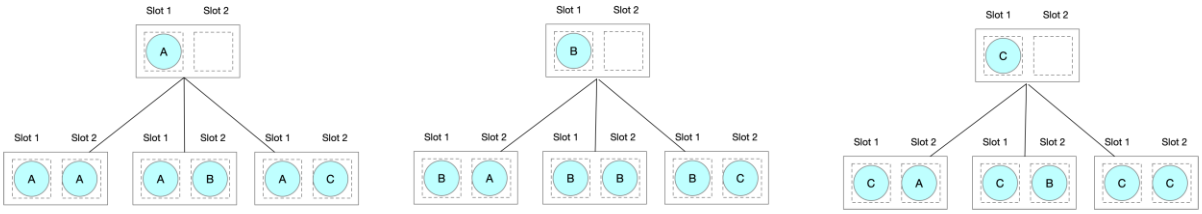
# K-Strings

## Description

Consider the set $S = \{A, B, C\}$ If we use this set to form two-character strings we have the following situation For each of the three possible values of the first slot we have 3 possible values for the second slot.

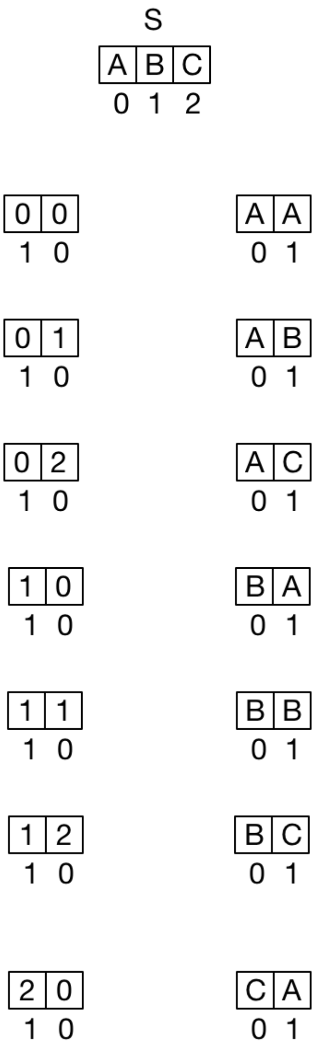



We have $3^2$ permutations. In full generality there are $n^k$ ways of forming different k-strings over a Set of size $n$ .Order is important so we can think of k-strings as permutations with repetition. The following code shows how we can generate the k-strings very simply using recursion.

## Algorithm One

`…\bitbucket\linqpad\Queries\algorithms\combinatorics\1.k-strings\Algorithm 1 (Iterative).linq`

This algorithm is based on incrementing integers.



```
public static IEnumerable<T[]> GenerateKStrings<T>( T[] S,int kStringLength)
{
   // There are n^k k-strings of a set of n items
   int numKStrings = (int)Math.Pow(S.Length,kStringLength);

   // Holds a k-digit number where each digit is of a base equal to
   // the number of elements in the set S. So if there are two
   // character in S,the digits in this number are binary.
   //
   // Each digit forms a index into S telling us exactly which
   // element of the S forms the character at the correspondong location
   // in the current kstring. So if we had k=3 and S{'a','b'} then
   // a seqIndices of {0,1,1} would correspond to the k-string of
   // {'a','b','c'}
   int[] seqIndices = new int[kStringLength];

   for (int kStringNumber = 0; kStringNumber <= numKStrings-1; kStringNumber++)
   {
        // Generate the current k-string by using the elements
        // of seqIndices to index into S.
        T[] kString = new T[kStringLength];
        for (int i = 0; i < kStringLength; i++)
               kString[i] = S[seqIndices[i]];

        // Return the k-string.
        yield return kString;

        // In this algorithm we treat  the indices array as an
        // n-digit number where the base of each digit is determined by the
        // number of elements in S.
        // Moving to the next n-tuple is then a  case of incrementing the
        // n-digit number held in seqIndices. To this we need to take care of
        // overflow which is what the following loop condition does.
        int digitIdx = 0;
        while (digitIdx <= seqIndices.Length - 2 && seqIndices[digitIdx] == (S.Length-1))
        {
               seqIndices[digitIdx] = 0;
               digitIdxToIncrement++;
        }

        seqIndices[digitIdxToIncrement]++;
        }
   }
```

The runtime of this algorithm is clearly $n^k$