LinksPlatform's Platform.Collections Class Library

./Platform.Collections/Arrays/ArrayFiller[TElement].cs

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Arrays
7   {
8       public class ArrayFiller<TElement>
9       {
10          protected readonly TElement[] _array;
11          protected long _position;
12
13          public ArrayFiller(TElement[] array, long offset)
14          {
15              _array = array;
16              _position = offset;
17          }
18
19          public ArrayFiller(TElement[] array) : this(array, 0) { }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public void Add(TElement element) => _array[_position++] = element;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public bool AddAndReturnTrue(TElement element)
26          {
27              _array[_position++] = element;
28              return true;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public bool AddFirstAndReturnTrue(IList<TElement> collection)
33          {
34              _array[_position++] = collection[0];
35              return true;
36          }
37      }
38  }
```

./Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Arrays
7   {
8       public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
9       {
10          protected readonly TReturnConstant _returnConstant;
11
12          public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
        ↪   base(array, offset) => _returnConstant = returnConstant;
13
14          public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
        ↪   returnConstant) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public TReturnConstant AddAndReturnConstant(TElement element)
18          {
19              _array[_position++] = element;
20              return _returnConstant;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
25          {
26              _array[_position++] = collection[0];
27              return _returnConstant;
28          }
29      }
30  }
```

./Platform.Collections/Arrays/ArrayPool.cs

```csharp
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
```

```csharp
namespace Platform.Collections.Arrays
{
    public static class ArrayPool
    {
        public static readonly int DefaultSizesAmount = 512;
        public static readonly int DefaultMaxArraysPerSize = 32;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
    }
}
```

./Platform.Collections/Arrays/ArrayPool[T].cs

```csharp
using System;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Arrays
{
    /// <remarks>
    /// Original idea from
    /// ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
    /// </remarks>
    public class ArrayPool<T>
    {
        public static readonly T[] Empty = new T[0];

        // May be use Default class for that later.
        [ThreadStatic]
        internal static ArrayPool<T> _threadInstance;
        internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
        ↪  = new ArrayPool<T>()); }

        private readonly int _maxArraysPerSize;
        private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
        ↪  Stack<T[]>>(ArrayPool.DefaultSizesAmount);

        public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;

        public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }

        public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);

        public Disposable<T[]> Resize(Disposable<T[]> source, long size)
        {
            var destination = AllocateDisposable(size);
            T[] sourceArray = source;
            T[] destinationArray = destination;
            Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
            ↪  sourceArray.Length);
            source.Dispose();
            return destination;
        }

        public virtual void Clear() => _pool.Clear();

        public virtual T[] Allocate(long size)
        {
            Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
            return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
            ↪  T[size];
        }

        public virtual void Free(T[] array)
        {
            Ensure.Always.ArgumentNotNull(array, nameof(array));
            if (array.Length == 0)
            {
                return;
            }
            var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
            if (stack.Count == _maxArraysPerSize) // Stack is full
```

```
60              {
61                  return;
62              }
63              stack.Push(array);
64          }
65      }
66  }
```

./Platform.Collections/Arrays/ArrayString.cs

```
1   using Platform.Collections.Segments;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Arrays
6   {
7       public class ArrayString<T> : Segment<T>
8       {
9           public ArrayString(int length) : base(new T[length], 0, length) { }
10          public ArrayString(T[] array) : base(array, 0, array.Length) { }
11          public ArrayString(T[] array, int length) : base(array, 0, length) { }
12      }
13  }
```

./Platform.Collections/Arrays/CharArrayExtensions.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Arrays
4   {
5       public static unsafe class CharArrayExtensions
6       {
7           /// <remarks>
8           /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783↲
              ↪   a3eda37d3d4cd10/mscorlib/system/string.cs#L833
9           /// </remarks>
10          public static int GenerateHashCode(this char[] array, int offset, int length)
11          {
12              var hashSeed = 5381;
13              var hashAccumulator = hashSeed;
14              fixed (char* pointer = &array[offset])
15              {
16                  for (char* s = pointer, last = s + length; s < last; s++)
17                  {
18                      hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
19                  }
20              }
21              return hashAccumulator + (hashSeed * 1566083941);
22          }
23
24          /// <remarks>
25          /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783↲
              ↪   a3eda37d3d4cd10/mscorlib/system/string.cs#L364
26          /// </remarks>
27          public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
              ↪   right, int rightOffset)
28          {
29              fixed (char* leftPointer = &left[leftOffset])
30              {
31                  fixed (char* rightPointer = &right[rightOffset])
32                  {
33                      char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
34                      if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
                          ↪   rightPointerCopy, ref length))
35                      {
36                          return false;
37                      }
38                      CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
                          ↪   ref length);
39                      return length <= 0;
40                  }
41              }
42          }
43
44          private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
              ↪   int length)
45          {
46              while (length >= 10)
47              {
48                  if ((*(int*)left != *(int*)right)
```

```
49                    || (*(int*)(left + 2) != *(int*)(right + 2))
50                    || (*(int*)(left + 4) != *(int*)(right + 4))
51                    || (*(int*)(left + 6) != *(int*)(right + 6))
52                    || (*(int*)(left + 8) != *(int*)(right + 8)))
53                {
54                    return false;
55                }
56                left += 10;
57                right += 10;
58                length -= 10;
59            }
60            return true;
61        }
62
63        private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
        ↪   int length)
64        {
65            // This depends on the fact that the String objects are
66            // always zero terminated and that the terminating zero is not included
67            // in the length. For odd string sizes, the last compare will include
68            // the zero terminator.
69            while (length > 0)
70            {
71                if (*(int*)left != *(int*)right)
72                {
73                    break;
74                }
75                left += 2;
76                right += 2;
77                length -= 2;
78            }
79        }
80    }
81 }
```

./Platform.Collections/Arrays/GenericArrayExtensions.cs

```
1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static class GenericArrayExtensions
8      {
9          public static T[] Clone<T>(this T[] array)
10         {
11             var copy = new T[array.Length];
12             Array.Copy(array, 0, copy, 0, array.Length);
13             return copy;
14         }
15     }
16 }
```

./Platform.Collections/BitString.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Numerics;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Ranges;
7
8  // ReSharper disable ForCanBeConvertedToForeach
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     /// <remarks>
14     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
        ↪   64 бит в массиве значений.
15     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
        ↪   байт в 8 байт.
16     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
        ↪   помощью которой можно быстро
17     /// проверять есть ли значения непосредственно далее (ниже по уровню).
18     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
19     /// </remarks>
20     public class BitString : IEquatable<BitString>
21     {
22         private static readonly byte[][] _bitsSetIn16Bits;
```

```csharp
        private long[] _array;
        private long _length;
        private long _minPositiveWord;
        private long _maxPositiveWord;

        public bool this[long index]
        {
            get => Get(index);
            set => Set(index, value);
        }

        public long Length
        {
            get => _length;
            set
            {
                if (_length == value)
                {
                    return;
                }
                Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
                // Currently we never shrink the array
                if (value > _length)
                {
                    var words = GetWordsCountFromIndex(value);
                    var oldWords = GetWordsCountFromIndex(_length);
                    if (words > _array.LongLength)
                    {
                        var copy = new long[words];
                        Array.Copy(_array, copy, _array.LongLength);
                        _array = copy;
                    }
                    else
                    {
                        // What is going on here?
                        Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
                    }
                    // What is going on here?
                    var mask = (int)(_length % 64);
                    if (mask > 0)
                    {
                        _array[oldWords - 1] &= (1L << mask) - 1;
                    }
                }
                else
                {
                    // Looks like minimum and maximum positive words are not updated
                    throw new NotImplementedException();
                }
                _length = value;
            }
        }

        #region Constructors

        static BitString()
        {
            _bitsSetIn16Bits = new byte[65536][];
            int i, c, k;
            byte bitIndex;
            for (i = 0; i < 65536; i++)
            {
                // Calculating size of array (number of positive bits)
                for (c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        c++;
                    }
                }
                var array = new byte[c];
                // Adding positive bits indices into array
                for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        array[c++] = bitIndex;
                    }
                    bitIndex++;
```

```csharp
                }
                _bitsSetIn16Bits[i] = array;
            }
        }

        public BitString(BitString other)
        {
            Ensure.Always.ArgumentNotNull(other, nameof(other));
            _length = other._length;
            _array = new long[GetWordsCountFromIndex(_length)];
            _minPositiveWord = other._minPositiveWord;
            _maxPositiveWord = other._maxPositiveWord;
            Array.Copy(other._array, _array, _array.LongLength);
        }

        public BitString(long length)
        {
            Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
            _length = length;
            _array = new long[GetWordsCountFromIndex(_length)];
            MarkBordersAsAllBitsReset();
        }

        public BitString(long length, bool defaultValue)
            : this(length)
        {
            if (defaultValue)
            {
                SetAll();
            }
        }

        #endregion

        public BitString Not()
        {
            for (var i = 0; i < _array.Length; i++)
            {
                _array[i] = ~_array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString VectorNot()
        {
            var thisVector = new Vector<long>(_array);
            var result = ~thisVector;
            result.CopyTo(_array, 0);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        public BitString And(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                _array[i] &= otherArray[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString VectorAnd(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            var thisVector = new Vector<long>(_array);
            var otherVector = new Vector<long>(other._array);
            var result = thisVector & otherVector;
            result.CopyTo(_array, 0);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }
```

```csharp
181         public BitString Or(BitString other)
182         {
183             EnsureBitStringHasTheSameSize(other, nameof(other));
184             GetCommonOuterBorders(this, other, out long from, out long to);
185             for (var i = from; i <= to; i++)
186             {
187                 _array[i] |= other._array[i];
188                 RefreshBordersByWord(i);
189             }
190             return this;
191         }
192
193         public BitString VectorOr(BitString other)
194         {
195             EnsureBitStringHasTheSameSize(other, nameof(other));
196             var thisVector = new Vector<long>(_array);
197             var otherVector = new Vector<long>(other._array);
198             var result = thisVector | otherVector;
199             result.CopyTo(_array, 0);
200             MarkBordersAsAllBitsSet();
201             TryShrinkBorders();
202             return this;
203         }
204
205         public BitString Xor(BitString other)
206         {
207             EnsureBitStringHasTheSameSize(other, nameof(other));
208             GetCommonOuterBorders(this, other, out long from, out long to);
209             for (var i = from; i <= to; i++)
210             {
211                 _array[i] ^= other._array[i];
212                 RefreshBordersByWord(i);
213             }
214             return this;
215         }
216
217         private void RefreshBordersByWord(long wordIndex)
218         {
219             if (_array[wordIndex] == 0)
220             {
221                 if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
222                 {
223                     _minPositiveWord++;
224                 }
225                 if (wordIndex == _maxPositiveWord && wordIndex != 0)
226                 {
227                     _maxPositiveWord--;
228                 }
229             }
230             else
231             {
232                 if (wordIndex < _minPositiveWord)
233                 {
234                     _minPositiveWord = wordIndex;
235                 }
236                 if (wordIndex > _maxPositiveWord)
237                 {
238                     _maxPositiveWord = wordIndex;
239                 }
240             }
241         }
242
243         public bool TryShrinkBorders()
244         {
245             GetBorders(out long from, out long to);
246             while (from <= to && _array[from] == 0)
247             {
248                 from++;
249             }
250             if (from > to)
251             {
252                 MarkBordersAsAllBitsReset();
253                 return true;
254             }
255             while (to >= from && _array[to] == 0)
256             {
257                 to--;
258             }
259             if (to < from)
```

```csharp
                {
                    MarkBordersAsAllBitsReset();
                    return true;
                }
                var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
                if (bordersUpdated)
                {
                    SetBorders(from, to);
                }
                return bordersUpdated;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Get(long index)
            {
                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
                return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Set(long index, bool value)
            {
                if (value)
                {
                    Set(index);
                }
                else
                {
                    Reset(index);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Set(long index)
            {
                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
                var wordIndex = GetWordIndexFromIndex(index);
                var mask = GetBitMaskFromIndex(index);
                _array[wordIndex] |= mask;
                RefreshBordersByWord(wordIndex);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Reset(long index)
            {
                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
                var wordIndex = GetWordIndexFromIndex(index);
                var mask = GetBitMaskFromIndex(index);
                _array[wordIndex] &= ~mask;
                RefreshBordersByWord(wordIndex);
            }

            public bool Add(long index)
            {
                var wordIndex = GetWordIndexFromIndex(index);
                var mask = GetBitMaskFromIndex(index);
                if ((_array[wordIndex] & mask) == 0)
                {
                    _array[wordIndex] |= mask;
                    RefreshBordersByWord(wordIndex);
                    return true;
                }
                else
                {
                    return false;
                }
            }

            public void SetAll(bool value)
            {
                if (value)
                {
                    SetAll();
                }
                else
                {
                    ResetAll();
                }
            }
```

```csharp
339
340         public void SetAll()
341         {
342             const long fillValue = unchecked((long)0xffffffffffffffff);
343             var words = GetWordsCountFromIndex(_length);
344             for (var i = 0; i < words; i++)
345             {
346                 _array[i] = fillValue;
347             }
348             MarkBordersAsAllBitsSet();
349         }
350
351         public void ResetAll()
352         {
353             const long fillValue = 0;
354             GetBorders(out long from, out long to);
355             for (var i = from; i <= to; i++)
356             {
357                 _array[i] = fillValue;
358             }
359             MarkBordersAsAllBitsReset();
360         }
361
362         public List<long> GetSetIndices()
363         {
364             var result = new List<long>();
365             GetBorders(out long from, out long to);
366             for (var i = from; i <= to; i++)
367             {
368                 var word = _array[i];
369                 if (word != 0)
370                 {
371                     AppendAllSetBitIndices(result, i, word);
372                 }
373             }
374             return result;
375         }
376
377         public List<ulong> GetSetUInt64Indices()
378         {
379             var result = new List<ulong>();
380             GetBorders(out ulong from, out ulong to);
381             for (var i = from; i <= to; i++)
382             {
383                 var word = _array[i];
384                 if (word != 0)
385                 {
386                     AppendAllSetBitIndices(result, i, word);
387                 }
388             }
389             return result;
390         }
391
392         public long GetFirstSetBitIndex()
393         {
394             var i = _minPositiveWord;
395             var word = _array[i];
396             if (word != 0)
397             {
398                 return GetFirstSetBitForWord(i, word);
399             }
400             return -1;
401         }
402
403         public long GetLastSetBitIndex()
404         {
405             var i = _maxPositiveWord;
406             var word = _array[i];
407             if (word != 0)
408             {
409                 return GetLastSetBitForWord(i, word);
410             }
411             return -1;
412         }
413
414         public long CountSetBits()
415         {
416             var total = 0L;
417             GetBorders(out long from, out long to);
```

```csharp
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    total += CountSetBitsForWord(word);
                }
            }
            return total;
        }

        public bool HaveCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                if (left != 0 && right != 0 && (left & right) != 0)
                {
                    return true;
                }
            }
            return false;
        }

        public long CountCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var total = 0L;
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    total += CountSetBitsForWord(combined);
                }
            }
            return total;
        }

        public List<long> GetCommonIndices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var result = new List<long>();
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
        }

        public List<ulong> GetCommonUInt64Indices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonBorders(this, other, out ulong from, out ulong to);
            var result = new List<ulong>();
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
```

```csharp
                        AppendAllSetBitIndices(result, i, combined);
                    }
                }
                return result;
            }

            public long GetFirstCommonBitIndex(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonInnerBorders(this, other, out long from, out long to);
                var otherArray = other._array;
                for (var i = from; i <= to; i++)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    var combined = left & right;
                    if (combined != 0)
                    {
                        return GetFirstSetBitForWord(i, combined);
                    }
                }
                return -1;
            }

            public long GetLastCommonBitIndex(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonInnerBorders(this, other, out long from, out long to);
                var otherArray = other._array;
                for (var i = to; i >= from; i--)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    var combined = left & right;
                    if (combined != 0)
                    {
                        return GetLastSetBitForWord(i, combined);
                    }
                }
                return -1;
            }

            public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
                false;

            public bool Equals(BitString other)
            {
                if (_length != other._length)
                {
                    return false;
                }
                var otherArray = other._array;
                if (_array.Length != otherArray.Length)
                {
                    return false;
                }
                if (_minPositiveWord != other._minPositiveWord)
                {
                    return false;
                }
                if (_maxPositiveWord != other._maxPositiveWord)
                {
                    return false;
                }
                GetCommonBorders(this, other, out ulong from, out ulong to);
                for (var i = from; i <= to; i++)
                {
                    if (_array[i] != otherArray[i])
                    {
                        return false;
                    }
                }
                return true;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
            {
                Ensure.Always.ArgumentNotNull(other, argumentName);
```

```
            if (_length != other._length)
            {
                throw new ArgumentException("Bit string must be the same size.", argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out long from, out long to)
        {
            from = _minPositiveWord;
            to = _maxPositiveWord;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out ulong from, out ulong to)
        {
            from = (ulong)_minPositiveWord;
            to = (ulong)_maxPositiveWord;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void SetBorders(long from, long to)
        {
            _minPositiveWord = from;
            _maxPositiveWord = to;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Range<long> GetValidIndexRange() => (0, _length - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Range<long> GetValidLengthRange() => (0, long.MaxValue);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
        ↪ wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪ bits32to47, out byte[] bits48to63);
            AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
            ↪ bits48to63);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
        ↪ wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪ bits32to47, out byte[] bits48to63);
            AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
            ↪ bits48to63);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long CountSetBitsForWord(long word)
        {
            GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
            ↪ out byte[] bits48to63);
            return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
            ↪ bits48to63.LongLength;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪ bits32to47, out byte[] bits48to63);
            return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetLastSetBitForWord(long wordIndex, long wordValue)
```

```csharp
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                bits32to47, out byte[] bits48to63);
            return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
        }

        private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
            byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
        {
            for (var j = 0; j < bits00to15.Length; j++)
            {
                result.Add(bits00to15[j] + (i * 64));
            }
            for (var j = 0; j < bits16to31.Length; j++)
            {
                result.Add(bits16to31[j] + 16 + (i * 64));
            }
            for (var j = 0; j < bits32to47.Length; j++)
            {
                result.Add(bits32to47[j] + 32 + (i * 64));
            }
            for (var j = 0; j < bits48to63.Length; j++)
            {
                result.Add(bits48to63[j] + 48 + (i * 64));
            }
        }

        private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
            byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
        {
            for (var j = 0; j < bits00to15.Length; j++)
            {
                result.Add(bits00to15[j] + (i * 64));
            }
            for (var j = 0; j < bits16to31.Length; j++)
            {
                result.Add(bits16to31[j] + 16UL + (i * 64));
            }
            for (var j = 0; j < bits32to47.Length; j++)
            {
                result.Add(bits32to47[j] + 32UL + (i * 64));
            }
            for (var j = 0; j < bits48to63.Length; j++)
            {
                result.Add(bits48to63[j] + 48UL + (i * 64));
            }
        }

        private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            bits32to47, byte[] bits48to63)
        {
            if (bits00to15.Length > 0)
            {
                return bits00to15[0] + (i * 64);
            }
            if (bits16to31.Length > 0)
            {
                return bits16to31[0] + 16 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[0] + 32 + (i * 64);
            }
            return bits48to63[0] + 48 + (i * 64);
        }

        private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            bits32to47, byte[] bits48to63)
        {
            if (bits48to63.Length > 0)
            {
                return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
            }
```

```csharp
716                    if (bits16to31.Length > 0)
717                    {
718                        return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
719                    }
720                    return bits00to15[bits00to15.Length - 1] + (i * 64);
721                }
722
723            private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪   byte[] bits32to47, out byte[] bits48to63)
724            {
725                bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
726                bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
727                bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
728                bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
729            }
730
731            [MethodImpl(MethodImplOptions.AggressiveInlining)]
732            public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪   out long to)
733            {
734                from = Math.Max(left._minPositiveWord, right._minPositiveWord);
735                to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
736            }
737
738            [MethodImpl(MethodImplOptions.AggressiveInlining)]
739            public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪   out long to)
740            {
741                from = Math.Min(left._minPositiveWord, right._minPositiveWord);
742                to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
743            }
744
745            [MethodImpl(MethodImplOptions.AggressiveInlining)]
746            public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
    ↪   ulong to)
747            {
748                from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
749                to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
750            }
751
752            [MethodImpl(MethodImplOptions.AggressiveInlining)]
753            public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
754
755            [MethodImpl(MethodImplOptions.AggressiveInlining)]
756            public static long GetWordIndexFromIndex(long index) => index >> 6;
757
758            [MethodImpl(MethodImplOptions.AggressiveInlining)]
759            public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
760
761            public override int GetHashCode() => base.GetHashCode();
762
763            public override string ToString() => base.ToString();
764        }
765    }
```

./Platform.Collections/BitStringExtensions.cs

```csharp
1    using Platform.Random;
2
3    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5    namespace Platform.Collections
6    {
7        public static class BitStringExtensions
8        {
9            public static void SetRandomBits(this BitString @string)
10            {
11                for (var i = 0; i < @string.Length; i++)
12                {
13                    var value = RandomHelpers.Default.NextBoolean();
14                    @string.Set(i, value);
15                }
16            }
17        }
18    }
```

./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

```csharp
1    using System.Collections.Concurrent;
2    using System.Collections.Generic;
```

```csharp
     using System.Runtime.CompilerServices;

     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

     namespace Platform.Collections.Concurrent
     {
         public static class ConcurrentQueueExtensions
         {
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
             {
                 while (queue.TryDequeue(out T item))
                 {
                     yield return item;
                 }
             }
         }
     }
```

./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```csharp
     using System.Collections.Concurrent;
     using System.Runtime.CompilerServices;

     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

     namespace Platform.Collections.Concurrent
     {
         public static class ConcurrentStackExtensions
         {
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
             ↪  value) ? value : default;

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
             ↪  value) ? value : default;
         }
     }
```

./Platform.Collections/EnsureExtensions.cs

```csharp
     using System;
     using System.Collections.Generic;
     using System.Diagnostics;
     using System.Runtime.CompilerServices;
     using Platform.Exceptions;
     using Platform.Exceptions.ExtensionRoots;

     #pragma warning disable IDE0060 // Remove unused parameter
     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

     namespace Platform.Collections
     {
         public static class EnsureExtensions
         {
             #region Always

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
             ↪  ICollection<T> argument, string argumentName, string message)
             {
                 if (argument.IsNullOrEmpty())
                 {
                     throw new ArgumentException(message, argumentName);
                 }
             }

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
             ↪  ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
             ↪  argumentName, null);

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
             ↪  ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
             ↪  string argument, string argumentName, string message)
             {
```

```
35              if (string.IsNullOrWhiteSpace(argument))
36              {
37                  throw new ArgumentException(message, argumentName);
38              }
39          }
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
            ↪ argument, argumentName, null);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
46
47          #endregion
48
49          #region OnDebug
50
51          [Conditional("DEBUG")]
52          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪ ICollection<T> argument, string argumentName, string message) =>
            ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
53
54          [Conditional("DEBUG")]
55          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪ ICollection<T> argument, string argumentName) =>
            ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
56
57          [Conditional("DEBUG")]
58          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
59
60          [Conditional("DEBUG")]
61          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪ root, string argument, string argumentName, string message) =>
            ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
62
63          [Conditional("DEBUG")]
64          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪ root, string argument, string argumentName) =>
            ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
65
66          [Conditional("DEBUG")]
67          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
            ↪ null, null);
68
69          #endregion
70      }
71  }
```

./Platform.Collections/ICollectionExtensions.cs

```
1  using System.Collections.Generic;
2  using System.Linq;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections
7  {
8      public static class ICollectionExtensions
9      {
10         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
           ↪ null || collection.Count == 0;
11
12         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
13         {
14             var equalityComparer = EqualityComparer<T>.Default;
15             return collection.All(item => equalityComparer.Equals(item, default));
16         }
17     }
18 }
```

./Platform.Collections/IDictionaryExtensions.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
```

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections
{
    public static class IDictionaryExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
        ↪ dictionary, TKey key)
        {
            dictionary.TryGetValue(key, out TValue value);
            return value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
        ↪ TKey key, Func<TKey, TValue> valueFactory)
        {
            if (!dictionary.TryGetValue(key, out TValue value))
            {
                value = valueFactory(key);
                dictionary.Add(key, value);
                return value;
            }
            return value;
        }
    }
}
```

./Platform.Collections/ISetExtensions.cs
```csharp
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections
{
    public static class ISetExtensions
    {
        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
        ↪ set.Remove(element);
        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
        ↪ !set.Contains(element);
    }
}
```

./Platform.Collections/Lists/CharIListExtensions.cs
```csharp
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Lists
{
    public static class CharIListExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783↵
        ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        public static unsafe int GenerateHashCode(this IList<char> list)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪ left.EqualTo(right, ContentEqualTo);

        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
        {
            for (var i = left.Count - 1; i >= 0; --i)
            {
                if (left[i] != right[i])
                {
```

```
31                      return false;
32                  }
33              }
34              return true;
35          }
36      }
37  }
```

./Platform.Collections/Lists/IListComparer.cs

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Lists
6   {
7       public class IListComparer<T> : IComparer<IList<T>>
8       {
9           public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
10      }
11  }
```

./Platform.Collections/Lists/IListEqualityComparer.cs

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Lists
6   {
7       public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
8       {
9           public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
10          public int GetHashCode(IList<T> list) => list.GenerateHashCode();
11      }
12  }
```

./Platform.Collections/Lists/IListExtensions.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Lists
7   {
8       public static class IListExtensions
9       {
10          public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
11          {
12              list.Add(element);
13              return true;
14          }
15
16          public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
17
18          public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
            ↪  right, ContentEqualTo);
19
20          public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
            ↪  IList<T>, bool> contentEqualityComparer)
21          {
22              if (ReferenceEquals(left, right))
23              {
24                  return true;
25              }
26              var leftCount = left.GetCountOrZero();
27              var rightCount = right.GetCountOrZero();
28              if (leftCount == 0 && rightCount == 0)
29              {
30                  return true;
31              }
32              if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
33              {
34                  return false;
35              }
36              return contentEqualityComparer(left, right);
37          }
38
39          public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
40          {
41              var equalityComparer = EqualityComparer<T>.Default;
```

```csharp
                for (var i = left.Count - 1; i >= 0; --i)
                {
                    if (!equalityComparer.Equals(left[i], right[i]))
                    {
                        return false;
                    }
                }
                return true;
            }

            public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
            {
                if (list == null)
                {
                    return null;
                }
                var result = new List<T>(list.Count);
                for (var i = 0; i < list.Count; i++)
                {
                    if (predicate(list[i]))
                    {
                        result.Add(list[i]);
                    }
                }
                return result.ToArray();
            }

            public static T[] ToArray<T>(this IList<T> list)
            {
                var array = new T[list.Count];
                list.CopyTo(array, 0);
                return array;
            }

            public static void ForEach<T>(this IList<T> list, Action<T> action)
            {
                for (var i = 0; i < list.Count; i++)
                {
                    action(list[i]);
                }
            }

            /// <remarks>
            /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
            ///         -overridden-system-object-gethashcode
            /// </remarks>
            public static int GenerateHashCode<T>(this IList<T> list)
            {
                var result = 17;
                for (var i = 0; i < list.Count; i++)
                {
                    result = unchecked((result * 23) + list[i].GetHashCode());
                }
                return result;
            }

            public static int CompareTo<T>(this IList<T> left, IList<T> right)
            {
                var comparer = Comparer<T>.Default;
                var leftCount = left.GetCountOrZero();
                var rightCount = right.GetCountOrZero();
                var intermediateResult = leftCount.CompareTo(rightCount);
                for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
                {
                    intermediateResult = comparer.Compare(left[i], right[i]);
                }
                return intermediateResult;
            }
        }
    }
```

./Platform.Collections/Segments/CharSegment.cs

```csharp
using System.Linq;
using System.Collections.Generic;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```csharp
 8   namespace Platform.Collections.Segments
 9   {
10       public class CharSegment : Segment<char>
11       {
12           public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
     ↪   length) { }
13
14           public override int GetHashCode()
15           {
16               // Base can be not an array, but still IList<char>
17               if (Base is char[] baseArray)
18               {
19                   return baseArray.GenerateHashCode(Offset, Length);
20               }
21               else
22               {
23                   return this.GenerateHashCode();
24               }
25           }
26
27           public override bool Equals(Segment<char> other)
28           {
29               bool contentEqualityComparer(IList<char> left, IList<char> right)
30               {
31                   // Base can be not an array, but still IList<char>
32                   if (Base is char[] baseArray && other.Base is char[] otherArray)
33                   {
34                       return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
35                   }
36                   else
37                   {
38                       return left.ContentEqualTo(right);
39                   }
40               }
41               return this.EqualTo(other, contentEqualityComparer);
42           }
43
44           public static implicit operator string(CharSegment segment)
45           {
46               if (!(segment.Base is char[] array))
47               {
48                   array = segment.Base.ToArray();
49               }
50               return new string(array, segment.Offset, segment.Length);
51           }
52
53           public override string ToString() => this;
54       }
55   }
```

./Platform.Collections/Segments/Segment.cs

```csharp
 1   using System;
 2   using System.Collections;
 3   using System.Collections.Generic;
 4   using Platform.Collections.Lists;
 5
 6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 7
 8   namespace Platform.Collections.Segments
 9   {
10       public class Segment<T> : IEquatable<Segment<T>>, IList<T>
11       {
12           public IList<T> Base { get; }
13           public int Offset { get; }
14           public int Length { get; }
15
16           public Segment(IList<T> @base, int offset, int length)
17           {
18               Base = @base;
19               Offset = offset;
20               Length = length;
21           }
22
23           public override int GetHashCode() => this.GenerateHashCode();
24
25           public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
26
27           public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
     ↪   false;
28
```

```csharp
            #region IList

            public T this[int i]
            {
                get => Base[Offset + i];
                set => Base[Offset + i] = value;
            }

            public int Count => Length;

            public bool IsReadOnly => true;

            public int IndexOf(T item)
            {
                var index = Base.IndexOf(item);
                if (index >= Offset)
                {
                    var actualIndex = index - Offset;
                    if (actualIndex < Length)
                    {
                        return actualIndex;
                    }
                }
                return -1;
            }

            public void Insert(int index, T item) => throw new NotSupportedException();

            public void RemoveAt(int index) => throw new NotSupportedException();

            public void Add(T item) => throw new NotSupportedException();

            public void Clear() => throw new NotSupportedException();

            public bool Contains(T item) => IndexOf(item) >= 0;

            public void CopyTo(T[] array, int arrayIndex)
            {
                for (var i = 0; i < Length; i++)
                {
                    array[arrayIndex++] = this[i];
                }
            }

            public bool Remove(T item) => throw new NotSupportedException();

            public IEnumerator<T> GetEnumerator()
            {
                for (var i = 0; i < Length; i++)
                {
                    yield return this[i];
                }
            }

            IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

            #endregion
        }
    }
```

./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase
    {
        public static readonly int DefaultMinimumStringSegmentLength = 2;
    }
}
```

./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs

```csharp
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
    {
```

```
  9         protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
        ↪  => new Segment<T>(elements, offset, length);
 10     }
 11 }
```

./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs

```
  1 using System.Collections.Generic;
  2
  3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  4
  5 namespace Platform.Collections.Segments.Walkers
  6 {
  7     public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
  8         where TSegment : Segment<T>
  9     {
 10         private readonly int _minimumStringSegmentLength;
 11
 12         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
        ↪  _minimumStringSegmentLength = minimumStringSegmentLength;
 13
 14         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
 15
 16         public virtual void WalkAll(IList<T> elements)
 17         {
 18             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
                ↪  offset <= maxOffset; offset++)
 19             {
 20                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                    ↪  offset; length <= maxLength; length++)
 21                 {
 22                     Iteration(CreateSegment(elements, offset, length));
 23                 }
 24             }
 25         }
 26
 27         protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
 28
 29         protected abstract void Iteration(TSegment segment);
 30     }
 31 }
```

./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs

```
  1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  2
  3 namespace Platform.Collections.Segments.Walkers
  4 {
  5     public static class AllSegmentsWalkerExtensions
  6     {
  7         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
            ↪  walker.WalkAll(@string.ToCharArray());
  8         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
            ↪  string @string) where TSegment : Segment<char> =>
            ↪  walker.WalkAll(@string.ToCharArray());
  9     }
 10 }
```

./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs

```
  1 using System.Collections.Generic;
  2
  3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  4
  5 namespace Platform.Collections.Segments.Walkers
  6 {
  7     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
  8     {
  9         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
            ↪  base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
 10         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪  dictionary, int minimumStringSegmentLength) : base(dictionary,
            ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 11         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪  dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
            ↪  DefaultResetDictionaryOnEachWalk) { }
 12         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            ↪  bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
            ↪  resetDictionaryOnEachWalk) { }
```

```
 13       protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
          ↪  base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 14       protected DictionaryBasedDuplicateSegmentsWalkerBase() :
          ↪  base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 15   }
 16 }
```

./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs

```
  1 using System;
  2 using System.Collections.Generic;
  3
  4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  5
  6 namespace Platform.Collections.Segments.Walkers
  7 {
  8     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
        ↪  DuplicateSegmentsWalkerBase<T, TSegment>
  9         where TSegment : Segment<T>
 10     {
 11         public static readonly bool DefaultResetDictionaryOnEachWalk;
 12
 13         private readonly bool _resetDictionaryOnEachWalk;
 14         protected IDictionary<TSegment, long> Dictionary;
 15
 16         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
 17             : base(minimumStringSegmentLength)
 18         {
 19             Dictionary = dictionary;
 20             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
 21         }
 22
 23         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪  dictionary, int minimumStringSegmentLength) : this(dictionary,
            ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 24
 25         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪  dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
            ↪  DefaultResetDictionaryOnEachWalk) { }
 26
 27         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            ↪  bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
            ↪  Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
            ↪  { }
 28
 29         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪  this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 30
 31         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            ↪  this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
 32
 33         public override void WalkAll(IList<T> elements)
 34         {
 35             if (_resetDictionaryOnEachWalk)
 36             {
 37                 var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
 38                 Dictionary = new Dictionary<TSegment, long>((int)capacity);
 39             }
 40             base.WalkAll(elements);
 41         }
 42
 43         protected override long GetSegmentFrequency(TSegment segment) =>
            ↪  Dictionary.GetOrDefault(segment);
 44
 45         protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
            ↪  Dictionary[segment] = frequency;
 46     }
 47 }
```

./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs

```
  1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  2
  3 namespace Platform.Collections.Segments.Walkers
  4 {
  5     public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
        ↪  Segment<T>>
  6     {
  7     }
  8 }
```

./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Segments.Walkers
4   {
5       public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
    ↪   TSegment>
6           where TSegment : Segment<T>
7       {
8           protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪   base(minimumStringSegmentLength) { }
9
10          protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
11
12          protected override void Iteration(TSegment segment)
13          {
14              var frequency = GetSegmentFrequency(segment);
15              if (frequency == 1)
16              {
17                  OnDublicateFound(segment);
18              }
19              SetSegmentFrequency(segment, frequency + 1);
20          }
21
22          protected abstract void OnDublicateFound(TSegment segment);
23          protected abstract long GetSegmentFrequency(TSegment segment);
24          protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
25      }
26  }
```

./Platform.Collections/Stacks/DefaultStack.cs

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Stacks
6   {
7       public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
8       {
9           public bool IsEmpty => Count <= 0;
10      }
11  }
```

./Platform.Collections/Stacks/IStack.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Stacks
4   {
5       public interface IStack<TElement>
6       {
7           bool IsEmpty { get; }
8           void Push(TElement element);
9           TElement Pop();
10          TElement Peek();
11      }
12  }
```

./Platform.Collections/Stacks/IStackExtensions.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Stacks
6   {
7       public static class IStackExtensions
8       {
9           public static void Clear<T>(this IStack<T> stack)
10          {
11              while (!stack.IsEmpty)
12              {
13                  _ = stack.Pop();
14              }
15          }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
            ↪   stack.Pop();
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
21        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
   ↪    stack.Peek();
22    }
23 }
```

./Platform.Collections/Stacks/IStackFactory.cs

```
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
8      {
9      }
10 }
```

./Platform.Collections/Stacks/StackExtensions.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Stacks
7  {
8      public static class StackExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
   ↪    default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
   ↪    : default;
15     }
16 }
```

./Platform.Collections/StringExtensions.cs

```
1  using System;
2  using System.Globalization;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections
7  {
8      public static class StringExtensions
9      {
10         public static string CapitalizeFirstLetter(this string @string)
11         {
12             if (string.IsNullOrWhiteSpace(@string))
13             {
14                 return @string;
15             }
16             var chars = @string.ToCharArray();
17             for (var i = 0; i < chars.Length; i++)
18             {
19                 var category = char.GetUnicodeCategory(chars[i]);
20                 if (category == UnicodeCategory.UppercaseLetter)
21                 {
22                     return @string;
23                 }
24                 if (category == UnicodeCategory.LowercaseLetter)
25                 {
26                     chars[i] = char.ToUpper(chars[i]);
27                     return new string(chars);
28                 }
29             }
30             return @string;
31         }
32
33         public static string Truncate(this string @string, int maxLength) =>
   ↪    string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
   ↪    Math.Min(@string.Length, maxLength));
34
35         public static string TrimSingle(this string @string, char charToTrim)
36         {
37             if (!string.IsNullOrEmpty(@string))
38             {
39                 if (@string.Length == 1)
```

```
40                {
41                    if (@string[0] == charToTrim)
42                    {
43                        return "";
44                    }
45                    else
46                    {
47                        return @string;
48                    }
49                }
50                else
51                {
52                    var left = 0;
53                    var right = @string.Length - 1;
54                    if (@string[left] == charToTrim)
55                    {
56                        left++;
57                    }
58                    if (@string[right] == charToTrim)
59                    {
60                        right--;
61                    }
62                    return @string.Substring(left, right - left + 1);
63                }
64            }
65            else
66            {
67                return @string;
68            }
69        }
70    }
71 }
```

./Platform.Collections/Trees/Node.cs

```
1  using System.Collections.Generic;
2
3  // ReSharper disable ForCanBeConvertedToForeach
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Trees
7  {
8      public class Node
9      {
10         private Dictionary<object, Node> _childNodes;
11
12         public object Value { get; set; }
13
14         public Dictionary<object, Node> ChildNodes => _childNodes ?? (_childNodes = new
           ↪  Dictionary<object, Node>());
15
16         public Node this[object key]
17         {
18             get
19             {
20                 var child = GetChild(key);
21                 if (child == null)
22                 {
23                     child = AddChild(key);
24                 }
25                 return child;
26             }
27             set => SetChildValue(value, key);
28         }
29
30         public Node(object value) => Value = value;
31
32         public Node() : this(null) { }
33
34         public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
35
36         public Node GetChild(params object[] keys)
37         {
38             var node = this;
39             for (var i = 0; i < keys.Length; i++)
40             {
41                 node.ChildNodes.TryGetValue(keys[i], out node);
42                 if (node == null)
43                 {
44                     return null;
45                 }
```

```
46                   }
47                   return node;
48               }
49
50               public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
51
52               public Node AddChild(object key) => AddChild(key, new Node(null));
53
54               public Node AddChild(object key, object value) => AddChild(key, new Node(value));
55
56               public Node AddChild(object key, Node child)
57               {
58                   ChildNodes.Add(key, child);
59                   return child;
60               }
61
62               public Node SetChild(params object[] keys) => SetChildValue(null, keys);
63
64               public Node SetChild(object key) => SetChildValue(null, key);
65
66               public Node SetChildValue(object value, params object[] keys)
67               {
68                   var node = this;
69                   for (var i = 0; i < keys.Length; i++)
70                   {
71                       node = SetChildValue(value, keys[i]);
72                   }
73                   node.Value = value;
74                   return node;
75               }
76
77               public Node SetChildValue(object value, object key)
78               {
79                   if (!ChildNodes.TryGetValue(key, out Node child))
80                   {
81                       child = AddChild(key, value);
82                   }
83                   child.Value = value;
84                   return child;
85               }
86           }
87   }
```

./Platform.Collections.Tests/BitStringTests.cs

```
1    using System;
2    using System.Collections;
3    using Xunit;
4    using Platform.Random;
5
6    namespace Platform.Collections.Tests
7    {
8        public static class BitStringTests
9        {
10           [Fact]
11           public static void BitGetSetTest()
12           {
13               const int n = 250;
14               var bitArray = new BitArray(n);
15               var bitString = new BitString(n);
16               for (var i = 0; i < n; i++)
17               {
18                   var value = RandomHelpers.Default.NextBoolean();
19                   bitArray.Set(i, value);
20                   bitString.Set(i, value);
21                   Assert.Equal(value, bitArray.Get(i));
22                   Assert.Equal(value, bitString.Get(i));
23               }
24           }
25
26           [Fact]
27           public static void BitAndTest()
28           {
29               TestToOperationsWithSameMeaning((x, y, w, v) =>
30               {
31                   x.VectorAnd(y);
32                   w.And(v);
33               });
34           }
35
36           [Fact]
```

```csharp
37      public static void BitNotTest()
38      {
39          TestToOperationsWithSameMeaning((x, y, w, v) =>
40          {
41              x.VectorNot();
42              w.Not();
43          });
44      }
45
46      [Fact]
47      public static void BitOrTest()
48      {
49          TestToOperationsWithSameMeaning((x, y, w, v) =>
50          {
51              x.VectorOr(y);
52              w.Or(v);
53          });
54      }
55
56      private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
        ↪  BitString, BitString> test)
57      {
58          const int n = 250;
59          var x = new BitString(n);
60          var y = new BitString(n);
61          x.SetRandomBits();
62          y.SetRandomBits();
63          var w = new BitString(x);
64          var v = new BitString(y);
65          test(x, y, w, v);
66          Assert.True(x.Equals(w));
67      }
68  }
69 }
```

./Platform.Collections.Tests/CharsSegmentTests.cs

```csharp
1  using Xunit;
2  using Platform.Collections.Segments;
3
4  namespace Platform.Collections.Tests
5  {
6      public static class CharsSegmentTests
7      {
8          [Fact]
9          public static void GetHashCodeEqualsTest()
10         {
11             const string testString = "test test";
12             var testArray = testString.ToCharArray();
13             var first = new CharSegment(testArray, 0, 4);
14             var firstHashCode = first.GetHashCode();
15             var second = new CharSegment(testArray, 5, 4);
16             var secondHashCode = second.GetHashCode();
17             Assert.Equal(firstHashCode, secondHashCode);
18         }
19
20         [Fact]
21         public static void EqualsTest()
22         {
23             const string testString = "test test";
24             var testArray = testString.ToCharArray();
25             var first = new CharSegment(testArray, 0, 4);
26             var second = new CharSegment(testArray, 5, 4);
27             Assert.True(first.Equals(second));
28         }
29     }
30 }
```

./Platform.Collections.Tests/StringTests.cs

```csharp
1  using Xunit;
2
3  namespace Platform.Collections.Tests
4  {
5      public static class StringTests
6      {
7          [Fact]
8          public static void CapitalizeFirstLetterTest()
9          {
10             var source1 = "hello";
11             var result1 = source1.CapitalizeFirstLetter();
```

```csharp
                Assert.Equal("Hello", result1);
                var source2 = "Hello";
                var result2 = source2.CapitalizeFirstLetter();
                Assert.Equal("Hello", result2);
                var source3 = "  hello";
                var result3 = source3.CapitalizeFirstLetter();
                Assert.Equal("  Hello", result3);
        }

        [Fact]
        public static void TrimSingleTest()
        {
                var source1 = "'";
                var result1 = source1.TrimSingle('\'');
                Assert.Equal("", result1);
                var source2 = "''";
                var result2 = source2.TrimSingle('\'');
                Assert.Equal("", result2);
                var source3 = "'hello'";
                var result3 = source3.TrimSingle('\'');
                Assert.Equal("hello", result3);
                var source4 = "hello'";
                var result4 = source4.TrimSingle('\'');
                Assert.Equal("hello", result4);
                var source5 = "'hello";
                var result5 = source5.TrimSingle('\'');
                Assert.Equal("hello", result5);
        }
    }
}
```

Index