The ability to programmatically create and use string permutations is an essential software testing skill. A string permutation is a rearrangement of a set of strings. For example, if you have an initial set of three strings—apple, banana, cherry—then there are a total of six permutations:

{ "apple", "banana", "cherry" } { "apple", "cherry", "banana" } { "banana", "apple", "cherry" } { "banana", "cherry", "apple" } { "cherry", "apple", "banana" } { "cherry", "banana", "apple" }

A typical use of permutations in software testing is generating test case data for unit, API, and module tests. Suppose you have a method that accepts three strings, and one of your test case inputs is "apple", "banana", "cherry". In most situations you want to create five additional test cases using the other input permutations. There are many other uses of permutations in software testing. In fact, permutations are so important and prevalent in software engineering that questions about permutations are among the most common type of interview questions for testing jobs at Microsoft.

A good way to see where I'm headed in this month's column is by way of a screenshot. **Figure 1** shows a run of a demo program. (The complete source code used to generate the demo is available in the download for this column.) As the output in **Figure 1** points out, the three fundamental skills related to string permutations are the ability to calculate exactly how many permutations there are for a given set of strings, the ability to generate all permutations, and the ability to generate a specified permutation.

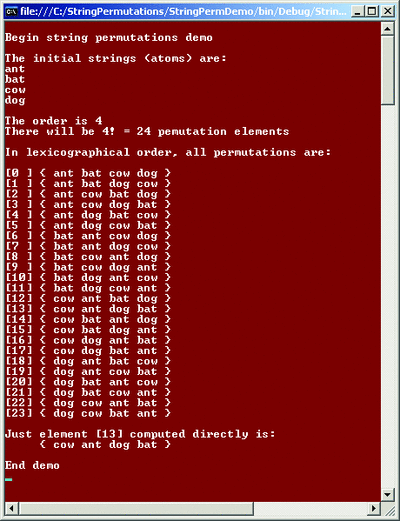


Figure 1**String Combinations Demonstration**(Click the image for a larger view)

Because permutation terminology varies somewhat between problem domains, let's establish a few key vocabulary terms that I'll be using throughout this column. The term permutation element refers to an ordered set of strings from some initial set. The term atom means any one of the individual strings in a permutation element. The order of the permutation is the number of atoms in any element. I will be using lexicographical permutations. This means the initial permutation element, also called the identity element, has atoms listed in dictionary order. Each permutation element is also ordered relative to the other elements. This means that there is a distinct identity permutation element, a distinct last element, and each element has a successor and predecessor element (except for the first element which has no predecessor, and the last element which has no successor). Finally, the term string permutation just means a permutation element where the atoms are strings. I use this term because mathematical permutations of order n have integer atoms from 0 to n-1, such as { 0, 1, 2 }. The total number of permutation elements of order n is n! (read as "n factorial"). For example, with four atoms, the total number of permutation elements is:

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

The reason for this should be easy to see. For the first atom in the element, any of the n atoms can be chosen. For the next, any of the n-1 remaining atoms can be chosen. And so on.

Permutations are closely related to, and sometimes confused with, combinations. String combinations are subsets of an initial set of strings, where order does not matter. For example, consider my initial set of strings that comprise apple, banana, and cherry. There are three combination elements of subset size k = 1:

{ "apple" }, { "banana" }, {"cherry" }

There are three combination elements of subset size k = 2:

{ "apple", "banana" } { "apple", "cherry" } { "banana", "cherry" }

And there is one combination element with subset size k = 3:

{ "apple", "banana", "cherry" }

The July 2004 issue of MSDN®Magazine includes a column on combinations, "[Using Combinations to Improve Your Software Test Case Generation](http://msdn.microsoft.com/msdnmag/issues/04/07/TestRun)".

In the remainder of this column I will present the overall design of a string permutation class, including a basic constructor, and I will show you how to generate all possible permutation elements by using a neat algorithm to generate the successor to a given permutation element. Next, I'll show you how to generate a specific permutation element using a cool technique that employs one of the most elegant algorithms I've ever seen. I'll then present techniques to determine how many permutation elements there are for an initial set of strings. I'll conclude with a brief discussion of how you can adapt and extend the code and algorithms presented here to meet your own needs. I'm confident that after you've read this column you will have added several useful new tools to your personal skill set.

**A String Permutation Class**

String permutations are perfectly suited for implementation using an object-oriented approach because permutations contain closely related data (the string atoms) and code. The overall structure of my class library is presented in **Figure 2**.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 2 StringPerm Class Library Structure**

using System; using System.Text; namespace StringPermLib { public class StringPerm { private string[] element; private int order; public StringPerm(string[] atoms) { ... } public StringPerm(string[] atoms, int k) { ... } public override string ToString() { ... } public static bool IsValid(string[] e) { ... } public StringPerm Successor() { ... } public static ulong FactorialCompute(int n) { ... } public static ulong FactorialLookup(int n) { ... } public static ulong FactorialRecursive(int n) { ... } } }

I've named my single class StringPerm; you can choose a different name of course. The StringPerm class has only two data fields, both private: a string array named element to hold the individual atom strings for a particular string permutation element, and an integer field to store the order of the permutation class. I do not implement a default constructor; instead I implement a constructor that accepts a string array as a single input parameter and creates the identity permutation element:

public StringPerm(string[] atoms) { if (atoms == null) throw new ArgumentNullException("atoms"); if (!IsValid(atoms)) throw new ArgumentException("atoms"); this.element = new string[atoms.Length]; atoms.CopyTo(this.element, 0); this.order = atoms.Length; }

I simply allocate memory space for the class array field, use the Array.CopyTo method to assign each value, and then use the input argument's Length property to assign a value to the order field—clean and simple, which is usually a good sign when designing a class. The following code snippet shows that class constructor being called:

string[] atoms = new string[] { "ant", "bat", "cow", "dog" }; StringPerm p = new StringPerm(atoms);

Because I am creating a lexicographical string permutation class, my constructor must accept a string array that is sorted. For that task, I've added a static method IsValid to my class, which I call from my constructor. Here is one way to implement IsValid:

public static bool IsValid(string[] e) { if (e == null) return false; if (e.Length < 2) return false; for (int i = 0; i < e.Length - 1; ++i) { if (e[i].CompareTo(e[i + 1]) >= 0) return false; } return true; }

My method accepts a string array and returns true if the array is valid as an input argument to my StringPerm constructor, false otherwise. I check to make sure that my input array is not null and that it has at least two cells. You may want to modify this part of the code to allow the possibility of a string permutation of order 1. Next I use the String.CompareTo method to make sure that each string atom in the input array is strictly less than its next neighbor or, in other words, I check to make sure that the string atoms are in order, and I do not allow duplicate atoms. Notice that I use reverse logic and return false if the string atom at index position i is greater than or equal to the string atom at index i+1. You could also allow for duplicates, such as in the following array:

{ "ant", "bat", "bat", "cow" }

To do so, just change the >= operator in the check to >.

It is convenient to have a small ToString helper method. Here is one way:

public override string ToString() { StringBuilder result = new StringBuilder(); result.Append("{ "); for (int i = 0; i < this.order; ++i) { result.Append(this.element[i]); result.Append(" "); } result.Append("}"); return result; }

All I'm doing here is simply building up a result string by concatenating each string atom separated by a blank space and surrounded by curly brace delimiters.

**Generating All Permutations of Order n**

One approach to generating all string permutations is to write a Successor method that returns the next lexicographical string permutation element for a given element. This is an interesting problem and has a pretty solution. Briefly, the algorithm to find the successor to a permutation element determines the locations of two atoms, swaps the atoms at those two positions, and then reverses the order of atoms on the right tail. It's easiest to see what I mean with an example. Suppose you are working with a string permutation of digits (this will make my explanation easier to follow) and the current permutation element is:

{ "1" "3" "5" "4" "2" "0" }

The successor element is:

{ "1" "4" "0" "2" "3" "5" }

First you find a left index and a right index, which point to two atoms that will be swapped. To find index left you start at the next to last atom (2 in this example), and move left until you find a condition where the atom on the left is less than the current atom. So, in this example I'd move left until reaching atom 3 because 3 is less than 5. So index left is [1]. To find index right, you start at the right-most atom (in this case 0) and move left until you find an atom that is greater than the atom at index left. Here I'd move left until hitting atom 4, so index right is [3]. Now swap the atoms at indexes left and right ([1] and [3]), yielding an intermediate result of:

{ "1" "4" "5" "3" "2" "0" }

I finish by reversing all atoms to the right of index left, in this case all atoms to the right of [1], which includes 5, 3, 2, and 0, giving us the final successor result:

{ "1" "4" "0" "2" "3" "5" }

The code to implement this algorithm is presented in **Figure 3**.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 3 Implementing the Successor Algorithm**

public StringPerm Successor() { StringPerm result = new StringPerm(this.element); int left, right; // Step #1 - Find left value left = result.order - 2; while ((result.element[left].CompareTo(result.element[left + 1])) >= 0 && (left >= 1)) { --left; } if ((left == 0) && (this.element[left].CompareTo(this.element[left + 1])) >= 0) { return null; } // Step #2 - find right; first value > left right = result.order - 1; while (result.element[left].CompareTo(result.element[right]) >= 0) { --right; } // Step #3 - swap [left] and [right] string temp = result.element[left]; result.element[left] = result.element[right]; result.element[right] = temp; // Step #4 - reverse order the tail int i = left + 1; int j = result.order - 1; while (i < j) { temp = result.element[i]; result.element[i++] = result.element[j]; result.element[j--] = temp; } return result; }

This Successor method only works for permutations with no duplicate atoms. A mildly tricky part of the Successor method is the part that determines what to do on the last permutation element. Suppose you are working with this element:

{ "ant", "bat", "cow", "dog" }

The last permutation element is:

{ "dog", "cow", "bat", "ant" }

There are several ways to identify this situation. The approach I take uses the fact that when determining the value of index left in the Successor code, if index left has value 0 and the atom at [0] is greater than the atom at [1], I must be at the last permutation element. In this case I return null. Alternatives to returning null that you might want to consider are returning the identity (first) element (thus wrapping around to the start of the permutation sequence), or throwing an exception.

With this Successor method, you can easily generate a single successor permutation to any given permutation element, or you can generate all permutation elements (if you do not have any duplicate atoms) like this:

string[] atoms = new string[] { "ant", "bat", "cow", "dog" }; Console.WriteLine("In lexicographical order, all permutations are:\n"); for(StringPerm p = new StringPerm(atoms); p != null; p = p.Successor()) { Console.WriteLine(p.ToString()); }

A subtle part of this algorithm concerns the issue of duplicate string atoms. As presented, my code does not allow duplicates. But if you want to allow duplicate atoms in your permutation elements, you can first modify the IsValid helper method described in the previous section. Then, to generate all permutation elements with duplicates, you must use a different technique such as the one I present in the next section.

**Determining a Specific Permutation Element**

Suppose you want to generate just one particular string permutation element. For example, suppose you are working with strings "ant", "bat", "cow", and "dog" and you want just element [13], which is:

{ "cow" "ant" "dog" "bat" }

At first I thought this would be an easy problem; just start with the identity permutation element and then call the Successor method from the previous section 13 times. However, instead of working with just 4 string atoms, suppose you are working with 20 string atoms. In this case there are 2,432,902,008,176,640,000 permutation elements! So, suppose you want just element [999,999,999,999]. Using the looping technique on my desktop computer, it took well over 24 hours to compute the answer. Furthermore, using the looping technique to generate all permutation elements won't work at all if you allow duplicate string atoms. But there is an elegant algorithm you can use that will compute the answer in well under one second and that allows duplicate atoms.

The basis of this graceful algorithm that directly computes a specific permutation element is a neat mathematical construct called the factoradic. A factoradic is an alternate representation of an integer. The trick is to compute the factoradic of the desired index value, then convert the factoradic into a permutation element.

Before I show you the details, let me explain a factoradic. Consider the number 859. If you set the order to n = 7, then the number 859 can be represented as:

(6! \* 1) + (5! \* 1) + (4! \* 0) + (3! \* 3) + (2! \* 0) + (1! \* 1) + (0! \* 0) = 720 + 120 + 0 + 18 + 0 + 1 = 859

So, the factoradic of 859 is [1 1 0 3 0 1 0]. It turns out that any integer can be uniquely represented as a factoradic with a specified order. What makes the factoradic useful is that there is a one-to-one mapping between factoradics and permutation elements. For example, if you are working with permutations of order 4, the table in **Figure 4** shows the relationship between an integer, its factoradic, and corresponding mathematical permutation.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 4 Relationships**

index factoradic permutation ================================= 0 [ 0 0 0 0 ] { 0 1 2 3 } 1 [ 0 0 1 0 ] { 0 1 3 2 } 2 [ 0 1 0 0 ] { 0 2 1 3 } 3 [ 0 1 1 0 ] { 0 2 3 1 } 4 [ 0 2 0 0 ] { 0 3 1 2 } 5 [ 0 2 1 0 ] { 0 3 2 1 } 6 [ 1 0 0 0 ] { 1 0 2 3 } 7 [ 1 0 1 0 ] { 1 0 3 2 } 8 [ 1 1 0 0 ] { 1 2 0 3 } 9 [ 1 1 1 0 ] { 1 2 3 0 } 10 [ 1 2 0 0 ] { 1 3 0 2 } 11 [ 1 2 1 0 ] { 1 3 2 0 } 12 [ 2 0 0 0 ] { 2 0 1 3 } 13 [ 2 0 1 0 ] { 2 0 3 1 } 14 [ 2 1 0 0 ] { 2 1 0 3 } 15 [ 2 1 1 0 ] { 2 1 3 0 } 16 [ 2 2 0 0 ] { 2 3 0 1 } 17 [ 2 2 1 0 ] { 2 3 1 0 } 18 [ 3 0 0 0 ] { 3 0 1 2 } 19 [ 3 0 1 0 ] { 3 0 2 1 } 20 [ 3 1 0 0 ] { 3 1 0 2 } 21 [ 3 1 1 0 ] { 3 1 2 0 } 22 [ 3 2 0 0 ] { 3 2 0 1 } 23 [ 3 2 1 0 ] { 3 2 1 0 }

Here's an example of what I'm getting at. Suppose you are dealing with the four string atoms "ant", "bat", "cow", and "dog", and you want to compute permutation [13] directly. First determine the factoradic of 13, which is [2 0 1 0], then map that factoradic to its corresponding mathematical permutation, which is { 2 0 3 1 }, and then map that permutation to the initial string atoms:

{ "cow", "ant", "dog", "bat" }

A detailed description of factoradics is outside the scope of this column. However, I dedicate an entire section of my book, .NET Test Automation Recipes (Apress, 2006), to the topic, and you can also find online information.

I decided to implement the subroutine of determining a specific permutation element as an auxiliary constructor. The code is presented in **Figure 5**. With the constructor shown in **Figure 5**, you can make calls like this:

string[] atoms = new string[] { "ant", "bat", "cow", "dog" }; Console.WriteLine("\nJust element [13] computed directly is:"); p = new StringPerm(atoms, 13); Console.WriteLine(" " + p.ToString());

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 5 Determining a Specific Permutation Element**

public StringPerm(string[] atoms, int k) { this.element = new string[atoms.Length]; this.order = atoms.Length; // Step #1 - Find factoradic of k int[] factoradic = new int[this.order]; for (int j = 1; j <= this.order; ++j) { factoradic[this.order - j] = k % j; k /= j; } // Step #2 - Convert factoradic[] to numeric permuatation in perm[] int[] temp = new int[this.order]; int[] perm = new int[this.order]; for (int i = 0; i < this.order; ++i) { temp[i] = ++factoradic[i]; } perm[this.order - 1] = 1; // right-most value is set to 1. for (int i = this.order - 2; i >= 0; --i) { perm[i] = temp[i]; for (int j = i + 1; j < this.order; ++j) { if (perm[j] >= perm[i]) ++perm[j]; } } for (int i = 0; i < this.order; ++i) // put in 0-based form --perm[i]; // Step #3 - map numeric permutation to string permutation for (int i = 0; i < this.order; ++i) this.element[i] = atoms[perm[i]]; }

This constructor also allows you to generate all permutation elements regardless of whether you have duplicate atoms or not. For example, see the following code:

string[] atoms = new string[] { "ant", "bat", "cow", "cow" }; StringPerm p = null; Console.WriteLine("All permutations are:\n"); for (int k = 0; (ulong)k < StringPerm.FactorialLookup(4); ++k) { p = new StringPerm(atoms, k); Console.WriteLine(k + " " + p.ToString()); }

This technique will generate duplicate permutation elements. If for some reason you want just unique elements, there are several approaches you can use. For example, you can load all the elements into a hashtable, checking duplicates before each Add operation.

**The Number of Permutation Elements of Order n**

Calculating the total number of string permutation elements for a given set of strings is paradoxically easy and hard at the same time. Suppose you have four strings, "ant", "bat", "cow", and "dog", and so the order is n = 4. There are a total of 24 permutation elements, as shown in **Figure 1**. Any element can have any one of the four string atoms in the first (left-most) position, then any one of the remaining three atoms in the next position, then any one of the remaining two atoms in the next position, and finally the sole remaining atom in the last position. In other words, for n = 4, there are 4 \* 3 \* 2 \* 1 = 24 permutation elements. You probably recognize this as the formula for n factorial (often written as n!). Therefore, the total number of string permutation elements of order n is just n factorial. A very easy Computer Science 101 problem, yes? Well, not necessarily. Now let's examine three different implementations of a factorial method.

One approach, often taken by recent college graduates, is to write a factorial method using recursion. For example:

public static ulong FactorialRecursive(int n) { if (n < 0) throw new ArgumentOutOfRangeException("n"); if (n == 0 || n == 1) return 1; else return (ulong)n \* FactorialRecursive(n - 1); }

This is a weak approach in most situations, and if you're not careful it can easily give you the wrong answer. The value of n! gets very large very quickly. For example, the value of 64! is roughly:

126,886,932,100,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000,000

In fact, the largest value that can be stored by a variable of type ulong is only 20! which is:

20 \* 19 \* 18 \*. .. \* 2 \* 1 = 2,432,902,008,176,640,000

What do you suppose would happen if you called

ulong result = FactorialRecursive(21)

in a program? You'd expect to throw an arithmetic overflow of some sort, but that is not what happens by default. The compiler will reach ulong.MaxValue (which is 18,446,744,073,709,551,615) and then wrap around, back to 0, and continue computing without any warnings. Note that the common language runtime (CLR) is capable of overflow and underflow checking on numeric operations, a feature you can enable at compile time. On top of this problem, there's really no compelling reason to use a recursive solution here. A second try at a factorial method is to compute the answer iteratively, something like this:

public static ulong FactorialCompute(int n) { if (n < 0 || n > 20) throw new Exception("Input argument must be between 0 and 20"); ulong answer = 1; for (int i = 1; i <= n; ++i) answer = checked(answer \* (ulong)i); return answer; }

This is certainly better in most cases. The method checks its input parameter and uses the C# checked keyword, which will cause a runtime exception if arithmetic overflow occurs. But, because the input argument set is so small—0 to 20—you might just as well store the 21 possible results in a lookup table, giving you a solution like:

public static ulong FactorialLookup(int n) { if (n < 0 || n > 20) throw new Exception("Input argument must be between 0 and 20"); ulong[] answers = new ulong[] { 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000 }; return answers[n]; }

In most software testing scenarios, the lookup approach is the most efficient. Obviously, my discussion of different ways to implement a factorial method just scratches the surface of this interesting subroutine.

**Wrapping It Up**

You can use string permutations in many ways. For example, suppose you are testing some software setup program and you are considering the effect of the order of installation of three additional but related programs. Let's call your software program "A", and the other three programs "B", "C", and "D". Using the techniques in this column you can easily determine that there are 24 installation orders, and easily generate those permutations. But suppose you are considering 19 other related programs. You will conclude that the 20! installation orders are just too many to test exhaustively so you'll have to prioritize your effort.

To summarize, string permutations are rearrangements of a set of strings. The ability to understand and use string permutations is a fundamental skill for software testers. In particular all testers should know how to calculate how many permutation elements there are for a given set of initial strings, produce all possible permutation elements, and efficiently generate a specified permutation element. This month's column showed you exactly how to accomplish those tasks. The code presented here can be used as is or can be modified to meet your own needs.