```
/// <summary>
/// Produces a pivot of the input sequence of sequences <paramref name="pivotSource"/>
/// of object <typeparamref name="TSource"/>.
/// </summary>
/// <typeparam name="TSource">
/// The type of object which the input sequences <paramref name="pivotSource"/>.
/// </typeparam>
/// <param name="pivotSource">
/// The <paramref name="pivotSource"/> is the sequence which is transformed.
/// The pivot transform takes each column of the input sequences
/// <paramref name="pivotSource"/> and make a row for that column.
/// The type of the output objects is the <typeparamref name="TSource"/> type.
/// </param>
/// <returns>
/// A series of rows, one for each column of the input sequences
/// <paramref name="pivotSource"/>.
/// The contents of the row is all of the values from the column.
/// </returns>
/// <exception cref="System.ArgumentNullException">
/// The <seealso cref="System.ArgumentNullException"/> ArgumentNullException is
/// thrown in two cases.
/// <list type="number">
///     <item>
///         This exception is thrown if the input <paramref name="pivotSource"/>
///         IEnumrable is null.
///     </item>
///     <item>
///         This exception is thrown if first inner IEnumerable in the input
///         <paramref name="pivotSource"/> is null.
///     </item>
/// </list>
/// </exception>
/// <exception cref="System.ArgumentOutOfRangeException">
/// This exception is throw in one of two cases.
///     <list type="number">
///         <item>
///         If the outer IEnumerable <paramref name="pivotSource"/>
///         contains no elements (is an empty sequence).
///         </item>
///         <item>
///         If the first element in the outer sequence <paramref name="pivotSource"/>
///         contains no elements(is an empty sequence).
///         </item>
///     </list>
```

```
/// </exception>
/// <remarks>
/// This extension method takes the input sequence <paramref name="pivotSource"/> and
/// converts the columns in the input to rows in the output.
/// </remarks>
public static IEnumerable<IEnumerable<TSource>> Pivot<TSource>(
            this IEnumerable<IEnumerable<TSource>> pivotSource)
{
    // Validation of the input arguments, and structure of those arguments.
    if (Object.ReferenceEquals(pivotSource, null))
        throw new ArgumentNullException("pivotSource",
            "The source IEnumerable cannot be null.");
    if (pivotSource.Count( ) == 0)
        throw new ArgumentOutOfRangeException("pivotSource",
            "The outer IEnumerable cannot be an empty sequence");
    if (pivotSource.Any(A => Object.Equals(A, null)))
        throw new ArgumentOutOfRangeException("pivotSource",
            "None of any inner IEnumerables in pivotSource can be null");
    if (pivotSource.All(A => A.Count( ) == 0))
        throw new ArgumentOutOfRangeException("pivotSource",
            "All of the input inner sequences have no columns of data.");
    // Get the row lengths to check if the data needs squaring out
    int maxRowLen = pivotSource.Select(a => a.Count( )).Max( );
    int minRowLen = pivotSource.Select(a => a.Count( )).Min( );
    // Set up the input to the Pivot
    IEnumerable<IEnumerable<TSource>> squared = pivotSource;
    // If a square out is required
    if (maxRowLen != minRowLen)
        // Fill the tail of short rows with the default value for the type
        squared = pivotSource.Select(row =>
            row.Concat(
                Enumerable.Repeat(default(TSource), maxRowLen - row.Count( ))));
    // Perform the Pivot operation on squared out data
    var result = Enumerable.Range(0, maxRowLen)
    .Select((ColumnNumber) =>
    {
        return squared.SelectMany
            (row => row
                .Where((Column, ColumnPosition) =>
                    ColumnPosition == ColumnNumber)
            );
    });
    return result;
}
```