```
1
2
3  public class LatinSquare
4  {
5      public int genericCounter;
6      public DateTime startTime;
7      public StreamWriter writer;
8      List<int[][]>? validSquare;
9
10     public LatinSquare()
11     {
12
13         //I could have done this with a loop but I think it's nice to look
               at
14         using (writer = new StreamWriter(File.Create("n=3.txt"))) Part_1
             (3);
15         using (writer = new StreamWriter(File.Create("n=4.txt"))) Part_1
             (4);
16         using (writer = new StreamWriter(File.Create("n=5.txt"))) Part_1
             (5);
17         using (writer = new StreamWriter(File.Create("n=6.txt"))) Part_1
             (6);
18         using (writer = new StreamWriter(File.Create("P2_n=9.txt")))
             Part_2(9);
19         using (writer = new StreamWriter(File.Create("P2_n=10.txt")))
             Part_2(10);
20         using (writer = new StreamWriter(File.Create("P2_n=11.txt")))
             Part_2(11);
21         using (writer = new StreamWriter(File.Create("P2_n=12.txt")))
             Part_2(12);
22         using (writer = new StreamWriter(File.Create("P2_n=13.txt")))
             Part_2(13);
23         using (writer = new StreamWriter(File.Create("P2_n=14.txt")))
             Part_2(14);
24         using (writer = new StreamWriter(File.Create("P2_n=15.txt")))
             Part_2(15);
25
26     }
27
28
29     #region PART 1 SPECIFIC
30     public void Part_1(int size)
31     {
32         writer.WriteLine("Running Algorithm with size " + size);
33         writer.WriteLine();
34
35         //genericCounter is used for running the program on n=7 and above,
               so you know how far in you are. It's referenced right after a
             valid square is found
```

```
36          genericCounter = 1;
37          validSquare = new List<int[][]>();
38
39          //For time tracking purposes. I couldn't get a way to record CPU
              time per run, but I proved some screenshots of executions in the
               accompanying document.
40          //If you want to get the specifics (and if you're using visual
              studio) then commenting out all the undesired part 1/all part2
              lines and hitting alt + f2 will give a value eventually.
41          startTime = DateTime.Now;
42
43          //Starts off the program
44          Backtrack(EmptySquare(size), 1, 1);
45
46          //Cleanup
47          writer.WriteLine("Process finished in: " + (DateTime.Now -
              startTime).TotalMilliseconds + " ms");
48          writer.WriteLine("Number of valid squares: " + validSquare.Count);
49          writer.WriteLine("Examples: ");
50          writer.WriteLine();
51
52          //printing squares
53          int i, max = 4;
54          if (validSquare.Count < 4 || size == 5) max = validSquare.Count;
55          for (i = 0; i < max; i++)
56          {
57              PrintSquare(validSquare[i]);
58          }
59
60      }
61
62      //It permutes the square by sorting one of the rows. Then, it sorts it
           so that the leftmost column is in increasing order.
63      //Elaborated upon in the doc.
64      public int[][] ValidationSort(int[][] square, int row)
65      {
66          int size = square.Length;
67          int[][] column_sorted = EmptySquare(square.Length);
68          int[][] full_sorted = EmptySquare(square.Length);
69          int i, j, k;
70
71          //Sorting rows such that the "row"-th row is in increasing order
72          for (i = 0; i < size; i++)
73          {
74              j = square[i][row] - 1;
75              column_sorted[j] = square[i];
76          }
77
78          //Sorting by rows so that the leftmost column is in increasing
```

```
            order
79          for (i = 0; i < size; i++)
80          {
81              k = column_sorted[0][i] - 1;
82              for (j = 0; j < size; j++)
83              {
84                  full_sorted[j][k] = column_sorted[j][i];
85              }
86          }
87          return full_sorted;
88      }
89
90      //The backtracking algorithm for the assignment.
91      public void Backtrack(int[][] square, int x, int y)
92      {
93          int i, tx = x, ty = y;
94          //this iterator is for values that actually go into the table, so
            i is one of [1...n]
95          for (i = 1; i <= square.Length; i++)
96          {
97              //if value i is valid at x,y then it is placed there and a new
                branch is made
98              if (PositionIsValid(square, x, y, i))
99              {
100                 //checks to see if the square is finished, or if the x and
                    y coordinates are at their max.
101                 square[x][y] = i;
102                 if (x == y && x == square.Length - 1)
103                 {
104
105                     //check against old squares using the sorting method,
                    then adds the square if it's inequivalent.
106                     if (!SquareExists(square))
107                     {
108                         validSquare.Add(CopySquare(square));
109                         //this is a means to keep track of how many
                    inequivalent squares have been found so far in batches
                    of 1000.
110                         if (validSquare.Count / 1000 == genericCounter)
111                         {
112                             writer.WriteLine(genericCounter + "000 squares
                     so far");
113                             writer.WriteLine();
114                             genericCounter++;
115                         }
116                     }
117                 }
118                 else //if the square's not done
119                 {
```

```
120                     //making sure the y value doesn't wrap around
121                     if (y + 1 == square.Length)
122                     {
123                         ty = 0;
124                         tx += 1;
125 
126                     }
127                     Backtrack(square, tx, ty + 1);
128                 }
129                 //voids the previously placed square so the process can repeat for every possible i
130                 square[x][y] = 0;
131             }
132         }
133     }
134 
135     //Checking to make sure if the square exists. Elaborated upon in the doc.
136     public bool SquareExists(int[][] square)
137     {
138         int i;
139         int[][] temp;
140         List<int[][]> permutations = new List<int[][]>();
141         //This ensures that any new square is checked with all of its possible transformations.
142         for (i = 0; i < square.Length; i++)
143         {
144             temp = ValidationSort(square, i);
145             permutations.Add(temp);
146         }
147         //Checking if any permutation of the inequivalency candidate exists elsewhere.
148         foreach (int[][] p in validSquare)
149         {
150             foreach (int[][] q in permutations)
151             {
152                 if (IsEquivalent(p, q)) return true;
153             }
154         }
155         return false;
156     }
157 
158     //Checks the equivalency of two squares. Elaborated upon in the doc.
159     public bool IsEquivalent(int[][] squareOne, int[][] squareTwo)
160     {
161         int i, j;
162         for (i = 1; i < squareOne.Length - 1; i++)
163         {
164             for (j = 1; j < squareOne.Length - 1; j++)
```

```
165             {
166                 if (squareOne[i][j] != squareTwo[i][j]) return false;
167             }
168         }
169 
170 
171         return true;
172     }
173     #endregion
174 
175     #region PART 2 SPECIFIC
176     public void Part_2(int size)
177     {
178         //used doubles for these values as they can get absolutely massive
               with n = [9..16]
179         double nodeSum = 0d, nodeAverage;
180         int[] nodesPerLevel;
181         //arbitrary, but > 5
182         int runCount = 10;
183         //runs the test x times,, then calculates the average and prints
              it
184         for (int i = 0; i < runCount; i++)
185         {
186             nodesPerLevel = BacktrackEst(EmptySquare(size), 1, 1, 0, new
                  int[size * size]);
187             nodeSum += NodeScale(nodesPerLevel);
188         }
189 
190         nodeAverage = nodeSum / runCount;
191         writer.WriteLine("Average over " + runCount + " runs: " +
              nodeAverage);
192     }
193 
194     //NodeScale takes in an array of nodes by depth, then prints out what
          I'm meant to submit. Returns the full amount of estimated nodes in
          the tree.
195     public double NodeScale(int[] estData)
196     {
197         writer.WriteLine("Depth:\t" + "# of Nodes:\t" + "Total # of
              estimated nodes:");
198         int i = 0;
199         double nodeCount = 1d;
200         while (estData[i] != 0)
201         {
202             nodeCount *= estData[i];
203             writer.WriteLine(i + "\t" + estData[i] + "\t\t" + nodeCount);
204             i++;
205         }
206         writer.WriteLine("\nTotal: " + nodeCount + "\n");
```

```
207
208         return nodeCount;
209     }
210
211     //The bactrack estimation function. Returns an array carrying the depth/node values
212     //It works similar to the backtrack algorithm in part 1.
213     public int[] BacktrackEst(int[][] square, int x, int y, int currentDepth, int[] nodesPerLevel)
214     {
215         Random r = new Random(DateTime.Now.Millisecond);
216         int i, placedValue;
217         int[] test = nodesPerLevel;
218         List<int> validSelections = new List<int>();
219         //if it actually managed to make a valid square, return early to prevent issues.
220         if (x == y && x == square.Length - 1) return nodesPerLevel;
221         //getting the possible moves
222         for (i = 1; i <= square.Length; i++)
223         {
224             {
225                 if (PositionIsValid(square, x, y, i)) validSelections.Add(i);
226                 test[currentDepth]++;
227             }
228
229         }
230         nodesPerLevel[currentDepth] = validSelections.Count;
231         //if there are no valid options, dodge.
232         if (validSelections.Count == 0) return test;
233
234         //random selection, then continuation. Since it's meant to be an estimation algorithm based on the one we used for part 1, I redid the same style of picking the next cell "in line".
235         //I could have run it on every possible placement beyond the x/y I already have (like putting down a 4 in the bottom right corner) but this reflects my actual implementation.
236         placedValue = validSelections[r.Next(0, validSelections.Count)];
237         square[x][y] = placedValue;
238         if (y + 1 == square.Length)
239         {
240             y = 0;
241             x += 1;
242         }
243         //recurse
244         return BacktrackEst(square, x, y + 1, currentDepth + 1, test);
245     }
246
247     #endregion
```

```
248
249     #region GENERAL UTILITY
250     public void PrintSquare(int[][] square)
251     {
252         int i, j;
253         for (i = 0; i < square.Length; i++)
254         {
255             for (j = 0; j < square.Length; j++)
256             {
257                 writer.Write(square[j][i]);
258             }
259             writer.WriteLine();
260         }
261         writer.WriteLine();
262     }
263
264     //Empty is a vestigial name, it creates a new 2d array with the top
          row and left column in natural order. Elaborated upon in the doc.
265     public int[][] EmptySquare(int size)
266     {
267         int[][] nSquare = new int[size][];
268         int i;
269         for (i = 0; i < size; i++)
270         {
271             nSquare[i] = new int[size];
272         }
273         for (i = 0; i < size; i++)
274         {
275             nSquare[0][i] = i + 1;
276             nSquare[i][0] = i + 1;
277         }
278
279         return nSquare;
280     }
281
282     public int[][] CopySquare(int[][] square)
283     {
284         int i, j;
285         int[][] temp = EmptySquare(square.Length);
286         for (i = 1; i < square.Length; i++)
287         {
288             for (j = 1; j < square.Length; j++)
289             {
290                 temp[i][j] = square[i][j];
291             }
292         }
293         return temp;
294     }
295
```

```
296     //checks if a given placement with a given value is valid. Searches a
          row or column
297     public bool PositionIsValid(int[][] square, int x, int y, int n)
298     {
299         int i;
300         for (i = 0; i < square[x].Length; i++)
301         {
302             if (square[i][y] == n) return false; //searching across
303         }
304
305         if (square[x].Contains(n)) return false; //searching down
306         return true;
307     }
308
309     #endregion
310
311     public static void Main(string[] args)
312     {
313         LatinSquare l = new LatinSquare();
314     }
315 }
```