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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
    {
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) : base(array, offset) => _returnConstant = returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0, returnConstant) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element) => _array.AddAndReturnConstant(ref _position, element, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements) => _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements) => _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements) => _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement>
    {
        protected readonly TElement[] _array;
        protected long _position;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, long offset)
        {
            _array = array;
            _position = offset;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array) : this(array, 0) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _array[_position++] = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref _position, element, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> elements) => _array.AddFirstAndReturnConstant(ref _position, elements, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAllAndReturnTrue(IList<TElement> elements) => _array.AddAllAndReturnConstant(ref _position, elements, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
        ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
    }
}
```

1.3 ./csharp/Platform.Collections/Arrays/ArrayPool.cs

```
using System.Runtime.CompilerServices;

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

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);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
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>
    {
        // May be use Default class for that later.
        [ThreadStatic]
        private static ArrayPool<T> _threadInstance;
        internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
        ↪ ArrayPool<T>());

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Disposable<T[]> Resize(Disposable<T[]> source, long size)
        {
            var destination = AllocateDisposable(size);
            T[] sourceArray = source;
            if (!sourceArray.IsNullOrEmpty())
            {
                T[] destinationArray = destination;
                Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                ↪ sourceArray.LongLength);
                source.Dispose();
            }
            return destination;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Clear() => _pool.Clear();

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
        ↪ _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Free(T[] array)
        {
            if (array.IsNullOrEmpty())
            {
                return;
            }
            var stack = _pool.GetOrAdd(array.LongLength, size => new
            ↪ Stack<T[]>(_maxArraysPerSize));
            if (stack.Count == _maxArraysPerSize) // Stack is full
            {
                return;
            }
            stack.Push(array);
        }
    }
}
```

1.5 ./csharp/Platform.Collections/Arrays/ArrayString.cs

```
using System.Runtime.CompilerServices;
using Platform.Collections.Segments;

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

namespace Platform.Collections.Arrays
{
    public class ArrayString<T> : Segment<T>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(int length) : base(new T[length], 0, length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(T[] array) : base(array, 0, array.Length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(T[] array, int length) : base(array, 0, length) { }
    }
}
```

1.6 ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Arrays
{
    public static unsafe class CharArrayExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GenerateHashCode(this char[] array, int offset, int length)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            fixed (char* arrayPointer = &array[offset])
            {
                for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
                ↪ < last; charPointer++)
                {
                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
                }
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
        ↪ right, int rightOffset)
```

```
32          {
33              fixed (char* leftPointer = &left[leftOffset])
34              {
35                  fixed (char* rightPointer = &right[rightOffset])
36                  {
37                      char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
38                      if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
                        ↪  rightPointerCopy, ref length))
39                      {
40                          return false;
41                      }
42                      CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
                        ↪  ref length);
43                      return length <= 0;
44                  }
45              }
46          }
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
            ↪  int length)
50          {
51              while (length >= 10)
52              {
53                  if ((*(int*)left != *(int*)right)
54                   || (*(int*)(left + 2) != *(int*)(right + 2))
55                   || (*(int*)(left + 4) != *(int*)(right + 4))
56                   || (*(int*)(left + 6) != *(int*)(right + 6))
57                   || (*(int*)(left + 8) != *(int*)(right + 8)))
58                  {
59                      return false;
60                  }
61                  left += 10;
62                  right += 10;
63                  length -= 10;
64              }
65              return true;
66          }
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
            ↪  int length)
70          {
71              // This depends on the fact that the String objects are
72              // always zero terminated and that the terminating zero is not included
73              // in the length. For odd string sizes, the last compare will include
74              // the zero terminator.
75              while (length > 0)
76              {
77                  if (*(int*)left != *(int*)right)
78                  {
79                      break;
80                  }
81                  left += 2;
82                  right += 2;
83                  length -= 2;
84              }
85          }
86      }
87  }
```

1.7 ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections.Arrays
8   {
9       public static class GenericArrayExtensions
10      {
11          /// <summary>
12          /// <para>Checks if an array exists, if so,  checks the array length using the  index
            ↪  variable type int, and if the array length is greater than the index - return
            ↪  array[index], otherwise - default value.</para>
13          /// <para>Проверяется, существует ли массив, если да - идет проверка длины массива с
            ↪  помощью переменной index, и если длина массива больше индекса - возвращается
            ↪  array[index], иначе - значение по умолчанию.</para>
```

```
14          /// </summary>
15          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
16          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
17          /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
            ↪  сравнения.</para></param>
18          /// <returns><para>Array element or default value.</para><para>Элемент массива или же
            ↪  значение по умолчанию.</para></returns>
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
            ↪  array.Length > index ? array[index] : default;
21
22          /// <summary>
23          /// <para>Checks whether the array exists, if so, checks the array length using the
            ↪  index variable type long, and if the array length is greater than the index - return
            ↪  array[index], otherwise - default value.</para>
24          /// <para>Проверяется, существует ли массив, если да - идет проверка длины массива с
            ↪  помощью переменной index, и если длина массива больше индекса - возвращается
            ↪  array[index], иначе - значение по умолчанию.</para>
25          /// </summary>
26          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
27          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
28          /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
            ↪  для сравнения.</para></param>
29          /// <returns><para>Array element or default value.</para><para>Элемент массива или же
            ↪  значение по умолчанию.</para></returns>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
            ↪  array.LongLength > index ? array[index] : default;
32
33          /// <summary>
34          /// <para>Checks whether the array exist, if so, checks the array length using the index
            ↪  varible type int, and if the array length is greater than the index, set the element
            ↪  variable to array[index] and return true.</para>
35          /// <para>Проверяется, существует ли массив, если да, то  идет проверка  длины массива с
            ↪  помощью переменной index  типа int, и если длина массива больше значения index,
            ↪  устанавливается значение переменной element - array[index] и возвращается
            ↪  true.</para>
36          /// </summary>
37          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
38          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
39          /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
            ↪  сравнения.</para></param>
40          /// <param name="element"><para>Passing the argument by reference, if successful, it
            ↪  will take the value array[index] otherwise default value.</para><para>Передается
            ↪  аргумент по ссылке, в случае успеха он примет значение array[index] в противном
            ↪  случае значение по умолчанию.</para></param>
41          /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
            ↪  в противном случае false</para></returns>
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static bool TryGetElement<T>(this T[] array, int index, out T element)
44          {
45              if (array != null && array.Length > index)
46              {
47                  element = array[index];
48                  return true;
49              }
50              else
51              {
52                  element = default;
53                  return false;
54              }
55          }
56
57          /// <summary>
58          /// <para>Checks whether the array exist, if so,  checks the array length using the
            ↪  index varible type long, and if the array length is greater than the index,  set the
            ↪  element variable to array[index] and return true.</para>
```

```
59          /// <para>Проверяется, существует ли массив, если да, то идет проверка длины массива с
                ↪ помощью переменной index  типа long, и если длина массива больше значения index,
                ↪ устанавливается значение переменной element - array[index] и возвращается
                ↪ true.</para>
60          /// </summary>
61          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
                ↪ массива.</para></typeparam>
62          /// <param name="array"><para>Array that will participate in
                ↪ verification.</para><para>Массив который будет учавствовать в
                ↪ проверке.</para></param>
63          /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
                ↪ для сравнения.</para></param>
64          /// <param name="element"><para>Passing the argument by reference, if successful, it
                ↪ will take the value array[index] otherwise default value.</para><para>Передаём
                ↪ аргумент по ссылке, в случае успеха он примет значение array[index] в противном
                ↪ случае значение по умолчанию.</para></param>
65          /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
                ↪ в противном случае false</para></returns>
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public static bool TryGetElement<T>(this T[] array, long index, out T element)
68          {
69              if (array != null && array.LongLength > index)
70              {
71                  element = array[index];
72                  return true;
73              }
74              else
75              {
76                  element = default;
77                  return false;
78              }
79          }
80
81          /// <summary>
82          /// <para>Copying  of elements from one array to another array.</para>
83          /// <para>Копируется  элементы из одного массива в другой массив.</para>
84          /// </summary>
85          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
                ↪ массива.</para></typeparam>
86          /// <param name="array"><para>The array  to copy.</para><para>Массив который необходимо
                ↪ скопировать.</para></param>
87          /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          public static T[] Clone<T>(this T[] array)
90          {
91              var copy = new T[array.LongLength];
92              Array.Copy(array, 0L, copy, 0L, array.LongLength);
93              return copy;
94          }
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
98
99          /// <summary>
100         /// <para>Extending the array boundaries to shift elements and then copying it, but with
                ↪ the condition that shift > 0. If shift = = 0, the extension will not occur, but
                ↪ cloning will still be applied. If shift < 0, a NotImplementedException is
                ↪ thrown.</para>
101         /// <para>Расширение границ массива на shift элементов и последующее его копирование, но
                ↪ с условием что shift > 0. Если же shift == 0 - расширение не произойдет , но
                ↪ клонирование все равно применится. Если shift < 0, выбросится исключение
                ↪ NotImplementedException.</para>
102         /// </summary>
103         /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
                ↪ массива.</para></typeparam>
104         /// <param name="array"><para>Array to expand  Elements.</para><para>Массив для
                ↪ расширения элементов.</para></param>
105         /// <param name="shift"><para>The number to expand the array</para><para>Число на
                ↪ которое необходимо рассширить массив.</para></param>
106         /// <returns>
107         /// <para>If the value of the shift variable is < 0, it returns a
                ↪ NotImplementedException exception. If shift = = 0, the array is cloned, but the
                ↪ extension will not be applied. Otherwise, if the value shift > 0, the length of the
                ↪ array is increased by the shift elements and the array is cloned.</para>
```

```
108         /// <para>Если значение переменной shift < 0,  возвращается исключение
            ↪  NotImplementedException. Если shift = = 0, то массив клонируется, но расширение не
            ↪  применяется. В противном случае, если значение shift > 0, длина массива
            ↪  увеличивается на shift элементов  и массив клонируется.</para>
109         /// </returns>
110         [MethodImpl(MethodImplOptions.AggressiveInlining)]
111         public static IList<T> ShiftRight<T>(this T[] array, long shift)
112         {
113             if (shift < 0)
114             {
115                 throw new NotImplementedException();
116             }
117             if (shift == 0)
118             {
119                 return array.Clone<T>();
120             }
121             else
122             {
123                 var restrictions = new T[array.LongLength + shift];
124                 Array.Copy(array, 0L, restrictions, shift, array.LongLength);
125                 return restrictions;
126             }
127         }
128
129         [MethodImpl(MethodImplOptions.AggressiveInlining)]
130         public static void Add<T>(this T[] array, ref int position, T element) =>
            ↪  array[position++] = element;
131
132         [MethodImpl(MethodImplOptions.AggressiveInlining)]
133         public static void Add<T>(this T[] array, ref long position, T element) =>
            ↪  array[position++] = element;
134
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, TElement element, TReturnConstant
            ↪  returnConstant)
137         {
138             array.Add(ref position, element);
139             return returnConstant;
140         }
141
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
            ↪  array[position++] = elements[0];
144
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪  returnConstant)
147         {
148             array.AddFirst(ref position, elements);
149             return returnConstant;
150         }
151
152         [MethodImpl(MethodImplOptions.AggressiveInlining)]
153         public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪  returnConstant)
154         {
155             array.AddAll(ref position, elements);
156             return returnConstant;
157         }
158
159         [MethodImpl(MethodImplOptions.AggressiveInlining)]
160         public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
161         {
162             for (var i = 0; i < elements.Count; i++)
163             {
164                 array.Add(ref position, elements[i]);
165             }
166         }
167
168         [MethodImpl(MethodImplOptions.AggressiveInlining)]
169         public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
            ↪  TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
            ↪  TReturnConstant returnConstant)
170         {
171             array.AddSkipFirst(ref position, elements);
```

```
            return returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
        ↪  => array.AddSkipFirst(ref position, elements, 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
        ↪  int skip)
        {
            for (var i = skip; i < elements.Count; i++)
            {
                array.Add(ref position, elements[i]);
            }
        }
    }
}
```

1.8 ./csharp/Platform.Collections/BitString.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;
using Platform.Exceptions;
using Platform.Ranges;

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

namespace Platform.Collections
{
    /// <remarks>
    /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
    ↪  64 бит в массиве значений.
    /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
    ↪  байт в 8 байт.
    /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
    ↪  помощью которой можно быстро
    /// проверять есть ли значения непосредственно далее (ниже по уровню).
    /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
    /// </remarks>
    public class BitString : IEquatable<BitString>
    {
        private static readonly byte[][] _bitsSetIn16Bits;
        private long[] _array;
        private long _length;
        private long _minPositiveWord;
        private long _maxPositiveWord;

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

        public long Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _length;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    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++;
            }
            _bitsSetIn16Bits[i] = array;
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    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);
    }

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString(long length, bool defaultValue)
```

```
        : this(length)
    {
        if (defaultValue)
        {
            SetAll();
        }
    }

    #endregion

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString Not()
    {
        for (var i = 0L; i < _array.LongLength; i++)
        {
            _array[i] = ~_array[i];
            RefreshBordersByWord(i);
        }
        return this;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString ParallelNot()
    {
        var threads = Environment.ProcessorCount / 2;
        if (threads <= 1)
        {
            return Not();
        }
        var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
        ↪  threads);
        Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪  MaxDegreeOfParallelism = threads }, range =>
        {
            var maximum = range.Item2;
            for (var i = range.Item1; i < maximum; i++)
            {
                _array[i] = ~_array[i];
            }
        });
        MarkBordersAsAllBitsSet();
        TryShrinkBorders();
        return this;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString VectorNot()
    {
        if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
        {
            return Not();
        }
        var step = Vector<long>.Count;
        if (_array.Length < step)
        {
            return Not();
        }
        VectorNotLoop(_array, step, 0, _array.Length);
        MarkBordersAsAllBitsSet();
        TryShrinkBorders();
        return this;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString ParallelVectorNot()
    {
        var threads = Environment.ProcessorCount / 2;
        if (threads <= 1)
        {
            return VectorNot();
        }
        if (!Vector.IsHardwareAccelerated)
        {
            return ParallelNot();
        }
        var step = Vector<long>.Count;
        if (_array.Length < (step * threads))
        {
```

```
            return VectorNot();
        }
        var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
        Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪  MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
        ↪  range.Item1, range.Item2));
        MarkBordersAsAllBitsSet();
        TryShrinkBorders();
        return this;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static private void VectorNotLoop(long[] array, int step, int start, int maximum)
    {
        var i = start;
        var range = maximum - start - 1;
        var stop = range - (range % step);
        for (; i < stop; i += step)
        {
            (~new Vector<long>(array, i)).CopyTo(array, i);
        }
        for (; i < maximum; i++)
        {
            array[i] = ~array[i];
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString And(BitString other)
    {
        EnsureBitStringHasTheSameSize(other, nameof(other));
        GetCommonOuterBorders(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;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString ParallelAnd(BitString other)
    {
        var threads = Environment.ProcessorCount / 2;
        if (threads <= 1)
        {
            return And(other);
        }
        EnsureBitStringHasTheSameSize(other, nameof(other));
        GetCommonOuterBorders(this, other, out long from, out long to);
        var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
        Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪  MaxDegreeOfParallelism = threads }, range =>
        {
            var maximum = range.Item2;
            for (var i = range.Item1; i < maximum; i++)
            {
                _array[i] &= other._array[i];
            }
        });
        MarkBordersAsAllBitsSet();
        TryShrinkBorders();
        return this;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString VectorAnd(BitString other)
    {
        if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
        {
            return And(other);
        }
        var step = Vector<long>.Count;
        if (_array.Length < step)
        {
            return And(other);
        }
```

```
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorAndLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorAnd(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorAnd(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelAnd(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorAnd(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
            ↪  step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
        ↪  int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] &= otherArray[i];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Or(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] |= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelOr(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Or(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
```

```
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions { MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
                for (var i = range.Item1; i < maximum; i++)
                {
                    _array[i] |= other._array[i];
                }
            });
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString VectorOr(BitString other)
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return Or(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return Or(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorOrLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorOr(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorOr(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelOr(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorOr(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions { MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array, step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start, int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] |= otherArray[i];
            }
        }
```

```
435
436         [MethodImpl(MethodImplOptions.AggressiveInlining)]
437         public BitString Xor(BitString other)
438         {
439             EnsureBitStringHasTheSameSize(other, nameof(other));
440             GetCommonOuterBorders(this, other, out long from, out long to);
441             for (var i = from; i <= to; i++)
442             {
443                 _array[i] ^= other._array[i];
444                 RefreshBordersByWord(i);
445             }
446             return this;
447         }
448
449         [MethodImpl(MethodImplOptions.AggressiveInlining)]
450         public BitString ParallelXor(BitString other)
451         {
452             var threads = Environment.ProcessorCount / 2;
453             if (threads <= 1)
454             {
455                 return Xor(other);
456             }
457             EnsureBitStringHasTheSameSize(other, nameof(other));
458             GetCommonOuterBorders(this, other, out long from, out long to);
459             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
460             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                ↪ MaxDegreeOfParallelism = threads }, range =>
461             {
462                 var maximum = range.Item2;
463                 for (var i = range.Item1; i < maximum; i++)
464                 {
465                     _array[i] ^= other._array[i];
466                 }
467             });
468             MarkBordersAsAllBitsSet();
469             TryShrinkBorders();
470             return this;
471         }
472
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public BitString VectorXor(BitString other)
475         {
476             if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
477             {
478                 return Xor(other);
479             }
480             var step = Vector<long>.Count;
481             if (_array.Length < step)
482             {
483                 return Xor(other);
484             }
485             EnsureBitStringHasTheSameSize(other, nameof(other));
486             GetCommonOuterBorders(this, other, out int from, out int to);
487             VectorXorLoop(_array, other._array, step, from, to + 1);
488             MarkBordersAsAllBitsSet();
489             TryShrinkBorders();
490             return this;
491         }
492
493         [MethodImpl(MethodImplOptions.AggressiveInlining)]
494         public BitString ParallelVectorXor(BitString other)
495         {
496             var threads = Environment.ProcessorCount / 2;
497             if (threads <= 1)
498             {
499                 return VectorXor(other);
500             }
501             if (!Vector.IsHardwareAccelerated)
502             {
503                 return ParallelXor(other);
504             }
505             var step = Vector<long>.Count;
506             if (_array.Length < (step * threads))
507             {
508                 return VectorXor(other);
509             }
510             EnsureBitStringHasTheSameSize(other, nameof(other));
511             GetCommonOuterBorders(this, other, out int from, out int to);
```

```
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
            ↪  step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
        ↪  int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] ^= otherArray[i];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void RefreshBordersByWord(long wordIndex)
        {
            if (_array[wordIndex] == 0)
            {
                if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
                {
                    _minPositiveWord++;
                }
                if (wordIndex == _maxPositiveWord && wordIndex != 0)
                {
                    _maxPositiveWord--;
                }
            }
            else
            {
                if (wordIndex < _minPositiveWord)
                {
                    _minPositiveWord = wordIndex;
                }
                if (wordIndex > _maxPositiveWord)
                {
                    _maxPositiveWord = wordIndex;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryShrinkBorders()
        {
            GetBorders(out long from, out long to);
            while (from <= to && _array[from] == 0)
            {
                from++;
            }
            if (from > to)
            {
                MarkBordersAsAllBitsReset();
                return true;
            }
            while (to >= from && _array[to] == 0)
            {
                to--;
            }
            if (to < from)
            {
                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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll(bool value)
        {
            if (value)
            {
                SetAll();
            }
            else
            {
                ResetAll();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll()
        {
            const long fillValue = unchecked((long)0xffffffffffffffff);
            var words = GetWordsCountFromIndex(_length);
```

```
            for (var i = 0; i < words; i++)
            {
                _array[i] = fillValue;
            }
            MarkBordersAsAllBitsSet();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ResetAll()
        {
            const long fillValue = 0;
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] = fillValue;
            }
            MarkBordersAsAllBitsReset();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<long> GetSetIndices()
        {
            var result = new List<long>();
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    AppendAllSetBitIndices(result, i, word);
                }
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSetUInt64Indices()
        {
            var result = new List<ulong>();
            GetBorders(out ulong from, out ulong to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    AppendAllSetBitIndices(result, i, word);
                }
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetFirstSetBitIndex()
        {
            var i = _minPositiveWord;
            var word = _array[i];
            if (word != 0)
            {
                return GetFirstSetBitForWord(i, word);
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetLastSetBitIndex()
        {
            var i = _maxPositiveWord;
            var word = _array[i];
            if (word != 0)
            {
                return GetLastSetBitForWord(i, word);
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long CountSetBits()
        {
            var total = 0L;
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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)
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
        ↪  false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
```

```
903                 }
904             }
905             return true;
906         }
907 
908         [MethodImpl(MethodImplOptions.AggressiveInlining)]
909         private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
910         {
911             Ensure.Always.ArgumentNotNull(other, argumentName);
912             if (_length != other._length)
913             {
914                 throw new ArgumentException("Bit string must be the same size.", argumentName);
915             }
916         }
917 
918         [MethodImpl(MethodImplOptions.AggressiveInlining)]
919         private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
920 
921         [MethodImpl(MethodImplOptions.AggressiveInlining)]
922         private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
923 
924         [MethodImpl(MethodImplOptions.AggressiveInlining)]
925         private void GetBorders(out long from, out long to)
926         {
927             from = _minPositiveWord;
928             to = _maxPositiveWord;
929         }
930 
931         [MethodImpl(MethodImplOptions.AggressiveInlining)]
932         private void GetBorders(out ulong from, out ulong to)
933         {
934             from = (ulong)_minPositiveWord;
935             to = (ulong)_maxPositiveWord;
936         }
937 
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         private void SetBorders(long from, long to)
940         {
941             _minPositiveWord = from;
942             _maxPositiveWord = to;
943         }
944 
945         [MethodImpl(MethodImplOptions.AggressiveInlining)]
946         private Range<long> GetValidIndexRange() => (0, _length - 1);
947 
948         [MethodImpl(MethodImplOptions.AggressiveInlining)]
949         private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
950 
951         [MethodImpl(MethodImplOptions.AggressiveInlining)]
952         private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
            ↪  wordValue)
953         {
954             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                ↪  bits32to47, out byte[] bits48to63);
955             AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
                ↪  bits48to63);
956         }
957 
958         [MethodImpl(MethodImplOptions.AggressiveInlining)]
959         private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
            ↪  wordValue)
960         {
961             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                ↪  bits32to47, out byte[] bits48to63);
962             AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
                ↪  bits48to63);
963         }
964 
965         [MethodImpl(MethodImplOptions.AggressiveInlining)]
966         private static long CountSetBitsForWord(long word)
967         {
968             GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
                ↪  out byte[] bits48to63);
969             return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
                ↪  bits48to63.LongLength;
970         }
971 
972         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
973         private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
974         {
975             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                ↪  bits32to47, out byte[] bits48to63);
976             return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
977         }
978 
979         [MethodImpl(MethodImplOptions.AggressiveInlining)]
980         private static long GetLastSetBitForWord(long wordIndex, long wordValue)
981         {
982             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                ↪  bits32to47, out byte[] bits48to63);
983             return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
984         }
985 
986         [MethodImpl(MethodImplOptions.AggressiveInlining)]
987         private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
                ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
988         {
989             for (var j = 0; j < bits00to15.Length; j++)
990             {
991                 result.Add(bits00to15[j] + (i * 64));
992             }
993             for (var j = 0; j < bits16to31.Length; j++)
994             {
995                 result.Add(bits16to31[j] + 16 + (i * 64));
996             }
997             for (var j = 0; j < bits32to47.Length; j++)
998             {
999                 result.Add(bits32to47[j] + 32 + (i * 64));
1000            }
1001            for (var j = 0; j < bits48to63.Length; j++)
1002            {
1003                result.Add(bits48to63[j] + 48 + (i * 64));
1004            }
1005        }
1006 
1007        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1008        private static void AppendAllSetBitIndices(List<ulong> result, ulong i, byte[] bits00to15,
                ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1009        {
1010            for (var j = 0; j < bits00to15.Length; j++)
1011            {
1012                result.Add(bits00to15[j] + (i * 64));
1013            }
1014            for (var j = 0; j < bits16to31.Length; j++)
1015            {
1016                result.Add(bits16to31[j] + 16UL + (i * 64));
1017            }
1018            for (var j = 0; j < bits32to47.Length; j++)
1019            {
1020                result.Add(bits32to47[j] + 32UL + (i * 64));
1021            }
1022            for (var j = 0; j < bits48to63.Length; j++)
1023            {
1024                result.Add(bits48to63[j] + 48UL + (i * 64));
1025            }
1026        }
1027 
1028        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029        private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
                ↪  bits32to47, byte[] bits48to63)
1030        {
1031            if (bits00to15.Length > 0)
1032            {
1033                return bits00to15[0] + (i * 64);
1034            }
1035            if (bits16to31.Length > 0)
1036            {
1037                return bits16to31[0] + 16 + (i * 64);
1038            }
1039            if (bits32to47.Length > 0)
1040            {
1041                return bits32to47[0] + 32 + (i * 64);
1042            }
1043            return bits48to63[0] + 48 + (i * 64);
1044        }
```

```
1045
1046            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1047            private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
                ↪  bits32to47, byte[] bits48to63)
1048            {
1049                if (bits48to63.Length > 0)
1050                {
1051                    return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1052                }
1053                if (bits32to47.Length > 0)
1054                {
1055                    return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1056                }
1057                if (bits16to31.Length > 0)
1058                {
1059                    return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1060                }
1061                return bits00to15[bits00to15.Length - 1] + (i * 64);
1062            }
1063
1064            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1065            private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
                ↪  byte[] bits32to47, out byte[] bits48to63)
1066            {
1067                bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1068                bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1069                bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1070                bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1071            }
1072
1073            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074            public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
                ↪  out long to)
1075            {
1076                from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1077                to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1078            }
1079
1080            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081            public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
                ↪  out long to)
1082            {
1083                from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1084                to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1085            }
1086
1087            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1088            public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
                ↪  out int to)
1089            {
1090                from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1091                to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1092            }
1093
1094            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1095            public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
                ↪  ulong to)
1096            {
1097                from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1098                to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1099            }
1100
1101            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1102            public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1103
1104            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1105            public static long GetWordIndexFromIndex(long index) => index >> 6;
1106
1107            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1108            public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1109
1110            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1111            public override int GetHashCode() => base.GetHashCode();
1112
1113            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1114            public override string ToString() => base.ToString();
1115        }
1116    }
```

1.9 ./csharp/Platform.Collections/BitStringExtensions.cs

```
using System.Runtime.CompilerServices;
using Platform.Random;

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

namespace Platform.Collections
{
    public static class BitStringExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void SetRandomBits(this BitString @string)
        {
            for (var i = 0; i < @string.Length; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                @string.Set(i, value);
            }
        }
    }
}
```

1.10 ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

```
using System.Collections.Concurrent;
using System.Collections.Generic;
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;
            }
        }
    }
}
```

1.11 ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```
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;
    }
}
```

1.12 ./csharp/Platform.Collections/EnsureExtensions.cs

```
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
    {
```

```
15          #region Always
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪  ICollection<T> argument, string argumentName, string message)
19          {
20              if (argument.IsNullOrEmpty())
21              {
22                  throw new ArgumentException(message, argumentName);
23              }
24          }
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪  ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
            ↪  argumentName, null);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪  ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪  string argument, string argumentName, string message)
34          {
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  }
```

1.13 ./csharp/Platform.Collections/ICollectionExtensions.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections
{
    public static class ICollectionExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection == null || collection.Count == 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AllEqualToDefault<T>(this ICollection<T> collection)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            return collection.All(item => equalityComparer.Equals(item, default));
        }
    }
}
```

1.14 ./csharp/Platform.Collections/IDictionaryExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#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;
        }
    }
}
```

1.15 ./csharp/Platform.Collections/Lists/CharIListExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#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/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static 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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪  left.EqualTo(right, ContentEqualTo);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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])
                {
                    return false;
                }
            }
            return true;
        }
    }
}
```

1.16 ./csharp/Platform.Collections/Lists/IListComparer.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Lists
{
    public class IListComparer<T> : IComparer<IList<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
    }
}
```

1.17 ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Lists
{
    public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(IList<T> list) => list.GenerateHashCode();
    }
}
```

1.18 ./csharp/Platform.Collections/Lists/IListExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Lists
{
    public static class IListExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
        ↪  list.Count > index ? list[index] : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
        {
            if (list != null && list.Count > index)
            {
                element = list[index];
                return true;
```

```
            }
            else
            {
                element = default;
                return false;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
        {
            list.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
        {
            list.AddFirst(elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
        ↪  list.Add(elements[0]);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
        {
            list.AddAll(elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAll<T>(this IList<T> list, IList<T> elements)
        {
            for (var i = 0; i < elements.Count; i++)
            {
                list.Add(elements[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
        {
            list.AddSkipFirst(elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
        ↪  list.AddSkipFirst(elements, 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
        {
            for (var i = skip; i < elements.Count; i++)
            {
                list.Add(elements[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
        ↪  right, ContentEqualTo);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
        ↪  IList<T>, bool> contentEqualityComparer)
        {
            if (ReferenceEquals(left, right))
            {
                return true;
            }
            var leftCount = left.GetCountOrZero();
            var rightCount = right.GetCountOrZero();
```

```
            if (leftCount == 0 && rightCount == 0)
            {
                return true;
            }
            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
            {
                return false;
            }
            return contentEqualityComparer(left, right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            for (var i = left.Count - 1; i >= 0; --i)
            {
                if (!equalityComparer.Equals(left[i], right[i]))
                {
                    return false;
                }
            }
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GenerateHashCode<T>(this IList<T> list)
        {
            var hashAccumulator = 17;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
            }
            return hashAccumulator;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] SkipFirst<T>(this IList<T> list, int skip)
        {
            if (list.IsNullOrEmpty() || list.Count <= skip)
            {
                return Array.Empty<T>();
            }
            var result = new T[list.Count - skip];
            for (int r = skip, w = 0; r < list.Count; r++, w++)
            {
                result[w] = list[r];
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
        {
            if (shift < 0)
            {
                throw new NotImplementedException();
            }
            if (shift == 0)
            {
                return list.ToArray();
            }
            else
            {
                var result = new T[list.Count + shift];
                for (int r = 0, w = shift; r < list.Count; r++, w++)
                {
                    result[w] = list[r];
                }
                return result;
            }
        }
    }
}
```

1.19 ./csharp/Platform.Collections/Lists/ListFiller.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Lists
{
    public class ListFiller<TElement, TReturnConstant>
    {
        protected readonly List<TElement> _list;
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list, TReturnConstant returnConstant)
        {
            _list = list;
            _returnConstant = returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list) : this(list, default) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
24          public void Add(TElement element) => _list.Add(element);
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
            ↪  _list.AddFirstAndReturnTrue(elements);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public bool AddAllAndReturnTrue(IList<TElement> elements) =>
            ↪  _list.AddAllAndReturnTrue(elements);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
            ↪  _list.AddSkipFirstAndReturnTrue(elements);
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TReturnConstant AddAndReturnConstant(TElement element)
40          {
41              _list.Add(element);
42              return _returnConstant;
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
47          {
48              _list.AddFirst(elements);
49              return _returnConstant;
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
54          {
55              _list.AddAll(elements);
56              return _returnConstant;
57          }
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
61          {
62              _list.AddSkipFirst(elements);
63              return _returnConstant;
64          }
65      }
66  }
```

1.20 ./csharp/Platform.Collections/Segments/CharSegment.cs

```
1   using System.Linq;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Arrays;
5   using Platform.Collections.Lists;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Collections.Segments
10  {
11      public class CharSegment : Segment<char>
12      {
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
            ↪  length) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public override int GetHashCode()
18          {
19              // Base can be not an array, but still IList<char>
20              if (Base is char[] baseArray)
21              {
22                  return baseArray.GenerateHashCode(Offset, Length);
23              }
24              else
25              {
26                  return this.GenerateHashCode();
27              }
28          }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public override bool Equals(Segment<char> other)
        {
            bool contentEqualityComparer(IList<char> left, IList<char> right)
            {
                // Base can be not an array, but still IList<char>
                if (Base is char[] baseArray && other.Base is char[] otherArray)
                {
                    return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
                }
                else
                {
                    return left.ContentEqualTo(right);
                }
            }
            return this.EqualTo(other, contentEqualityComparer);
        }

        public override bool Equals(object obj) => obj is Segment<char> charSegment ?
        ↪  Equals(charSegment) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator string(CharSegment segment)
        {
            if (!(segment.Base is char[] array))
            {
                array = segment.Base.ToArray();
            }
            return new string(array, segment.Offset, segment.Length);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => this;
    }
}
```

1.21 ./csharp/Platform.Collections/Segments/Segment.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;

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

namespace Platform.Collections.Segments
{
    public class Segment<T> : IEquatable<Segment<T>>, IList<T>
    {
        public IList<T> Base
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Offset
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Segment(IList<T> @base, int offset, int length)
        {
            Base = @base;
            Offset = offset;
            Length = length;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => this.GenerateHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(Segment<T> other) => this.EqualTo(other);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
45          public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
            ↪  false;
46
47          #region IList
48
49          public T this[int i]
50          {
51              [MethodImpl(MethodImplOptions.AggressiveInlining)]
52              get => Base[Offset + i];
53              [MethodImpl(MethodImplOptions.AggressiveInlining)]
54              set => Base[Offset + i] = value;
55          }
56
57          public int Count
58          {
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              get => Length;
61          }
62
63          public bool IsReadOnly
64          {
65              [MethodImpl(MethodImplOptions.AggressiveInlining)]
66              get => true;
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public int IndexOf(T item)
71          {
72              var index = Base.IndexOf(item);
73              if (index >= Offset)
74              {
75                  var actualIndex = index - Offset;
76                  if (actualIndex < Length)
77                  {
78                      return actualIndex;
79                  }
80              }
81              return -1;
82          }
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public void Insert(int index, T item) => throw new NotSupportedException();
86
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public void RemoveAt(int index) => throw new NotSupportedException();
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public void Add(T item) => throw new NotSupportedException();
92
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          public void Clear() => throw new NotSupportedException();
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public bool Contains(T item) => IndexOf(item) >= 0;
98
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public void CopyTo(T[] array, int arrayIndex)
101         {
102             for (var i = 0; i < Length; i++)
103             {
104                 array.Add(ref arrayIndex, this[i]);
105             }
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public bool Remove(T item) => throw new NotSupportedException();
110
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public IEnumerator<T> GetEnumerator()
113         {
114             for (var i = 0; i < Length; i++)
115             {
116                 yield return this[i];
117             }
118         }
119
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
122
123         #endregion
```

```
    }
}
```

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

```
#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;
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
        where TSegment : Segment<T>
    {
        private readonly int _minimumStringSegmentLength;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
        ↪  _minimumStringSegmentLength = minimumStringSegmentLength;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void WalkAll(IList<T> elements)
        {
            for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
            ↪  offset <= maxOffset; offset++)
            {
                for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                ↪  offset; length <= maxLength; length++)
                {
                    Iteration(CreateSegment(elements, offset, length));
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void Iteration(TSegment segment);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#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>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
        ↪  => new Segment<T>(elements, offset, length);
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
```

```
    public static class AllSegmentsWalkerExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
        ↪  walker.WalkAll(@string.ToCharArray());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
        ↪  string @string) where TSegment : Segment<char> =>
        ↪  walker.WalkAll(@string.ToCharArray());
    }
}
```

1.26 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
    ↪  DuplicateSegmentsWalkerBase<T, TSegment>
        where TSegment : Segment<T>
    {
        public static readonly bool DefaultResetDictionaryOnEachWalk;

        private readonly bool _resetDictionaryOnEachWalk;
        protected IDictionary<TSegment, long> Dictionary;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
            : base(minimumStringSegmentLength)
        {
            Dictionary = dictionary;
            _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪  dictionary, int minimumStringSegmentLength) : this(dictionary,
        ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪  dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
        ↪  DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪  bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
        ↪  Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪  this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
        ↪  this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void WalkAll(IList<T> elements)
        {
            if (_resetDictionaryOnEachWalk)
            {
                var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
                Dictionary = new Dictionary<TSegment, long>((int)capacity);
            }
            base.WalkAll(elements);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long GetSegmentFrequency(TSegment segment) =>
        ↪  Dictionary.GetOrDefault(segment);

```

```
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
            ↪ Dictionary[segment] = frequency;
56      }
57  }
```

1.27 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].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.Segments.Walkers
7   {
8       public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
        ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
            ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
            ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
            ↪ DefaultResetDictionaryOnEachWalk) { }
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
            ↪ resetDictionaryOnEachWalk) { }
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27      }
28  }
```

1.28 ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].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.Segments.Walkers
6   {
7       public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
        ↪ TSegment>
8           where TSegment : Segment<T>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪ base(minimumStringSegmentLength) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override void Iteration(TSegment segment)
18          {
19              var frequency = GetSegmentFrequency(segment);
20              if (frequency == 1)
21              {
22                  OnDublicateFound(segment);
23              }
24              SetSegmentFrequency(segment, frequency + 1);
25          }
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected abstract void OnDublicateFound(TSegment segment);
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract long GetSegmentFrequency(TSegment segment);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
    }
}
```

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

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

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

1.30 ./csharp/Platform.Collections/Sets/ISetExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Sets
{
    public static class ISetExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) => set.Remove(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
        {
            set.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            AddFirst(set, elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddFirst<T>(this ISet<T> set, IList<T> elements) => set.Add(elements[0]);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            set.AddAll(elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAll<T>(this ISet<T> set, IList<T> elements)
        {
            for (var i = 0; i < elements.Count; i++)
            {
                set.Add(elements[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            set.AddSkipFirst(elements);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
        ↪  set.AddSkipFirst(elements, 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
        {
            for (var i = skip; i < elements.Count; i++)
            {
                set.Add(elements[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
        ↪  !set.Contains(element);
    }
}
```

1.31 ./csharp/Platform.Collections/Sets/SetFiller.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Sets
{
    public class SetFiller<TElement, TReturnConstant>
    {
        protected readonly ISet<TElement> _set;
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
        {
            _set = set;
            _returnConstant = returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set) : this(set, default) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _set.Add(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddFirstAndReturnTrue(elements);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAllAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddAllAndReturnTrue(elements);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddSkipFirstAndReturnTrue(elements);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
        {
            _set.Add(element);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
        {
            _set.AddFirst(elements);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
        {
            _set.AddAll(elements);
            return _returnConstant;
        }
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
        {
            _set.AddSkipFirst(elements);
            return _returnConstant;
        }
    }
}
```

1.32 ./csharp/Platform.Collections/Stacks/DefaultStack.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
    {
        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Count <= 0;
        }
    }
}
```

1.33 ./csharp/Platform.Collections/Stacks/IStack.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public interface IStack<TElement>
    {
        bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Push(TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Pop();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Peek();
    }
}
```

1.34 ./csharp/Platform.Collections/Stacks/IStackExtensions.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public static class IStackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Clear<T>(this IStack<T> stack)
        {
            while (!stack.IsEmpty)
            {
                _ = stack.Pop();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Pop();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Peek();
```

```
    }
}
```

1.35 ./csharp/Platform.Collections/Stacks/IStackFactory.cs

```
using Platform.Interfaces;

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

namespace Platform.Collections.Stacks
{
    public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
    {
    }
}
```

1.36 ./csharp/Platform.Collections/Stacks/StackExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public static class StackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
        ↪  default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
        ↪  : default;
    }
}
```

1.37 ./csharp/Platform.Collections/StringExtensions.cs

```
using System;
using System.Globalization;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections
{
    public static class StringExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string CapitalizeFirstLetter(this string @string)
        {
            if (string.IsNullOrWhiteSpace(@string))
            {
                return @string;
            }
            var chars = @string.ToCharArray();
            for (var i = 0; i < chars.Length; i++)
            {
                var category = char.GetUnicodeCategory(chars[i]);
                if (category == UnicodeCategory.UppercaseLetter)
                {
                    return @string;
                }
                if (category == UnicodeCategory.LowercaseLetter)
                {
                    chars[i] = char.ToUpper(chars[i]);
                    return new string(chars);
                }
            }
            return @string;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Truncate(this string @string, int maxLength) =>
        ↪  string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
        ↪  Math.Min(@string.Length, maxLength));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string TrimSingle(this string @string, char charToTrim)
        {
            if (!string.IsNullOrEmpty(@string))
```

```
            {
                if (@string.Length == 1)
                {
                    if (@string[0] == charToTrim)
                    {
                        return "";
                    }
                    else
                    {
                        return @string;
                    }
                }
                else
                {
                    var left = 0;
                    var right = @string.Length - 1;
                    if (@string[left] == charToTrim)
                    {
                        left++;
                    }
                    if (@string[right] == charToTrim)
                    {
                        right--;
                    }
                    return @string.Substring(left, right - left + 1);
                }
            }
            else
            {
                return @string;
            }
        }
    }
}
```

1.38 ./csharp/Platform.Collections/Trees/Node.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Trees
{
    public class Node
    {
        private Dictionary<object, Node> _childNodes;

        public object Value
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public Dictionary<object, Node> ChildNodes
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
        }

        public Node this[object key]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => GetChild(key) ?? AddChild(key);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => SetChildValue(value, key);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node(object value) => Value = value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node() : this(null) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public Node GetChild(params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node.ChildNodes.TryGetValue(keys[i], out node);
                if (node == null)
                {
                    return null;
                }
            }
            return node;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key) => AddChild(key, new Node(null));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, object value) => AddChild(key, new Node(value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, Node child)
        {
            ChildNodes.Add(key, child);
            return child;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(params object[] keys) => SetChildValue(null, keys);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(object key) => SetChildValue(null, key);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChildValue(object value, params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node = SetChildValue(value, keys[i]);
            }
            node.Value = value;
            return node;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChildValue(object value, object key)
        {
            if (!ChildNodes.TryGetValue(key, out Node child))
            {
                child = AddChild(key, value);
            }
            child.Value = value;
            return child;
        }
    }
}
```

1.39 ./csharp/Platform.Collections.Tests/ArrayTests.cs

```
using Xunit;
using Platform.Collections.Arrays;

namespace Platform.Collections.Tests
{
    public class ArrayTests
    {
        [Fact]
        public void GetElementTest()
        {
            var nullArray = (int[])null;
            Assert.Equal(0, nullArray.GetElementOrDefault(1));
            Assert.False(nullArray.TryGetElement(1, out int element));
            Assert.Equal(0, element);
            var array = new int[] { 1, 2, 3 };
            Assert.Equal(3, array.GetElementOrDefault(2));
            Assert.True(array.TryGetElement(2, out element));
            Assert.Equal(3, element);
```

```
            Assert.Equal(0, array.GetElementOrDefault(10));
            Assert.False(array.TryGetElement(10, out element));
            Assert.Equal(0, element);
        }
    }
}
```

1.40 ./csharp/Platform.Collections.Tests/BitStringTests.cs

```
using System;
using System.Collections;
using Xunit;
using Platform.Random;

namespace Platform.Collections.Tests
{
    public static class BitStringTests
    {
        [Fact]
        public static void BitGetSetTest()
        {
            const int n = 250;
            var bitArray = new BitArray(n);
            var bitString = new BitString(n);
            for (var i = 0; i < n; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                bitArray.Set(i, value);
                bitString.Set(i, value);
                Assert.Equal(value, bitArray.Get(i));
                Assert.Equal(value, bitString.Get(i));
            }
        }

        [Fact]
        public static void BitVectorNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitParallelNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitParallelVectorNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitVectorAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorAnd(y);
                w.And(v);
            });
        }

        [Fact]
        public static void BitParallelAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelAnd(y);
```

```
            w.And(v);
        });
    }

    [Fact]
    public static void BitParallelVectorAndTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelVectorAnd(y);
            w.And(v);
        });
    }

    [Fact]
    public static void BitVectorOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.VectorOr(y);
            w.Or(v);
        });
    }

    [Fact]
    public static void BitParallelOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelOr(y);
            w.Or(v);
        });
    }

    [Fact]
    public static void BitParallelVectorOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelVectorOr(y);
            w.Or(v);
        });
    }

    [Fact]
    public static void BitVectorXorTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.VectorXor(y);
            w.Xor(v);
        });
    }

    [Fact]
    public static void BitParallelXorTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelXor(y);
            w.Xor(v);
        });
    }

    [Fact]
    public static void BitParallelVectorXorTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelVectorXor(y);
            w.Xor(v);
        });
    }

    private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
    ↪  BitString, BitString> test)
    {
        const int n = 5654;
```

```
            var x = new BitString(n);
            var y = new BitString(n);
            while (x.Equals(y))
            {
                x.SetRandomBits();
                y.SetRandomBits();
            }
            var w = new BitString(x);
            var v = new BitString(y);
            Assert.False(x.Equals(y));
            Assert.False(w.Equals(v));
            Assert.True(x.Equals(w));
            Assert.True(y.Equals(v));
            test(x, y, w, v);
            Assert.True(x.Equals(w));
        }
    }
}
```

1.41 ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs

```
using Xunit;
using Platform.Collections.Segments;

namespace Platform.Collections.Tests
{
    public static class CharsSegmentTests
    {
        [Fact]
        public static void GetHashCodeEqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
            var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
            Assert.Equal(firstHashCode, secondHashCode);
        }

        [Fact]
        public static void EqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
            var second = new CharSegment(testArray, 5, 4);
            Assert.True(first.Equals(second));
        }
    }
}
```

1.42 ./csharp/Platform.Collections.Tests/ListTests.cs

```
using System.Collections.Generic;
using Xunit;
using Platform.Collections.Lists;


namespace Platform.Collections.Tests
{
    public class ListTests
    {
        [Fact]
        public void GetElementTest()
        {
            var nullList = (IList<int>)null;
            Assert.Equal(0, nullList.GetElementOrDefault(1));
            Assert.False(nullList.TryGetElement(1, out int element));
            Assert.Equal(0, element);
            var list = new List<int>() { 1, 2, 3 };
            Assert.Equal(3, list.GetElementOrDefault(2));
            Assert.True(list.TryGetElement(2, out element));
            Assert.Equal(3, element);
            Assert.Equal(0, list.GetElementOrDefault(10));
            Assert.False(list.TryGetElement(10, out element));
            Assert.Equal(0, element);
        }
    }
}
```

1.43 ./csharp/Platform.Collections.Tests/StringTests.cs

```
using Xunit;

namespace Platform.Collections.Tests
{
    public static class StringTests
    {
        [Fact]
        public static void CapitalizeFirstLetterTest()
        {
            Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
            Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
            Assert.Equal("  Hello", "  hello".CapitalizeFirstLetter());
        }

        [Fact]
        public static void TrimSingleTest()
        {
            Assert.Equal("", "''".TrimSingle('\''));
            Assert.Equal("", "'''".TrimSingle('\''));
            Assert.Equal("hello", "'hello'".TrimSingle('\''));
            Assert.Equal("hello", "hello'".TrimSingle('\''));
            Assert.Equal("hello", "'hello".TrimSingle('\''));
        }
    }
}
```

Index