```
        /// <summary>
        /// Method to generate an arbitrary n-way Cartesian Product
        ///     using the LINQ Join extension method
        /// </summary>
        /// <typeparam name="TSource">
        /// The type of the object which all of the IEnumerable inputs contain.<br/>
        /// Working with multiple object types requires the caller to cast
        /// everything as the Object type.
        /// </typeparam>
        /// <param name="source">
        /// This is the argument which does two things.
        /// Firstly, it is the part of the function signature which indicates
        /// this is an IEnumerable extension method.
        /// Secondly, is is the argument which the innovation of the
        /// extension method appears as a method call.
        /// </param>
        /// <param name="inputs">
        /// This is the IEnumerable which contains IEnumerable sequences that
        /// will be joined to form the resulting Cartesian product.
        /// </param>
        /// <returns>
        /// An IEnumerable sequence, one element for each resulting row of Cartesian product.
        /// Each row is an IEnumerable which contains an element from each input sequence.
        /// </returns>
        /// <remarks>
        /// Working with multiple object types requires the caller to cast
        /// all elements in the input IEnumerable to the Object type.
        /// <seealso cref=
        /// "System.Linq.Enumerable.Join&lt;TOuter, TInner, TKey, TResult&gt;(IEnumerable&lt;TOuter&gt; ,IEnumerable&lt;TInner&gt; ,Func&lt;
TOuter, TKey&gt; ,Func&lt;TInner, TKey&gt; , Func&lt;TOuter, TInner, TResult&gt;)"
        /// />
        /// <seealso cref=
        /// "System.Linq.Enumerable.Skip&lt;TSource&gt;(IEnumerable&lt;TSource&gt; ,int)"
        /// />
        /// <seealso cref=
        /// "System.Linq.Enumerable.Union&lt;TSource&gt;(IEnumerable&lt;TSource&gt; , IEnumerable&lt;TSource&gt; )"
        /// />
        /// <seealso cref=
        /// "System.Linq.Enumerable.First&lt;TSource&gt;(IEnumerable&lt;TSource&gt;)"
        /// />
        /// <seealso cref="ToIEnumerable&lt;TSource&gt;(TSource)"/>
        /// <seealso cref="ToIEnumerable&lt;TSource&gt;(TSource, TSource)"/>
        /// /// </remarks>
        /// <example>
```

```
/// To be provided later.
/// </example>
public static IEnumerable<IEnumerable<TSource>> CartesianProduct<TSource>
    (this IEnumerable<TSource> source,
    IEnumerable<IEnumerable<TSource>> inputs)
{
    // Build the remaining work to be done
    var RemaimingJoinTarget = inputs.Skip(1);
    // Recursion end condition, no more to be done.
    if (RemaimingJoinTarget.Count() == 0)
    {
        // At the end of the inputs, return the Cartesian of two elements
        return source.Join(inputs.First(),
            (outer) => 1, (inner) => 1,
            (outer, inner) => ToIEnumerable(outer, inner));
    }
    else
    {
        // Recursive call down the list of inputs.
        // When coming back up add the source to the results already built.
        return source.Join(
            inputs.First().CartesianProduct(RemaimingJoinTarget),
            (outer) => 1, (inner) => 1,
            (outer, inner)
                => ToIEnumerable(outer).Union(inner));
    }
}

/// <summary>
/// Method which wraps an IEnumerable around a single value
/// </summary>
/// <typeparam name="TSource">
/// This is the type of the object which is passed in as the <paramref name="val"/> argument.
/// </typeparam>
/// <param name="val">
/// The object which is to be wrapped in an IEnumerable.
/// </param>
/// <returns>
/// An IEnumerable which contains only one object the <paramref name="val"/> argument
/// </returns>
/// <remarks>
/// This may not be the most efficient way to implement the function,
/// but it is a clear and concise implementation.
/// The is because using yield return generates a large amount of compiler generated code.
```

```
/// </remarks>
/// <example>
/// To be provided.
/// </example>
public static IEnumerable<TSource> ToIEnumerable<TSource>(TSource val)
{
    yield return val;
}

/// <summary>
/// Method which wraps an IEnumerable around a pair of values.
/// </summary>
/// <typeparam name="TSource">
/// This is the type of the object which is passed in as the <paramref name="val1"/> and <paramref name="val2"/> argument.
/// </typeparam>
/// <param name="val1">
/// The first object which is to be wrapped into the IEnumerable return.
/// /// </param>
/// <param name="val2">
/// The second object which is to be wrapped into the IEnumerable return.
/// </param>
/// <returns>
/// An IEnumerable which contains objects <paramref name="val1"/> and <paramref name="val2"/> argument.
///
/// </returns>
/// <remarks>
/// This may not be the most efficient way to implement the function,
/// but it is a clear and concise implementation.
/// The is because using yield return generates a large amount of compiler generated code.
/// </remarks>
/// <example>
/// To be provided.
/// </example>
public static IEnumerable<TSource> ToIEnumerable<TSource>(TSource val1, TSource val2)
{
    yield return val1;
    yield return val2;
}
```