Permutations

The permutation problem is as follows: Given a list of items, list all the possible orderings of those items.

Generically, we list permutations as all the orderings of the integers from 0 to n-1, inclusive. For example, if we wanted to list all the permutations for n = 3, in lexicographical ordering, these would be:

0, 1, 2
0, 2, 1
1, 0, 2
1, 2, 0
2, 0, 1
2, 1, 0.

There are several different permutation algorithms, but since recursion an emphasis of the course, a recursive algorithm to solve this problem will be presented. (Feel free to come up with an iterative algorithm on your own.)

We utilize recursion as follows, using the following parameters to our recursive function:

1) An array with a partially filled in permutation.
2) A used array, storing which items have already been partially filled in
3) An integer, k, representing how many items have already been filled in.

Technically, one could have all of this information with just item number 1, but in C, it makes life easier to pass in items 2 and 3.

The job of our function will be to list all permutations of the given partially filled in permutation that have their first k values fixed.

Thus, for example, if k = 1 and our partially filled in array looked like:

2		

Then the goal of the algorithm would be to print out (or process in some way) the following two permutations:

2, 0, 1
2, 1, 0

in that order.

As a second example, imagine the following partially filled in array with k = 4 (for permutations with n = 7):

3	**6**	**0**	**4**			

The algorithm should print out the following permutations:

3, 6, 0, 4, 1, 2, 5
3, 6, 0, 4, 1, 5, 2
3, 6, 0, 4, 2, 1, 5
3, 6, 0, 4, 2, 5, 1
3, 6, 0, 4, 5, 1, 2
3, 6, 0, 4, 5, 2, 1

Recursively, our code ought to do the following:

1) Check if k is equal to n, the length of our permutation array. If so, just print out the fully filled in permutation.

2) If not, iterate through each un-used item, placing it in slot k (in numerical order), and recursively calling the function, noting that now, k+1 items are fixed.

In code, we have the following (assume that the appropriate functions and variables are declared):

```
void printperms(int* perm, int* used, int k,
int n) {

    if (k == n) print(perm, n);

    int i;
    for (i=0; i<n; i++) {
        if (!used[i]) {
            used[i] = 1;
            perm[k] = i;
            printperms(perm, used, k+1, n);
            used[i] = 0;
        }
    }
}
```

Applying Permutation Algorithm to Objects

Say we wanted to go through all the permutations of some array of objects, call it items. Then we can just do the following:

```
void printperms(int* perm, int* used, type*
items, int k, int n) {

     if (k == n) process(perm, items, n);

     int i;
     for (i=0; i<n; i++) {
          if (!used[i]) {
               used[i] = 1;
               perm[k] = i;
               printperms(perm, used, k+1, n);
               used[i] = 0;
          }
     }
}
```

The process function we do whatever it needs to do with the items in the following order:

```
items[perm[0]],
items[perm[1]],
items[perm[2]], …,
items[perm[n-1]]
```

Iterative Permutation Algorithm - Background

Another algorithm that cycles through permutations goes through each of them in lexicographical ordering. Roughly speaking, lexicographical ordering is the same as alphabetical ordering. To determine which of two permutations should appear first in a lexicographical ordering, start comparing individual items from the left until you hit a difference. The permutation that should come first is the one with the item that comes earlier when comparing the two different items from the two different permutations.

For example, when comparing permutations of ACT, we find that CAT comes before CTA, because A comes before C. For a numerical example, the permutation 4,6,2,8,3,7,5,1 comes before 4,6,2,8,5,1,3,7, since 3 is smaller than 5 at the spot of the first discrepancy.

Given this definition for comparing two permutations of a set of items, a complete natural ordering is imposed on all the permutations on the list. For example, for the letters A, C, and T, the natural ordering of the permutations, using this definition is as follows:

ACT
ATC
CAT
CTA
TAC
TCA

Similarly, the ordering of the permutations of 1, 2, 3 and 4 are as follows:

1234	**2134**	**3124**	**4123**
1243	**2143**	**3142**	**4132**
1324	**2314**	**3214**	**4213**
1342	**2341**	**3241**	**4231**
1423	**2413**	**3412**	**4312**
1432	**2431**	**3421**	**4321**

In order to come up with an algorithm that iterates through all of the permutations of a set of items in this order, we need to have a successor function. Namely, we need a function that advances an array storing one permutation to the following permutation.

Once we write this successor function, we simply need to start with the first permutation (in this case, 1234 or ACT), and call the successor function the correct number of times. Since there are n! (read, “n factorial”) orderings of n items, we must call the successor function n!-1 times.

Note: We can derive the total number of permutations of n distinct objects as follows:

For the first object, we have n choices.

For the second object, we have n-1 choices (all but what we chose for the first object.)

For the third object, we have n-2 choices, etc.

Since each of these choices is independent of the rest, to calculate the number of different permutations, we simply need to multiply each of these numbers:

$n \times (n-1) \times (n-2) \times \ldots \times 1$

This product is so common in mathematics, that it has a special name (factorial) and symbol (!). Symbolically, we have:

$n \times (n-1) \times (n-2) \times \ldots \times 1 = n!$

Next Permutation Function

Let's examine an example of finding the next permutation of

4,6,2,8,3,7,5,1

and utilize it to come up with a general algorithm.

We know that the fewer items we change on the right the better. The reason is that if we change the 4 to a 5, for example, then we are definitely missing other permutations that might start with 4 that come after the current one. In essence, our goal is to maximize the number of items, starting from the right, that stay fixed.

A real quick inspection will reveal that we can keep 4, 6, 2, and 8 fixed.

The reason is that 3, 7, 5, and 1 can be rearranged to form a "higher" permutation.

BUT, notice that 3 CAN NOT be fixed because it is IMPOSSIBLE to rearrange

7, 5, 1

to create a higher permutation. (This is the highest one since all the values are in descending order.)

Thus, the key to our successor algorithm is to determine the first item, from the left, that has to be changed.

Simply put, all of the items after this item have to be arranged in descending order.

So, here is step #1 of the algorithm:

Scan from the right side of the permutation, going backwards, continue scanning until you find the first pair of values in ascending order. The first value in this pair is the item that will be switched out.

Thus, if our input was 4,6,2,8,3,7,5,1,

We note that (5,1) is descending.
We note that (7,5) is descending.
But, we find that (3,7) is ascending.

Now that we've identified this value, our key is to determine WHICH value to switch into its place.

First, we know that all the values to its left will stay fixed, so we are NOT switching with any of these values. (In our example, that means we won't switch 3 with 4, 6, 2 or 8.)

With this analysis, we have determined the following about the next permutation of 4,6,2,8,3,7,5,1:

1) The values, 4, 6, 2, and 8 will be fixed.
2) The value 3 must be changed.
3) It must be changed to 1, 5, or 7

Next, we know we must switch it with a higher value, otherwise our permutation won't be a higher one. This reduces our list of possible values in our example to 7 and 5. (More generally, these are all the items that appear AFTER our designated item and are larger than it.)

Now, of the possibilities left, we MUST switch it with the lowest value left (5,7 in our example). The reason for this is that any permutation that starts with 5 precedes any permutation that starts with 7. In general, we want the lowest permutation possible of the ones left, and we can achieve this by minimizing our next choice.

Thus, we know that our permutation must start:

4, 6, 2, 8, 5

In doing so, we have exchanged the 3 for the 5, so our array currently looks like this:

4, 6, 2, 8, 5, <u>7, 3, 1</u>

Now, we can state step #2 of the algorithm:

Determine the smallest value larger than the value to be exchanged in the permutation that comes AFTER the value to be exchanged, and swap these two values.

In our example, 3 was the value that needed to be exchanged and 5 was the smallest value listed after 3 that was also larger than 3.

Now, that we have made that change, we would like to "minimize" the rest of the permutation. For any permutation, we can minimize it by listing the items in ascending order. Currently, however, our items are listed in descending order:

4, 6, 2, 8, 5, <u>7, 3, 1</u>

In a nutshell, we must simply take all the values that come after our original swapped location in the array, and reverse these values to obtain:

4, 6, 2, 8, 5, 1, 3, 7.

This is the next permutation.

Now, let's look at the steps of the algorithm all together:

Iterative Permutation Algorithm

1. Scan from the right side of the permutation, going backwards, continue scanning until you find the first pair of values in ascending order. The first value in this pair is the item that will be switched out.

2. Scan to the right of the item identified in step 1, looking for the smallest item that is greater than the item identified in step 1. Swap these two items.

3. Reverse the part of the permutation that starts from the original location first identified in the array and ends at the end of the array.

A function that implements this algorithm is located on the next page.

```
void nextPerm(int perm[], int length) {

    // Find the spot that needs to change.
    int i = length-1;
    while (i>0 && perm[i] < perm[i-1]) i--;

    i--; // Advance to swap location.

    // So last perm doesn't cause a problem.
    if (i == -1) return;

    // Find the spot with which to swap.
    int j=length-1;
    while (j>i && perm[j]<perm[i]) j--;

    // Swap it.
    int temp = perm[i];
    perm[i] = perm[j];
    perm[j] = temp;

   // reverse from index i+1 to length-1.
   int k,m;
   for (k=i+1,m=length-1; k<m; k++,m--) {
       temp = perm[k];
       perm[k] = perm[m];
       perm[m] = temp;
    }
}
```