Generating the mth Lexicographical Element of a Mathematical Combination

James McCaffrey
Volt Information Sciences, Inc

July 2004

Summary: Presents an elegant algorithm to generate an arbitrary mathematical combination element from a given lexicographical index, and explains how this algorithm can benefit you developer skills. (11 printed pages)

Download the combinations.msi file.

Contents

Introduction
The Combinadic of an Integer
The Combination.Element() Method
Discussion
References

Introduction

In this article I will show you an elegant algorithm to generate an arbitrary mathematical combination element from a given lexicographical index. Huh? If you stick with me, I will explain exactly what that means. I think you'll find the algorithm quite interesting and potentially a valuable addition to your developer skill set.

I recently wrote an article in *MSDN Magazine* (http://msdn.microsoft.com/msdnmag/issues/04/07/TestRun/default.aspx) that describes mathematical combinations, implements a Combination class with C#, and explains how combinations can be useful for software test automation. Combinations have an enormous number of uses in a wide range of areas beyond software testing. This short article essentially extends the ideas and code in the MSDN Magazine article. In particular, I will describe a method **Combination.Element()** that accepts as input an index value and returns the appropriate **Combination** object.

The best way to start the explanation is with a screen shot. Figure 1 shows the output resulting from a console application that demonstrates where I'm headed. The code that generated the output in Figure 1 is listed in the code snipped below.

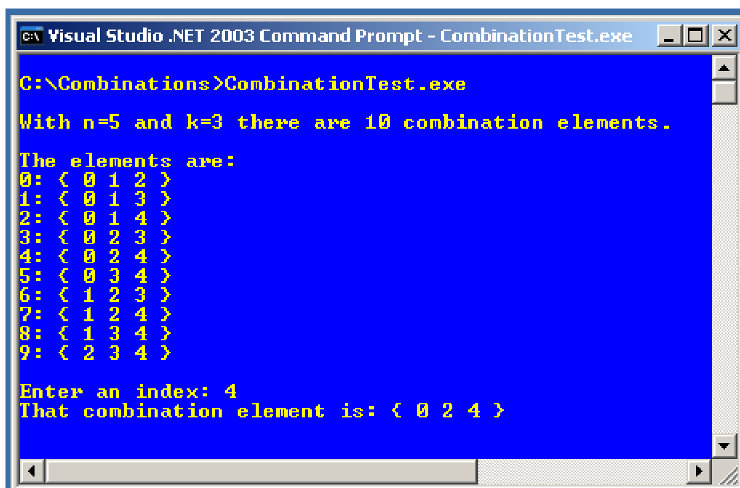

Figure 1. Combinations demonstration

A mathematical combination is a subset of size k from a set of integers from 0 to n-1, where order does not matter. In the program that I ran to get the output shown in Figure 1, the value of n is set to 5 and the value of k is set to 3. When listed in lexicographical order, the first combination element is { 0 1 2 } and the last element is { 2 3 4 }. When using lexicographical order, combination elements increase in magnitude if you think of the elements as numbers. Notice that I don't list an element like { 1 0 2 } because it is the same as { 0 1 2 }.

```
Combination c = new Combination(5,3);
```

```
Console.WriteLine("\nWith n=5 and k=3 there are " + Combination.Choose(5,3) + " combination e
lements.");

Console.WriteLine("\nThe elements are:");
for (int i = 0 ; i < 10; ++i)
{
  Console.WriteLine(i + ": " + c.ToString());
  c = c.Successor();
}

c = new Combination(5,3);
Console.Write("\nEnter an index: ");
int m = int.Parse(Console.ReadLine());
Console.WriteLine("That combination element is: " + c.Element(m));
```

If you look at the previous code block, you'll see that I call a method **Combination.Element()** that returns a combination element for a specified lexicographical index. The purpose of this article is to describe the **Element()** method. The code above also shows that I use a method **Choose(n,k)**, which returns the total number of combination elements for given n and k values, and a **Successor()** method that returns the next lexicographical combination element for a given element. The combination constructor, **Choose()** and **Successor()** methods, are explained in detail in the MSDN article at http://msdn.microsoft.com/msdnmag/issues/04/07/TestRun/default.aspx, but I have re-listed their code in the code block below for your reference.

```
public class Combination
{
  private long n = 0;
  private long k = 0;
  private long[] data = null;

  public Combination(long n, long k)
  {
    if (n < 0 || k < 0) // normally n >= k
      throw new Exception("Negative parameter in constructor");

    this.n = n;
    this.k = k;
    this.data = new long[k];
    for (long i = 0; i < k; ++i)
      this.data[i] = i;
  } // Combination(n,k)

  public Combination(long n, long k, long[] a) // Combination from a[]
  {
    if (k != a.Length)
      throw new Exception("Array length does not equal k");

    this.n = n;
    this.k = k;
    this.data = new long[k];
    for (long i = 0; i < a.Length; ++i)
      this.data[i] = a[i];

    if (!this.IsValid())
      throw new Exception("Bad value from array");
    } // Combination(n,k,a)

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

    for (long i = 0; i < this.k; ++i)
    {
      if (this.data[i] < 0 || this.data[i] > this.n - 1)
        return false; // value out of range
```

```
      for (long j = i+1; j < this.k; ++j)
        if (this.data[i] >= this.data[j])
          return false; // duplicate or not lexicographic
    }

    return true;
  } // IsValid()

  public override string ToString()
  {
    string s = "{ ";
    for (long i = 0; i < this.k; ++i)
      s += this.data[i].ToString() + " ";
    s += "}";
    return s;
  } // ToString()

  public Combination Successor()
  {
    if (this.data[0] == this.n - this.k)
      return null;

    Combination ans = new Combination(this.n, this.k);

    long i;
    for (i = 0; i < this.k; ++i)
      ans.data[i] = this.data[i];

    for (i = this.k - 1; i > 0 && ans.data[i] == this.n - this.k + i; --i)
      ;

    ++ans.data[i];

    for (long j = i; j < this.k - 1; ++j)
      ans.data[j+1] = ans.data[j] + 1;

    return ans;
  } // Successor()

  public static long Choose(long n, long k)
  {
    if (n < 0 || k < 0)
      throw new Exception("Invalid negative parameter in Choose()");
    if (n < k)
      return 0;  // special case
    if (n == k)
      return 1;

    long delta, iMax;

    if (k < n-k) // ex: Choose(100,3)
    {
      delta = n-k;
      iMax = k;
    }
    else         // ex: Choose(100,97)
    {
      delta = k;
      iMax = n-k;
    }

    long ans = delta + 1;

    for (long i = 2; i <= iMax; ++i)
    {
      checked { ans = (ans * (delta + i)) / i; }
    }
```

```
    return ans;
  } // Choose()

} // Combination class
```

Let me explain why a **Combination.Element()** method is useful. An obvious technique to get the **mth** element is to start with the first element and then iterate, calling a successor method or code, m times. A typical implementation of this naive approach might look like:

```
// standard but naive technique to generate the mth element
Console.WriteLine("Element [4] of a combination with n=5 and k=3 is: ");
Combination c = new Combination(5,3);
long m = 4;
for (long i = 1; i <= m; ++i)
{
  c = c.Successor();
}
Console.WriteLine(c.ToString());
```

This implementation yields the correct combination element { 0, 2, 4 }. So, what is the problem? It's not hard for you to guess that this technique is bad when the value of m is large. And unfortunately m is often very, very large. For example, if we have a combination of n = 200 items taken k = 10 at a time, there are 22,451,004,309,013,280 possible elements. Using the naive looping technique above, I calculated **element [999,999,999,999]** for n = 200 and k = 10 on a reasonably fast desktop machine and it took over 100 hours—not very good performance.

I discovered a very helpful mathematical idea that I call the *combinadic* of a number, which allows me to code an **Element()** method that calculates the [999,999,999,999] element for n = 200 and k = 10 in approximately 1 second.

The Combinadic of an Integer

The combinadic of an integer is an alternative representation of the number based on combinations that maps nicely to a combination element. Consider for example the number 27. If we fix n = 7 and k = 4, it turns out that the combinadic of 27 is (6 5 2 1). This means that:

```
27 = Choose(6,4) + Choose(5,3) + Choose(2,2) + Choose(1,1).
```

With n = 7 and k = 4, any number z between 0 and 34 (the total number of combination elements for n and k) can be uniquely represented as:

```
z = Choose(c1,4) + Choose(c2,3) + Choose(c3,2) + Choose(c4,1)
```

This representation is where n > c1 > c2 > c3 > c4. Notice that n is analogous to the base because all combinadic digits are between 0 and n-1 (just like all digits in ordinary base 10 are between 0 and 9). The k value determines the number of terms in the combinadic. The combinadic of a number can be calculated fairly quickly.

Let me show you a second example of the combinadic of a number, but this time I'll show you how to calculate it. Suppose we have the integer 8, with n and k set to 7 and 4 respectively, and we want the combinadic. The combinadic of 8 will have the form:

```
8 = Choose(c1,4) + Choose(c2,3) + Choose(c3,2) + Choose(c4,1)
```

We first determine the value of c1. We try 6 (the largest number less than n = 7) and get **Choose(6,4)** = 15, which is too much because we're over 8. Next, we try 5 and get **Choose(5,4)** = 5, which is less than 8, so bingo, c1 = 5. Now we have used up 5 of the original number 8 so we have 3 left to account for. To determine the value of c2, we try 4 (the largest number less than the 5 we got for c1), but get **Choose(4,3)** = 4, which is barely too much. Working down we get to 3 and **Choose(3,3)** = 1, so c2 = 3. We used up 1 of the remaining 3 we had to account for, so we have 2 left to consume. Using the same ideas we'll get c3 = 2 with **Choose(2,2)** = 1, so we have 1 left to account for. And then we'll find that c4 = 1 because **Choose(1,1)** = 1. Putting our four c values together we conclude that the combinadic of 8 with n = 7 and k = 4 is (5 3 2 1).

Well, this is all rather strange and perhaps somewhat interesting but what makes the combinadic useful is that the combinadic of a number indirectly maps to its lexicographic element and gives us a way to quickly compute an arbitrarily specified combination element.

The Combination.Element() Method

Before I show you how to use the combinadic of a number to determine the mth lexicographical element of a combination, I need to explain the idea of the dual of each lexicographic index. Suppose n = 7 and k = 4. There are **Choose(7,4)** = 35 combination elements, indexed from 0 to 34. The dual indexes are the ones on opposite ends so to speak: indexes 0 and 34 are duals, indexes 1 and 33 are duals, indexes 2 and 32, and so forth. Notice that each pair of dual indexes sum to 34, so if we know any index it is easy to find its dual.

Now, continuing the first example above for the number 27 with n = 7 and k = 4, suppose we are able to find the combinadic of 27 and get (6 5 2 1). Now suppose we subtract each digit in the combinadic from n-1 = 6 and get (0 1 4 5). Amazingly, this gives us the combination element [7], the dual index of 27! Putting these idea together we have an elegant algorithm to determine an arbitrarily specified combination element for given n and k values. To find the combination element for index m, first find its dual and call it x. Next, find the combinadic of x. Then subtract each digit of the combinadic of x from n-1 and the result is the mth lexicographic combination element. The table below shows the relationships among m, the dual of m, **Comination.Element(m)**, the combinadic of m, and (n-1) - ci for n=5 and k=3.

```
m  dual(m)  Element(m)  combinadic(m)  (n-1) - ci
==================================================
0    9      { 0 1 2 }   ( 2 1 0 )      ( 2 3 4 )
1    8      { 0 1 3 }   ( 3 1 0 )      ( 1 3 4 )
2    7      { 0 1 4 }   ( 3 2 0 )      ( 1 2 4 )
3    6      { 0 2 3 }   ( 3 2 1 )      ( 1 2 3 )
4    5      { 0 2 4 }   ( 4 1 0 )      ( 0 3 4 )
5    4      { 0 3 4 }   ( 4 2 0 )      ( 0 2 4 )
6    3      { 1 2 3 }   ( 4 2 1 )      ( 0 2 3 )
7    2      { 1 2 4 }   ( 4 3 0 )      ( 0 1 4 )
8    1      { 1 3 4 }   ( 4 3 1 )      ( 0 1 3 )
9    0      { 2 3 4 }   ( 4 3 2 )      ( 0 1 2 )
```

With all the parts of the puzzle in place, I implemented the **Element()** method using C# as shown in the following code snippet:

```
// return the mth lexicographic element of combination C(n,k)
public Combination Element(long m)
{
  long[] ans = new long[this.k];

  long a = this.n;
  long b = this.k;
  long x = (Choose(this.n, this.k) - 1) - m; // x is the "dual" of m

  for (long i = 0; i < this.k; ++i)
  {
    ans[i] = LargestV(a,b,x); // largest value v, where v < a and vCb < x
    x = x - Choose(ans[i],b);
    a = ans[i];
    b = b-1;
  }

  for (long i = 0; i < this.k; ++i)
  {
    ans[i] = (n-1) - ans[i];
  }

  return new Combination(this.n, this.k, ans);
} // Element()


// return largest value v where v < a and  Choose(v,b) <= x
private static long LargestV(long a, long b, long x)
{
  long v = a - 1;
```

```
    while (Choose(v,b) > x)
      --v;

    return v;
  } // LargestV()
```

Let's walk through how the **Element()** method calculates and returns the [6] lexicographic combination element for n = 7 and k = 4, which is { 0, 1, 3, 6 }. **Element()** starts by creating an array of size 4 to hold the digits of our answer. Next, we compute the dual index of the input parameter m, making use of the fact that the duals sum to **Choose(n,k)** - 1. Because m is 6, the dual index x is 34 - 6 = 28. Now we will compute the combinadic of 28. Most of the work is done with a strange little helper function, **LargestV()**. We know the basic structure of the combinadic of 28 will be (c1, c2, c3, c4) where:

```
28 = Choose(c1,4) + Choose(c2,3) + Choose(c3,2) + Choose(c4,1)
```

So, we need to find values c1, c2, c3, and c4. The **LargestV(a,b,x)** function returns the largest value v that is less than a given value a, and so that **Choose(v,b)** is less than or equal to x. Believe me, this made my head spin when I was first working out the details. To compute c1, we call **LargestV(7,4,28)**, the largest value v less than 7, so that **Choose(v,4)** is less than or equal to 28. In this case, **LargestV()** returns 6 because **Choose(6,4)** = 15, which is less than 28. The value 6 is the first number c1, of the combinadic.

Now to compute the c2 value, we subtract 15 from 28 and see that we now only have 13 left to consume because we used up 15 for the c1 coefficient. We call **LargestV(6,3,13)**, which returns 5 and note that **Choose(5,3)** is 10, leaving us with 3. The combinadic is now (6 5 ? ?). Next, we call **LargestV(4,2,10)** and get 3 for c3, noting that **Choose(3,2)** is 3, leaving us with 0 left. Finally, to compute c4, we call **LargestV(3,1,0)**, which returns 0.

Now that we have the combinadic (6 5 3 0) in our answer array, we map it to a combination element by subtracting each of the combinadic values from n-1 = 6, giving us (0 1 3 6). Finally, we feed our answer array to our auxiliary constructor to convert it into a combination object and get { 0, 1, 3, 6 }—combination element [6] in lexicographical order for n = 7 and k = 4. Whew!

Notice that the **LargestV(a,b,x)** function calls the **Choose(n,k)** method in such a way that n can be less than k. This is why we allow this possibility in the **Choose()** method, and also in the Combination constructor.

To summarize, an important fundamental operation on combinations is generating the mth lexicographic combination element for given n and k values. A naive approach that iterates from the first element up to the mth element is not always effective because the number of elements can be very large. I discovered that the little-known combinadic of a number could be used to quickly generate the mth combination element.

Discussion

The idea of the combinadic of a number is closely related to the *factoradic* of a number. In an MSDN article (http://msdn.microsoft.com/library/enus/dnnetsec/html/permutations.asp) on mathematical permutations (all possible orderings of a set of integers from 0 to n-1), I describe how the factoradic of a number is a representation based on factorials. For example, the factoradic of integer value 53 is (2 0 1 3) because:

```
53 = (2 * 4!)  +  (0 * 3!)  +  (1 * 2!)  +  (3 * 1!)
```

The factoradic can be used to generate the mth permutation element for a given n in a way that is analogous to the way the combinadic can be used to generate the mth combination element. I have observed a general theme that when you are trying to map from an index value to a vector value, alternate number representations are often the key to an elegant solution.

Although the combinadic can calculate the mth combination element much faster than an iterative solution for large values of m, my **Element()** method was not coded for optimum speed. The algorithm I implemented in this article was designed for clarity rather than performance. In particular, I used a crude way to find the values of the combinadic. As you may recall, I started at n-1, computed **Choose(n-1,k)** and if that result was greater than m, then I tried 1 less, and so forth until I found a value for the combinadic. For large values of n, a binary search approach could improve performance. I could also have improved performance by recasting the **Element()** method to eliminate calls to the **Choose()** and **LargestV()** helper methods.

While researching this article, I discovered that B. P. Buckles and M. Lybanon published an article and algorithm titled, "Algorithm 515: Generation of a Vector from the Lexicographical Index" in the June, 1977 issue of ACM Transactions on Mathematical Software. I was not able to access the article online because it required a subscription. The article implements an alternative, but

copyrighted algorithm in Fortran.

Acknowledgements

I am grateful to Laci Lovasz (Microsoft Research) for suggesting that the combinadic of a number maps to a combination element; to Chris Sells (MSDN) for several helpful observations which got me started on the right track; and to Stephen Toub (MSDN) for some valuable observations on the performance of the combinadic algorithm.

References

- "Algorithm 515: Generation of a Vector from the Lexicographical Index"; Buckles, B. P., and Lybanon, M. ACM Transactions on Mathematical Software, Vol. 3, No. 2, June 1977.
- "Using Permutations in .NET for Improved Systems Security"; McCaffrey, James. http://msdn.microsoft.com/library/enus/dnnetsec/html/permutations.asp
- "The Art of Computer Programming, 2nd Ed."; Knuth, Donald. Addison-Wesley, 1998.
- "Permutation Generation Methods"; Sedgewick, Robert. Computing Surveys, Volume 9, Number 2, June 1977.

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