Permutations (and Combinations)

It is sometimes very useful to be able to generate all permutations of some set in a computer program. For example, perhaps we need to do some exhaustive testing. In this exercise, you will write three short programs.

1. Generating the permutations of some set. A set is simply a list (of numbers) where there are no duplicates. A set of n elements should have n! permutations. For example, [7, 9, 8, 6] should have 24 permutations. Here is an algorithm for generating permutations, based on Johnsonbaugh's Discrete Mathematics book. Let's assume that list/array indices begin with zero. You will need to implement a factorial function along the way.

```
s = the given set
n = the number of elements
immediately print s as given:  this is our first permutation
for i = 2 to n! do
   m = n – 2

   // find the first decrease working from the right
   while s[m] > s[m + 1] do
      m = m – 1
   k = n – 1

  // find the rightmost element s[k] with s[m] < s[k].
   while s[m] > s[k] do
      k = k – 1
   swap s[m] with s[k]
   p = m + 1
   q = n – 1

   # swap s[m+1] with s[n–1], s[m+2] with s[n–2], etc.
   while p < q do
      swap s[p] with s[q]
      p = p + 1
      q = q – 1
   print s
```

2. While we are in the neighborhood, let's also consider combinations. Given a set of numbers in the range 1..n, a combination is a subset of these numbers. Let's write out all the combinations of size r. This algorithm is also inspired by Johnsonbaugh. Note that the notation C(n, r) refers to the "combination" or binomial coefficient formula. You will need to implement both a factorial and a combination formula along the way. I recommend that you also keep track of how many combinations your program generates, so that you can verify correctness.

```
n = size of list (1 .. n) from which a subset will be selected
r = desired size of selection

for i = 1 to r do
   L[i] = i

// print the first combination automatically
print L

for i = 2 to C(n, r) do
   m = r – 1
   maxVal = n
   while L[m] == maxVal
      // find the rightmost element not at its max value
      m = m – 1
      maxVal = maxVal – 1

   // increment rightmost element
   L[m] = L[m] + 1

   // other elements are the successors of L[m]
   for j = m + 1 to r – 1 do
      L[j] = L[j – 1] + 1
   print L
```

3. A multiset is a list (of numbers) that allows duplicates. Let's enumerate all of the combinations of size r from a multiset of n items. I don't have an algorithm that works for all possible values of r, but let me illustrate an unsophisticated algorithm that works when r = 6.

```
L = the given multiset
n = size of L
selection = initially empty list
answer = initially empty list

for a = 0 to n – 1 do
   for b = a + 1 to n – 1 do
      for c = b + 1 to n – 1 do
         for d = c + 1 to n – 1 do
            for e = d + 1 to n – 1 do
               for f = e + 1 to n – 1 do
                  append the list/sextuple
                  (L[a],L[b],L[c],L[d],L[e],L[f])
                  to the selection

// definitely put the first selection into our answer
append selection[0] to the answer list

// for all i from 1 to the end, see if selection already matches
// something in our answer
for i = 1 to size of the selection do
   matchFound = false
   for j = 0 to 5 do
      if selection[i][k] == answer[j][k] for all k in 0..5
         matchFound = true
   if matchFound == false
      append selection[i] to our answer list

// Once the i loop is finished, the answer list contains all
unduplicated selections
```

As an illustration, test your program on this input list:
1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 25, 50, 75, 100
It turns out that the selection list will contain 134,596 elements, but the answer list will contain just 13,243 elements.