Sparse Int Range

```
//http://isotopescreencapture.codeplex.com/
//The MIT License (MIT)
using System.Collections.Generic;
using System.Linq;

namespace Isotope.Collections.Ranges
{
public static partial class EnumerableUtil
{
public static IEnumerable Single(T item)
{
yield return item;
}

///

/// Given a range (start,end) and a number of steps, will yield that a number for each step
///
///  ///  ///  ///
public static IEnumerable RangeSteps(double start, double end, int steps)
{
// for non-positive number of steps, yield no points
if (steps < 1) { yield break; } // for exactly 1 step, yield the start value if (steps == 1) { yield return start; yield break; } // for exactly 2 stesp, yield the start value, and then the end value if (steps == 2) { yield return start; yield return end; yield break; } // for 3 steps or above, start yielding the segments // notice that the start and end values are explicitly returned so that there // is no possibility of rounding error affecting their values int segments = steps - 1; double total_length = end - start; double stepsize = total_length / segments; yield return start; for (int i = 1; i < (steps - 1); i++) { double p = start + (stepsize * i); yield
```

```
return p; } yield return end; } public static IEnumerable
GroupByCount(
IEnumerable items,
int group_size,
System.Func func_new_col,
System.Action func_add) where TGROUP : class
{

if (items == null)
{
throw new System.ArgumentNullException(“items”);
}

if (group_size < 1) { throw new
System.ArgumentOutOfRangeException("group_size"); } if
(func_new_col == null) { throw new
System.ArgumentNullException("func_new_col"); } if (func_add
== null) { throw new System.ArgumentNullException("func_add");
} int cur_group_size = 0; TGROUP cur_group = null; foreach
(var item in items) { if (cur_group_size == 0) { if (cur_group
== null) { cur_group = func_new_col(group_size); } else {
throw new System.InvalidOperationException(); } }
func_add(cur_group, cur_group_size, item); cur_group_size++;
if (cur_group_size == group_size) { yield return cur_group;
cur_group = null; cur_group_size = 0; } } if (cur_group_size >
0)
{
if (cur_group == null)
{
throw new System.InvalidOperationException();
}
yield return cur_group;
}

}

public static IEnumerable GroupByCount(ICollection items,
ICollection group_sizes, System.Func func_new_col,
```

```
System.Action func_add)
{

if (items == null)
{
throw new System.ArgumentNullException(“items”);
}

if (group_sizes == null)
{
throw new System.ArgumentNullException(“group_sizes”);
}

if (func_new_col == null)
{
throw new System.ArgumentNullException(“func_new_col”);
}

if (func_add == null)
{
throw new System.ArgumentNullException(“func_add”);
}

int total_group_sizes = group_sizes.Sum();
if (total_group_sizes != items.Count)
{
throw new System.ArgumentException(“group_sizes must account
for all items”);
}

int items_added = 0;
for (int group_index = 0; group_index < group_sizes.Count;
group_index++)     {     int     cur_group_size     =
group_sizes.ElementAt(group_index); if (cur_group_size < 0) {
throw new System.ArgumentException("group_sizes contains a
negative    numver");    }    var    cur_group    =
func_new_col(cur_group_size); for (int row_index = 0;
row_index < cur_group_size; row_index++) { var cur_item =
items.ElementAt(items_added); func_add(cur_group, row_index,
```

```
cur_item); items_added++; } yield return cur_group; } } public static IDictionary Bucketize(IEnumerable items, System.Func func_get_key, IEqualityComparer ieq)
{
if (items == null)
{
throw new System.ArgumentNullException(“items”);
}

if (func_get_key == null)
{
throw new System.ArgumentNullException(“func_get_key”);
}

var dic = new Dictionary(ieq);
foreach (var item in items)
{
var key = func_get_key(item);
List list = null;
bool found = dic.TryGetValue(key, out list);
if (!found)
{
list = new List();
dic[key] = list;
}
list.Add(item);

}

return dic;
}

public static IDictionary Bucketize(IEnumerable items, System.Func func_get_key)
{
IEqualityComparer ieq = null;
return Bucketize(items, func_get_key, ieq);
}
```

```
public static IDictionary Histogram(IEnumerable items,
System.Func func_get_key, IEqualityComparer ieq)
{
if (items == null)
{
throw new System.ArgumentNullException(“items”);
}

if (func_get_key == null)
{
throw new System.ArgumentNullException(“func_get_key”);
}

var dic = new Dictionary(ieq);
foreach (var item in items)
{
var key = func_get_key(item);
int old_value = 0;
bool found = dic.TryGetValue(key, out old_value);
if (!found)
{
dic[key] = 1;
}
else
{
dic[key] = old_value + 1;
}

}

return dic;
}

public static IDictionary Histogram(IEnumerable items)
{
var dic = Histogram(items, i => i, null);
return dic;
}
```

```
public static List Chunkify(IEnumerable items, int chunksize)
{
var chunks = new List();
List cur_chunk = null;
Chunkify(items, chunksize,
() =>
{
cur_chunk = new List(chunksize); chunks.Add(cur_chunk);
},
item =>
{
cur_chunk.Add(item);
});

return chunks;
}

public static void Chunkify(IEnumerable items, int chunksize,
System.Action create_chunk, System.Action add_item)
{
if (items == null)
{
throw new System.ArgumentNullException(“items”);
}

if (chunksize < 1) { throw new System.ArgumentOutOfRangeException("chunksize"); } int item_count = 0; int curchunk_size = 0; foreach (T item in items) { if ((item_count % chunksize) == 0) { create_chunk(); curchunk_size = 0; } add_item(item); item_count++; curchunk_size++; } } } public struct IntRange { public readonly int _Lower; public readonly int _Length; public int Lower { get { return this._Lower; } } public int Upper { get { return this.Lower + this.Length - 1; } } public int Length { get { return this._Length; } } private IntRange(int lower, int length) { this._Lower = lower; this._Length = length; if (this.Lower > this.Upper)
{
```

```
throw new System.OverflowException();
}
}

///

/// Creates a range from the lower and upper values
///
///
/// (0,0) -> [0]
/// (0,1) -> [0,1]
/// (0,5) -> [0,1,2,3,4,5]
/// (2,5) -> [2,3,4,5]
///
///  ///  ///
public static IntRange FromEndpoints(int lower, int upper)
{
return new IntRange(lower, upper – lower + 1);
}

///

/// Creates a range of a single number
///
///
///  ///
public static IntRange FromInteger(int value)
{
return new IntRange(value, 1);
}

///

/// Creates a new range from a starting value and including
all the numbers following that
///
///
/// (2,0) -> []
/// (2,1) -> [2]
```

```
/// (2,2) -> [2,3]
/// (2,3) -> [2,3,4]
/// (2,4) -> [2,3,4,5]
///
///  ///  ///
public static IntRange FromLower(int lower, int length)
{
if (length < 0) { throw new System.ArgumentOutOfRangeException("length", "must be >= 0″);
}

return new IntRange(lower, length);
}

public override string ToString()
{
var invariant_culture = System.Globalization.CultureInfo.InvariantCulture;
return string.Format(invariant_culture, “Range({0},{1})”, this.Lower, this.Upper);
}

private static bool _intersects(IntRange range1, IntRange range2)
{
return ((range1.Lower <= range2.Lower) && (range1.Upper >= range2.Lower));
}

///

/// Tests if this range interests with another
///
/// the other range /// true if they intersect
public bool Intersects(IntRange range)
{
return (_intersects(this, range) || _intersects(range, this));
}
```

```
private static int Delta = 1;

private static bool _touches(IntRange range1, IntRange range2)
{
var val = (range1.Upper + Delta) == range2.Lower;
return val;
}

///

/// Tests if this range touches another. For example (1-2)
touches (3,5) but (1,2) does not touch (4,5)
///
/// the other range /// true if they touch
public bool Touches(IntRange range)
{
var val = _touches(this, range) || _touches(range, this);
return val;
}

///

/// Tests if this range contains a specific integer
///
/// the integet /// true if the number is contained
public bool Contains(int n)
{
return ((this.Lower <= n) && (n <= this.Upper)); } ///
/// Join this range with another and return a single range
that contains them both. The ranges must either touch or
interest.
/// for example (0,2) amd (3,7) will yield (0,7)
///

/// the other range /// the merged range
public IntRange JoinWith(IntRange range)
{
if (this.Intersects(range) || this.Touches(range))
{
```

```
int new_Upper = System.Math.Max(this.Upper, range.Upper);
int new_Lower = System.Math.Min(this.Lower, range.Lower);
return IntRange.FromEndpoints(new_Lower, new_Upper);
}
else
{
throw new System.ArgumentException();
}
}

///

/// Get each int int he range from the lower value to the upper value
///
/// each int in the range
public IEnumerable Values
{
get
{
for (int i = this.Lower; i <= this.Upper; i++) { yield return i; } } } } public class SparseIntRange { private List ranges;

public SparseIntRange()
{
this.clear();
}

public IEnumerable Ranges
{
get
{
foreach (var range in this.ranges)
{
yield return range;
}
}
}
```

```
public IEnumerable Values
{
get
{
foreach (var rng in this.Ranges)
{
foreach (int i in rng.Values)
{
yield return i;
}
}
}
}

private void clear()
{
this.ranges = new List();
}

public int RangeCount
{
get { return this.ranges.Count; }
}

public void Add(int n)
{
var rng = IntRange.FromInteger(n);
this.Add(rng);
}

public void AddInclusive(int lower, int upper)
{
var rng = IntRange.FromEndpoints(lower, upper);
this.Add(rng);
}

public void Add(IntRange range)
{
```

```
var left = new List();
var right = new List();
foreach (var rng in this.ranges)
{
if (range.Intersects(rng) || range.Touches(rng))
{
range = range.JoinWith(rng);
}
else if (rng.Upper < range.Lower) { left.Add(rng); } else if (range.Upper < rng.Lower) { right.Add(rng); } else { throw new System.InvalidOperationException("Internal Error"); } } this.ranges = left.Concat(EnumerableUtil.Single(range)).Concat(right).ToList(); } public int Lower { get { if (this.ranges.Count < 1) { throw new System.InvalidOperationException("There are no ranges"); } return this.ranges[0].Lower; } } public int Upper { get { if (this.ranges.Count < 1) { throw new System.InvalidOperationException("empty"); } return this.ranges[this.ranges.Count - 1].Upper; } } public int Count { get { int length = this.ranges.Aggregate(0, (old, rng) => old + rng.Length);
return length;
}
}

public void Remove(int value)
{
var rng = IntRange.FromInteger(value);
this.Remove(rng);
}

public void RemoveInclusive(int lower, int upper)
{
var rng = IntRange.FromEndpoints(lower, upper);
this.Remove(rng);
}

public void Remove(IntRange range)
```

```
{
// if the range doesn't intersect this collection do nothing
if    (!range.Intersects(IntRange.FromEndpoints(this.Lower,
this.Upper)))
{
return;
}

var middle = new List();

foreach (var S in this.ranges)
{
if (!range.Intersects(S))
{
middle.Add(S);
continue;
}

if ((range.Lower <= S.Lower) && (range.Upper >= S.Upper))
{
// disregard S completely
continue;
}

if (range.Lower > S.Lower)
{
if  (S.Lower  <=  (range.Lower  -  1))  {  var  X  =
IntRange.FromEndpoints(S.Lower,  range.Lower  -  1);
middle.Add(X); } } if (range.Upper <= S.Upper) { if
((range.Upper  +  1)  <=  S.Upper)  {  var  X  =
IntRange.FromEndpoints(range.Upper  +  1,  S.Upper);
middle.Add(X);    }    }    else    {    throw    new
System.InvalidOperationException("internal  error");  }  }
this.ranges   =   middle;   }   public   IntRange?
FindRangeContainingNumber(int n) { foreach (var rng in
this.ranges) { if (rng.Contains(n)) { return rng; } } return
null; } public bool Contains(int n) { var rng =
this.FindRangeContainingNumber(n); return rng != null ? true :
```

false; } public override string ToString() { var sb = new System.Text.StringBuilder(); sb.AppendFormat("{0}(", this.GetType().Name); foreach (var rng in this.ranges) { sb.Append(rng.ToString()); } sb.Append(")"); return sb.ToString(); } } } [/csharp]