

Original Source (2003):

<http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnnetsec/html/permutations.asp>

Taken From (2019):

http://delphi.ktop.com.tw/download/upload/41338_Using%20Permutations%20in%20.NET%20for%20Improved%20Systems%20Security.doc

Using Permutations in .NET for Improved Systems Security

Dr. James McCaffrey

Volt Information Sciences, Inc.

August 2003

Applies to:

Â Â Â Microsoft® .NET

Â Â Â C#

Summary: Permutations are the backbone of all modern cryptosystems. By using permutations, you can increase the security of software systems. This article briefly explains what permutations are and provides the basis of a `Permutation` class implemented in C#, presents a short algorithm that solves the difficult problem of generating an arbitrary permutation by using a mathematical construct, and shows several practical applications of permutations, including block diffusion ciphers and programmatic generation of test cases. (22 printed pages)

Contents

[Introduction](#)

[A Permutation Class](#)

[The kth Permutation of Order n](#)

[The Successor of a Given Permutation](#)

[Block Ciphers](#)

[Programmatic Generation of Test Cases](#)

[Random Permutations](#)

[Conclusion](#)

Introduction

Permutations have a surprisingly wide range of uses in computer security. For example, the very first step in the Data Encryption Standard (DES) is to permute the input. In fact, permutations, along with substitutions, form the backbone of all modern cryptographic systems. In this article, I briefly describe what permutations are, provide you with a working **Permutation** class that contains a previously unpublished algorithm using a mathematical construct called the "factoradic," and show you how you can apply permutations to improve and test the security of software systems.

A permutation is an arrangement of items where order matters. For example, all six permutations of the three words "apple", "banana", and "cherry" are:

```
{ cherry, banana, apple }
```

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$
[illegible]

One of the staples of sophomore-level computer science classes is the problem of generating all possible permutations of a set of strings. A typical bad solution written using C# looks like the following code.

```
public static void Permute(string[] strings, int start, int finish)
{
    if (start == finish)
    {
        for (int i = 0; i <= finish; ++i)
        {
            Console.Write(strings[i] + " ");
        }
        Console.WriteLine("");
    }
    else
    {
```

```

    for (int i = start; i <= finish; ++i)
    {
        string temp = strings[start];
        strings[start] = strings[i];
        strings[i] = temp;

        Permute(strings, start+1, finish);

        temp = strings[start];
        strings[start] = strings[i];
        strings[i] = temp;
    }
}

} // Permute()

```

Why is this solution bad? After all, it works doesn't it? There are at least five reasons why this approach is weak. First, the solution uses recursion so the memory-use characteristics are not easily predictable. Second, it has an unnatural signature by requiring indexes (**start** and **finish**) as input parameters. Third, it does not easily generalize. Fourth, there is no easy way to generate a particular permutation without generating all permutations. And finally, I don't know about you, but to me it's not even obvious that the code works at all!

Three fundamental problems when dealing with permutations are:

1. How can I generate all permutations of a list?
2. Given a particular permutation, what is the next permutation?
3. What is the *k*th permutation of a list?

I will present code that answers these three questions and fully explain the algorithms I use. You will be able to use the code as is, or you can extend the code into a full-fledged **Permutation** class as part of a high-quality general scientific library, or you can use the algorithms to create specialized methods based on permutations. The ability to understand and use permutations and their associated algorithms will be a valuable addition to your developer skills set. I will also show you how to use permutations to increase the security of systems using a technique called diffusion, and how you can use permutations to improve security testing of systems.

A Permutation Class

A mathematical permutation lends itself nicely to implementation as a class. For data members, we really only need an array of integers and a single-integer value representing the order of the permutation. The basic code is shown below that represents a **Permutation** object and a constructor to create an **Identity** permutation object along with code to represent it as a string. I decided to use C#, but you can easily adapt the .NET language of your choice.

Permutation Class Definition

```
public class Permutation
```

```

{
    private int[] data = null;
    private int order = 0;

    public Permutation(int n)
    {
        this.data = new int[n];
        for (int i = 0; i < n; ++i)
        {
            this.data[i] = i;
        }

        this.order = n;
    }

    public override string ToString()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        sb.Append("( ");
        for (int i = 0; i < this.order; ++i)
        {
            sb.Append(this.data[i].ToString() + " ");
        }
        sb.Append(")");

        return sb.ToString();
    } // ToString()

    // other methods

} // class Permutation

```

When we put this code into a class library (arbitrarily named GeneralScientific), we can call it from a .NET console application:

```

Permutation p1 = new Permutation(5);
Console.WriteLine("\nIdentity permutation of order 5 is:");
Console.WriteLine(p1.ToString());

```

The resulting output is shown in Figure 1.

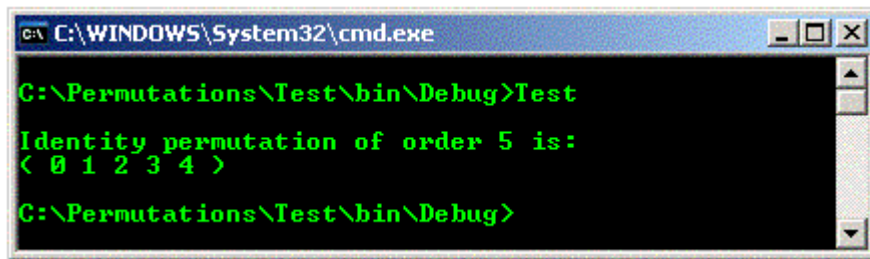


Figure 1. Creating and displaying identity permutations

This is not a very impressive start, but things will get more interesting quickly. When the **Permutation** constructor is invoked with

```
Permutation p1 = new Permutation(5)
```

We get an object in memory that can be represented as shown in Figure 2. The constructor code that creates an **Identity** permutation is as simple as it gets. The **ToString()** code that returns a string representation of a **Permutation** object uses the **StringBuilder** class, which is slightly more efficient than using string objects. Our representation conforms to the style normally used in math texts.

p1

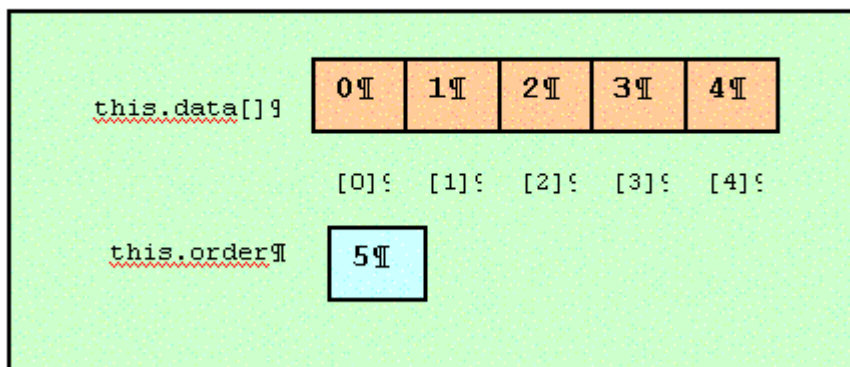


Figure 2. Permutation memory representation

Because the valid values of a mathematical permutation of order n are $0..n-1$, we could have declared the data array as an array of unsigned integers. Also note that order field is synonymous with the **data.Length** property, so we could have left it out altogether. Deciding whether or not to include the order field and deciding whether to use **int** or **uint** are typical clarity-versus-efficiency design choices that must be confronted when using object-oriented programming. Because the data array has conceptual meaning only as a whole, I did not name the array "Items", as is usual for collection classes.

It is very useful to have a constructor that can create a **Permutation** object with a specified value. This leads to a validity check method, too. The following code will create a **Permutation** object from an array of integers and a method that checks if a **Permutation** object is valid.

Permutation Construction from Integer Array

```
public Permutation(int[] a)
{
    this.data = new int[a.Length];
```

```

        a.CopyTo(this.data, 0);
        this.order = a.Length;
    }

    public bool IsValid()
    {
        if (this.data.Length != this.order)
            return false;

        bool[] checks = new bool[this.data.Length];

        for (int i = 0; i < this.order; ++i)
        {
            if (this.data[i] < 0 || this.data[i] >= this.order)
                return false; // value out of range

            if (checks[this.data[i]] == true)
                return false; // duplicate value

            checks[this.data[i]] = true;
        }

        return true;
    } // IsValid()

```

We can call and demonstrate the code like so:

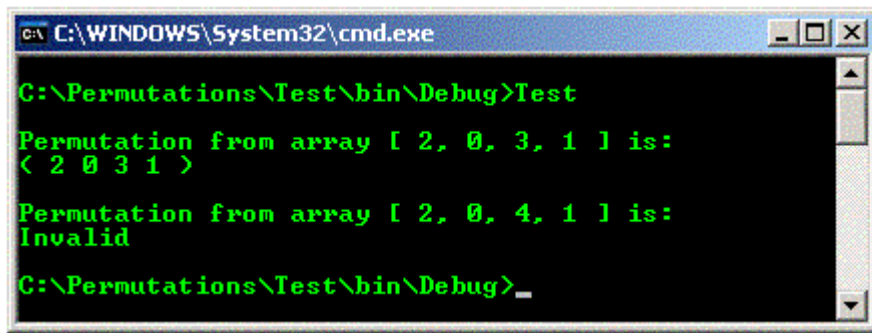
```

Permutation p2 = new Permutation(new int[] { 2, 0, 3, 1 });
Console.WriteLine("\nPermutation from array [ 2, 0, 3, 1 ] is:");
Console.WriteLine( (p2.IsValid() ? p2.ToString() : "Invalid") );

int[] arr = new int[] { 2, 0, 4, 1 };
Permutation p3 = new Permutation(arr);
Console.WriteLine("\nPermutation from array [ 2, 0, 4, 1 ] is:");
Console.WriteLine( (p3.IsValid() ? p3.ToString() : "Invalid") );

```

The corresponding output is shown in Figure 3. The constructor code that initializes a **Permutation** object from an **int[]** array is easy, and we even save a line of code by using the **CopyTo()** method instead of assigning values in a **for** loop.



```
C:\WINDOWS\System32\cmd.exe

C:\Permutations\Test\bin\Debug>Test
Permutation from array [ 2, 0, 3, 1 ] is:
< 2 0 3 1 >
Permutation from array [ 2, 0, 4, 1 ] is:
Invalid
C:\Permutations\Test\bin\Debug>_
```

Figure 3. Permutation constructed from array test

Checking if a **Permutation** object is valid or not can be done in several ways. There are two main things to check for. First, all the values in the **this.data[]** array must be in the range $0..n-1$, where n is the size of the array. Second, there can be no duplicate values. I could have sorted the **this.data[]** array and then iterated through, checking that the first cell had 0 and each successor cell was 1 greater. But this does more work than is necessary, so instead I used an auxiliary array of Booleans as a kind of minimal hash table to keep track of which values are in the **data[]** array.

More difficult than the validity algorithm itself is the decision on how to structure validity checking. Implementing a public **IsValid()** method as I did places the burden of validity checking on the calling program. A more robust strategy would be to create a custom **Exception** that deals with invalid **Permutation** objects.

The *k*th Permutation of Order *n*

The heart of the **Permutation** class is a constructor that creates the *k*th permutation of order *n*. This is a problem that is easy to state but very hard to answer.

What does it mean when we say the *k*th permutation? For example, the 0th permutation of order 6 is (0 1 2 3 4 5) and the 4th permutation is (0 1 2 5 3 4). There are a variety of ways that a set of permutations can be ordered. The most usual way is lexicographic ordering, sometimes called dictionary order. The easiest way to explain lexicographic ordering is with an example. The set of all permutations of order 3 in lexicographic order is:

(0 1 2)

(0 2 1)

(1 0 2)

(1 2 0)

(2 0 1)

(2 1 0)

For mathematical permutations, if each permutation is interpreted as an ordinary number in base 10, they are arranged in order from smallest to largest in magnitude. For the permutations above, the 0th (and first) permutation is (0 1 2) and the 5th (and last) permutation is (2 1 0).

Writing a method that returns the k th permutation of order n using "obvious" means is surprisingly difficult. However, I discovered a little-known algorithm dating back to the late 1800s that leads to an astonishingly elegant and efficient solution. The essence of the algorithm is to take k , compute its "factoradic," and then use that to compute its permutation.

You can think of a factoradic as an alternate representation of an integer. Let's consider the integer 859. It can be represented as

$$(8 * 100) + (5 * 10) + (9 * 1)$$

Or another way of looking at it is as based on a fixed radix (base) of powers of 10

$$(8 * 10^2) + (5 * 10^1) + (9 * 10^0)$$

The factoradic of an integer is its representation based on a variable base corresponding to the values of n factorial. It turns out that any integer can be uniquely represented in the form

$$(a_0 * 1!) + (a_1 * 2!) + (a_2 * 3!) + \dots = (a_0 * 1) + (a_1 * 2) + (a_2 * 6) + (a_3 * 24) + \dots \text{ where } 0 \leq a_k \leq k+1$$

For example, the integer 859 can be represented as

$$(1 * 1!) + (0 * 2!) + (3 * 3!) + (0 * 4!) + (1 * 5!) + (1 * 6!) = (1 * 1) + (3 * 6) + (1 * 120) + (1 * 720)$$

So we can represent 859 in factoradic form as $\{ 1 \ 1 \ 0 \ 3 \ 0 \ 1 \}$ where the right-most digit is the value of the 1!s. It will be useful for us to append a trailing 0 onto the right end of all factoradics so we get $\{ 1 \ 1 \ 0 \ 3 \ 0 \ 1 \ 0 \}$ as the final form.

Furthermore, it turns out that there is a one-to-one mapping between the factoradic of an integer k and the k th permutation of order n , meaning that each factoradic uniquely determines a permutation. To illustrate this, Table 1 shows the values of k , the factoradic of k , and the k th permutation for order 4.

Table 1. Factoradic and Permutations of Order 4

k	factoradic(k)	permutation(k)
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)

For $k = 5$, the factoradic is

$$\{ 0 \ 2 \ 1 \ 0 \} = (0 * 3!) + (2 * 2!) + (1 * 1!) + 0$$

The 5th permutation of order 4 is (0 3 2 1).

The clever and efficient way to derive the k th permutation of order n is to first find the factoradic of k and then to generate the corresponding permutation from the factoradic.

I implemented these two steps in a constructor **Permutation(int n, int k)** that creates the k th permutation object of order n . This code is listed in the [Constructing the kth Permutation](#) section. We can demonstrate the constructor using

```
Permutation p4 = new Permutation(4, 5);
Console.WriteLine("\nThe 5th permutation of order 4 is:");
Console.WriteLine(p4.ToString());
p4 = new Permutation(10, 999999);
Console.WriteLine("\nThe 999,999th permutation of order 10 is:");
Console.WriteLine(p4.ToString());
```

The resulting output is shown in Figure 4.

Constructing the k th Permutation

```
public Permutation(int n, int k)
{
    this.data = new int[n];
    this.order = this.data.Length;
```

```

// Step #1 - Find factoradic of k
int[] factoradic = new int[n];

for (int j = 1; j <= n; ++j)
{
    factoradic[n-j] = k % j;
    k /= j;
}

// Step #2 - Convert factoradic to permutation
int[] temp = new int[n];

for (int i = 0; i < n; ++i)
{
    temp[i] = ++factoradic[i];
}

this.data[n-1] = 1; // right-most element is set to 1.

for (int i = n-2; i >= 0; --i)
{
    this.data[i] = temp[i];
    for (int j = i+1; j < n; ++j)
    {
        if (this.data[j] >= this.data[i])
            ++this.data[j];
    }
}

for (int i = 0; i < n; ++i) // put in 0-based form
{
    --this.data[i];
}

} // Permutation(n,k)

```

```

C:\WINDOWS\System32\cmd.exe
C:\Permutations\Test\bin\Debug>Test
The 5th permutation of order 4 is:
< 0 3 2 1 >
The 999,999th permutation of order 10 is:
< 2 7 8 3 9 1 5 4 6 0 >
C:\Permutations\Test\bin\Debug>_

```

Figure 4. *k*th permutation of order *n* test

Computing the factoradic of an integer is very much like determining a fixed base representation. After creating an **int[]** array to hold the factoradic, the loop

```

for (int j = 1; j <= n; ++j)
{
    factoradic[n-j] = k % j;
    k /= j;
}

```

does all the work. On each pass, the remainder is calculated using the modulus (%) operator and stored in the right-most available cell (*n-j*) of the array. Then *k* is reduced by division (*k /= j*); in effect changing the base for the next pass by doing a reverse factorial calculation.

The trickiest part of the algorithm is the computation of the permutation that corresponds to the factoradic. Let's look at how the algorithm converts the factoradic { 1 2 3 2 1 1 0 } into its corresponding permutation. We first create a **temp[]** array and copy into it the factoradic values incremented by 1:

[2 3 4 3 2 2 1]

We seed the right-most cell of the result **data[]** array with 1:

[? ? ? ? ? 1]

Now starting with the second value from the right-most value (we skip over the right-most because it will always be 1 since it came from the padded 0 value), we add it to the **data[]** array:

[? ? ? ? 2 1]

Now we scan through all the values to the right of the new value and increment by 1 all values that are greater than or equal to the new value.

Continuing this process generates:

[? ? ? 2 3 1]

[? ? 3 2 4 1]

[? 4 3 2 5 1]

```
[ 3 5 4 2 6 1 ]
```

```
[ 2 4 6 5 3 7 1 ]
```

Last we traverse the **data[]** array and decrement all values by 1 to put the resulting permutation in 0-based form:

```
( 1 3 5 4 2 6 0 )
```

To summarize, if we want to generate the *k*th permutation of order *n*, first we compute the factoradic of *k*, and then use that result to compute the corresponding permutation. In the example above, we started with *k* = 1,047 and then computed its factoradic = { 1 2 3 2 1 1 0 } and then computed the permutation (1 3 5 4 2 6 0). So the 1047th permutation of order 7 is (1 3 5 4 2 6 0).

The Successor of a Given Permutation

A key method when using permutations is one that returns the successor permutation of a given permutation in lexicographic order.

I implemented a **Successor()** method shown in the [Permutation Successor\(\) Method](#) section. We can demonstrate its use with:

```
Permutation p5 = new Permutation(new int[] {2,1,3,5,4,0});  
Console.WriteLine("\nThe successor to ( 2 1 3 5 4 0 ) is");  
Console.WriteLine(p5.Successor().ToString());
```

```
p5 = new Permutation(3);  
Console.WriteLine("\nAll permutations of order 3 are:");  
Console.WriteLine(p5.ToString());  
while ( (p5 = p5.Successor()) != null )  
    Console.WriteLine(p5.ToString());
```

The resulting output is shown below. Notice that the ability to generate the lexicographic successor of a given permutation gives us an easy way to generate all permutations.

Permutation Successor() Method

```
public Permutation Successor()  
{  
    Permutation result = new Permutation(this.order);  
  
    int left, right;  
  
    for (int k = 0; k < result.order; ++k) // Step #0 - copy current data into result  
    {  
        result.data[k] = this.data[k];  
    }  
  
    left = result.order - 2; // Step #1 - Find left value  
    while ((result.data[left] > result.data[left+1]) && (left >= 1))  
    {  
        --left;
```

```

}
if ((left == 0) && (this.data[left] > this.data[left+1]))
    return null;

right = result.order - 1; // Step #2 - find right; first value > left
while (result.data[left] > result.data[right])
{
    --right;
}

int temp = result.data[left]; // Step #3 - swap [left] and [right]
result.data[left] = result.data[right];
result.data[right] = temp;

int i = left + 1;           // Step #4 - order the tail
int j = result.order - 1;

while (i < j)
{
    temp = result.data[i];
    result.data[i++] = result.data[j];
    result.data[j--] = temp;
}

return result;
} // Successor()

```

```

C:\WINDOWS\System32\cmd.exe
C:\Permutations\Test\bin\Debug>Test
The successor to < 2 1 3 5 4 0 > is
< 2 1 4 0 3 5 >

All permutations of order 3 are:
< 0 1 2 >
< 0 2 1 >
< 1 0 2 >
< 1 2 0 >
< 2 0 1 >
< 2 1 0 >
< 2 1 0 >

C:\Permutations\Test\bin\Debug>_

```

Figure 5. Permutation successor test

The algorithm I used to generate the successor is straightforward even if it isn't as elegant as the *k*th permutation code. Essentially we locate two values to swap, swap them, and then shuffle the tail that is to the right of the swap position.

The only tricks involve finding the two swap positions that I've called "left" and "right" in the code. I'll demonstrate how the algorithm finds the successor to (2 1 3 5 4 0). To find position "left," we start with an index at the second value from the right and move left until the value at index+1 is greater than that at index. In this example "left" stops when it points to the 3 in the permutation. To find position "right," we start with an index at the right-most value and move left until we find a value that is greater than that pointed to by "left." In this example, "right" stops when it points to the 4. Now we swap getting an intermediate result of

(2 1 4 5 3 0)

Finally we perform a shuffle of the values between "left" and the right-end to get the successor permutation:

(2 1 4 0 3 5)

The **Successor()** method returns null when applied to the last permutation of a particular order. Although I could have checked to see if the permutation is in the form (n-1 n-2 . . . 0), it is easier to observe that this state will occur when index "left" walks all the way down to the data element at index 0.

There are many well-known algorithms that return the lexicographic successor of a permutation, and the one I used is nothing new except in the details. An alternative that merits investigation is to take the given permutation, reverse compute its factoradic, reverse compute the corresponding k from the factoradic, add 1 to k , compute the factoradic of $k+1$, and then compute the corresponding permutation. This approach would have the advantage of making the implementation of a **Predecessor()** method symmetric and easy.

Block Ciphers

Now that we have the foundations of a **Permutation** class, I will show you some applications centered on security. Block ciphers are encryption schemes, such that the encryption of every plaintext block is a ciphertext block of the same length. A highly influential example of a block cipher is the Data Encryption Standard (DES).

The Data Encryption Standard (DES) was approved as a Federal Information Processing Standard (FIPS) in 1977. Although DES has been recently replaced by the Advanced Encryption Standard (AES), DES will be around for many years.

An advantage of block ciphers over other techniques is that, because the plaintext and ciphertext blocks are the same size, block ciphers are efficient in terms of communication bandwidth, and they integrate easily into existing software protocols and hardware components.

The two primary techniques of encrypting a block of information are substitution—replacing a symbol with another—and permutation. In security literature, substitution and permutation are sometimes called confusion and diffusion, respectively. Although neither technique works well alone, together they form the backbone of modern cryptosystems.

Let's look at how we can permute, or diffuse, a block of bits. I wrote **ApplyTo()** and **Inverse()** methods as shown in the following code. Some sample code that calls them is shown under the [Applying a Permutation](#) section and the resulting output is shown in Figure 6.

ApplyTo() and Inverse() Methods

```
public object[] ApplyTo(object[] arr)
```

```

{
    if (arr.Length != this.order)
        return null;

    object[] result = new object[arr.Length];
    for (int i = 0; i < result.Length; ++i)
    {
        result[i] = arr[this.data[i]];
    }

    return result;
} // ApplyTo()

public Permutation Inverse()
{
    int[] inverse = new int[this.order];

    for (int i = 0; i < inverse.Length; ++i)
    {
        inverse[this.data[i]] = i;
    }

    return new Permutation(inverse);
} // Inverse()

```

Applying a Permutation

```

Permutation p6 = new Permutation(8, 1395);
Console.WriteLine("\nThe permutation is:");
Console.WriteLine(p6.ToString());

object[] bits = new object[] { "1", "0", "1", "0", "1", "0", "1", "0" };

Console.WriteLine("\nBits before permutation:");
for (int i = 0; i < 8; ++i)
{
    Console.Write(bits[i].ToString());
}
Console.WriteLine("");

bits = p6.ApplyTo(bits);

```

```

Console.WriteLine("\nBits after permutation:");
for (int i = 0; i < 8; ++i)
{
    Console.Write(bits[i].ToString());
}
Console.WriteLine("");

bits = p6.Inverse().ApplyTo(bits);

Console.WriteLine("\nBits after inverting the permutation:");
for (int i = 0; i < 8; ++i)
{
    Console.Write(bits[i].ToString());
}
Console.WriteLine("");

```

The screenshot shows a Windows command prompt window titled "C:\WINDOWS\System32\cmd.exe". The command prompt is at the directory "C:\Permutations\Test\bin\Debug". The user has entered the command "Test". The output is as follows:

```

C:\Permutations\Test\bin\Debug>Test

The permutation is:
< 0 2 7 5 1 4 6 3 >

Bits before permutation:
10101010

Bits after permutation:
11000110

Bits after inverting the permutation:
10101010

C:\Permutations\Test\bin\Debug>

```

Figure 6. Applying a permutation test

The **ApplyTo()** method showcases one of the many advantages of the .NET Framework by allowing us to write one method that can handle an array of anything instead of having to resort to templates or multiple overloading. **ApplyTo()** effectively rearranges the objects in its input parameter array according to the values in the **Permutation** object. In the example above, the bit array is simulated using strings as

{ 1 0 1 0 1 0 1 0 }

[0 1 2 3 4 5 6 7]

The permutation is

(0 2 7 5 1 4 6 3)

[0 1 2 3 4 5 6 7]

where I've put the indexes below the bit array and permutation atoms.

We've been thinking of mathematical permutations as an ordered set of integers, but they can also be interpreted as rearrangement instructions. When thought of in this way, it is usual to put the indexes below the permutation atoms. **ApplyTo()** interprets the permutation as instructions to "put what is in cell 0 of the bits into cell 0 of the result, put what is in cell 2 of the bits into cell 1 of the result, put what is in cell 7 of the bits into cell 2 of the results," and so forth.

The **Inverse()** method returns a **Permutation** object that will undo any rearrangements indicated by the original **Permutation**.

Performing rudimentary plaintext block cipher diffusion will by no means provide your code with security in any formal sense, but permutations are an excellent way to add obfuscation to your systems. Permutations can be applied anywhere from the bit level all the way up to the string level. For example, in any of my applications that have information that travels via HTTP, I always add permutation of the information to remove the "low-hanging fruit," so to speak. Permutation will not defeat a determined attempt to reveal data, but it will often persuade the attacker to look for easier pickings.

It is possible to greatly increase the effectiveness of block cipher permutation schemes by adding rotations. A rotation of a permutation is the original permutation, but with all atoms rotated to the right (or left). For example, if $p = (3 \ 1 \ 0 \ 2)$, its right rotation is $(2 \ 3 \ 1 \ 0)$. Instead of applying the same permutation to each block of plaintext, you can apply successive rotations to each block of plaintext in the input. Again, this will not create a truly secure system, but will help guard against rudimentary attacks. I will leave it to you to implement a **Rotated()** method.

Programmatic Generation of Test Cases

Another security-related area where permutations are useful is programmatic generation of test cases. We've all seen examples of security flaws in systems that might have been found with more thorough testing. Even when the testing effort uses test automation, most of the time the test cases themselves are manually created. This introduces a practical limit on the number of test cases that a system can be exposed to.

One solution to this problem is to supplement manually created test cases for test automation with programmatically generated test cases. For example, I remember working on a system that accepted multiple words as input. The input string could consist of an arbitrarily large number of words. After I created the test automation, I then spent months generating test cases manually. It was painful. A more efficient strategy would have been to supplement the test cases with programmatically generated permutations of the input words.

Let's suppose that some system accepts a string of the four color words "aqua," "blue," "cyan," and "darkgreen" separated by white space as input. For example, "cyan blue darkgreen aqua." Let's suppose that the system compares the first and third words and returns the lower word alphabetically unless the fourth word is "blue," in which case the correct return value is "darkgreen." A manual generation of the $4! = 24$ test cases in a flat text file might look something like:

000001:aqua:blue:cyan:darkgreen:aqua

000002:aqua:blue:darkgreen:cyan:aqua

000003:aqua:cyan:blue:darkgreen:aqua

000004:aqua:cyan:darkgreen:blue:darkgreen

etc.

The first field separated by the ':' character is a test case ID, the next four fields are the inputs, and the last field is the expected result. Even with this artificially simple example, it would not be pleasant to manually create the test case input.

The following code shows how we can use **Permutation** objects to easily generate a file of test case information programmatically. The output from running the code is shown in the [Automatically Generated Test Case Data](#) section.

Code to Programmatically Generate Test Cases

```
Console.WriteLine("\nCreating file testcases.txt");
FileStream fs = new FileStream("testcases.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
int id = 1;

object[] colors = new object[] { "aqua", "blue", "cyan", "darkgreen" };
object[] casedata = new object[colors.Length];

Permutation p7 = new Permutation(colors.Length);

while ( p7 != null )
{
    sw.Write(id++.ToString().PadLeft(6, '0') + ":");
    casedata = p7.ApplyTo(colors);
    sw.Write(casedata[0].ToString() + ":" + casedata[1].ToString() + ":");
    sw.Write(casedata[2].ToString() + ":" + casedata[3].ToString() + ":");

    if (casedata[3].ToString() == "blue")
        sw.WriteLine("darkgreen");
    else if ( casedata[0].ToString().CompareTo(casedata[2].ToString()) < 0 )
        sw.WriteLine(casedata[0].ToString());
    else
        sw.WriteLine(casedata[2].ToString());

    p7 = p7.Successor();
}

sw.Flush();
sw.Close();
fs.Close();
```

Automatically Generated Test Case Data

```
000001:aqua:blue:cyan:darkgreen:aqua
000002:aqua:blue:darkgreen:cyan:aqua
000003:aqua:cyan:blue:darkgreen:aqua
```

```

000004:aqua:cyan:darkgreen:blue:darkgreen
000005:aqua:darkgreen:blue:cyan:aqua
000006:aqua:darkgreen:cyan:blue:darkgreen
000007:blue:aqua:cyan:darkgreen:blue
000008:blue:aqua:darkgreen:cyan:blue
000009:blue:cyan:aqua:darkgreen:aqua
000010:blue:cyan:darkgreen:aqua:blue
000011:blue:darkgreen:aqua:cyan:aqua
000012:blue:darkgreen:cyan:aqua:blue
000013:cyan:aqua:blue:darkgreen:blue
000014:cyan:aqua:darkgreen:blue:darkgreen
000015:cyan:blue:aqua:darkgreen:aqua
000016:cyan:blue:darkgreen:aqua:cyan
000017:cyan:darkgreen:aqua:blue:darkgreen
000018:cyan:darkgreen:blue:aqua:blue
000019:darkgreen:aqua:blue:cyan:blue
000020:darkgreen:aqua:cyan:blue:darkgreen
000021:darkgreen:blue:aqua:cyan:aqua
000022:darkgreen:blue:cyan:aqua:cyan
000023:darkgreen:cyan:aqua:blue:darkgreen
000024:darkgreen:cyan:blue:aqua:blue

```

The code that generates test cases is straightforward. An identity permutation of the appropriate order is created. The **while** loop iterates through all possible permutations by calling the **Successor()** method and checking for null to indicate when finished. The **ApplyTo()** method generates the correct permutation of inputs in lexicographic order. Then we apply some logic to determine the expected result and write to a file.

In a realistic example, you would likely write the test case data to a SQL database or an XML file, of course. And the determination of the expected result would certainly be the most time-consuming task you face. Programmatic generation of test cases is not always feasible, but the ability to use this technique can increase the security of your system by greatly increasing the test coverage.

Random Permutations

A problem that is closely related to programmatic generation of test cases is the generation of random test cases. This is useful when creating all possible permutations is not practical. Within the context of a **Permutation** object, if we can generate a random permutation, then we can apply it to an input set to get a random test case.

An easy way to do this is to combine the **Permutation** constructor that creates the *kth* permutation of order *n* with the **System.Math.Random** class. Suppose for example we wish to generate a random permutation of the set { "ant", "bear", "cat", "duck" }. It is simple to generate a random permutation of the set like this:

```

object[] animals = new object[] { "ant", "bear", "cat", "duck" };
int maxValue = Methods.Factorial(animals.Length) - 1;
System.Random r = new System.Random(0);

```

```
Permutation p8 = new Permutation(animals.Length, r.Next(maxValue));
animals = p8.ApplyTo(animals);
foreach (object o in animals)
    Console.Write(o.ToString() + " " );
Console.WriteLine("");
```

The **Permutation** constructor that creates the k th permutation of order n needs to know the order and a value for k . The order is just the number of items in the set to permute; in this case, the **animals.Length** property equals 4. Because there are a total of $n!$ permutations for a list of n items, we compute the value of **(animals.Length)!** and then subtract 1 because we are 0-based. In this case, we get $4! - 1 = 23$ and assign it to a variable **maxValue**. A **System.Random** object is constructed using an arbitrary seed of 0, and then we generate a random integer between 0 and **maxValue** by invoking the **Random.Next()** method. This generates one of the 24 possible permutations of order 4 randomly, which is applied to the **animals[]** array using the **ApplyTo()** method.

The technique just described is fairly standard in the literature for mathematical permutations, but has a pragmatic limitation. Notice that the technique requires the calculation of $n!$ as a preliminary step. For the C# data type **int**, the largest value is only +2,147,483,648, so a **Factorial()** method that returns an **int** can only calculate up to 12! because 13! is greater than **int.MaxValue**. Even using data type **ulong** will only allow a maximum value of 20!, or in other words, the permutation of at most 20 items.

An alternate technique to generate random permutations of more than 20 items is to generate an identity permutation of order n and then repeatedly select 2 indexes between 0 and $n-1$ and swap the values at those indexes. This would not necessarily generate a random permutation in the mathematical sense, but it would generate a shuffled permutation that would be suitable for many security-testing purposes.

Conclusion

In this article, I present the algorithms and code for a basic **Permutation** class. Permutations are the backbone of all modern cryptosystems and I demonstrate a few ways that you can use them to increase the security of your code and enhance the security-related testing of your system.

Although the code in this article is usable as is, you could extend it into a full-fledged **Permutation** class by adding error handling and custom **Exceptions** and other methods such as a **Permutation.ComposeWith(Permutation p)** that returns the composition of two permutations. The .NET development environment is still young, and it does not yet have large scientific libraries.

Security is no longer an afterthought of the development process. By incorporating security from the earliest stages of product design, you can avoid recurring security nightmares after product release. The ability to understand and use permutations is an easy and effective way to increase the security of your systems.

Related Articles

[Defend Your Code with Top Ten Security Tips Every Developer Must Know](#)

[Protect Private Data with the Cryptography Namespaces of the .NET Framework](#)

Background Information

"Permutation Generation Methods," Sedgewick, Robert. Computing Surveys, Volume 9, Number 2, June 1977.

"Standing the Test of Time: The Data Encryption Standard," Landau, Susan. Notices of the ACM, March 2000.

About the Author

Dr. James McCaffrey works for Volt Information Sciences, Inc., where he manages technical training for software engineers working at the Redmond, Washington campus of Microsoft. He has worked on several Microsoft products including Internet Explorer and MSN Search. James can be reached at jmccaffrey@volt.com.

Acknowledgment

I am grateful to Dr. Peter Cameron of Queen Mary, University of London for suggesting that the factoradic of a number can be use to generate the k th permutation of order n , and to Dr. Kurt Bryan of Rose-Hulman Institute of Technology for reviewing the mathematics in this article.