

Aufgabe 5: Rominos

Team-ID: 00587

Team-Name: Doge.NET

Bearbeiter dieser Aufgabe:
Nikolas Kilian & Johannes von Stoephasius

22. November 2019

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Definitionen	1
1.2	Kernidee	2
1.2.1	Beispiel	2
1.3	Hinzufügen von Blöcken	2
1.3.1	Beispiel	2
1.4	Eliminierung von Duplikaten	3
1.4.1	Verschiebung	3
1.4.2	Rotation und Spiegelung	3
1.4.3	Endgültige Duplikat-Eliminierung	4
1.4.4	Beispiel	5
2	Umsetzung	6
2.1	Vector2Int	7
2.2	BitBuffer512	7
2.3	Romino	7
3	Beispiele	7
3.1	Rominos of size 2 (1)	7
3.2	Rominos of size 3 (3)	7
3.3	Rominos of size 4 (17)	8
3.4	Rominos of size 5 (82)	8
4	Quellcode	11

1 Lösungsidee

1.1 Definitionen

Romino Ein Romino ist eine Ansammlung von Blöcken auf einem 2d Integer-Gitter.

Block Ein Block ist eine Position auf dem 2d Gitter, die einem Romino angehört.

(echte) Diagonale Die (echte) Diagonale ist die in der Aufgabenstellung beeschriebene diagonale Verbindung zweier Blöcke mit zwei fehlenden Blöcken.

Diagonalenblockade Die Diagonalenblockade sind die Blöcke die für das Schützen der echten Diagonale von Blocksadditionsmöglichkeit ausgeschlossen sind.

Blocksadditionsmöglichkeit Eine Blocksadditionsmöglichkeit eines Rominos ist eine Position im 2d-Gitter, wo in zukünftigen Schritten ein neuer Block angefügt werden kann.

In den Grafiken als grauer Kreis dargestellt.

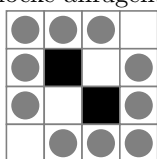
1.2 Kernidee

Rominos mit n Blöcken können gefunden werden, indem zu Rominos mit $n - 1$ Blöcken ein Block angefügt wird. Hierbei muss beachtet werden, dass der Rominostein zusammenhängend bleiben muss und dass mindestens eine Diagonale bleiben muss.

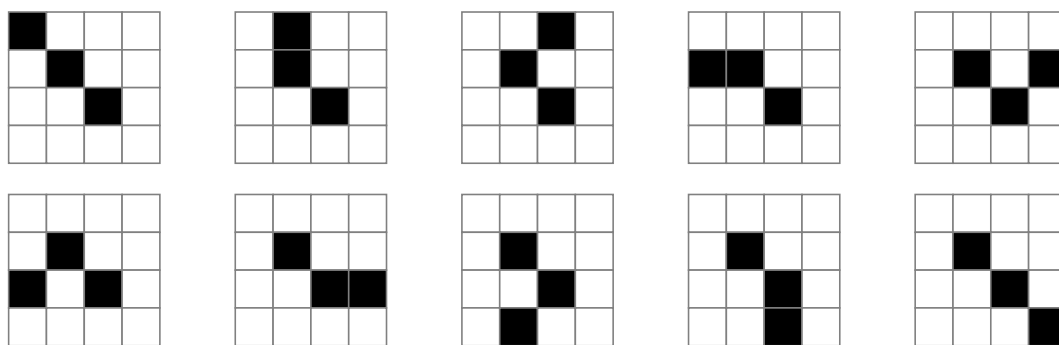
Um alle möglichen Rominos mit n Blöcken zu finden, muss man also alle Rominos mit $n - 1$ Blöcken finden, und für diese alle Rominos die durch hinzufügen eines weiteren Blocks entstehen können ermitteln. Dabei wird es Duplikate geben. Eliminiert man diese hat man alle möglichen n -Rominos eindeutig gefunden.

1.2.1 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.



Kleinstes Rominostein

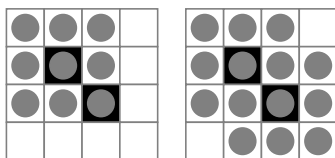
Um alle n -Rominos für ein beliebiges n zu finden, würde man den obigen Algorithmus verwenden um aus dem 2er-Romino alle 3-Rominos zu folgern, dann aus diesen alle 4-Rominos etc., bis man alle n -Rominos errechnet hat.

1.3 Hinzufügen von Blöcken

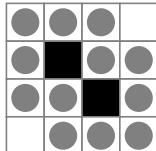
Bei Rominos werden konstant die Blocksadditionsmöglichkeit verfolgt, also die Positionen, wo Blöcke angefügt werden können, sodass das Romino zusammenhängend bleibt. Hierfür werden die Nachbarn jedes Blocks des Rominos ermittelt, daraufhin werden Duplikate und bereits belegte Blöcke eliminiert.

1.3.1 Beispiel

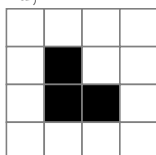
Nehme man beispielsweise wieder das 2er-Romino, würden die Nachbarn aller Blöcke wie folgt ermittelt werden:



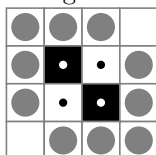
Entfernen bereits existierender Blöcke von den Blocksadditionsmöglichkeit



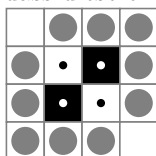
Es lässt sich hier erkennen, dass die Existenz einer echten Diagonale nicht zwingend aufrecht erhalten wird;



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



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



1.4 Eliminierung von Duplikaten

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

1.4.1 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 $y=0$.

1.4.2 Rotation und Spiegelung

Um Rotation und Spiegelung eines Rominos zu eliminieren, werden zuerst alle seine Permutationen (also alle Kombinationen von Rotation und Spiegelung) ermittelt, denen jeweils ein eindeutiger Wert zugewiesen wird. 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

Es wird quasi der Block als Maske für folgende Werte verwendet:

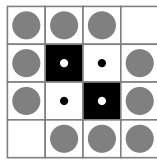
2^0	2^1	2^2
2^3	2^4	2^5
2^6	2^7	2^8

Abbildung 1: Faktoren

1.4.3 Endgültige Duplikat-Eliminierung

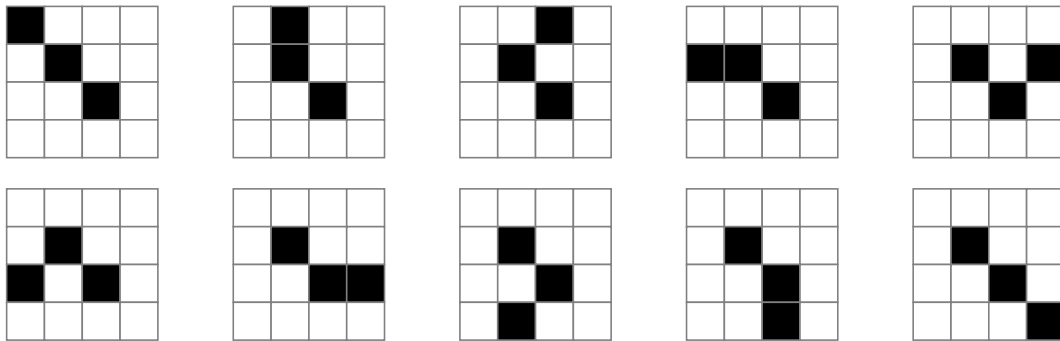
Zum endgültigen eliminieren der Duplikate werden zuerst alle Rominos wie oben beschrieben orientiert, dann werden die eindeutigen Werte dieser Rominos verglichen, um schnell Gleichheit zu ermitteln. Durch Verwendung dieser Vergleichsmethode lassen sich schnell Duplikate entfernen.

1.4.4 Beispiel

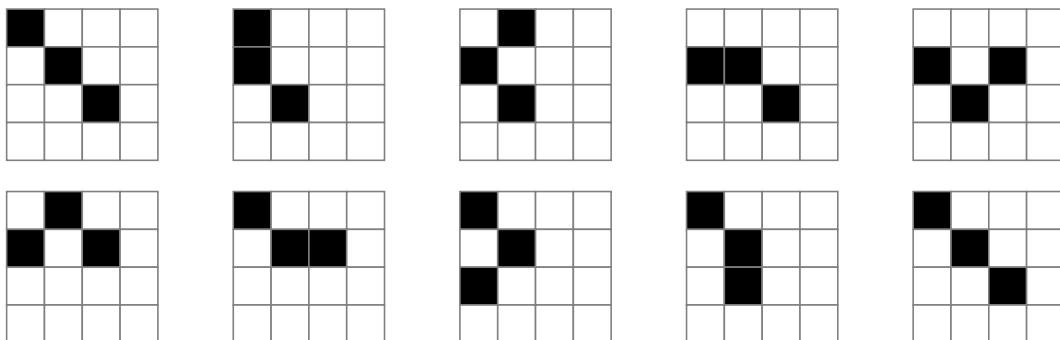


Ausgangsromino

Ermittlung möglicher Rominos

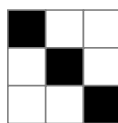


Eliminierung von Verschiebungen

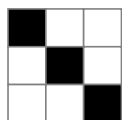


Eliminierung von Rotation und Spiegelung

Ausgehend von dem Romino;

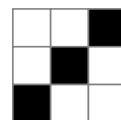


werden folgende Permutationen festgestellt:



Permutation 1

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

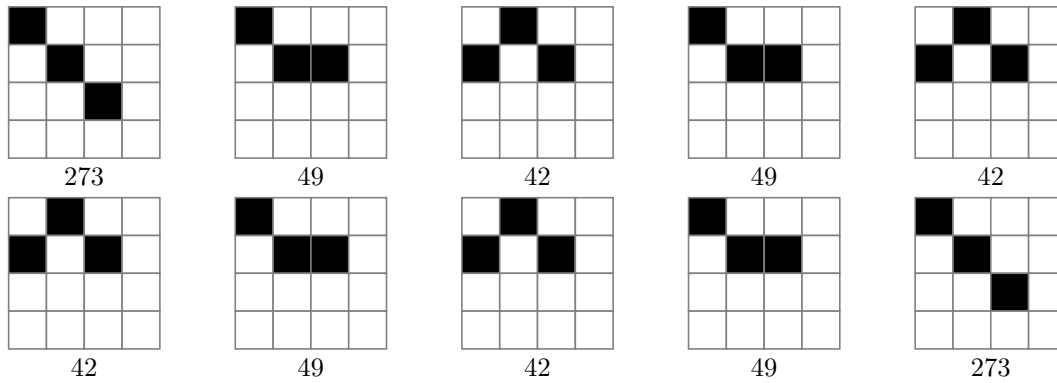


Permutation 2

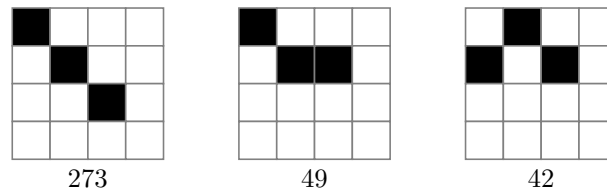
$$Wert_2 = 2^2 + 2^4 + 2^6 = 84$$

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.

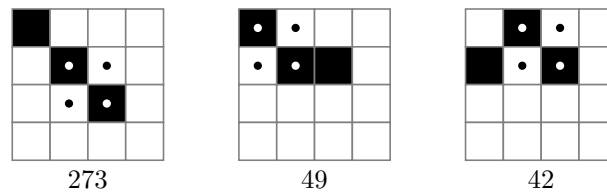
Analog auf alle Rominos angewendet ergibt sich:



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 platziert:



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 struct 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 Konvertierung zum Array unnötig Rechenzeit kosten würde.
3. `Vector2Int DiagonalRoot` - Die linke obere Ecke der geschützten Diagonale.
4. `Vector2Int Max` - Die rechte untere Ecke des Rominos. Verwendet für die Korrektur der Verschiebung ohne über alle Blöcke zu iterieren.
5. `BitBuffer512 _uniqueCode` - Der eindeutige Wert, errechnet wie in Abschnitt 1.4.2.

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 Abschnitt 1.3 die Rominos der nächsten Generation. Danach werden nach dem Verfahren aus Abschnitt 1.4 die Duplikate entfernt.

Die eindeutigen Werte aus Abschnitt 1.4.2 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 des eindeutigen Werts angewendet, ohne das Romino zu modifizieren. Erst wenn die eindeutige Rotation nach Abschnitt 1.4 gefunden wurde, wird das Romino so modifiziert, dass es als diese Permutation dargestellt wird.

2.1 Vector2Int

`readonly struct` `Vector2Int` ist ein 2-dimensionaler `Vector` von `System.Int32`.

2.2 BitBuffer512

`struct` `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 unsigned Binärzahlen vergleicht. Das `struct` wird zum speichern des eindeutigen Werts aus Abschnitt 1.4.2 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.

2.3 Romino

`readonly struct` `Romino` ist das Her(t)zstück des Codes; es speichert ein Romino ab, mit den in Abschnitt 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 Abschnitt 1.4.

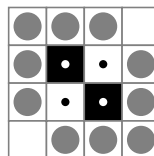
3 Beispiele

```
--size 10 --stopwatch
```

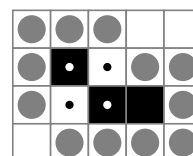
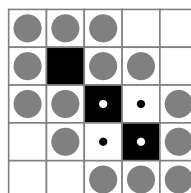
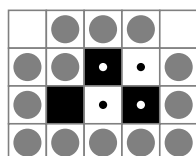
```
Starting
2 done - 1
3 done - 3
4 done - 17
5 done - 82
6 done - 489
7 done - 2924
8 done - 18406
9 done - 116883
10 done - 753905
Done 9370ms
```

```
--size 5 --latex --highlightPossibleExtensions --highlightDiagonalBlockade
```

3.1 Rominos of size 2 (1)



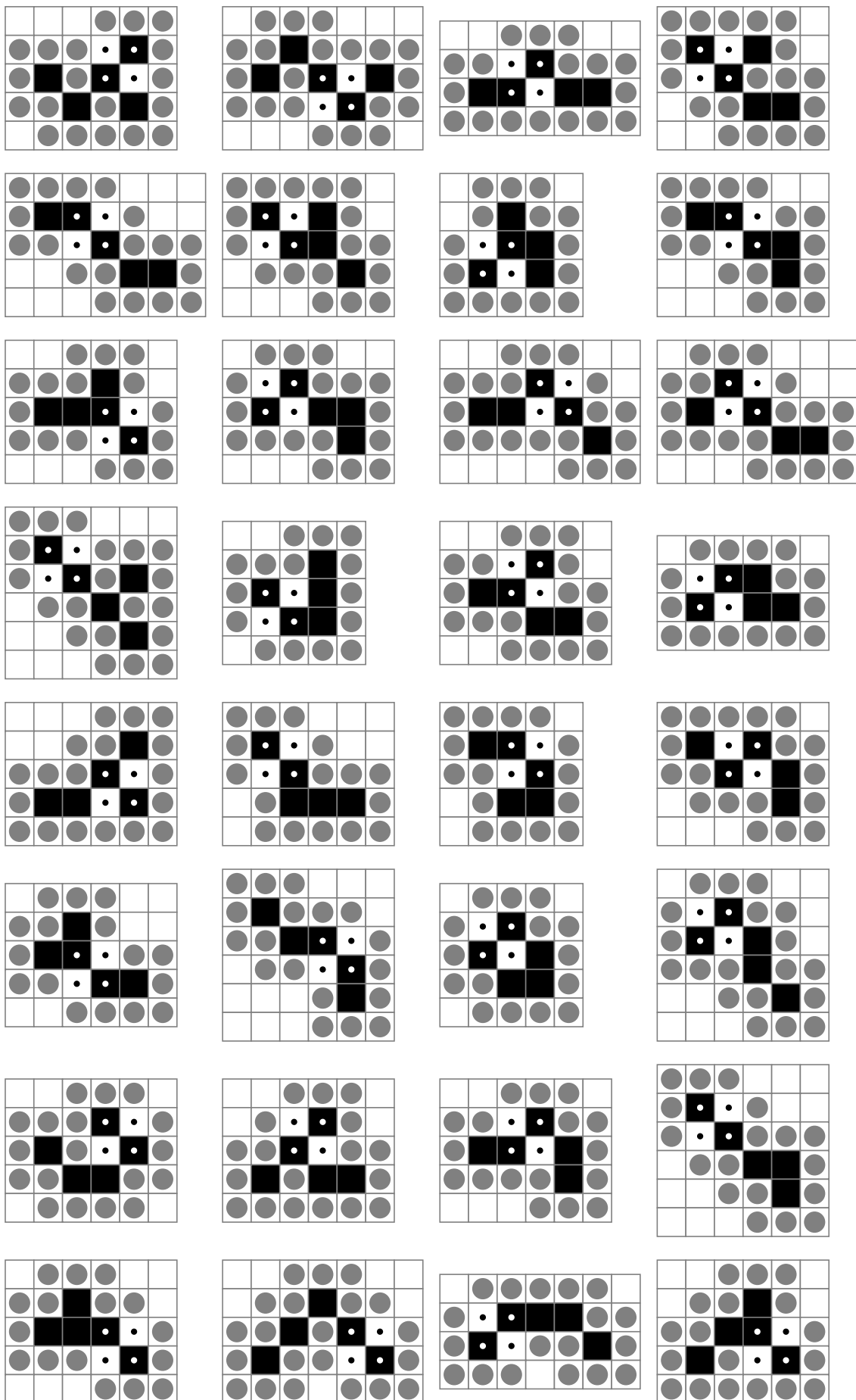
3.2 Rominos of size 3 (3)

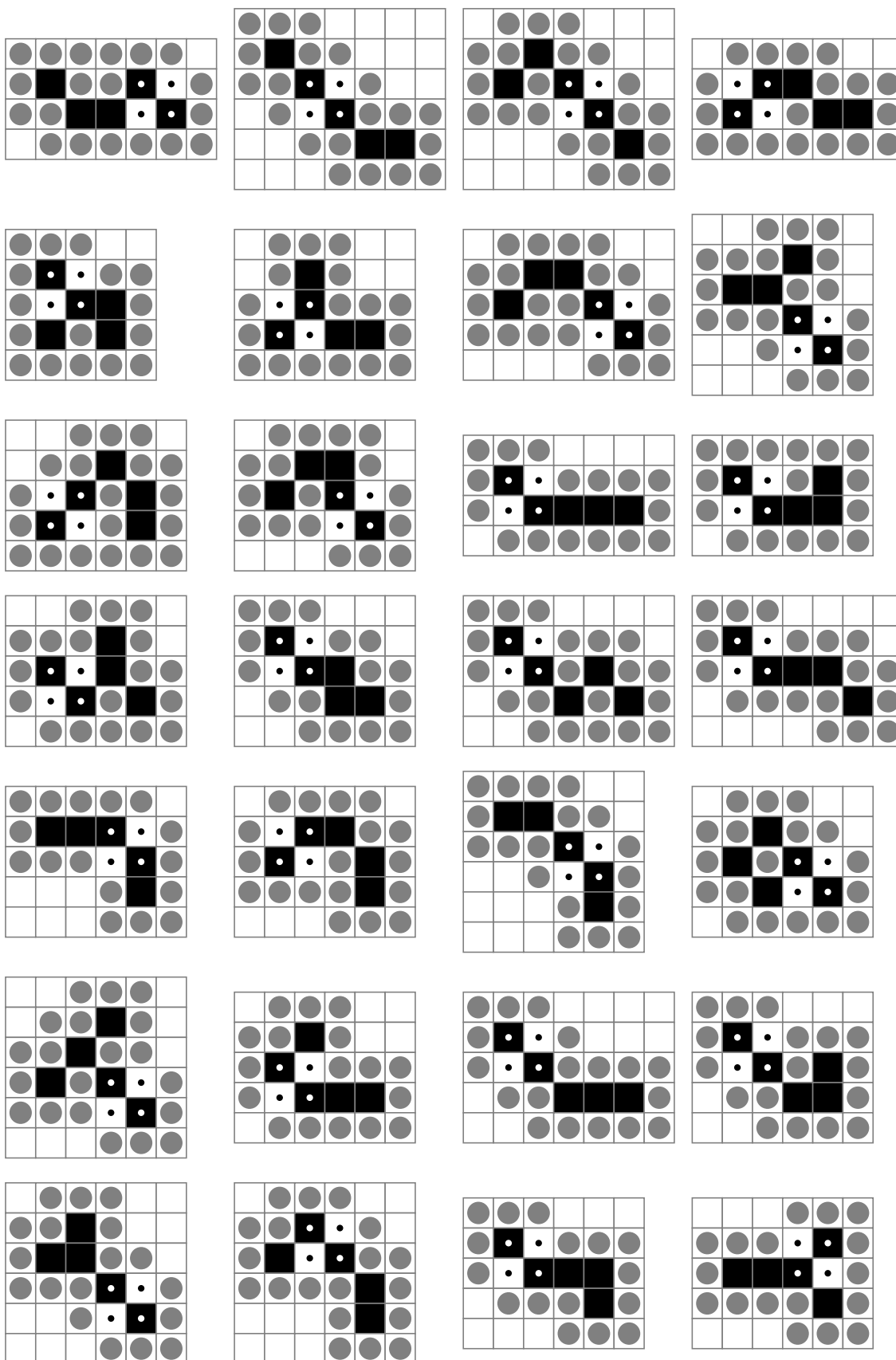


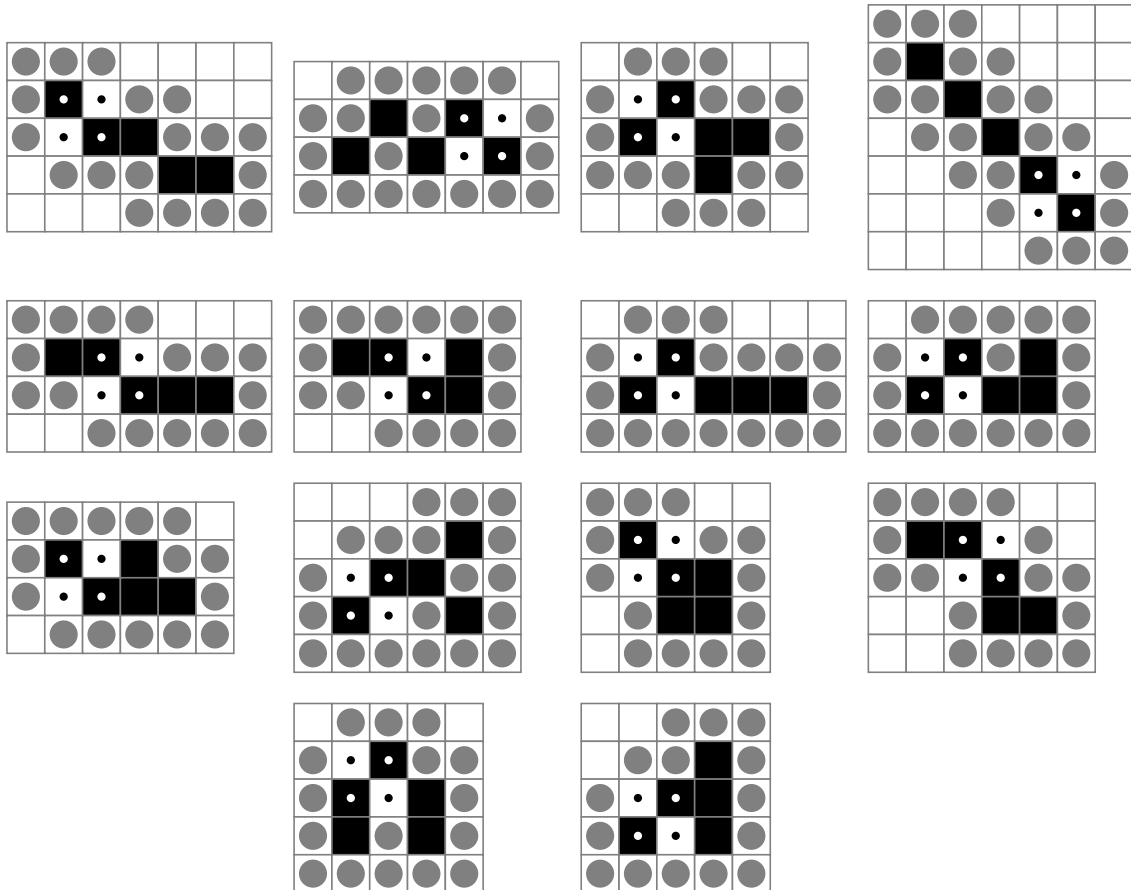
The image displays a 4x4 grid of 16 6x6 matrices. Each matrix contains a unique pattern of black squares, white squares with dots, and gray circles. The patterns are as follows:

- Matrix 1 (top-left):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 2 (top-second):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 3 (top-third):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 4 (top-fourth):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 5 (second-row first):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 6 (second-row second):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 7 (second-row third):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 8 (second-row fourth):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 9 (third-row first):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 10 (third-row second):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 11 (third-row third):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).
- Matrix 12 (third-row fourth):** Black squares at (2,2), (3,3), (4,4), (5,5), (6,6). White squares with dots at (2,3), (3,4), (4,5), (5,6). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5).
- Matrix 13 (bottom row):** Black squares at (2,3), (3,2), (3,4), (4,3), (4,4), (5,3), (5,4). White squares with dots at (2,2), (3,3), (4,2), (5,2). Gray circles at (1,2), (1,3), (1,4), (2,1), (2,4), (2,5), (2,6), (3,1), (3,5), (3,6), (4,1), (4,5), (4,6), (5,1), (5,5), (5,6), (6,2), (6,3), (6,4), (6,5), (6,6).

Figure 1 displays eight 10x10 grids, each containing a unique pattern of black, white, and gray cells. The patterns are arranged in two rows of four. Each grid represents a different state or configuration in a cellular automaton or similar system.







4 Quellcode

```

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
18    /// romino
19    /// to the rotated/mirrored romino.
20    /// Different from BlockMap because the DiagonalRoot is always the upper left of a
21    /// square
22    /// of 4 coords;
23    /// </para>
24    /// <para> e.g. when mirroring along the y-Axis (x => (-x.X, x.Y)):
25    ///
26    /// 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
48              // by appending, rather than computed once.
49              // As this first romino can not be computed like other rominos, this won't be
50              // populated using normal methods.
51              new[] { new Vector2Int(-1, -1), new Vector2Int(0, -1), new Vector2Int(1, -1),
52                  new Vector2Int(-1, 0), new Vector2Int(2, 0),
53                  new Vector2Int(-1, 1), new Vector2Int(2, 1),
54                  new Vector2Int(0, 2), new Vector2Int(1, 2), new
55                      Vector2Int(2, 2), }
56          .ToList(),
57          diagonalRoot: new Vector2Int(0, 0),
58          max: new Vector2Int(1, 1));
59
60  /// <summary>
61  /// All the Blocks composing the Romino.
62  /// </summary>
63  public readonly Vector2Int[] Blocks;
64
65  /// <summary>
66  /// All possible positions for adding new blocks.
67  /// </summary>
68  /// <remarks>
69  /// This is a list, yet the length is fixed.
70  /// Reason for this is, that at the point of creation, the size of this is not known,
71  /// and converting to an array after the size is known adds unnecessary overhead.
72  /// </remarks>
73  public readonly List<Vector2Int> PossibleExtensions;
74
75  /// <summary>
76  /// The upper left (lowest x, y) corner of the protected diagonal.
77  /// </summary>
78  public readonly Vector2Int DiagonalRoot;
79
80  /// <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
106     /// blocks.</param>
107     /// <param name="diagonalRoot">The upper left (lowest x, y) corner of the protected
108     /// diagonal.</param>
109     /// <param name="max">The highest x and y coordinates of any block inside the
110     /// romino.</param>
111     public Romino(Vector2Int[] blocks, List<Vector2Int> possibleExtensions, Vector2Int
112     diagonalRoot, Vector2Int max)
113     {
114         Blocks = blocks;
115         DiagonalRoot = diagonalRoot;
116         PossibleExtensions = possibleExtensions;
117         Max = max;
118
119         _uniqueCode = default; // Needs to be assigned in order to call methods, including
120         CalculateUniqueCode.
121         _uniqueCode = CalculateUniqueCode();
122
123         // Find highest unique Code.
124         // Start of with asserting the current permutation to be the one with the highest
125         unique code.
126         int maxIndex = 0;
127         BitBuffer512 maxCode = _uniqueCode;
128
129         // Check against all other permutations, skipping 1, as thats already been
130         calculated.
131         for (int i = 1; i < Maps.Length; i++)
132         {
133             var uniqueCode = CalculateUniqueCode(Maps[i].BlockMap);
134             if (maxCode < uniqueCode)
135             {

```

```

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 smallest 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
185         // new generation.
186         lastRominos = newRominos;
187         // Yield this generations rominos with their size.
188         yield return (i, newRominos);
189     }
190 }
191
192 // Generate IEnumerable<T> instead of allocating 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
212     cref="PossibleExtensions"/>
213 /// </summary>
214 /// <remarks>Does not remove duplicates, but orients results.</remarks>
215 /// <returns>All, non-unique rominos generated by adding one block from <see
216     cref="PossibleExtensions"/>.</returns>
217 public readonly IEnumerable<Romino> AddOneNotUnique()
218 {
219     foreach (var newBlock in PossibleExtensions)
220     {
221         // If the new block has x or y smaller than 0, move the entire romino such that
222         // the lowest x and y are 0.
223         // This offset will need to be applied to anything inside the romino.
224         var offset = new Vector2Int(Math.Max(-newBlock.X, 0), Math.Max(-newBlock.Y, 0));
225
226         // If the new block is outside of the old rominos bounds, i.e. has bigger x or
227         // y coords than Max,
228         // increase size.
229         var newSize = new Vector2Int(Math.Max(newBlock.X, Max.X), Math.Max(newBlock.Y,
230             Max.Y));
231         // or if the new block has coordinates x or y smaller than 0, increase size.
232         + offset;
233
234         HashSet<Vector2Int> newPossibleExtensions =
235             // Get the direct neighbours, i.e. the blocks that will be possible spots
236             // for adding blocks after newBlock has been added
237             new HashSet<Vector2Int>(GetDirectNeighbours(newBlock + offset));
238     }
239 }

```

```

234     // Remove already occupied positions
235     newPossibleExtensions.ExceptWith(Blocks.Select(x => x + offset));
236     // Exclude positions blocked by the protected diagonal
237     newPossibleExtensions.ExceptWith(DiagonalRootBlockade.Select(x => x + offset));
238
239     // Re-use old extension spots.
240     newPossibleExtensions.UnionWith(PossibleExtensions.Select(x => x + offset));
241
242     // Remove the newly added block.
243     newPossibleExtensions.Remove(newBlock + offset);
244
245     // Allocate a new array for the new romino, with one more space then right now
246     // to store the new block in.
247     Vector2Int[] newBlocks = new Vector2Int[Blocks.Length + 1];
248
249     for (int i = 0; i < Blocks.Length; i++)
250     {
251         // Copy elements from current romino and apply offset.
252         newBlocks[i] = Blocks[i] + offset;
253     }
254
255     // Insert the new block, also, with offset.
256     newBlocks[Blocks.Length] = newBlock + offset;
257
258     yield return new Romino(
259         newBlocks,
260         new List<Vector2Int>(newPossibleExtensions),
261         // Apply offset to the diagonal root as well.
262         DiagonalRoot + offset,
263         newSize);
264 }
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 ( $2^{((y * len) + x)} = 1 \ll ((y * len) + x)$ )
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
295    /// applying a given function
296    /// in order to still have the lowest x and y be equal to 0.
297    /// </summary>
298    /// <remarks>The function <paramref name="map"/> may not apply any translations, only
299    /// scaling and rotation around the origin (0, 0) is handled.</remarks>
300    /// <param name="map">The function to calculate the offset for.</param>
301    /// <returns>The offset that needs to be applied to set the minimum x and y
302    /// coordinates after applying <paramref name="map"/> back to 0.</returns>
303    private readonly Vector2Int CalculateOffset(Func<Vector2Int, Vector2Int> map)
304    {
305        var mappedSize = map(Max);
306        // We only need to offset if the blocks are being moved into the negative,
307        // as translations from map are forbidden, and such the min will only change by
308        // mirroring around an axis or rotating.
309        return new Vector2Int(Math.Max(-mappedSize.X, 0), Math.Max(-mappedSize.Y, 0));
310    }
311
312    /// <remarks>Returns invalid results for comparisons between rominos of different
313    /// sizes</remarks>
314    public override readonly bool Equals(object obj) => obj is Romino romino &&
315        Equals(romino);
316
317    public override readonly int GetHashCode() => _uniqueCode.GetHashCode();
318
319    /// <remarks>Returns invalid results for comparisons between rominos of different
320    /// sizes</remarks>
321    public readonly bool Equals(Romino romino) => _uniqueCode == romino._uniqueCode;
322
323    /// <remarks>Returns invalid results for comparisons between rominos of different
324    /// sizes</remarks>
325    public readonly int CompareTo(Romino other) =>
326        _uniqueCode.CompareTo(other._uniqueCode);
327 }
```