June, 2003

Advisor Answers

Generating combinations

VFP 8/7/6

Q: How can I find all the groups of ten numbers out of a set of 45? More generally, how can I find all the subsets of a particular size in a set of numbers? For example, if I have these six numbers:

```
5, 6, 7, 8, 9, 10
```

and I want the subsets with three members, the result would be:

```
5, 6, 7
5, 6, 8
5, 6, 9
5, 6, 10
5, 7, 8
5, 7, 9
5, 7, 10
...
```

–Wayne Stephenson (via Advisor Forums)

A: Generating all the combinations of a list is a classic problem in mathematics. It's also a great example of a problem where recursion produces a much simpler solution than other approaches. Let's start by talking about the general solution, then we'll dig into recursion, and finally, I'll show a recursive solution to this problem.

First, it's important to note that we're talking here about combinations, not permutations. Combinations are all the subsets without regard to order; permutations are the subsets where the order of elements varies. So, in the example above, the list of combinations includes (5, 6, 7), but not (6, 5, 7) or (7, 6, 5) or any of the other orderings of those three numbers. The list of permutations would include all of them.

How do you find all the combinations of a set? The list above gives you a clue. In pseudo-code, it looks something like:

```
FOR nI = 1 TO nListSize-nSubsetSize + 1
  * Make the item at position nI the first item
  FOR nJ = nI+1 TO nListSize-nSubsetSize+2
    * Make the item at position nJ the second item
    FOR nK = nJ+1 TO nListSize-nSubsetSize+3
```

```
      * Make the item at position nK the third item
        ...
    ENDFOR
  ENDFOR
ENDFOR
```

If you know the subset size ahead of time, you can write code with nested loops like this. However, if you want to solve the general problem, where you can have a list of any size and subsets of any size, you have to take another approach.

Enter recursion. Recursion is the act of a program calling itself. The most familiar example of recursion is probably the definition of the factorial function, the product of all the numbers from 1 to n, normally expressed with an exclamation point:

```
0! = 1
n! = n * (n-1)! for n>=1
```

You can write code to compute factorials in VFP like this:

```
FUNCTION Factorial( nNumber AS Number )
LOCAL nResult
IF nNumber = 0
   nResult = 1
ELSE
   nResult = nNumber * Factorial(nNumber-1)
ENDIF
RETURN nResult
```

This example points out an important rule for recursive functions—there must be a termination point defined. For factorial, it's passing 0.

How can we solve the combination problem recursively? The idea is to call a function once for each number in the subset; the function chooses one item. The termination point is having the specified number of items.

When writing recursive code, you need to be extra careful about the scope of variables. Be sure to declare any variables used only within the current call as local, so another call doesn't step on their values. Sometimes, you have items you want modified by successive calls. For the combination problem, we track the number of subsets found so far. Make sure to declare those variables private prior to calling the recursive routine.

It's not unusual to have two functions involved in a recursive situation. The first sets things up, including declaring any private variables, and makes the initial call to the actual recursive routine. That's the case for solving this problem. The first function creates a cursor to hold the results, initializes a private counter variable and then calls the recursive function.

Rather than worrying about the actual items that need to go into subsets, these functions assume that they're stored in an array or cursor where each item can be accessed by its position. So, each subset in the resulting cursor is the list of positions where the appropriate items can be found.

Here's the code:

```
LPARAMETER nCount, nSetSize
* Parameter checking omitted
PRIVATE nRecCount
* Create a cursor to hold results
cFieldList = ""
FOR nItem = 1 TO nSetSize
   cFieldList = cFieldList + "Item" + ;
                  TRANSFORM(nItem) + " N(4),"
ENDFOR
cFieldList = LEFT(cFieldList, LEN(cFieldList)-1)
CREATE CURSOR Results (&cFieldList)
nRecCount = 0
GetNextValue(nCount, nSetSize, 1, 1)
RETURN nRecCount
PROCEDURE GetNextValue
LPARAMETERS nCount, nSetSize, nStartPos, nResultPos
* Parameter checking omitted
LOCAL nPos, nOldRecCount
FOR nPos = nStartPos TO nCount-nSetSize+nResultPos
   nOldRecCount = nRecCount

   * Drill down first
   IF nResultPos<nSetSize
      GetNextValue(nCount,nSetSize,nPos+1,nResultPos+1)
   ENDIF

   * Save results
   IF nResultPos=nSetSize
      nRecCount = nRecCount + 1
      APPEND BLANK IN Results
   ENDIF
   GO nOldRecCount + 1
   SCAN REST
      REPLACE ("Item"+TRANSFORM(nResultPos)) ;
              WITH nPos IN Results
   ENDSCAN
ENDFOR
RETURN
```

The trickiest part of this particular problem turns out to be putting the values into the result. Having determined the value for a particular position, it may apply to a whole series of records in the result. But at the time we figure it out, we don't know how many. So we wait to save the value until we've drilled all the way down and come back. By that point, all the appropriate records have been added. The nOldRecCount variable keeps track of the record pointer position when we enter the routine each time. (Note that it's local so that each call to the function has its own variable.) When we return from the recursive call, we need to put the value for this position into the appropriate field of every record added after that.

Let's look at a simple example. Suppose we have four items and want all combinations with two items. The first time GetNextValue is called, it receives the following parameters:

```
4, 2, 1, 1
```

indicating that there are four items total, we want subsets with two items, this item should start with position 1 and we're working on the first element in the resulting subsets.

The loop starts by setting nPos to 1; in other words, the first item in the subsets we're working on now is item 1. Since nRecCount is 0, nOldRecCount is set to 0. The recursive call passes these parameters:

```
4, 2, 2, 2
```

So, on the second call to GetNextValue, nPos starts at 2; that is, the second item in the subset we're now working on is item 2. Again, nOldRecCount is set to 0. Because nResultPos (2) is the same as nSetSize (2), there's no recursive call and we add a record to the cursor. The REPLACE command sets field Item2 to 2 in the newly added record (leaving field Item1 blank for now).

We return to the top of the loop and nPos becomes 3. Again, there's no recursive call and we add another record, this time setting Item2 to 3. We go through the loop one more time, adding another record, setting Item2 to 4.

At this point, we return to the original call to GetNextValue. nResultPos (1) isn't equal to nSetSize (2), so we don't add a record. We move the record pointer to one beyond nOldRecCount, in this case, record 1, and we loop through all records from there to the end of the cursor (there are three at this point), setting field Item1 to 1.

The loop continues, with nPos=2, but this should be enough to demonstrate the technique.

Recursion in VFP is limited by the program stack, 128 in recent versions. For this program, that means it can't handle subsets of more than 127 items.

The program above is included as GetSetsToCursor.PRG on this month's Professional Resource CD.

Incidentally, there's a connection between the factorial function and combinations. The number of combinations can be expressed as:

```
nCount!/(nSetSize! * (nCount-nSetSize)!)
```

–Tamar