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)
```

```
        {
            fixed (char* leftPointer = &left[leftOffset])
            {
                fixed (char* rightPointer = &right[rightOffset])
                {
                    char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
                    if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
                    ↪  rightPointerCopy, ref length))
                    {
                        return false;
                    }
                    CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
                    ↪  ref length);
                    return length <= 0;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
        ↪  int length)
        {
            while (length >= 10)
            {
                if ((*(int*)left != *(int*)right)
                 || (*(int*)(left + 2) != *(int*)(right + 2))
                 || (*(int*)(left + 4) != *(int*)(right + 4))
                 || (*(int*)(left + 6) != *(int*)(right + 6))
                 || (*(int*)(left + 8) != *(int*)(right + 8)))
                {
                    return false;
                }
                left += 10;
                right += 10;
                length -= 10;
            }
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
        ↪  int length)
        {
            // This depends on the fact that the String objects are
            // always zero terminated and that the terminating zero is not included
            // in the length. For odd string sizes, the last compare will include
            // the zero terminator.
            while (length > 0)
            {
                if (*(int*)left != *(int*)right)
                {
                    break;
                }
                left += 2;
                right += 2;
                length -= 2;
            }
        }
    }
}
```

1.7 ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.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.Arrays
{
    public static class GenericArrayExtensions
    {
        /// <summary>
        /// <para>We check whether the array exists, if so, we check the array length using the
        ↪  index variable type int, and if the array length is greater than the index, we
        ↪  return array[index], otherwise-default value.</para>
        /// <para>Мы проверяем, существует ли массив, если да - мы проверяем длину массива с
        ↪  помощью переменной index, и если длина массива больше индекса - возвращаем
        ↪  array[index], иначе - default value.</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
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
            ↪  array.Length > index ? array[index] : default;
22
23          /// <summary>
24          /// <para>We check whether the array exists, if so, we check the array length using the
            ↪  index variable type long, and if the array length is greater than the index, we
            ↪  return array[index], otherwise-default value.</para>
25          /// <para>Мы проверяем, существует ли массив, если да - мы проверяем длину массива с
            ↪  помощью переменной index, и если длина массива больше индекса - возвращаем
            ↪  array[index], иначе - значение по умолчанию.</para>
26          /// </summary>
27          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
28          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
29          /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
            ↪  для сравнения.</para></param>
30          /// <returns><para>Array element or default value.</para><para>Элемент массива или же
            ↪  значение по умолчанию.</para></returns>
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
            ↪  array.LongLength > index ? array[index] : default;
34
35          /// <summary>
36          /// <para>We check whether the array exist, if so, we check the array length using the
            ↪  index varible type int, and if the array length is greater than the index, we set
            ↪  the element variable to array[index] and return true.</para>
37          /// <para>Мы проверяем, существует ли массив, если да, то мы проверяем длину массива с
            ↪  помощью переменной index  типа int, и если длина массива больше значения index, мы
            ↪  устанавливаем значение переменной element - array[index] и возвращаем true.</para>
38          /// </summary>
39          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
40          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
41          /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
            ↪  сравнения.</para></param>
42          /// <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>
43          /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
            ↪  в противном случае false</para></returns>
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool TryGetElement<T>(this T[] array, int index, out T element)
47          {
48              if (array != null && array.Length > index)
49              {
50                  element = array[index];
51                  return true;
52              }
53              else
54              {
55                  element = default;
56                  return false;
57              }
58          }
59
60          /// <summary>
```

```
61          /// <para>We check whether the array exist, if so, we check the array length using the
            ↪  index varible type long, and if the array length is greater than the index, we set
            ↪  the element variable to array[index] and return true.</para>
62          /// <para>Мы проверяем, существует ли массив, если да, то мы проверяем длину массива с
            ↪  помощью переменной index  типа long, и если длина массива больше значения index, мы
            ↪  устанавливаем значение переменной element - array[index] и возвращаем true.</para>
63          /// </summary>
64          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
65          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать в
            ↪  проверке.</para></param>
66          /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
            ↪  для сравнения.</para></param>
67          /// <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>
68          /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
            ↪  в противном случае false</para></returns>
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public static bool TryGetElement<T>(this T[] array, long index, out T element)
72          {
73              if (array != null && array.LongLength > index)
74              {
75                  element = array[index];
76                  return true;
77              }
78              else
79              {
80                  element = default;
81                  return false;
82              }
83          }
84
85          /// <summary>
86          /// <para>Copying a range of elements from one array to another array.</para>
87          /// <para>Копируем диапазон элементов из одного массива в другой массив.</para>
88          /// </summary>
89          /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
            ↪  массива.</para></typeparam>
90          /// <param name="array"><para>The array you want to copy.</para><para>Массив который
            ↪  необходимо скопировать.</para></param>
91          /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
92
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          public static T[] Clone<T>(this T[] array)
95          {
96              var copy = new T[array.LongLength];
97              Array.Copy(array, 0L, copy, 0L, array.LongLength);
98              return copy;
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public static IList<T> ShiftRight<T>(this T[] array, long shift)
106         {
107             if (shift < 0)
108             {
109                 throw new NotImplementedException();
110             }
111             if (shift == 0)
112             {
113                 return array.Clone<T>();
114             }
115             else
116             {
117                 var restrictions = new T[array.LongLength + shift];
118                 Array.Copy(array, 0L, restrictions, shift, array.LongLength);
119                 return restrictions;
120             }
121         }
122
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
124         public static void Add<T>(this T[] array, ref int position, T element) =>
            ↪ array[position++] = element;
125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public static void Add<T>(this T[] array, ref long position, T element) =>
            ↪ array[position++] = element;
128
129         [MethodImpl(MethodImplOptions.AggressiveInlining)]
130         public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
            ↪ TElement[] array, ref long position, TElement element, TReturnConstant
            ↪ returnConstant)
131         {
132             array.Add(ref position, element);
133             return returnConstant;
134         }
135
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
            ↪ array[position++] = elements[0];
138
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
            ↪ TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪ returnConstant)
141         {
142             array.AddFirst(ref position, elements);
143             return returnConstant;
144         }
145
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
            ↪ TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪ returnConstant)
148         {
149             array.AddAll(ref position, elements);
150             return returnConstant;
151         }
152
153         [MethodImpl(MethodImplOptions.AggressiveInlining)]
154         public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
155         {
156             for (var i = 0; i < elements.Count; i++)
157             {
158                 array.Add(ref position, elements[i]);
159             }
160         }
161
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
            ↪ TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
            ↪ TReturnConstant returnConstant)
164         {
165             array.AddSkipFirst(ref position, elements);
166             return returnConstant;
167         }
168
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
            ↪ => array.AddSkipFirst(ref position, elements, 1);
171
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
            ↪ int skip)
174         {
175             for (var i = skip; i < elements.Count; i++)
176             {
177                 array.Add(ref position, elements[i]);
178             }
179         }
180     }
181 }
```

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

```
1  using System;
2  using System.Collections.Concurrent;
3  using System.Collections.Generic;
4  using System.Numerics;
5  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];
            }
```

```
235         }
236
237         [MethodImpl(MethodImplOptions.AggressiveInlining)]
238         public BitString And(BitString other)
239         {
240             EnsureBitStringHasTheSameSize(other, nameof(other));
241             GetCommonOuterBorders(this, other, out long from, out long to);
242             var otherArray = other._array;
243             for (var i = from; i <= to; i++)
244             {
245                 _array[i] &= otherArray[i];
246                 RefreshBordersByWord(i);
247             }
248             return this;
249         }
250
251         [MethodImpl(MethodImplOptions.AggressiveInlining)]
252         public BitString ParallelAnd(BitString other)
253         {
254             var threads = Environment.ProcessorCount / 2;
255             if (threads <= 1)
256             {
257                 return And(other);
258             }
259             EnsureBitStringHasTheSameSize(other, nameof(other));
260             GetCommonOuterBorders(this, other, out long from, out long to);
261             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
262             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                ↪  MaxDegreeOfParallelism = threads }, range =>
263             {
264                 var maximum = range.Item2;
265                 for (var i = range.Item1; i < maximum; i++)
266                 {
267                     _array[i] &= other._array[i];
268                 }
269             });
270             MarkBordersAsAllBitsSet();
271             TryShrinkBorders();
272             return this;
273         }
274
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         public BitString VectorAnd(BitString other)
277         {
278             if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
279             {
280                 return And(other);
281             }
282             var step = Vector<long>.Count;
283             if (_array.Length < step)
284             {
285                 return And(other);
286             }
287             EnsureBitStringHasTheSameSize(other, nameof(other));
288             GetCommonOuterBorders(this, other, out int from, out int to);
289             VectorAndLoop(_array, other._array, step, from, to + 1);
290             MarkBordersAsAllBitsSet();
291             TryShrinkBorders();
292             return this;
293         }
294
295         [MethodImpl(MethodImplOptions.AggressiveInlining)]
296         public BitString ParallelVectorAnd(BitString other)
297         {
298             var threads = Environment.ProcessorCount / 2;
299             if (threads <= 1)
300             {
301                 return VectorAnd(other);
302             }
303             if (!Vector.IsHardwareAccelerated)
304             {
305                 return ParallelAnd(other);
306             }
307             var step = Vector<long>.Count;
308             if (_array.Length < (step * threads))
309             {
310                 return VectorAnd(other);
311             }
```

```
312             EnsureBitStringHasTheSameSize(other, nameof(other));
313             GetCommonOuterBorders(this, other, out int from, out int to);
314             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
315             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                ↪  MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
                ↪  step, range.Item1, range.Item2));
316             MarkBordersAsAllBitsSet();
317             TryShrinkBorders();
318             return this;
319         }
320 
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
            ↪  int maximum)
323         {
324             var i = start;
325             var range = maximum - start - 1;
326             var stop = range - (range % step);
327             for (; i < stop; i += step)
328             {
329                 (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
330             }
331             for (; i < maximum; i++)
332             {
333                 array[i] &= otherArray[i];
334             }
335         }
336 
337         [MethodImpl(MethodImplOptions.AggressiveInlining)]
338         public BitString Or(BitString other)
339         {
340             EnsureBitStringHasTheSameSize(other, nameof(other));
341             GetCommonOuterBorders(this, other, out long from, out long to);
342             for (var i = from; i <= to; i++)
343             {
344                 _array[i] |= other._array[i];
345                 RefreshBordersByWord(i);
346             }
347             return this;
348         }
349 
350         [MethodImpl(MethodImplOptions.AggressiveInlining)]
351         public BitString ParallelOr(BitString other)
352         {
353             var threads = Environment.ProcessorCount / 2;
354             if (threads <= 1)
355             {
356                 return Or(other);
357             }
358             EnsureBitStringHasTheSameSize(other, nameof(other));
359             GetCommonOuterBorders(this, other, out long from, out long to);
360             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
361             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                ↪  MaxDegreeOfParallelism = threads }, range =>
362             {
363                 var maximum = range.Item2;
364                 for (var i = range.Item1; i < maximum; i++)
365                 {
366                     _array[i] |= other._array[i];
367                 }
368             });
369             MarkBordersAsAllBitsSet();
370             TryShrinkBorders();
371             return this;
372         }
373 
374         [MethodImpl(MethodImplOptions.AggressiveInlining)]
375         public BitString VectorOr(BitString other)
376         {
377             if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
378             {
379                 return Or(other);
380             }
381             var step = Vector<long>.Count;
382             if (_array.Length < step)
383             {
384                 return Or(other);
385             }
```

```
            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];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Xor(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 ParallelXor(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Xor(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 VectorXor(BitString other)
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return Xor(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return Xor(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorXorLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

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

```
534
535         [MethodImpl(MethodImplOptions.AggressiveInlining)]
536         private void RefreshBordersByWord(long wordIndex)
537         {
538             if (_array[wordIndex] == 0)
539             {
540                 if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
541                 {
542                     _minPositiveWord++;
543                 }
544                 if (wordIndex == _maxPositiveWord && wordIndex != 0)
545                 {
546                     _maxPositiveWord--;
547                 }
548             }
549             else
550             {
551                 if (wordIndex < _minPositiveWord)
552                 {
553                     _minPositiveWord = wordIndex;
554                 }
555                 if (wordIndex > _maxPositiveWord)
556                 {
557                     _maxPositiveWord = wordIndex;
558                 }
559             }
560         }
561
562         [MethodImpl(MethodImplOptions.AggressiveInlining)]
563         public bool TryShrinkBorders()
564         {
565             GetBorders(out long from, out long to);
566             while (from <= to && _array[from] == 0)
567             {
568                 from++;
569             }
570             if (from > to)
571             {
572                 MarkBordersAsAllBitsReset();
573                 return true;
574             }
575             while (to >= from && _array[to] == 0)
576             {
577                 to--;
578             }
579             if (to < from)
580             {
581                 MarkBordersAsAllBitsReset();
582                 return true;
583             }
584             var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
585             if (bordersUpdated)
586             {
587                 SetBorders(from, to);
588             }
589             return bordersUpdated;
590         }
591
592         [MethodImpl(MethodImplOptions.AggressiveInlining)]
593         public bool Get(long index)
594         {
595             Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
596             return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
597         }
598
599         [MethodImpl(MethodImplOptions.AggressiveInlining)]
600         public void Set(long index, bool value)
601         {
602             if (value)
603             {
604                 Set(index);
605             }
606             else
607             {
608                 Reset(index);
609             }
610         }
611
612         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
613         public void Set(long index)
614         {
615             Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
616             var wordIndex = GetWordIndexFromIndex(index);
617             var mask = GetBitMaskFromIndex(index);
618             _array[wordIndex] |= mask;
619             RefreshBordersByWord(wordIndex);
620         }
621 
622         [MethodImpl(MethodImplOptions.AggressiveInlining)]
623         public void Reset(long index)
624         {
625             Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
626             var wordIndex = GetWordIndexFromIndex(index);
627             var mask = GetBitMaskFromIndex(index);
628             _array[wordIndex] &= ~mask;
629             RefreshBordersByWord(wordIndex);
630         }
631 
632         [MethodImpl(MethodImplOptions.AggressiveInlining)]
633         public bool Add(long index)
634         {
635             var wordIndex = GetWordIndexFromIndex(index);
636             var mask = GetBitMaskFromIndex(index);
637             if ((_array[wordIndex] & mask) == 0)
638             {
639                 _array[wordIndex] |= mask;
640                 RefreshBordersByWord(wordIndex);
641                 return true;
642             }
643             else
644             {
645                 return false;
646             }
647         }
648 
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public void SetAll(bool value)
651         {
652             if (value)
653             {
654                 SetAll();
655             }
656             else
657             {
658                 ResetAll();
659             }
660         }
661 
662         [MethodImpl(MethodImplOptions.AggressiveInlining)]
663         public void SetAll()
664         {
665             const long fillValue = unchecked((long)0xffffffffffffffff);
666             var words = GetWordsCountFromIndex(_length);
667             for (var i = 0; i < words; i++)
668             {
669                 _array[i] = fillValue;
670             }
671             MarkBordersAsAllBitsSet();
672         }
673 
674         [MethodImpl(MethodImplOptions.AggressiveInlining)]
675         public void ResetAll()
676         {
677             const long fillValue = 0;
678             GetBorders(out long from, out long to);
679             for (var i = from; i <= to; i++)
680             {
681                 _array[i] = fillValue;
682             }
683             MarkBordersAsAllBitsReset();
684         }
685 
686         [MethodImpl(MethodImplOptions.AggressiveInlining)]
687         public List<long> GetSetIndices()
688         {
689             var result = new List<long>();
690             GetBorders(out long from, out long to);
691             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;
            }
        }
        return true;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
    {
        Ensure.Always.ArgumentNotNull(other, argumentName);
        if (_length != other._length)
        {
            throw new ArgumentException("Bit string must be the same size.", argumentName);
        }
    }

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

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

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

```
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             {
```

```
                result.Add(bits16to31[j] + 16 + (i * 64));
            }
            for (var j = 0; j < bits32to47.Length; j++)
            {
                result.Add(bits32to47[j] + 32 + (i * 64));
            }
            for (var j = 0; j < bits48to63.Length; j++)
            {
                result.Add(bits48to63[j] + 48 + (i * 64));
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
        {
            if (bits48to63.Length > 0)
            {
                return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
            }
            if (bits16to31.Length > 0)
            {
                return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
            }
            return bits00to15[bits00to15.Length - 1] + (i * 64);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47, out byte[] bits48to63)
        {
            bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
            bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
```

```
            bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
            bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
        ↪  out long to)
        {
            from = Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
        ↪  out long to)
        {
            from = Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
        ↪  out int to)
        {
            from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
        ↪  ulong to)
        {
            from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordIndexFromIndex(long index) => index >> 6;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);

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

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

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
    {
        #region Always

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

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

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

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

```
            if (string.IsNullOrWhiteSpace(argument))
            {
                throw new ArgumentException(message, argumentName);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
        ↪  string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);

        #endregion

        #region OnDebug

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName, string message) =>
        ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName) =>
        ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument, string argumentName, string message) =>
        ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument, string argumentName) =>
        ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
        ↪  null, null);

        #endregion
    }
}
```

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/3b1eaf5203992df69de44c783
        ↪  a3eda37d3d4cd10/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)]
        public void Add(TElement element) => _list.Add(element);

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
        {
            _list.AddFirst(elements);
```

```
            return _returnConstant;
        }

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

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

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

```
using System.Linq;
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 CharSegment : Segment<char>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
        ↪  length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode()
        {
            // Base can be not an array, but still IList<char>
            if (Base is char[] baseArray)
            {
                return baseArray.GenerateHashCode(Offset, Length);
            }
            else
            {
                return this.GenerateHashCode();
            }
        }

        [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)]
        public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
        ↪  false;

        #region IList

        public T this[int i]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Base[Offset + i];
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => Base[Offset + i] = value;
        }

        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Length;
        }

        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(T item)
        {
            var index = Base.IndexOf(item);
```

```
            if (index >= Offset)
            {
                var actualIndex = index - Offset;
                if (actualIndex < Length)
                {
                    return actualIndex;
                }
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, T item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(T item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(T item) => IndexOf(item) >= 0;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(T item) => throw new NotSupportedException();

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        #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);

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

1.27 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[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 DictionaryBasedDuplicateSegmentsWalkerBase<T> : DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long> dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) : base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long> dictionary, int minimumStringSegmentLength) : base(dictionary, minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
        ↪ DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
        ↪ resetDictionaryOnEachWalk) { }

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

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

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

```
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 DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
    ↪ TSegment>
        where TSegment : Segment<T>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪ base(minimumStringSegmentLength) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Iteration(TSegment segment)
        {
            var frequency = GetSegmentFrequency(segment);
            if (frequency == 1)
            {
                OnDublicateFound(segment);
            }
            SetSegmentFrequency(segment, frequency + 1);
        }

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

```
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
            ↪  : default;
15      }
16  }
```

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

```
1   using System;
2   using System.Globalization;
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
8   {
9       public static class StringExtensions
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public static string CapitalizeFirstLetter(this string @string)
13          {
14              if (string.IsNullOrWhiteSpace(@string))
15              {
16                  return @string;
17              }
18              var chars = @string.ToCharArray();
19              for (var i = 0; i < chars.Length; i++)
20              {
21                  var category = char.GetUnicodeCategory(chars[i]);
22                  if (category == UnicodeCategory.UppercaseLetter)
23                  {
24                      return @string;
25                  }
26                  if (category == UnicodeCategory.LowercaseLetter)
27                  {
28                      chars[i] = char.ToUpper(chars[i]);
29                      return new string(chars);
30                  }
31              }
32              return @string;
33          }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public static string Truncate(this string @string, int maxLength) =>
            ↪  string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
            ↪  Math.Min(@string.Length, maxLength));
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public static string TrimSingle(this string @string, char charToTrim)
40          {
41              if (!string.IsNullOrEmpty(@string))
42              {
43                  if (@string.Length == 1)
44                  {
45                      if (@string[0] == charToTrim)
46                      {
47                          return "";
48                      }
49                      else
50                      {
51                          return @string;
52                      }
53                  }
54                  else
55                  {
56                      var left = 0;
57                      var right = @string.Length - 1;
58                      if (@string[left] == charToTrim)
59                      {
60                          left++;
61                      }
62                      if (@string[right] == charToTrim)
63                      {
64                          right--;
65                      }
66                      return @string.Substring(left, right - left + 1);
67                  }
68              }
69              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()
```

```
98          {
99              TestToOperationsWithSameMeaning((x, y, w, v) =>
100             {
101                 x.ParallelOr(y);
102                 w.Or(v);
103             });
104         }
105
106         [Fact]
107         public static void BitParallelVectorOrTest()
108         {
109             TestToOperationsWithSameMeaning((x, y, w, v) =>
110             {
111                 x.ParallelVectorOr(y);
112                 w.Or(v);
113             });
114         }
115
116         [Fact]
117         public static void BitVectorXorTest()
118         {
119             TestToOperationsWithSameMeaning((x, y, w, v) =>
120             {
121                 x.VectorXor(y);
122                 w.Xor(v);
123             });
124         }
125
126         [Fact]
127         public static void BitParallelXorTest()
128         {
129             TestToOperationsWithSameMeaning((x, y, w, v) =>
130             {
131                 x.ParallelXor(y);
132                 w.Xor(v);
133             });
134         }
135
136         [Fact]
137         public static void BitParallelVectorXorTest()
138         {
139             TestToOperationsWithSameMeaning((x, y, w, v) =>
140             {
141                 x.ParallelVectorXor(y);
142                 w.Xor(v);
143             });
144         }
145
146         private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
            ↪ BitString, BitString> test)
147         {
148             const int n = 5654;
149             var x = new BitString(n);
150             var y = new BitString(n);
151             while (x.Equals(y))
152             {
153                 x.SetRandomBits();
154                 y.SetRandomBits();
155             }
156             var w = new BitString(x);
157             var v = new BitString(y);
158             Assert.False(x.Equals(y));
159             Assert.False(w.Equals(v));
160             Assert.True(x.Equals(w));
161             Assert.True(y.Equals(v));
162             test(x, y, w, v);
163             Assert.True(x.Equals(w));
164         }
165     }
166 }
```

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

```
1  using Xunit;
2  using Platform.Collections.Segments;
3
4  namespace Platform.Collections.Tests
5  {
6      public static class CharsSegmentTests
7      {
```

```
        [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