Understanding combinations and permutations is a fundamental skill in software testing. In this month’s Test Run column I show you how to work with combinations and permutations using code written in the new F# language.

A mathematical combination is a subset of k items selected from a set of n items, where order does not matter. For ex-ample, if n = 5 and k = 2, all possible ways to select two items from five items are:

{0,1}, {0,2}, {0,3}, {0,4}, {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}

Notice that I do not list the combination {1,0} because it is considered the same as {0,1}. Also, I have listed the 10 combinations of five items selected two at a time using what is called lexicographical order, where the values in each combination element are listed in increasing order.

A mathematical permutation is all possible rearrangements of n items. For example, if n = 4, all possible permutations listed in lexicographical order are:

{0,1,2,3}, {0,1,3,2}, {0,2,1,3}, {0,2,3,1}, {0,3,1,2}, {0,3,2,1}, {1,0,2,3}, {1,0,3,2}, {1,2,0,3}, {1,2,3,0}, {1,3,0,2}, {1,3,2,0}, {2,0,1,3}, {2,0,3,1}, {2,1,0,3}, {2,1,3,0}, {2,3,0,1}, {2,3,1,0}, {3,0,1,2}, {3,0,2,1}, {3,1,0,2}, {3,1,2,0}, {3,2,0,1}, {3,2,1,0}

When working with combinations and permutations, two important functions are Choose(n,k) and Factorial(n). You are probably familiar with the Factorial(n) function, which is often abbreviated as n! The Factorial(n) function returns the total number of permutations of order n. For example:

4! = 4 \* 3 \* 2 \* 1 = 24.

The Choose(n,k) function returns the total number of combinations of k items selected from n items.

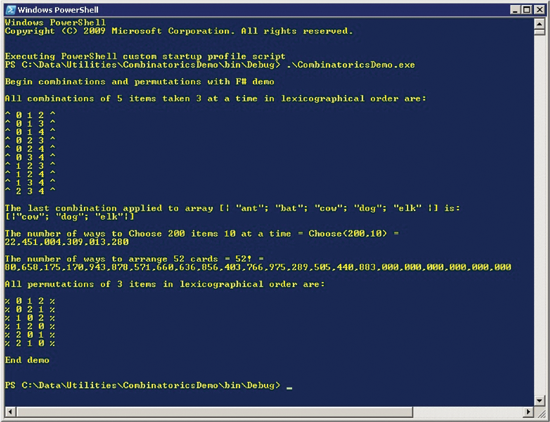
Choose(n,k) = n! / (k! \* (n-k)!)

For example:

Choose(5,2) = 5! / (2! \* (5-2)!) = 5! / (2! \* 3!) = 120 / 12 = 10.

Combinations and permutations are part of an area of study usually called combinatorial mathematics, or just combinatorics for short.

A good way for you to see where I’m headed in this month’s column is to take a look at the screenshot in**Figure 1**. I used Windows PowerShell to host my F# demo application, but I could have just as easily used a command shell. I modified my Windows PowerShell startup script to automatically navigate to the location of my CombinatoricsDemo.exe program.

  
Figure 1 **Combinations and Permutations with F#**

Behind the scenes, the demo program references and calls into an F# code library named CombinatoricsLib. The demo begins by listing all mathematical combinations of five items selected three at a time. Next, it uses a helper method to apply the last combination element {2,3,4}, to an array of strings {ant, bat, cow, dog, elk} to yield {cow, dog, elk}—that is, the three strings from the original set that are located at index values 2, 3 and 4.

My demo program continues by computing and displaying the value of Choose(200,10), the number of ways to select 10 items from a set of 200 items where order does not matter.

Next, the demo code computes and displays the value of Factorial(52), which is the total number of ways to arrange the cards in a standard deck of 52 cards. Notice that the result is a very, very large number. As I’ll explain, F# has the ability to work with arbitrarily large integer values.

My demo program concludes by listing all mathematical permutations of order n = 3.

In the sections that follow, I describe in detail the F# code in the CombinatoricsLib module and the code in the F# demo program shown running in **Figure 1**. Along the way, I compare the use of F# with other languages such as Visual Basic and C# for working with combinations and permutations.

This column assumes you have beginner-to-intermediate experience with a .NET language such as C#, and a very basic familiarity with F#. But even if you are completely new to F#, you should be able to follow my explanations without too much difficulty.

## The F# CombinatoricsLib Library

To create my combinatorics library, I used the beta 2 release of Visual Studio 2010, which has the F# language and tools built in. I expect the F# code I present here to work with the release version of Visual Studio 2010 without any significant changes. If you are using an earlier version of Visual Studio, you can find the F# tools on the Microsoft F# Developer Center ([msdn.microsoft.com/fsharp](http://msdn.microsoft.com/fsharp)). In addition to the F# language, Visual Studio 2010 ships with the Microsoft .NET Framework 4, which my library uses.

I created my library by launching Visual Studio 2010 and selecting File | New | Project. On the new project dialog, I selected the F# Library template and named my library CombinatoricsLib. The overall structure of my CombinatoricsLib is listed in **Figure 2**.

Figure 2 **F# CombinatoricsLib Structure**

module Module1

open System

open System.Numerics // BigInteger class

type Combination(n : int, k : int, a : int[]) =

// primary constructor code

new(n : int, k : int) =

...

member this.IsLast() : bool =

...

member this.Successor() : Combination =

...

member this.ApplyTo(a : string[]) : string[] =

...

override this.ToString() : string =

...

static member Choose(n : int, k : int) : BigInteger =

...

static member Factorial(n : int) : BigInteger =

...

// end type Combination

type Permutation(n : int, a : int[]) =

// primary constructor code

new(n : int) =

...

override this.ToString() : string =

...

member this.Successor() : Permutation =

...

member this.ApplyTo(a : string[]) : string[] =

...

member this.IsLast() : bool =

// end type Permutation

The library code begins with Visual Studio-generated code that names my library Module1. I used this rather nondescript default name instead of changing to something more descriptive so that the syntax for accessing the library will stand out later in this article.

Next, I added two F# open statements to the top-level System namespace and the new System.Numerics namespace so I can access the classes in these namespaces without fully qualifying the class names. The System.Numerics namespace is part of the .NET Framework 4 and contains a BigInteger definition that allows me to work with arbitrarily large integer values.

Defining a type in F# is different from defining a conceptually equivalent C# class. The type definition has a signature that contains the input arguments for the primary type constructor:

type Combination(n : int, k : int, a : int[]) =

This signature means, “I am defining a type named Combination that has a primary constructor that accepts int arguments n (the total number of items), and k (the subset size), and an integer array a that specifies the individual combination values, such as {0,1,4}.”

F# types may have optional secondary constructors, such as the one listed in **Figure 2**:

new(n : int, k : int) =

Notice that secondary constructors use the explicit new keyword as opposed to type primary constructors. This secondary constructor accepts values just for n and k. With other programming languages such as C#, you would likely define the simpler constructor as the primary constructor, that is, before constructors that accept arguments. But as you’ll see in F#, for types with multiple constructors, it is usually better to create a type in such a way that the primary constructor is the one with the most parameters. The structure of my Combination type continues with three member functions:

member this.IsLast() : bool =

...

member this.Successor() : Combination =

...

member this.ApplyTo(a : string[]) : string[] =

...

The this keyword binds member methods that are publicly visible to external calling code. The IsLast function returns true if the associated Combination object is the last element in lexicographical order, such as {2,3,4} for n = 5 and k = 3, as shown in **Figure 1**.

The Successor function returns the next Combination element in lexicographical order to the current element.

The ApplyTo function accepts an array of strings and returns an array of strings that corresponds to the current Combination element.

My next type member function provides a way to display a Combination element:

override this.ToString() : string =

I use the override keyword to distinguish my custom ToString function from the base ToString method. Because F# is a .NET language, all objects inherit from a common base Object that has a ToString method. Notice that even though the overridden ToString function has public scope, I do not use the member keyword.

The last two member functions in the Combination type define the Choose and Factorial functions:

static member Choose(n : int, k : int) : BigInteger = ...

static member Factorial(n : int) : BigInteger = ...

Both functions are static, which means that the functions are associated with and called directly from the context of the Combination type rather than a particular instance of a Combination object. Both functions return type BigInteger, defined in namespace System.Numerics, which is directly visible by default to F# code.

In addition to a Combination type, I define a Permutation type as shown in the listing in **Figure 2**.

## The F# Combinatorics Library Implementation

Now let’s go over the details of the implementation of the CombinatoricsLib library. The Combination primary constructor begins:

type Combination(n : int, k : int, a : int[]) =

do if n < 0 || k < 0 then failwith

"Negative argument in Combination ctor"

do if n < k then failwith

"Subset size k is larger than n in Combination"

Interestingly, to perform input argument validation in a primary constructor you should use the do keyword, which indicates an action, rather than the default of a value, which is typically assumed by functional languages such as F#. The failwith keyword throws an exception, which can be caught by calling code.

Next, I set up the Combination type private members:

let n : int = n // private

let k : int = k

let data =

[| for i = 0 to a.Length-1 do yield a.[i] |]

The let keyword binds values to private members. Notice that I can use lowercase n and k as both input parameters and as private fields. This looks a bit awkward and so in most situations I use uppercase notation for either the parameters or the private members.

Copying the values from the input argument array a into the type field data uses a standard F# idiom. The [| . . |] delimiters indicate a mutable array. The secondary Combination constructor creates an initial Combination object and is defined as:

new(n : int, k : int) =

do if n < 0 || k < 0 then failwith

"Negative argument in Combination ctor"

do if n < k then failwith

"Subset size k is larger than n in Combination"

let starters = [| for i in 0..k-1 -> i |]

new Combination(n,k,starters)

In F#, secondary constructors must call the primary constructor. So I define an int array named starters with values 0 through k-1 and pass it along with n and k to the primary constructor. This F# mechanism is why it is advisable to define any primary constructor as the constructor with the most parameters.

The IsLast member function is defined as:

member this.IsLast() : bool =

if data.[0] = n-k then true

else false

In **Figure 1**, notice that only the last element in a list of all combinations has value n-k located in the first position of the array. F# does not use an explicit return keyword as most languages do; the implied return is the last value in a function, in this case either true or false. The = token checks for equality in F# and is not an assignment operator. The Combination.Successor function is:

member this.Successor() : Combination =

// copy input to temp array

let temp = [| for i in 0..k-1 -> data.[i] |]

// find "x" - right-most index to change

let mutable x = k-1

while x > 0 && temp.[x] = n - k + x do

x <- x - 1

temp.[x] <- temp.[x] + 1 // increment value at x

// increment all values to the right of x

for j = x to k-2 do

temp.[j+1] <- temp.[j] + 1

// use primary ctor

let result = new Combination(n, k, temp)

result

I begin by copying the values of the current Combination object context into a mutable array named temp. Next, I define an index variable named x and position it at the end of the temp array. I must use the mutable keyword so that I can decrement this index variable because, by default, most variables in F# are immutable. I use the <- assignment operator.

Once I locate the key index of the current Combination object, I increment that value and all values to the right of the key index. Then I pass the temp array, which now has the value of the successor Combination element, into the primary constructor and return the newly created object.

Notice that I do not return null when I am at the last Combination element—in F# it is considered poor style to do so. The code I present in this article uses a style that is not very F#-ish. F# experts would likely use a recursive approach, but because I am assuming you are new to F#, I wanted to make my F# code as familiar as possible.

An alternative approach to writing a Successor function is to implement the .NET IEnumerable interface.

The ApplyTo function provides a way to map a combination element to a set of string values:

member this.ApplyTo(a : string[]) : string[] =

if a.Length <> n then failwith

"Invalid array size in ApplyTo()"

// array of int

let result = Array.zeroCreate k

for i = 0 to k-1 do

// bind to array of string

result.[i] <- a.[data.[i]]

result

When performing input argument checks in a member function, I do not need to use the do keyword as is required in type constructors. The static Array.zeroCreate method creates an integer array initialized to all 0 values as you might expect. The ApplyTo function is easy because the range of values in a mathematical combination with subset size k (0..k-1) is exactly the same as the indexes of any .NET array of size k.

The overridden ToString member function simply builds a string made up of the context object’s values:

override this.ToString() : string =

let mutable s : string = "^ "

for i in 0..k-1 do

s <- s + data.[i].ToString() + " "

s <- s + "^"

s

I decided to delimit my Combination elements with the ^ (caret) character, which starts with the letter c, and to delimit my Permutation elements with the % (percent) character, which starts with p, to help me identify whether a string of digits represents a Combination or a Permutation object.

The static Choose function is coded as shown in **Figure 3**.

Figure 3 **The Choose Function**

static member Choose(n : int, k : int) : BigInteger =

if n < 0 || k < 0 then failwith

"Negative argument in Choose()"

if n < k then failwith

"Subset size k is larger than n in Choose()"

let (delta, iMax) =

if k < n-k then

(n-k, k)

else

(k, n-k)

let mutable answer : BigInteger =

bigint delta + bigint 1

for i = 2 to iMax do

answer <- (answer \* (bigint delta + bigint i ))

/ bigint i

answer

Instead of computing Choose from the definition described earlier in this article, I use two optimizations. First, I use the fact that Choose(n, k) = Choose(n, n-k). For example Choose(9,6) = Choose(9,3). Second, rather than compute three separate factorials, each of which can be very large, I compute a series of partial products. In order to explicitly convert int values to type BigInteger, I use the built-in F# bigint function.

The implementation of the Permutation type is quite similar to the implementation of the Combination type. You can get the  complete source code for the CombinationLib library from the Microsoft Code Gallery Web site at code.msdn.microsoft.com.

## Using the CombinatoricsLib

In this section I explain how to call the function in the CombinatoricsLib library to produce the run shown in the screenshot in **Figure 1**. I begin by launching Visual Studio 2010 and creating a new F# Application project named CombinatoricsDemo. The entire program is listed in **Figure 4**.

Figure 4 **Using the CobinatoricsLib**

open System

open Module1 // the Combinatorics Lib

try

printfn "\nBegin combinations and permutations with F# demo\n"

printfn "All combinations of 5 items 3 at a time in lexicographical order are: \n"

let mutable c = new Combination(5,3)

printfn "%A" c // print initial combination

// objects cannot be null in F# so use an explicit method

while c.IsLast() = false do

c <- c.Successor()

printfn "%A" c

printf "\nThe last combination applied to array [| \"ant\"; \"bat\"; \"cow\"; \"dog\"; \"elk\" |] is: \n"

let animals = [| "ant"; "bat"; "cow"; "dog"; "elk" |]

//let result = c.ApplyTo(animals)

let result = animals |> c.ApplyTo

printfn "%A" result

printfn "\nThe number of ways to Choose 200 items 10 at a time = Choose(200,10) ="

let Choose\_200\_10 = Combination.Choose(200,10).ToString("000,000")

printfn "%s" Choose\_200\_10

printfn "\nThe number of ways to arrange 52 cards = 52! = "

let Factorial\_52 = Combination.Factorial(52).ToString("000,000")

printfn "%s" Factorial\_52

printfn "\nAll permutations of 3 items in lexicographical order are: \n"

let mutable p = new Permutation(3)

printfn "%A" p // print initial permutation

while p.IsLast() = false do

p <- p.Successor()

printfn "%A" p

printfn "\nEnd demo\n"

Console.ReadLine() |> ignore

with

| Failure(errorMsg) -> printfn "Fatal error: %s" errorMsg

// end program

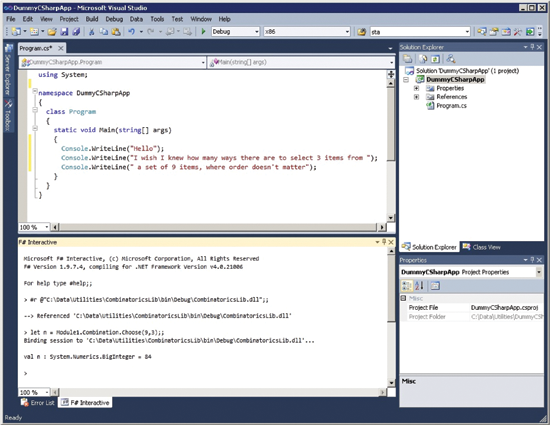
Before writing any code, I right-clicked on the Project name in the Solution Explorer window of Visual Studio and selected the Add Reference option from the context menu. I then selected the Browse tab and navigated to the CombinatoricsLib.dll assembly.

I begin the demo program code by adding open statements to the System and Module1 assemblies. Recall that the module name of the CombinatorcsLib is Module1. I wrap all program statements in a try/with block to capture and handle exceptions. I instantiate a Combination object using the secondary constructor to make an initial mathematical combination object c of five items taken three at a time: {0,1,2}. I use the neat F# %A format specifier, which instructs F# to infer how to print my Combination object. I could also have used the %s string format.

Next, I use the F# while..do loop to iterate through and display all 10 Combination(5,3) elements. At this point, the Combination object c is the last element and I call the ApplyTo function to map that combination onto an array of strings.

Notice that I call the Choose and Factorial functions from the context of the Combination type rather than the c Combination object. After calling the Permutation type code in a similar way, the demo program concludes by pausing for user input with the Console.ReadLine method, where I pipe the return value to the built-in ignore object. I handle any exceptions in the with block, by simply displaying the exception error message.

In addition to calling an F# library from an F# program as I’ve just demonstrated, you can call an F# library from any .NET-compliant language. Additionally, Visual Studio allows you to use the handy F# interactive window to make ad hoc calls, as shown in **Figure 5**.

  
Figure 5 **Interactive Use of an F# Library**

In the F# interactive window at the bottom of the screen, I add a reference to the CombinatoricsLib assembly by typing:

#r @"C:\(path)\CombinatoricsLib.dll";;

In this case #r means add a reference, and ;; terminates an interactive F# statement. Now I can interactively call the functions in the library. Neat!

 In my opinion, there are several pros and cons to using F#. On the negative side, I found that the learning curve for F# was much steeper than I expected. Writing in a functional style was a big paradigm shift for me. Also, much more so than other languages, F# has multiple ways of coding a particular task, which led me to feel uneasy about whether any F# code I wrote was written in the optimal way.

However, in my case at least, I feel the benefits of learning F# definitely outweigh the costs. When talking to experienced F# coders, most told me that in many cases, even though there are in fact several ways to code a task, the approach taken is more a matter of personal preference than technical efficiency. Also, grappling with F# syntax and coding paradigms (such as default immutability) gave me what I felt was some good insight into coding in procedural languages such as C#.