1 Lösungsidee

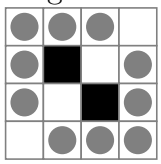
1.1 Kernidee

Rominos mitt n Blöcken können gefunden werden, in dem zu Rominos mit (n-1) Blöcken ein Block angefügt wird. Hierbei muss beachtet werden das der Rominostein zusammenhängend bleiben muss, und dass mindestens eine Diagonale bleiben muss.

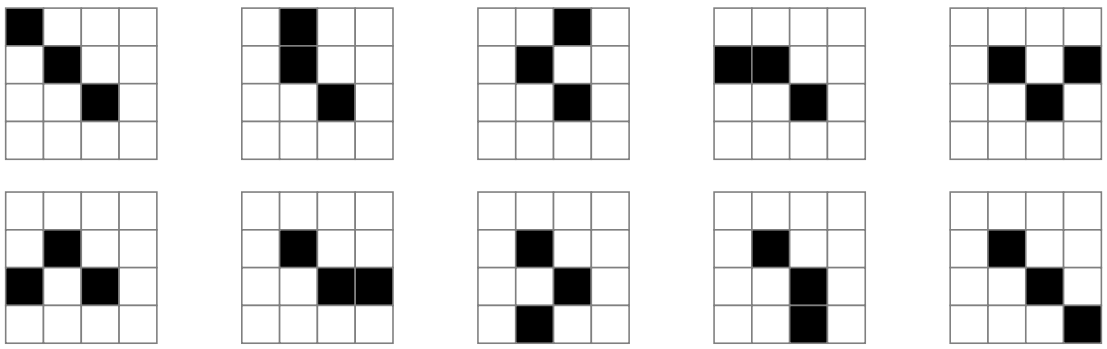
Um alle möglichen Rominos mit n Blöcken zu finden, muss man also alle Rominos mit (n-1) Blöcken finden, und für diese alle Rominos die durch hinzufügen eines weiteren Blocks enstehen können ermitteln. Dabei wird es Duplikate geben. Eliminiert man diese hat man alle möglichen n-Rominos eindeutig gefunden.

1.1.1 Beispiel

Nehme man beispielsweise das 2er-Romino, kann man zum finden aller 3 (= 2 + 1) - Rominos wie folgt Blöcke anfügen:



Somit ergeben sich folgende 3-Rominos:



Da Rominos mindestens zwei Steine haben müssen um eine Diagonale zu besitzen, ist der Rominostein mit den wenigsten Blöcken eine 2er Diagonale.

Kleinster Rominostein

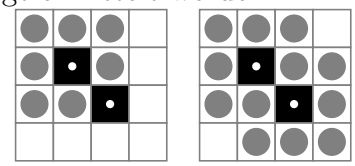
Um alle n-Rominos für ein beliebiges n zu finden, würde man den obigen Algorithmus verwenden um aus dem 2er-Romino alle 3-Rominos zu folgern, dann aus diesen alle 4-Rominos etc. bis man alle n-Rominos errechnet hat.

1.2 Hinzufügen von Blöcken

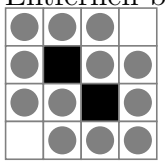
Um Blöcke hinzuzufügen, werden zuerst die Stellen ermittelt, wo Blöcke angefügt werden können, sodass das Romino zusammenhängend bleibt. Hierfür werden die Nachbarn jedes Blocks des Rominos ermittelt, daraufhin werden Duplikate und bereits belegte Blöcke eliminiert.

1.2.1 Beispiel

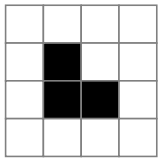
Nehme man beispielsweise wieder das 2er-Romino, würden die Nachbarn aller Blöcke wie folgt ermittelt werden:



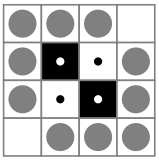
Entfernen bereits existierender Blöcke



Es lässt sich hier erkennen, das die Existenz einer echten Diagonale nicht zwingend aufrecht erhalten wird;



Um dafür zu sorgen, dass diese echte Diagonale immer existiert, wird eine spezifische Diagonale immer beschützt. Bei den Möglichen Block-Additionen beim 2er-Romino beispielsweise würden hierfür die für die Diagonale relevanten Blöcke aus den Block-Additionsmöglichkeiten entfernt:



Diese 4 beschützten Blöcke werden auch bei Spiegelungen, Verschiebungen und Rotationen mitverfolgt, sodass diese eine Diagonale immer besteht.

1.3 Eliminierung von Duplikaten

Zur Eliminierung von Duplikaten werden die Rominos zuerst eindeutig orientiert, um Vergleiche zwischen gleichen, aber transformierten Rominos zu erleichtern.

1.3.1 Verschiebung

Die Verschiebung wird eliminiert durch Verschiebung des Rominos in die linke obere Ecke des Gitters; also wird der Block mit der geringsten x-Koordinate auf x=0 verschoben, und der Block mit der geringsten y-Koordinate auf y=0.

1.3.2 Rotation und Spiegelung

Um Rotation und Spiegelung eines Rominos zu eliminieren, werden zuerst alle seine Permutationen (also alle Kombinationen von Rotation und Spiegelung) ermittelt, und denen wird ein eindeutiger Wert zugewiesen. Daraufhin wird das Romino mit dem höchsten dieser eindeutigen Werte ausgewählt. Hierbei ist es eigentlich egal, ob der niedrigste oder höchste Wert genommen wird, solange das Ergebnis eindeutig ist.

Die Bestimmung dieses eindeutigen Werts haben wir einen trivialen Algorithmus verwendet wie folgt:

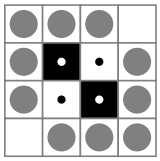
1. Nehme einen Block b aus der Permutation des Rominos
2. Seien die Koordinaten (x, y) die Koordinaten des Blocks b, wobei die minimale x-Koordinate und die minimale y-Koordinate aus allen Blöcken der Permutation 0 ist.
3. Man weise dem Block b den Wert $2^{(y*<\text{Anzahl an Blöcken}>)+x}$ zu
4. Addiere die Werte aller Blöcke der Permutation, sei dies der Wert der Permutation

Dabei ist zwar noch viel Raum für Optimierung, aber dieser Algorithmus ist ausreichend und O(n).

1.3.3 Endgültige Duplikat-Eliminierung

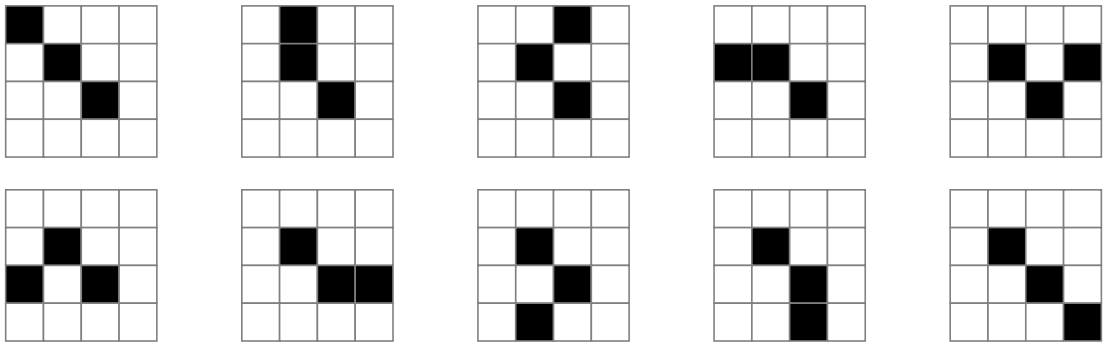
Zum endgültigen eliminieren der Duplikate werden zuerst alle Rominos wie oben beschrieben orientiert, dann werden die eindeutigen Werte dieser verglichen, um schnell Gleichheit zu ermitteln. Durch Verwendung dieser Vergleichsmethode lassen sich schnell Duplikate entfernen.

1.3.4 Beispiel

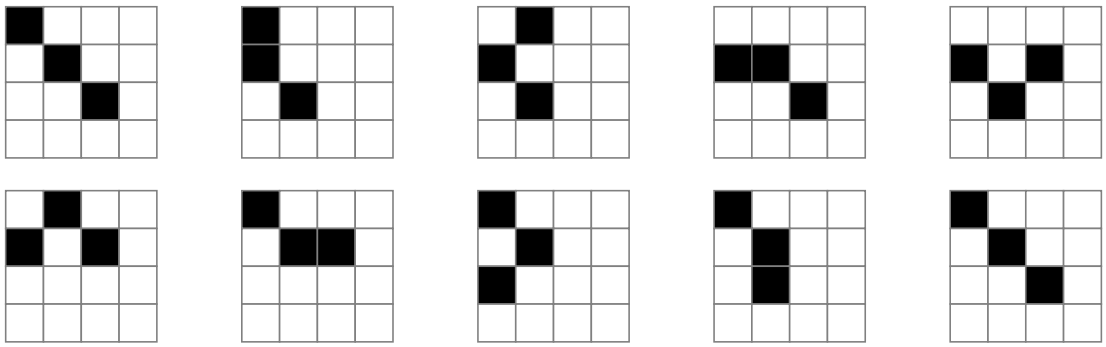


Ausgangsromino

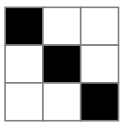
Nächste Rominos



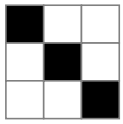
Verschiebung eliminieren



Rotation und Spiegelung eliminieren Ausgehend von dem Romino;

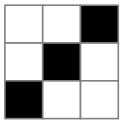


werden folgende Permutationen festgestellt:



Permutation 1

$Wert_1 = 2^0 + 2^4 + 2^8 = 273$

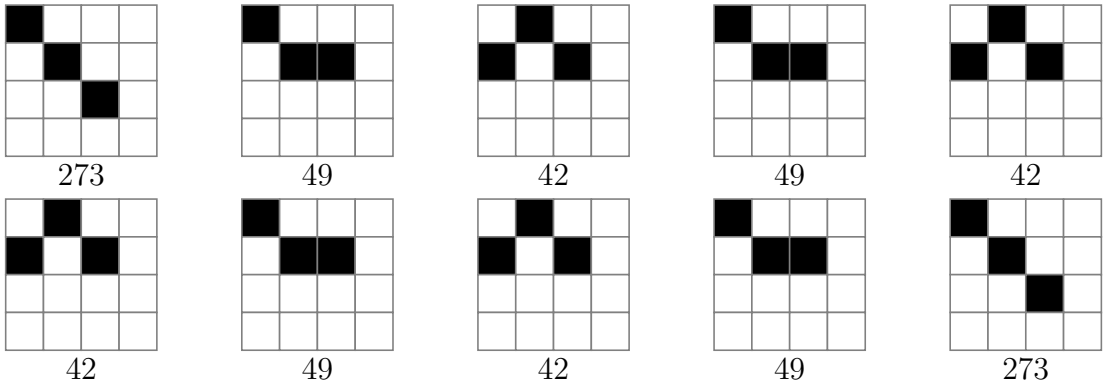


Permutation 2

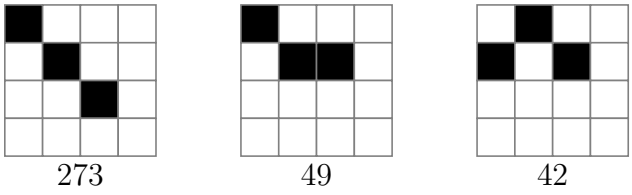
$Wert_2 = 2^2 + 2^4 + 2^6 = 84$

Hierbei ist $Wert_2 = 84 < 273 = Wert_1$. Da Permutation 1 mit $Wert_1$ den höchsten Wert hat, wird Permutation 1 als die eindeutige Rotierung festgelegt.

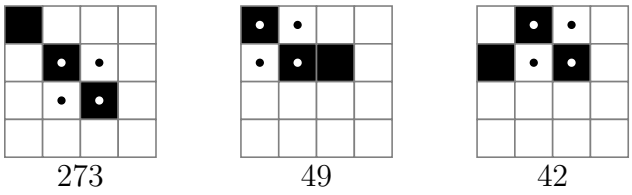
Analog auf alle Rominos angewendet ergibt sich:



Nun lassen sich trivialerweise die Duplikate eliminieren;



Über den gesamten Prozess hinweg wird auch die geschützte Diagonale mitverfolgt, bei den 3er-Rominos ist sie wie folgt plaziert:



2 Umsetzung

Zur Umsetzung haben wir den obigen Algorithmus in C# 8.0 mit .NET Core 3.0 implementiert.

Die Rominos werden in Form eines readonly structs *Romino* gespeichert. Das struct beinhaltet

1. *Vector2Int[] Blocks* - Das Array mit allen Blöcken des Rominos.
2. *List<Vector2Int> PossibleExtensions* - Die Liste mit allen Block-Additionsmöglichkeiten. Hierbei ist zu bemerken, dass die Größe der Liste konstant bleibt; es wird hier eine Liste statt einem Array verwendet, da bei der Erstellung die Größe unbekannt ist, und die Liste noch in ein Array zu konvertieren unnötig Rechenzeit kostet.

3. *Vector2Int DiagonalRoot* - Die linke obere Ecke der geschützten Diagonale.

4. *Vector2Int Max* - Die rechte untere Ecke des Rominos. Verwendet für korrigieren der Verschiebung ohne über alle Blöcke zu iterieren.

5. *BitBuffer512 _uniqueCode* - Der eindeutige Wert, errechnet wie in 1.3.2.

Die Hauptmethode ist die statische Methode *IEnumerable<(int Size, List<Romino> Rominos)> Romino.GetRominosUntilSize(int size)* welche für eine gegebene Größe alle Rominos aller Größen, bis zu dieser Größe ausgibt. Diese ruft intern parallelisiert für alle Rominos aus einer Generation die Methode *IEnumerable<Romino> Romino.AddOneNotUnique()* auf. Diese Methode errechnet nach dem Verfahren aus 1.2 die Rominos der nächsten Generation. Danach werden nach dem Verfahren aus 1.3 die Duplikate entfernt.

Die eindeutigen Werte aus 1.3.2 werden hierbei berechnet, ohne dass der Romino modifiziert wird, alle Modifikationen die an dem Romino gemacht werden müssten, um den Wert einer Permutation zu bestimmen, werden beim orientieren direkt in der Ausrechnung angewendet, ohne das Romino zu modifizieren. Erst wenn die eindeutige Rotation nach 1.3 gefunden wurde, wird das Romino so modifiziert, dass es als diese Permutation dargestellt wird.

3 Quellcode

3.1 *readonly struct Vector2Int*

Vector2Int ist ein 2-dimensionaler Vector von *System.Int32*.

3.2 *struct BitBuffer512*

BitBuffer512 hält 512 bits an Daten, wobei die individuellen Bits mit dem Indexer *BitBuffer512[int bitIndex]* gelesen und geschrieben werden können. Weiterdem überlädt *BitBuffer512* Vergleichsoperatoren, die 2 Instanzen wie eine 512 stellige unsignierte Binärzahlen vergleicht. Das struct wird zum speichern des eindeutigem Werts aus 1.3.2 verwendet, da der größte vorimplementierte Zahlentype, *ulong* bereits mit 8er-Rominos komplett gefüllt wird. Im Vergleich kann *BitBuffer512* Rominos von bis zu 22 Blöcken speichern.

3.3 readonly struct Romino

Romino ist das Hertzstück des Codes; es speichert ein Romino ab, mit den in 2 benannten Feldern. Dabei ist das gesamte struct readonly, und auch die Listen/Arrays werden, auch wenn dies nicht explizit versichert ist, nie modifiziert, nach dem der Konstruktor durchgelaufen ist. Der Konstruktor orientiert hier das Romino nach dem Verfahren aus 1.3.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
```

```
4  using System.Text;
5
6  public readonly struct Romino : IEquatable<Romino>, IComparable<Romino>
7  {
8      /// <summary>
9      /// <para>
10     /// All different combinations of rotating and mirroring an
           arbitrary romino.
11     /// </para>
12     /// <para>
13     /// BlockMap represents the functor mapping a block coordinate
           from the origin romino
14     ///   to the rotated/mirrored romino.
15     /// </para>
16     /// <para>
17     /// DiagonalRootMap represents the functor mapping the
           DiagonalRoot from the origin romino
18     ///   to the rotated/mirrored romino.
19     ///   Different from BlockMap because the DiagonalRoot is always
           the upper left of a square
20     ///   of 4 coords;
21     /// </para>
22     /// <para>    e.g. when mirroring along the y-Axis (x => (-x.X,
           x.Y)):
23     ///
24     ///     Before  After
25     ///       |       |
26     ///       |       |
27     /// </para>
28     /// </summary>
29     private static readonly (Func<Vector2Int, Vector2Int> BlockMap,
           Func<Vector2Int, Vector2Int> DiagonalRootMap)[] Maps = new
           (Func<Vector2Int, Vector2Int> BlockMap, Func<Vector2Int,
           Vector2Int> DiagonalRootMap)[]
30     {
31         (x => new Vector2Int(+x.X, +x.Y), x => new Vector2Int(+x.X,
               +x.Y)),
32         (x => new Vector2Int(+x.X, -x.Y), x => new Vector2Int(+x.X, ~
               x.Y)),
33         (x => new Vector2Int(-x.X, +x.Y), x => new Vector2Int(~x.X,
               +x.Y)),
34         (x => new Vector2Int(-x.X, -x.Y), x => new Vector2Int(~x.X, ~
               x.Y)),
35         (x => new Vector2Int(+x.Y, +x.X), x => new Vector2Int(+x.Y,
               +x.X)),
36         (x => new Vector2Int(+x.Y, -x.X), x => new Vector2Int(+x.Y, ~
               x.X)),
37         (x => new Vector2Int(-x.Y, +x.X), x => new Vector2Int(~x.Y,
```

```
            +x.X)),
38          (x => new Vector2Int(-x.Y, -x.X), x => new Vector2Int(~x.Y, ~
               x.X)),
39      };
40
41      /// <summary>
42      /// The smallest Romino possible
43      /// </summary>
44      public static Romino One =
45          new Romino(blocks: new[] { new Vector2Int(0, 0), new
               Vector2Int(1, 1) },
46              possibleExtensions:
47              // These are hardcoded in by hand, because this list is
                   only populated lazily by appending, rather than
                   computed once.
48              // As this first romino can not be computed like other
                   rominos, this won't be populated using normal methods.
49              new[] { new Vector2Int(-1, -1), new Vector2Int(0, -1), new
                   Vector2Int(1, -1),
50                      new Vector2Int(-1, 0),

                           new Vector2Int(2, 0),
51                      new Vector2Int(-1, 1),

                           new Vector2Int(2, 1),
52                                      new Vector2Int(0, 2),  new
                                           Vector2Int(1, 2),  new
                                           Vector2Int(2, 2), }
53                  .ToList(),
54              diagonalRoot: new Vector2Int(0, 0),
55              max: new Vector2Int(1, 1));
56
57      /// <summary>
58      /// All the Blocks composing the Romino.
59      /// </summary>
60      public readonly Vector2Int[] Blocks;
61
62      /// <summary>
63      /// All possible positions for adding new blocks.
64      /// </summary>
65      /// <remarks>
66      /// This is a list, yet the length is fixed.
67      /// Reason for this is, that at the point of creation, the size of
           this is not known,
68      /// and converting to an array after the size is known adds
           unnecessary overhead.
69      /// </remarks>
70      public readonly List<Vector2Int> PossibleExtensions;
```

```
71
72      /// <summary>
73      /// The upper left (lowest x, y) corner of the protected diagonal.
74      /// </summary>
75      public readonly Vector2Int DiagonalRoot;
76
77      /// <summary>
78      /// The highest x and y coordinates of any block inside the romino.
79      /// </summary>
80      public readonly Vector2Int Max;
81
82      /// <summary>
83      /// The unique code assigned to this romino.
84      /// </summary>
85      private readonly BitBuffer512 _uniqueCode;
86
87      /// <summary>
88      /// Gets all the blocks blocked by the protected diagonal.
89      /// </summary>
90      public readonly IEnumerable<Vector2Int> DiagonalRootBlockade
91      {
92          get
93          {
94              yield return DiagonalRoot + new Vector2Int(0, 0);
95              yield return DiagonalRoot + new Vector2Int(0, 1);
96              yield return DiagonalRoot + new Vector2Int(1, 0);
97              yield return DiagonalRoot + new Vector2Int(1, 1);
98          }
99      }
100
101     /// <summary>
102     /// Initializes and orients a new instance of the <see
        cref="Romino"/> structure.
103     /// </summary>
104     /// <param name="blocks">All the Blocks composing the
        Romino.</param>
105     /// <param name="possibleExtensions">All possible positions for
        adding new blocks.</param>
106     /// <param name="diagonalRoot">The upper left (lowest x, y) corner
        of the protected diagonal.</param>
107     /// <param name="max">The highest x and y coordinates of any block
        inside the romino.</param>
108     public Romino(Vector2Int[] blocks, List<Vector2Int>
        possibleExtensions, Vector2Int diagonalRoot, Vector2Int max)
109     {
110         Blocks = blocks;
111         DiagonalRoot = diagonalRoot;
112         PossibleExtensions = possibleExtensions;
```

```
113         Max = max;
114
115         _uniqueCode = default; // Needs to be assigned in order to
                call methods, including CalculateUniqueCode.
116         _uniqueCode = CalculateUniqueCode();
117
118         // Find highest unique Code.
119         // Start of with asserting the current permutation to be the
                one with the highest unique code.
120         int maxIndex = 0;
121         BitBuffer512 maxCode = _uniqueCode;
122
123         // Check against all other permutations, skipping 1, as thats
                already been calculated.
124         for (int i = 1; i < Maps.Length; i++)
125         {
126             var uniqueCode = CalculateUniqueCode(Maps[i].BlockMap);
127             if (maxCode < uniqueCode)
128             {
129                 maxIndex = i;
130                 maxCode = uniqueCode;
131             }
132         }
133
134         // Only make changes if the highest unique Code isn't the
                initial state
135         // (Maps[0] = (x => x, x => x))
136         if (maxIndex != 0)
137         {
138             (Func<Vector2Int, Vector2Int> blockMap, Func<Vector2Int,
                    Vector2Int> diagonalRootMap) = Maps[maxIndex];
139
140             var offset = CalculateOffset(blockMap);
141
142             for (int i = 0; i < Blocks.Length; i++) Blocks[i] =
                    blockMap(Blocks[i]) + offset;
143             for (int i = 0; i < PossibleExtensions.Count; i++)
                    PossibleExtensions[i] = blockMap(PossibleExtensions[i])
                    + offset;
144
145             DiagonalRoot = diagonalRootMap(DiagonalRoot) + offset;
146
147             // Don't add offset to max, it might end up with x or y
                    equal to 0.
148             var mappedMax = blockMap(Max);
149             // Take the absolute of both components, we only care
                    about swapping of x and y, not inversion.
150             Max = new Vector2Int(Math.Abs(mappedMax.X),
```

```
                Math.Abs(mappedMax.Y));
151
152             // Recalculate the unique code, as the currently saved one
                    is for Maps[0].
153             _uniqueCode = CalculateUniqueCode();
154         }
155     }
156
157     public static IEnumerable<(int Size, List<Romino> Rominos)>
            GetRominosUntilSize(int size)
158     {
159         // Validate arguments outside of iterator block, to prevent
                the exception being thrown lazily.
160         if (size < 2) throw new
                ArgumentOutOfRangeException(nameof(size));
161
162         return GetRominosUntilSizeInternal();
163
164         IEnumerable<(int Size, List<Romino> Rominos)>
                GetRominosUntilSizeInternal()
165         {
166             // Start out with the smalles romino
167             List<Romino> lastRominos = new List<Romino> { One };
168
169             // The size of the smallest Romino is 2 blocks; yield it
                    as such.
170             yield return (2, lastRominos);
171
172             for (int i = 3; i <= size; i++)
173             {
174                 var newRominos = lastRominos
175                     // Enable parallelization using PLINQ.
176                     .AsParallel()
177                     // Map every romino to all rominos generated by
                            adding one block to it.
178                     .SelectMany(x => x.AddOneNotUnique())
179                     // Remove duplicates, rominos are already oriented
                            here.
180                     .Distinct()
181                     // Execute Query by iterating into a list. Cheaper
                            than .ToArray()
182                     .ToList();
183
184                 // We don't need last generations rominos anymore.
                        Replace them with the new generation.
185                 lastRominos = newRominos;
186                 // Yield this generations rominos with their size.
187                 yield return (i, newRominos);
```

```
188             }
189         }
190     }
191 
192     // Generate IEnumerable<T> instead of allocing a new array
193     /// <summary>
194     /// Gets all direct neighbours of a given block, not including the
        block itself.
195     /// </summary>
196     /// <param name="block">The block to get the neighbours of</param>
197     /// <returns>An <see cref="IEnumerable{Vector2Int}"/> yielding all
        neighbours</returns>
198     private static IEnumerable<Vector2Int>
        GetDirectNeighbours(Vector2Int block)
199     {
200         yield return block + new Vector2Int(0, -1);
201         yield return block + new Vector2Int(0, 1);
202         yield return block + new Vector2Int(1, 0);
203         yield return block + new Vector2Int(1, -1);
204         yield return block + new Vector2Int(1, 1);
205         yield return block + new Vector2Int(-1, 0);
206         yield return block + new Vector2Int(-1, -1);
207         yield return block + new Vector2Int(-1, 1);
208     }
209 
210     /// <summary>
211     /// Returns all rominos generated by adding one block from <see
        cref="PossibleExtensions"/>
212     /// </summary>
213     /// <remarks>Does not remove duplicates, but orients
        results.</remarks>
214     /// <returns>All, non-unique rominos generated by adding one block
        from <see cref="PossibleExtensions"/>.</returns>
215     public readonly IEnumerable<Romino> AddOneNotUnique()
216     {
217         foreach (var newBlock in PossibleExtensions)
218         {
219             // If the new block has x or y smaller than 0, move the
                entire romino such that
220             // the lowest x and y are 0.
221             // This offset will need to be applied to anything inside
                the romino.
222             var offset = new Vector2Int(Math.Max(-newBlock.X, 0),
                Math.Max(-newBlock.Y, 0));
223 
224             // If the new block is outside of the old rominos bounds,
                i.e. has bigger x or y coords than Max,
225             // increase size.
```

```
226             var newSize = new Vector2Int(Math.Max(newBlock.X, Max.X),
                    Math.Max(newBlock.Y, Max.Y))
227                 // or if the new block has coordinates x or y smaller
                        than 0, increase size.
228                 + offset;
229
230             HashSet<Vector2Int> newPossibleExtensions =
231                 // Get the direct neighbours, i.e. the blocks that
                        will be possible spots
232                 // for adding blocks after newBlock has been added
233                 new HashSet<Vector2Int>(GetDirectNeighbours(newBlock +
                        offset));
234
235             // Remove already occupied positions
236             newPossibleExtensions.ExceptWith(Blocks.Select(x => x +
                    offset));
237             // Exclude positions blocked by the protected diagonal
238             newPossibleExtensions.ExceptWith(DiagonalRootBlockade.Select(x
                    => x + offset));
239
240             // Re-use old extension spots.
241             newPossibleExtensions.UnionWith(PossibleExtensions.Select(x
                    => x + offset));
242
243             // Remove the newly added block.
244             newPossibleExtensions.Remove(newBlock + offset);
245
246             // Allocate a new array for the new romino, with one more
                    space then right now
247             // to store the new block in.
248             Vector2Int[] newBlocks = new Vector2Int[Blocks.Length + 1];
249
250             for (int i = 0; i < Blocks.Length; i++)
251             {
252                 // Copy elements from current romino and apply offset.
253                 newBlocks[i] = Blocks[i] + offset;
254             }
255
256             // Insert the new block, also, with offset.
257             newBlocks[Blocks.Length] = newBlock + offset;
258
259             yield return new Romino(
260                 newBlocks,
261                 new List<Vector2Int>(newPossibleExtensions),
262                 // Apply offset to the diagonal root as well.
263                 DiagonalRoot + offset,
264                 newSize);
265         }
```

```
266     }
267
268     private readonly BitBuffer512 CalculateUniqueCode()
269     { /* CalculateUniqueCode(x => x) with the parameter inlined */ }
270
271     private readonly BitBuffer512 CalculateUniqueCode(Func<Vector2Int,
            Vector2Int> func)
272     {
273         var bits = new BitBuffer512();
274
275         // "Definitely very useful caching"
276         int length = Blocks.Length;
277
278         // Calculate the offset to be applied.
279         var offset = CalculateOffset(func);
280
281         for (int i = 0; i < Blocks.Length; i++)
282         {
283             // Map the block and apply the offset.
284             var mapped = func(Blocks[i]) + offset;
285
286             // Assign the relevant bit (2^((y * len) + x) = 1 << ((y *
                    len) + x))
287             bits[(mapped.Y * length) + mapped.X] = true;
288         }
289
290         return bits;
291     }
292
293     /// <summary>
294     /// Calculates the offset by which blocks inside the romino need
            to be moved after applying a given function
295     /// in order to still have the lowest x and y be equal to 0.
296     /// </summary>
297     /// <remarks>The function <paramref name="map"/> may not apply any
            translations, only
298     /// scaling and rotation around the origin (0, 0) is
            handled.</remarks>
299     /// <param name="map">The function to calculate the offset
            for.</param>
300     /// <returns>The offset that needs to be applied to set the
            minimum x and y coordinates after applying <paramref
            name="map"/> back to 0.</returns>
301     private readonly Vector2Int CalculateOffset(Func<Vector2Int,
            Vector2Int> map)
302     {
303         var mappedSize = map(Max);
304         // We only need to offset if the blocks are being moved into
```

```
            the negative,
305         // as translations from map are forbidden, and such the min
            will only change by
306         // mirroring around an axis or rotating.
307         return new Vector2Int(Math.Max(-mappedSize.X, 0),
            Math.Max(-mappedSize.Y, 0));
308     }
309
310     /// <remarks>Returns invalid results for comparisons between
        rominos of different sizes</remarks>
311     public override readonly bool Equals(object obj) => obj is Romino
        romino && Equals(romino);
312
313     public override readonly int GetHashCode() =>
        _uniqueCode.GetHashCode();
314
315     /// <remarks>Returns invalid results for comparisons between
        rominos of different sizes</remarks>
316     public readonly bool Equals(Romino romino) => _uniqueCode ==
        romino._uniqueCode;
317
318     /// <remarks>Returns invalid results for comparisons between
        rominos of different sizes</remarks>
319     public readonly int CompareTo(Romino other) =>
        _uniqueCode.CompareTo(other._uniqueCode);
320 }
```