur inside all the blocks is nI_{n-1}. Now let us see how we go from the last permutation in block-i to the first permutation in block-$(i+1)$. The last permutation in block-i contains i in the first place followed by the remaining elements in decreasing order. As illustrated by the worst case example above, it takes $\lfloor (n+1)/2 \rfloor$ swaps to compute the lexicographic successor of this permutation. This reasoning leads to the following recurrence relation for I_n, for any integer $n > 1$:

$$I_n = nI_{n-1} + (n-1)\left\lfloor \frac{n+1}{2} \right\rfloor$$

Obviously, $I_1 = 0$. Here is a simple *Mathematica* function that computes I_n.

```
Swaps[1] := 0
Swaps[n_Integer?Positive] := n Swaps[n - 1] + (n - 1) Floor[ (n + 1)/2]
```

As expected I_n grows quite rapidly. Here is a table that shows the first 10 values of I_n.

```
In[7]= Table[Swaps[i], {i, 10}]

Out[7]= {0, 1, 7, 34, 182, 1107, 7773, 62212, 559948, 5599525}
```

However, the following table which shows the first 20 values of $I_n/n!$ is more interesting.

```
In[8]= Table[Swaps[i]/i! // N, {i, 20}]

Out[8]= {0., 0.5, 1.16667, 1.41667, 1.51667, 1.5375, 1.54226, 1.54296, 1.54307,
         1.54308, 1.54308, 1.54308, 1.54308, 1.54308, 1.54308, 1.54308,
         1.54308, 1.54308, 1.54308, 1.54308}
```

The above experiment leads to the conjecture that as $n \to \infty$, the ratio $I_n/n! \to 1.54308$. In other words, for large n, we perform about 1.5 swaps per permutation, in generating all n-permutations in lexicographic order. We will figure out how to solve recurrence relations such as the one above when we get to *generating functions*. We will be able to show that

$$\lim_{n\to\infty} \frac{I_n}{n!} = \cosh(1) \approx 1.5438.$$

The above exercise raises the question: can we generate permutations in such an order that we perform exactly one swap in going from a permutation to its successor. In other words, can be generate permutations in *minimum change order*?

Hamiltonian Paths. It is very useful to think about this problem from a completely different point of view. Let $P_n = (V_n, E_n)$ be a directed graph with vertex set V_n being the set of all n-permutations and edge set E_n defined as

$$E_n = \{(p, q) \mid q \text{ is obtained from } p \text{ by 1 swap}\}.$$

You should verify that this graph has $n!$ vertices and $n! \cdot n(n-1)/2$ edges. If q is obtained from p by a swap then p can be obtained from q by a swap as well and this implies that for any pair of permutations p and q, $(p,q) \in E_n$ if and only if $(q,p) \in E_n$. So we can equivalently view P_n as an undirected graph with $n!$ vertices and $n! \cdot n(n-1)/4$ edges. A *Hamiltonian path* in a graph is a simple path that visits every vertex in the graph. A *Hamiltonian cycle* in a graph is a simple cycle that visits every vertex in the graph. A Hamiltonian path in P_n corresponds to minimum change ordering of the set of all n-permutations. The question "Can permutations be listed in minimum change order?" is therefore equivalent to the question "Does P_n have a Hamiltonian path?" Figure 3 shows P_3 along with a Hamiltonian path (highlighted by the thick gray line segments). In fact the first and the last vertices of the path are adjacent and so this is a Hamiltonian cycle. At first glance this approach seems rather unprofitable because the problem of finding a Hamiltonian path (or a Hamiltonian cycle) in a graph belongs to the class of NP-complete problems and is therefore considered computationally intractable. But, examples we will study later will reveal that the Hamiltonian path point of view helps in two ways:

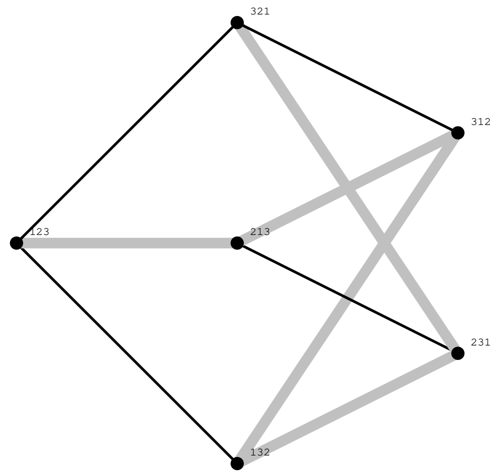

Figure 1: A Hamiltonian path in P_3

(i) While there are no computationally tractable necessary *and* sufficient conditions for a graph to contain a Hamiltonian path, there are various simple sufficient conditions (that are not necessary). Occasionally, we will be able to show that a sufficient condition holds and the graph is Hamiltonian, thereby proving that a minimum change sequence exists.

(ii) Graphs such as P_n, derived from combinatorial objects display an abundance of symmetry. This symmetry makes long paths likely. Some amount of research has been done in trying to characterize this symmetry and connect this characterization to the problem of testing for Hamiltonian paths. As we will see later, the main feature of this body of work is a conjecture that remains tantalizingly open.

Johnson-Trotter Algorithm. It turns out that it is not too hard to enumerate permutations in minimum change order. In fact, permutations can be enumerated in an order so that each permutation is obtained from the previous permutation by a *adjacent swap*. There is a simple algorithm, called the *Johnson-Trotter* algorithm that does this.

Suppose $p_1, p_2, \ldots, p_{(n-1)!}$ is a sequence of $(n-1)$-permutations listed so that each permutation differs from the previous by an adjacent swap. To extend this ordering to n-permutations we start by appending n to p_1. Then the element n is moved to the left, one adjacent swap at a time. When n reaches the first position we have generated n permutations by performing adjacent swaps. The next permutation is generated by keeping n fixed in the first position and transforming p_1 to p_2 by an adjacent swap. After this n begins its rightward journey, again one adjacent swap at a time. When n reaches the last position, p_2 is changed into p_3. This continues until n has traveled across $p_{(n-1)!}$, by which time all $n!$ permutations have been generated, each from the previous by an adjacent swap. For example, below is the list of 4-permutations listed according to the “Johnson-Trotter order:”

(1, 2, 3, 4)	(1, 2, 4, 3)	(1, 4, 2, 3)	(4, 1, 2, 3)
(4, 1, 3, 2)	(1, 4, 3, 2)	(1, 3, 4, 2)	(1, 3, 2, 4)
(3, 1, 2, 4)	(3, 1, 4, 2)	(3, 4, 1, 2)	(4, 3, 1, 2)
(4, 3, 2, 1)	(3, 4, 2, 1)	(3, 2, 4, 1)	(3, 2, 1, 4)
(2, 3, 1, 4)	(2, 3, 4, 1)	(2, 4, 3, 1)	(4, 2, 3, 1)
(4, 2, 1, 3)	(2, 4, 1, 3)	(2, 1, 4, 3)	(2, 1, 3, 4)