

LinksPlatform's Platform.Collections Class Library

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
9     {
10         protected readonly TReturnConstant _returnConstant;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
14             ↪ base(array, offset) => _returnConstant = returnConstant;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
18             ↪ returnConstant) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public TReturnConstant AddAndReturnConstant(TElement element) =>
22             ↪ _array.AddAndReturnConstant(ref _position, element, _returnConstant);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements) =>
26             ↪ _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements) =>
30             ↪ _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements) =>
34             ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
35     }
36 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement>
9     {
10         protected readonly TElement[] _array;
11         protected long _position;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayFiller(TElement[] array, long offset)
15         {
16             _array = array;
17             _position = offset;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ArrayFiller(TElement[] array) : this(array, 0) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _array[_position++] = element;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
28             ↪ _position, element, true);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
32             ↪ _array.AddFirstAndReturnConstant(ref _position, elements, true);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
36             ↪ _array.AddAllAndReturnConstant(ref _position, elements, true);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
40             ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
41     }
42 }
```

```

36         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
           ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
37     }
38 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static class ArrayPool
8      {
9          public static readonly int DefaultSizesAmount = 512;
10         public static readonly int DefaultMaxArraysPerSize = 32;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
17     }
18 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Stacks;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Arrays
10 {
11     /// <remarks>
12     /// Original idea from
13     ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
14     /// </remarks>
15     public class ArrayPool<T>
16     {
17         // May be use Default class for that later.
18         [ThreadStatic]
19         private static ArrayPool<T> _threadInstance;
20         internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
           ↪ ArrayPool<T>());
21
22         private readonly int _maxArraysPerSize;
23         private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
           ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
36         {
37             var destination = AllocateDisposable(size);
38             T[] sourceArray = source;
39             if (!sourceArray.IsNullOrEmpty())
40             {
41                 T[] destinationArray = destination;
42                 Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                   ↪ sourceArray.LongLength);
43                 source.Dispose();
44             }
45             return destination;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public virtual void Clear() => _pool.Clear();
50     }
51 }

```

```

50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
    ↪     _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public virtual void Free(T[] array)
55     {
56         if (array.IsNullOrEmpty())
57         {
58             return;
59         }
60         var stack = _pool.GetOrAdd(array.LongLength, size => new
    ↪         Stack<T[]>(_maxArraysPerSize));
61         if (stack.Count == _maxArraysPerSize) // Stack is full
62         {
63             return;
64         }
65         stack.Push(array);
66     }
67 }
68 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Segments;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Arrays
7  {
8      public class ArrayString<T> : Segment<T>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ArrayString(int length) : base(new T[length], 0, length) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayString(T[] array) : base(array, 0, array.Length) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public ArrayString(T[] array, int length) : base(array, 0, length) { }
18     }
19 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static unsafe class CharArrayExtensions
8      {
9          /// <remarks>
10         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11         ↪     ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12         /// </remarks>
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static int GenerateHashCode(this char[] array, int offset, int length)
15         {
16             var hashSeed = 5381;
17             var hashAccumulator = hashSeed;
18             fixed (char* arrayPointer = &array[offset])
19             {
20                 for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
21                     ↪     ↪ < last; charPointer++)
22                 {
23                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
24                 }
25             }
26             return hashAccumulator + (hashSeed * 1566083941);
27         }
28
29         /// <remarks>
30         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
31         ↪     ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
32         /// </remarks>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
35             ↪     ↪ right, int rightOffset)

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Arrays
8  {
9      public static class GenericArrayExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static T[] Clone<T>(this T[] array)
13         {
14             var copy = new T[array.LongLength];
15             Array.Copy(array, 0L, copy, 0L, array.LongLength);
16             return copy;
17         }

```

```

18 [MethodImpl(MethodImplOptions.AggressiveInlining)]
19 public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
20
21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 public static IList<T> ShiftRight<T>(this T[] array, long shift)
23 {
24     if (shift < 0)
25     {
26         throw new NotImplementedException();
27     }
28     if (shift == 0)
29     {
30         return array.Clone<T>();
31     }
32     else
33     {
34         var restrictions = new T[array.LongLength + shift];
35         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
36         return restrictions;
37     }
38 }
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 public static void Add<T>(this T[] array, ref int position, T element) =>
42     ↪ array[position++] = element;
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 public static void Add<T>(this T[] array, ref long position, T element) =>
46     ↪ array[position++] = element;
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
50     ↪ TElement[] array, ref long position, TElement element, TReturnConstant
51     ↪ returnConstant)
52 {
53     array.Add(ref position, element);
54     return returnConstant;
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
59     ↪ array[position++] = elements[0];
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
63     ↪ TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
64     ↪ returnConstant)
65 {
66     array.AddFirst(ref position, elements);
67     return returnConstant;
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
72     ↪ TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
73     ↪ returnConstant)
74 {
75     array.AddAll(ref position, elements);
76     return returnConstant;
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
81 {
82     for (var i = 0; i < elements.Count; i++)
83     {
84         array.Add(ref position, elements[i]);
85     }
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
90     ↪ TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
91     ↪ TReturnConstant returnConstant)
92 {
93     array.AddSkipFirst(ref position, elements);
94     return returnConstant;
95 }

```

```

85     }
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
89     ↪ => array.AddSkipFirst(ref position, elements, 1);
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
93     ↪ int skip)
94     {
95         for (var i = skip; i < elements.Count; i++)
96         {
97             array.Add(ref position, elements[i]);
98         }
99     }

```

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;
6  using System.Threading.Tasks;
7  using Platform.Exceptions;
8  using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17     ↪ 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19     ↪ байт в 8 байт.
20     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21     ↪ помощью которой можно быстро
22     /// проверять есть ли значения непосредственно далее (ниже по уровню).
23     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24     /// </remarks>
25     public class BitString : IEquatable<BitString>
26     {
27         private static readonly byte[][] _bitsSetIn16Bits;
28         private long[] _array;
29         private long _length;
30         private long _minPositiveWord;
31         private long _maxPositiveWord;
32
33         public bool this[long index]
34         {
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             get => Get(index);
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             set => Set(index, value);
39         }
40
41         public long Length
42         {
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             get => _length;
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             set
47             {
48                 if (_length == value)
49                 {
50                     return;
51                 }
52                 Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
53                 // Currently we never shrink the array
54                 if (value > _length)
55                 {
56                     var words = GetWordsCountFromIndex(value);
57                     var oldWords = GetWordsCountFromIndex(_length);
58                     if (words > _array.LongLength)
59                     {
60                         var copy = new long[words];

```

```

58         Array.Copy(_array, copy, _array.LongLength);
59         _array = copy;
60     }
61     else
62     {
63         // What is going on here?
64         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
65     }
66     // What is going on here?
67     var mask = (int)(_length % 64);
68     if (mask > 0)
69     {
70         _array[oldWords - 1] &= (1L << mask) - 1;
71     }
72 }
73 else
74 {
75     // Looks like minimum and maximum positive words are not updated
76     throw new NotImplementedException();
77 }
78 _length = value;
79 }
80 }
81
82 #region Constructors
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 static BitString()
86 {
87     _bitsSetIn16Bits = new byte[65536] [];
88     int i, c, k;
89     byte bitIndex;
90     for (i = 0; i < 65536; i++)
91     {
92         // Calculating size of array (number of positive bits)
93         for (c = 0, k = 1; k <= 65536; k <= 1)
94         {
95             if ((i & k) == k)
96             {
97                 c++;
98             }
99         }
100         var array = new byte[c];
101         // Adding positive bits indices into array
102         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
103         {
104             if ((i & k) == k)
105             {
106                 array[c++] = bitIndex;
107             }
108             bitIndex++;
109         }
110         _bitsSetIn16Bits[i] = array;
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public BitString(BitString other)
116 {
117     Ensure.Always.ArgumentNotNull(other, nameof(other));
118     _length = other._length;
119     _array = new long[GetWordsCountFromIndex(_length)];
120     _minPositiveWord = other._minPositiveWord;
121     _maxPositiveWord = other._maxPositiveWord;
122     Array.Copy(other._array, _array, _array.LongLength);
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public BitString(long length)
127 {
128     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
129     _length = length;
130     _array = new long[GetWordsCountFromIndex(_length)];
131     MarkBordersAsAllBitsReset();
132 }
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public BitString(long length, bool defaultValue)
136     : this(length)

```

```

137 {
138     if (defaultValue)
139     {
140         SetAll();
141     }
142 }
143
144 #endregion
145
146 [MethodImpl(MethodImplOptions.AggressiveInlining)]
147 public BitString Not()
148 {
149     for (var i = 0L; i < _array.LongLength; i++)
150     {
151         _array[i] = ~_array[i];
152         RefreshBordersByWord(i);
153     }
154     return this;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public BitString ParallelNot()
159 {
160     var threads = Environment.ProcessorCount / 2;
161     if (threads <= 1)
162     {
163         return Not();
164     }
165     var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
166 ↪ threads);
167     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
168 ↪ MaxDegreeOfParallelism = threads }, range =>
169     {
170         var maximum = range.Item2;
171         for (var i = range.Item1; i < maximum; i++)
172         {
173             _array[i] = ~_array[i];
174         }
175     });
176     MarkBordersAsAllBitsSet();
177     TryShrinkBorders();
178     return this;
179 }
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public BitString VectorNot()
183 {
184     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
185     {
186         return Not();
187     }
188     var step = Vector<long>.Count;
189     if (_array.Length < step)
190     {
191         return Not();
192     }
193     VectorNotLoop(_array, step, 0, _array.Length);
194     MarkBordersAsAllBitsSet();
195     TryShrinkBorders();
196     return this;
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public BitString ParallelVectorNot()
201 {
202     var threads = Environment.ProcessorCount / 2;
203     if (threads <= 1)
204     {
205         return VectorNot();
206     }
207     if (!Vector.IsHardwareAccelerated)
208     {
209         return ParallelNot();
210     }
211     var step = Vector<long>.Count;
212     if (_array.Length < (step * threads))
213     {
214         return VectorNot();
215     }

```



```

213     }
214     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
215     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
        ↪ range.Item1, range.Item2));
216     MarkBordersAsAllBitsSet();
217     TryShrinkBorders();
218     return this;
219 }
220
221 [MethodImpl(MethodImplOptions.AggressiveInlining)]
222 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
223 {
224     var i = start;
225     var range = maximum - start - 1;
226     var stop = range - (range % step);
227     for (; i < stop; i += step)
228     {
229         (~new Vector<long>(array, i)).CopyTo(array, i);
230     }
231     for (; i < maximum; i++)
232     {
233         array[i] = ~array[i];
234     }
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 {
316         ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
317         ↪ step, range.Item1, range.Item2));
318     MarkBordersAsAllBitsSet();
319     TryShrinkBorders();
320     return this;
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
325 ↪ int maximum)
326 {
327     var i = start;
328     var range = maximum - start - 1;
329     var stop = range - (range % step);
330     for (; i < stop; i += step)
331     {
332         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
333     }
334     for (; i < maximum; i++)
335     {
336         array[i] &= otherArray[i];
337     }
338 }
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 public BitString Or(BitString other)
342 {
343     EnsureBitStringHasTheSameSize(other, nameof(other));
344     GetCommonOuterBorders(this, other, out long from, out long to);
345     for (var i = from; i <= to; i++)
346     {
347         _array[i] |= other._array[i];
348         RefreshBordersByWord(i);
349     }
350     return this;
351 }
352
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public BitString ParallelOr(BitString other)
355 {
356     var threads = Environment.ProcessorCount / 2;
357     if (threads <= 1)
358     {
359         return Or(other);
360     }
361     EnsureBitStringHasTheSameSize(other, nameof(other));
362     GetCommonOuterBorders(this, other, out long from, out long to);
363     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
364     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
365         ↪ 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     }
386     EnsureBitStringHasTheSameSize(other, nameof(other));
387     GetCommonOuterBorders(this, other, out int from, out int to);
388     VectorOrLoop(_array, other._array, step, from, to + 1);
389     MarkBordersAsAllBitsSet();
390     TryShrinkBorders();
391     return this;
392 }
393
394 [MethodImpl(MethodImplOptions.AggressiveInlining)]
395 public BitString ParallelVectorOr(BitString other)
396 {
397     var threads = Environment.ProcessorCount / 2;
398     if (threads <= 1)
399     {
400         return VectorOr(other);
401     }
402     if (!Vector.IsHardwareAccelerated)
403     {
404         return ParallelOr(other);
405     }
406     var step = Vector<long>.Count;
407     if (_array.Length < (step * threads))
408     {
409         return VectorOr(other);
410     }
411     EnsureBitStringHasTheSameSize(other, nameof(other));
412     GetCommonOuterBorders(this, other, out int from, out int to);
413     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
414     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
415         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
416         ↪ step, range.Item1, range.Item2));
417     MarkBordersAsAllBitsSet();
418     TryShrinkBorders();
419     return this;
420 }
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
424     ↪ int maximum)
425 {
426     var i = start;
427     var range = maximum - start - 1;
428     var stop = range - (range % step);
429     for (; i < stop; i += step)
430     {
431         (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
432     }
433     for (; i < maximum; i++)
434     {
435         array[i] |= otherArray[i];
436     }
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

513     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
514         ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
515         ↪ step, range.Item1, range.Item2));
516     MarkBordersAsAllBitsSet();
517     TryShrinkBorders();
518     return this;
519 }
520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
521 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
522 ↪ int maximum)
523 {
524     var i = start;
525     var range = maximum - start - 1;
526     var stop = range - (range % step);
527     for (; i < stop; i += step)
528     {
529         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
530     }
531     for (; i < maximum; i++)
532     {
533         array[i] ^= otherArray[i];
534     }
535 }
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 private void RefreshBordersByWord(long wordIndex)
538 {
539     if (_array[wordIndex] == 0)
540     {
541         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
542         {
543             _minPositiveWord++;
544         }
545         if (wordIndex == _maxPositiveWord && wordIndex != 0)
546         {
547             _maxPositiveWord--;
548         }
549     }
550     else
551     {
552         if (wordIndex < _minPositiveWord)
553         {
554             _minPositiveWord = wordIndex;
555         }
556         if (wordIndex > _maxPositiveWord)
557         {
558             _maxPositiveWord = wordIndex;
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++)
692     {
693         var word = _array[i];
694         if (word != 0)
695         {
696             AppendAllSetBitIndices(result, i, word);
697         }
698     }
699     return result;
700 }
701
702 [MethodImpl(MethodImplOptions.AggressiveInlining)]
703 public List<ulong> GetSetUInt64Indices()
704 {
705     var result = new List<ulong>();
706     GetBorders(out ulong from, out ulong to);
707     for (var i = from; i <= to; i++)
708     {
709         var word = _array[i];
710         if (word != 0)
711         {
712             AppendAllSetBitIndices(result, i, word);
713         }
714     }
715     return result;
716 }
717
718 [MethodImpl(MethodImplOptions.AggressiveInlining)]
719 public long GetFirstSetBitIndex()
720 {
721     var i = _minPositiveWord;
722     var word = _array[i];
723     if (word != 0)
724     {
725         return GetFirstSetBitForWord(i, word);
726     }
727     return -1;
728 }
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public long GetLastSetBitIndex()
732 {
733     var i = _maxPositiveWord;
734     var word = _array[i];
735     if (word != 0)
736     {
737         return GetLastSetBitForWord(i, word);
738     }
739     return -1;
740 }
741
742 [MethodImpl(MethodImplOptions.AggressiveInlining)]
743 public long CountSetBits()
744 {
745     var total = 0L;
746     GetBorders(out long from, out long to);

```

```

747     for (var i = from; i <= to; i++)
748     {
749         var word = _array[i];
750         if (word != 0)
751         {
752             total += CountSetBitsForWord(word);
753         }
754     }
755     return total;
756 }
757
758 [MethodImpl(MethodImplOptions.AggressiveInlining)]
759 public bool HaveCommonBits(BitString other)
760 {
761     EnsureBitStringHasTheSameSize(other, nameof(other));
762     GetCommonInnerBorders(this, other, out long from, out long to);
763     var otherArray = other._array;
764     for (var i = from; i <= to; i++)
765     {
766         var left = _array[i];
767         var right = otherArray[i];
768         if (left != 0 && right != 0 && (left & right) != 0)
769         {
770             return true;
771         }
772     }
773     return false;
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 public long CountCommonBits(BitString other)
778 {
779     EnsureBitStringHasTheSameSize(other, nameof(other));
780     GetCommonInnerBorders(this, other, out long from, out long to);
781     var total = 0L;
782     var otherArray = other._array;
783     for (var i = from; i <= to; i++)
784     {
785         var left = _array[i];
786         var right = otherArray[i];
787         var combined = left & right;
788         if (combined != 0)
789         {
790             total += CountSetBitsForWord(combined);
791         }
792     }
793     return total;
794 }
795
796 [MethodImpl(MethodImplOptions.AggressiveInlining)]
797 public List<long> GetCommonIndices(BitString other)
798 {
799     EnsureBitStringHasTheSameSize(other, nameof(other));
800     GetCommonInnerBorders(this, other, out long from, out long to);
801     var result = new List<long>();
802     var otherArray = other._array;
803     for (var i = from; i <= to; i++)
804     {
805         var left = _array[i];
806         var right = otherArray[i];
807         var combined = left & right;
808         if (combined != 0)
809         {
810             AppendAllSetBitIndices(result, i, combined);
811         }
812     }
813     return result;
814 }
815
816 [MethodImpl(MethodImplOptions.AggressiveInlining)]
817 public List<ulong> GetCommonUInt64Indices(BitString other)
818 {
819     EnsureBitStringHasTheSameSize(other, nameof(other));
820     GetCommonBorders(this, other, out ulong from, out ulong to);
821     var result = new List<ulong>();
822     var otherArray = other._array;
823     for (var i = from; i <= to; i++)
824     {
825         var left = _array[i];

```



```

826         var right = otherArray[i];
827         var combined = left & right;
828         if (combined != 0)
829         {
830             AppendAllSetBitIndices(result, i, combined);
831         }
832     }
833     return result;
834 }
835
836 [MethodImpl(MethodImplOptions.AggressiveInlining)]
837 public long GetFirstCommonBitIndex(BitString other)
838 {
839     EnsureBitStringHasTheSameSize(other, nameof(other));
840     GetCommonInnerBorders(this, other, out long from, out long to);
841     var otherArray = other._array;
842     for (var i = from; i <= to; i++)
843     {
844         var left = _array[i];
845         var right = otherArray[i];
846         var combined = left & right;
847         if (combined != 0)
848         {
849             return GetFirstSetBitForWord(i, combined);
850         }
851     }
852     return -1;
853 }
854
855 [MethodImpl(MethodImplOptions.AggressiveInlining)]
856 public long GetLastCommonBitIndex(BitString other)
857 {
858     EnsureBitStringHasTheSameSize(other, nameof(other));
859     GetCommonInnerBorders(this, other, out long from, out long to);
860     var otherArray = other._array;
861     for (var i = to; i >= from; i--)
862     {
863         var left = _array[i];
864         var right = otherArray[i];
865         var combined = left & right;
866         if (combined != 0)
867         {
868             return GetLastSetBitForWord(i, combined);
869         }
870     }
871     return -1;
872 }
873
874 [MethodImpl(MethodImplOptions.AggressiveInlining)]
875 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
    ↪ false;
876
877 [MethodImpl(MethodImplOptions.AggressiveInlining)]
878 public bool Equals(BitString other)
879 {
880     if (_length != other._length)
881     {
882         return false;
883     }
884     var otherArray = other._array;
885     if (_array.Length != otherArray.Length)
886     {
887         return false;
888     }
889     if (_minPositiveWord != other._minPositiveWord)
890     {
891         return false;
892     }
893     if (_maxPositiveWord != other._maxPositiveWord)
894     {
895         return false;
896     }
897     GetCommonBorders(this, other, out ulong from, out ulong to);
898     for (var i = from; i <= to; i++)
899     {
900         if (_array[i] != otherArray[i])
901         {
902             return false;
903         }

```

```

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

```

```

974 {
975     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
        ↳ bits32to47, out byte[] bits48to63);
976     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
977 }
978
979 [MethodImpl(MethodImplOptions.AggressiveInlining)]
980 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
981 {
982     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
        ↳ bits32to47, out byte[] bits48to63);
983     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
984 }
985
986 [MethodImpl(MethodImplOptions.AggressiveInlining)]
987 private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
        ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
988 {
989     for (var j = 0; j < bits00to15.Length; j++)
990     {
991         result.Add(bits00to15[j] + (i * 64));
992     }
993     for (var j = 0; j < bits16to31.Length; j++)
994     {
995         result.Add(bits16to31[j] + 16 + (i * 64));
996     }
997     for (var j = 0; j < bits32to47.Length; j++)
998     {
999         result.Add(bits32to47[j] + 32 + (i * 64));
1000     }
1001     for (var j = 0; j < bits48to63.Length; j++)
1002     {
1003         result.Add(bits48to63[j] + 48 + (i * 64));
1004     }
1005 }
1006
1007 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1008 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
        ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1009 {
1010     for (var j = 0; j < bits00to15.Length; j++)
1011     {
1012         result.Add(bits00to15[j] + (i * 64));
1013     }
1014     for (var j = 0; j < bits16to31.Length; j++)
1015     {
1016         result.Add(bits16to31[j] + 16UL + (i * 64));
1017     }
1018     for (var j = 0; j < bits32to47.Length; j++)
1019     {
1020         result.Add(bits32to47[j] + 32UL + (i * 64));
1021     }
1022     for (var j = 0; j < bits48to63.Length; j++)
1023     {
1024         result.Add(bits48to63[j] + 48UL + (i * 64));
1025     }
1026 }
1027
1028 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
        ↳ bits32to47, byte[] bits48to63)
1030 {
1031     if (bits00to15.Length > 0)
1032     {
1033         return bits00to15[0] + (i * 64);
1034     }
1035     if (bits16to31.Length > 0)
1036     {
1037         return bits16to31[0] + 16 + (i * 64);
1038     }
1039     if (bits32to47.Length > 0)
1040     {
1041         return bits32to47[0] + 32 + (i * 64);
1042     }
1043     return bits48to63[0] + 48 + (i * 64);
1044 }
1045

```

```

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

```

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

```
1 using System.Runtime.CompilerServices;
2 using Platform.Random;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {
8     public static class BitStringExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void SetRandomBits(this BitString @string)
12         {
13             for (var i = 0; i < @string.Length; i++)
14             {
15                 var value = RandomHelpers.Default.NextBoolean();
16                 @string.Set(i, value);
17             }
18         }
19     }
20 }
```

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

```
1 using System.Collections.Concurrent;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Concurrent
8 {
9     public static class ConcurrentQueueExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
13         {
14             while (queue.TryDequeue(out T item))
15             {
16                 yield return item;
17             }
18         }
19     }
20 }
```

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

```
1 using System.Collections.Concurrent;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Concurrent
7 {
8     public static class ConcurrentStackExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
12             ↪ value) ? value : default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
16             ↪ value) ? value : default;
17     }
18 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Exceptions.ExtensionRoots;
7
8 #pragma warning disable IDE0060 // Remove unused parameter
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15     }
```

```

15 #region Always
16
17 [MethodImpl(MethodImplOptions.AggressiveInlining)]
18 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
19     ↪ ICollection<T> argument, string argumentName, string message)
20 {
21     if (argument.IsNullOrEmpty())
22     {
23         throw new ArgumentException(message, argumentName);
24     }
25 }
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
29     ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
30     ↪ argumentName, null);
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
34     ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
38     ↪ string argument, string argumentName, string message)
39 {
40     if (string.IsNullOrEmpty(argument))
41     {
42         throw new ArgumentException(message, argumentName);
43     }
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
48     ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
49     ↪ argument, argumentName, null);
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
53     ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
54
55 #endregion
56
57 #region OnDebug
58
59 [Conditional("DEBUG")]
60 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
61     ↪ ICollection<T> argument, string argumentName, string message) =>
62     ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
63
64 [Conditional("DEBUG")]
65 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
66     ↪ ICollection<T> argument, string argumentName) =>
67     ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
68
69 [Conditional("DEBUG")]
70 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
71     ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
72
73 [Conditional("DEBUG")]
74 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
75     ↪ root, string argument, string argumentName, string message) =>
76     ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
77
78 [Conditional("DEBUG")]
79 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
80     ↪ root, string argument, string argumentName) =>
81     ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
82
83 [Conditional("DEBUG")]
84 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
85     ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
86     ↪ null, null);
87
88 #endregion
89 }
90 }

```

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

```
1 using System.Collections.Generic;
2 using System.Linq;
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 ICollectionExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
13             ↪ null || collection.Count == 0;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
17         {
18             var equalityComparer = EqualityComparer<T>.Default;
19             return collection.All(item => equalityComparer.Equals(item, default));
20         }
21     }
22 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class IDictionaryExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
13             ↪ dictionary, TKey key)
14         {
15             dictionary.TryGetValue(key, out TValue value);
16             return value;
17         }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
21             ↪ TKey key, Func<TKey, TValue> valueFactory)
22         {
23             if (!dictionary.TryGetValue(key, out TValue value))
24             {
25                 value = valueFactory(key);
26                 dictionary.Add(key, value);
27                 return value;
28             }
29             return value;
30         }
31     }
32 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Lists
7 {
8     public static class CharListExtensions
9     {
10         /// <remarks>
11         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12         /// </remarks>
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static int GenerateHashCode(this IList<char> list)
15         {
16             var hashSeed = 5381;
17             var hashAccumulator = hashSeed;
18             for (var i = 0; i < list.Count; i++)
19             {
20             }
```

```

20         hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
21     }
22     return hashAccumulator + (hashSeed * 1566083941);
23 }
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public static bool EqualTo(this IList<char> left, IList<char> right) =>
    ↪ left.EqualTo(right, ContentEqualTo);
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public static bool ContentEqualTo(this IList<char> left, IList<char> right)
30 {
31     for (var i = left.Count - 1; i >= 0; --i)
32     {
33         if (left[i] != right[i])
34         {
35             return false;
36         }
37     }
38     return true;
39 }
40 }
41 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Lists
7 {
8     public class IListComparer<T> : IComparer<IList<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
12     }
13 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Lists
7 {
8     public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
15     }
16 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Lists
8 {
9     public static class IListExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
13         {
14             list.Add(element);
15             return true;
16         }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
20         {
21             list.AddFirst(elements);

```



```

22         return true;
23     }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
27         ↪ list.Add(elements[0]);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
31     {
32         list.AddAll(elements);
33         return true;
34     }
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public static void AddAll<T>(this IList<T> list, IList<T> elements)
38     {
39         for (var i = 0; i < elements.Count; i++)
40         {
41             list.Add(elements[i]);
42         }
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
47     {
48         list.AddSkipFirst(elements);
49         return true;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
54         ↪ list.AddSkipFirst(elements, 1);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
58     {
59         for (var i = skip; i < elements.Count; i++)
60         {
61             list.Add(elements[i]);
62         }
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
70         ↪ right, ContentEqualTo);
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
74         ↪ IList<T>, bool> contentEqualityComparer)
75     {
76         if (ReferenceEquals(left, right))
77         {
78             return true;
79         }
80         var leftCount = left.GetCountOrZero();
81         var rightCount = right.GetCountOrZero();
82         if (leftCount == 0 && rightCount == 0)
83         {
84             return true;
85         }
86         if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
87         {
88             return false;
89         }
90         return contentEqualityComparer(left, right);
91     }
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
95     {
96         var equalityComparer = EqualityComparer<T>.Default;
97         for (var i = left.Count - 1; i >= 0; --i)
98         {
99             if (!equalityComparer.Equals(left[i], right[i]))
100             {

```

```

97         return false;
98     }
99 }
100 return true;
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
105 {
106     if (list == null)
107     {
108         return null;
109     }
110     var result = new List<T>(list.Count);
111     for (var i = 0; i < list.Count; i++)
112     {
113         if (predicate(list[i]))
114         {
115             result.Add(list[i]);
116         }
117     }
118     return result.ToArray();
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static T[] ToArray<T>(this IList<T> list)
123 {
124     var array = new T[list.Count];
125     list.CopyTo(array, 0);
126     return array;
127 }
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static void ForEach<T>(this IList<T> list, Action<T> action)
131 {
132     for (var i = 0; i < list.Count; i++)
133     {
134         action(list[i]);
135     }
136 }
137
138 /// <remarks>
139 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
140 /// ↪ -overridden-system-object-gethashcode
141 /// </remarks>
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public static int GenerateHashCode<T>(this IList<T> list)
144 {
145     var hashAccumulator = 17;
146     for (var i = 0; i < list.Count; i++)
147     {
148         hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
149     }
150     return hashAccumulator;
151 }
152
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 public static int CompareTo<T>(this IList<T> left, IList<T> right)
155 {
156     var comparer = Comparer<T>.Default;
157     var leftCount = left.GetCountOrZero();
158     var rightCount = right.GetCountOrZero();
159     var intermediateResult = leftCount.CompareTo(rightCount);
160     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
161     {
162         intermediateResult = comparer.Compare(left[i], right[i]);
163     }
164     return intermediateResult;
165 }
166
167 [MethodImpl(MethodImplOptions.AggressiveInlining)]
168 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
169
170 [MethodImpl(MethodImplOptions.AggressiveInlining)]
171 public static T[] SkipFirst<T>(this IList<T> list, int skip)
172 {
173     if (list.IsNullOrEmpty() || list.Count <= skip)
174     {
175         return Array.Empty<T>();
176     }
177 }

```

```

175     }
176     var result = new T[list.Count - skip];
177     for (int r = skip, w = 0; r < list.Count; r++, w++)
178     {
179         result[w] = list[r];
180     }
181     return result;
182 }
183
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
186
187 [MethodImpl(MethodImplOptions.AggressiveInlining)]
188 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
189 {
190     if (shift < 0)
191     {
192         throw new NotImplementedException();
193     }
194     if (shift == 0)
195     {
196         return list.ToArray();
197     }
198     else
199     {
200         var result = new T[list.Count + shift];
201         for (int r = 0, w = shift; r < list.Count; r++, w++)
202         {
203             result[w] = list[r];
204         }
205         return result;
206     }
207 }
208 }
209 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Lists
7 {
8     public class ListFiller<TElement, TReturnConstant>
9     {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
15         {
16             _list = list;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ListFiller(List<TElement> list) : this(list, default) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _list.Add(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
31             ↪ _list.AddFirstAndReturnTrue(elements);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddAllAndReturnTrue(IList<TElement> elements) =>
35             ↪ _list.AddAllAndReturnTrue(elements);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
39             ↪ _list.AddSkipFirstAndReturnTrue(elements);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public TReturnConstant AddAndReturnConstant(TElement element)

```

```

40     {
41         _list.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
47     {
48         _list.AddFirst(elements);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
54     {
55         _list.AddAll(elements);
56         return _returnConstant;
57     }
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
61     {
62         _list.AddSkipFirst(elements);
63         return _returnConstant;
64     }
65 }
66 }

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public CharSegment(ICollection<char> @base, int offset, int length) : base(@base, offset,
15             ↪ length) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override int GetHashCode()
19         {
20             // Base can be not an array, but still ICollection<char>
21             if (Base is char[] baseArray)
22             {
23                 return baseArray.GenerateHashCode(Offset, Length);
24             }
25             else
26             {
27                 return this.GenerateHashCode();
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override bool Equals(Segment<char> other)
33         {
34             bool contentEqualityComparer(ICollection<char> left, ICollection<char> right)
35             {
36                 // Base can be not an array, but still ICollection<char>
37                 if (Base is char[] baseArray && other.Base is char[] otherArray)
38                 {
39                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
40                 }
41                 else
42                 {
43                     return left.ContentEqualTo(right);
44                 }
45             }
46             return this.EqualTo(other, contentEqualityComparer);
47         }
48
49         public override bool Equals(object obj) => obj is Segment<char> charSegment ?
50             ↪ Equals(charSegment) : false;

```

```

49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public static implicit operator string(CharSegment segment)
51     {
52         if (!(segment.Base is char[] array))
53         {
54             array = segment.Base.ToArray();
55         }
56         return new string(array, segment.Offset, segment.Length);
57     }
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public override string ToString() => this;
62 }
63 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
13     {
14         public IList<T> Base
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19         public int Offset
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23         }
24         public int Length
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Segment(IList<T> @base, int offset, int length)
32         {
33             Base = @base;
34             Offset = offset;
35             Length = length;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public override int GetHashCode() => this.GenerateHashCode();
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
46             ↪ false;
47
48         #region IList
49         public T this[int i]
50         {
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             get => Base[Offset + i];
53             [MethodImpl(MethodImplOptions.AggressiveInlining)]
54             set => Base[Offset + i] = value;
55         }
56
57         public int Count
58         {
59             [MethodImpl(MethodImplOptions.AggressiveInlining)]
60             get => Length;
61         }
62     }

```

```

63     public bool IsReadOnly
64     {
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         get => true;
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public int IndexOf(T item)
71     {
72         var index = Base.IndexOf(item);
73         if (index >= Offset)
74         {
75             var actualIndex = index - Offset;
76             if (actualIndex < Length)
77             {
78                 return actualIndex;
79             }
80         }
81         return -1;
82     }
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     public void Insert(int index, T item) => throw new NotSupportedException();
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public void RemoveAt(int index) => throw new NotSupportedException();
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public void Add(T item) => throw new NotSupportedException();
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public void Clear() => throw new NotSupportedException();
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public bool Contains(T item) => IndexOf(item) >= 0;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public void CopyTo(T[] array, int arrayIndex)
101    {
102        for (var i = 0; i < Length; i++)
103        {
104            array.Add(ref arrayIndex, this[i]);
105        }
106    }
107
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public bool Remove(T item) => throw new NotSupportedException();
110
111    [MethodImpl(MethodImplOptions.AggressiveInlining)]
112    public IEnumerator<T> GetEnumerator()
113    {
114        for (var i = 0; i < Length; i++)
115        {
116            yield return this[i];
117        }
118    }
119
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
122
123    #endregion
124 }
125 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class AllSegmentsWalkerBase
6      {
7          public static readonly int DefaultMinimumStringSegmentLength = 2;
8      }
9  }

```

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

```

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

```

```

4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9         where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
15             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public virtual void WalkAll(ICollection<T> elements)
22         {
23             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
24                 ↪ offset <= maxOffset; offset++)
25             {
26                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
27                     ↪ offset; length <= maxLength; length++)
28                 {
29                     Iteration(CreateSegment(elements, offset, length));
30                 }
31             }
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected abstract void Iteration(TSegment segment);
38         }
39     }
40 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override Segment<T> CreateSegment(ICollection<T> elements, int offset, int length)
12             ↪ => new Segment<T>(elements, offset, length);
13     }
14 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public static class AllSegmentsWalkerExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
11             ↪ walker.WalkAll(@string.ToCharArray());
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
15             ↪ string @string) where TSegment : Segment<char> =>
16             ↪ walker.WalkAll(@string.ToCharArray());
17     }
18 }

```

1.26 ./csharp/Platform.Collections.Segments.Walkers.DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment<T>].cs

```

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

```

```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Segments.Walkers
8 {
9     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
10         ↪ DuplicateSegmentsWalkerBase<T, TSegment>
11         where TSegment : Segment<T>
12     {
13         public static readonly bool DefaultResetDictionaryOnEachWalk;
14
15         private readonly bool _resetDictionaryOnEachWalk;
16         protected IDictionary<TSegment, long> Dictionary;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
20             ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
21             : base(minimumStringSegmentLength)
22         {
23             Dictionary = dictionary;
24             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
29             ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
30             ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
34             ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
35             ↪ DefaultResetDictionaryOnEachWalk) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
39             ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
40             ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
41             ↪ { }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
45             ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
49             ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public override void WalkAll(ICollection<T> elements)
53         {
54             if (_resetDictionaryOnEachWalk)
55             {
56                 var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
57                 Dictionary = new Dictionary<TSegment, long>((int)capacity);
58             }
59             base.WalkAll(elements);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override long GetSegmentFrequency(TSegment segment) =>
64             ↪ Dictionary.GetOrDefault(segment);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
68             ↪ Dictionary[segment] = frequency;
69     }
70 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
9         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

11     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
    ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27 }
28 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
    ↪ TSegment>
    ↪ where TSegment : Segment<T>
8     {
9
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override void Iteration(TSegment segment)
18         {
19             var frequency = GetSegmentFrequency(segment);
20             if (frequency == 1)
21             {
22                 OnDuplicateFound(segment);
23             }
24             SetSegmentFrequency(segment, frequency + 1);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected abstract void OnDuplicateFound(TSegment segment);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract long GetSegmentFrequency(TSegment segment);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
35     }
36 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Sets
7  {
8      public static class ISetExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
15             ↪ set.Remove(element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
19         {
20             set.Add(element);
21             return true;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
26         {
27             AddFirst(set, elements);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
33             ↪ set.Add(elements[0]);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
37         {
38             set.AddAll(elements);
39             return true;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public static void AddAll<T>(this ISet<T> set, IList<T> elements)
44         {
45             for (var i = 0; i < elements.Count; i++)
46             {
47                 set.Add(elements[i]);
48             }
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
53         {
54             set.AddSkipFirst(elements);
55             return true;
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
60             ↪ set.AddSkipFirst(elements, 1);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
64         {
65             for (var i = skip; i < elements.Count; i++)
66             {
67                 set.Add(elements[i]);
68             }
69         }
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         public static bool DoNotContains<T>(this ISet<T> set, T element) =>
73             ↪ !set.Contains(element);
74     }
75 }

```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Sets
7 {
8     public class SetFiller<TElement, TReturnConstant>
9     {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
15         {
16             _set = set;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public SetFiller(ISet<TElement> set) : this(set, default) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _set.Add(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
31             => _set.AddFirstAndReturnTrue(elements);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
35             => _set.AddAllAndReturnTrue(elements);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
39             => _set.AddSkipFirstAndReturnTrue(elements);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public TReturnConstant AddAndReturnConstant(TElement element)
43         {
44             _set.Add(element);
45             return _returnConstant;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
50         {
51             _set.AddFirst(elements);
52             return _returnConstant;
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
57         {
58             _set.AddAll(elements);
59             return _returnConstant;
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
64         {
65             _set.AddSkipFirst(elements);
66             return _returnConstant;
67         }
68     }
69 }
```

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

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

```

9     {
10         public bool IsEmpty
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get => Count <= 0;
14         }
15     }
16 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public interface IStack<TElement>
8     {
9         bool IsEmpty
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         void Push(TElement element);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         TElement Pop();
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         TElement Peek();
23     }
24 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public static class IStackExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static void Clear<T>(this IStack<T> stack)
11        {
12            while (!stack.IsEmpty)
13            {
14                _ = stack.Pop();
15            }
16        }
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
20            ↪ stack.Pop();
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
24            ↪ stack.Peek();
25    }
26 }

```

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

```

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

```

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

```

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

```

```

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

```

```

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.IsNullOrEmpty(@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) =>
37             => @string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
38             => Math.Min(@string.Length, maxLength));
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static string TrimSingle(this string @string, char charToTrim)
42         {
43             if (!@string.IsNullOrEmpty(@string))
44             {
45                 if (@string.Length == 1)
46                 {
47                     if (@string[0] == charToTrim)
48                     {
49                         return "";
50                     }
51                     else
52                     {
53                         return @string;
54                     }
55                 }
56                 else
57                 {
58                     var left = 0;
59                     var right = @string.Length - 1;
60                     if (@string[left] == charToTrim)
61                     {
62                         left++;
63                     }
64                     if (@string[right] == charToTrim)
65                     {
66                         right--;
67                     }
68                     return @string.Substring(left, right - left + 1);
69                 }
70             }
71             return @string;
72         }
73     }
74 }

```

```

61     }
62     if (@string[right] == charToTrim)
63     {
64         right--;
65     }
66     return @string.Substring(left, right - left + 1);
67 }
68 }
69 else
70 {
71     return @string;
72 }
73 }
74 }
75 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  // ReSharper disable ForCanBeConvertedToForeach
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Trees
8  {
9      public class Node
10     {
11         private Dictionary<object, Node> _childNodes;
12
13         public object Value
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get;
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             set;
19         }
20
21         public Dictionary<object, Node> ChildNodes
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
25         }
26
27         public Node this[object key]
28         {
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             get => GetChild(key) ?? AddChild(key);
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             set => SetChildValue(value, key);
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public Node(object value) => Value = value;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public Node() : this(null) { }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public Node GetChild(params object[] keys)
46         {
47             var node = this;
48             for (var i = 0; i < keys.Length; i++)
49             {
50                 node.ChildNodes.TryGetValue(keys[i], out node);
51                 if (node == null)
52                 {
53                     return null;
54                 }
55             }
56             return node;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public Node AddChild(object key) => AddChild(key, new Node(null));

```

```

64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public Node AddChild(object key, Node child)
69 {
70     ChildNodes.Add(key, child);
71     return child;
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 public Node SetChild(object key) => SetChildValue(null, key);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public Node SetChildValue(object value, params object[] keys)
82 {
83     var node = this;
84     for (var i = 0; i < keys.Length; i++)
85     {
86         node = SetChildValue(value, keys[i]);
87     }
88     node.Value = value;
89     return node;
90 }
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public Node SetChildValue(object value, object key)
94 {
95     if (!ChildNodes.TryGetValue(key, out Node child))
96     {
97         child = AddChild(key, value);
98     }
99     child.Value = value;
100     return child;
101 }
102 }
103 }
104 }

```

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

```

1 using System;
2 using System.Collections;
3 using Xunit;
4 using Platform.Random;
5
6 namespace Platform.Collections.Tests
7 {
8     public static class BitStringTests
9     {
10         [Fact]
11         public static void BitGetSetTest()
12         {
13             const int n = 250;
14             var bitArray = new BitArray(n);
15             var bitString = new BitString(n);
16             for (var i = 0; i < n; i++)
17             {
18                 var value = RandomHelpers.Default.NextBoolean();
19                 bitArray.Set(i, value);
20                 bitString.Set(i, value);
21                 Assert.Equal(value, bitArray.Get(i));
22                 Assert.Equal(value, bitString.Get(i));
23             }
24         }
25
26         [Fact]
27         public static void BitVectorNotTest()
28         {
29             TestToOperationsWithSameMeaning((x, y, w, v) =>
30             {
31                 x.VectorNot();
32                 w.Not();
33             });
34         }
35
36         [Fact]

```

```

37 public static void BitParallelNotTest()
38 {
39     TestToOperationsWithSameMeaning((x, y, w, v) =>
40     {
41         x.ParallelNot();
42         w.Not();
43     });
44 }
45
46 [Fact]
47 public static void BitParallelVectorNotTest()
48 {
49     TestToOperationsWithSameMeaning((x, y, w, v) =>
50     {
51         x.ParallelVectorNot();
52         w.Not();
53     });
54 }
55
56 [Fact]
57 public static void BitVectorAndTest()
58 {
59     TestToOperationsWithSameMeaning((x, y, w, v) =>
60     {
61         x.VectorAnd(y);
62         w.And(v);
63     });
64 }
65
66 [Fact]
67 public static void BitParallelAndTest()
68 {
69     TestToOperationsWithSameMeaning((x, y, w, v) =>
70     {
71         x.ParallelAnd(y);
72         w.And(v);
73     });
74 }
75
76 [Fact]
77 public static void BitParallelVectorAndTest()
78 {
79     TestToOperationsWithSameMeaning((x, y, w, v) =>
80     {
81         x.ParallelVectorAnd(y);
82         w.And(v);
83     });
84 }
85
86 [Fact]
87 public static void BitVectorOrTest()
88 {
89     TestToOperationsWithSameMeaning((x, y, w, v) =>
90     {
91         x.VectorOr(y);
92         w.Or(v);
93     });
94 }
95
96 [Fact]
97 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     [Fact]
116     public static void BitVectorXorTest()
117     {
118         TestToOperationsWithSameMeaning((x, y, w, v) =>
119         {
120             x.VectorXor(y);
121             w.Xor(v);
122         });
123     }
124
125     [Fact]
126     public static void BitParallelXorTest()
127     {
128         TestToOperationsWithSameMeaning((x, y, w, v) =>
129         {
130             x.ParallelXor(y);
131             w.Xor(v);
132         });
133     }
134
135     [Fact]
136     public static void BitParallelVectorXorTest()
137     {
138         TestToOperationsWithSameMeaning((x, y, w, v) =>
139         {
140             x.ParallelVectorXor(y);
141             w.Xor(v);
142         });
143     }
144
145     private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
146     ↪ 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 }

```

1.40 ./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      {
8          [Fact]
9          public static void GetHashCodeEqualsTest()
10         {
11             const string testString = "test test";
12             var testArray = testString.ToCharArray();
13             var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
14             var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
15             Assert.Equal(firstHashCode, secondHashCode);
16         }
17
18         [Fact]
19         public static void EqualsTest()
20         {
21             const string testString = "test test";
22             var testArray = testString.ToCharArray();
23             var first = new CharSegment(testArray, 0, 4);
24             var second = new CharSegment(testArray, 5, 4);

```

```

25         Assert.True(first.Equals(second));
26     }
27 }
28 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Collections.Tests
4  {
5      public static class StringTests
6      {
7          [Fact]
8          public static void CapitalizeFirstLetterTest()
9          {
10             Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
11             Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
12             Assert.Equal(" Hello", " hello".CapitalizeFirstLetter());
13         }
14
15         [Fact]
16         public static void TrimSingleTest()
17         {
18             Assert.Equal("", "".TrimSingle('\'));
19             Assert.Equal("", "''.TrimSingle('\'));
20             Assert.Equal("hello", "'hello'.TrimSingle('\'));
21             Assert.Equal("hello", "hello'".TrimSingle('\'));
22             Assert.Equal("hello", "'hello".TrimSingle('\'));
23         }
24     }
25 }

```

Index

- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 39
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 41
- ./csharp/Platform.Collections.Tests/StringTests.cs, 42
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayPool.cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayString.cs, 3
- ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 3
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 4
- ./csharp/Platform.Collections/BitString.cs, 6
- ./csharp/Platform.Collections/BitStringExtensions.cs, 20
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 21
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 21
- ./csharp/Platform.Collections/EnsureExtensions.cs, 21
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 22
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 23
- ./csharp/Platform.Collections/Lists/CharListExtensions.cs, 23
- ./csharp/Platform.Collections/Lists/ICollectionComparer.cs, 24
- ./csharp/Platform.Collections/Lists/ICollectionEqualityComparer.cs, 24
- ./csharp/Platform.Collections/Lists/ICollectionExtensions.cs, 24
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 27
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 28
- ./csharp/Platform.Collections/Segments/Segment.cs, 29
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 30
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 30
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 31
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 31
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 31
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 32
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 33
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 33
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 34
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 34
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 35
- ./csharp/Platform.Collections/Stacks/IStack.cs, 36
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 36
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 36
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 36
- ./csharp/Platform.Collections/StringExtensions.cs, 37
- ./csharp/Platform.Collections/Trees/Node.cs, 38