Combinatorics of Rotationally Symmetric Grid Regions

Noah Morris

March 8, 2023

1 INTRO

1.1 INSPIRATION

Sudoku was invented by Howard Garns, a modest architect from Indiana who died a decade before anyone attached his name to it. His coworkers, in interviews conducted after his death, noted that he would create obtuse number crosswords based on nines on the company's drawing boards, seemingly as a game. They only caught brief glimpses of it; he usually tried to hide it from people. He sent some in to a local magazine anonymously, *Dell Pencil Puzzles and Word Games*, but sought neither credit nor profit in his lifetime[1].

The puzzle became popular in Japan, rechristened "Sudoku", by the publisher *Nikoli*, which publishes many similar pen-and-paper logic games. They heard about it from *Dell*, under the name "Number Place". You could safely find Sudoku throughout much of Japan by the late 1980s, but it didn't take root in the rest of the world until the 21st century, when Wayne Gould, a retired

judge from New Zealand, spent six years on an algorithm that generated them, and started offering unique puzzles to newspapers across the world for free, with solutions hosted at his website. His Sudoku books became very popular, selling like hotcakes to an enthusiastic public. He was named one of "The World's Most Influential People" by TIME in 2006[2].

Sudoku is, at its core, a game of deduction that asks a solver to recreate a mathematical structure from an incomplete set of elements within it. This set of elements consists of nine numbers, meaning each square of the grid is deduced to be one of nine elements. This makes Sudoku a nonary-determination puzzle on a latin square, the only nonary-determination puzzle that I am aware of.

It would be fair to state that Sudoku has been appropriately considered, at this point, from a mathematical perspective, but it is far from the only pencil-and-paper logic game to involve unraveling a full, recognizable mathematical structure from an incomplete set of hints. The puzzle I have chosen to focus this project around, *Galaxies*, uses as its underlying structure a grid partitioned into rotationally symmetric regions. They were also popularized in Japan thanks to Nikoli, although by a different name[3]. I first came across them in *Simon Tatham's Puzzle Pack*[4], and over Covid I became deeply addicted to them. My goal in this project is to understand what's going on with these puzzles, so that the thrall they maintain over me may at least fall within my purview.

1.2 TERMINOLOGY

This paper is fundamentally about grids, like this one.

A **nxn grid** is a two-dimensional square array of length and width **n**.

In this project, the interior squares, of which there are n^2, will be referred to as **cells**. A cell is **located** at an **index**, which is a tuple denoting its row and column. The "first" cell, for us, will be that cell located at (1, 1), or, $[c_1 @ (1,1)]$. The second, $[c_2 @ (2,1)]$, the n^2th, $[c_{n^2} @ (n,n)]$. We can fill in the above grid like

(1,1)	(2,1)	(3,1)
(1,2)	(2,2)	(3,2)
(1,3)	(2,3)	(3,3)

.

This is the same as how we notate elements of a matrix; in fact, we will sometimes write partitions as matrices, with the same indices.

The interior points touching the corners of four cells, of which there are $(n-1)^2$, will be **intersections**. Their indices are the mean of the indeces of all four of their neighboring squares. Because we started indexing at the top left, let intersection i touch cell c at (x, y) from the southeast. Then, i's index is $(x + \frac{1}{2}, y + \frac{1}{2})$. Like so:

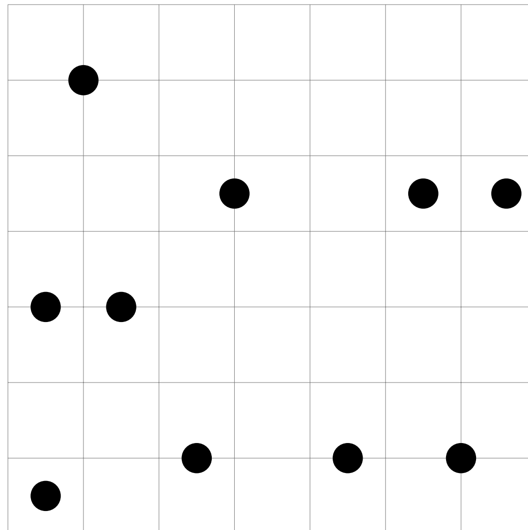
$(\frac{3}{2},\frac{3}{2})$ $(\frac{5}{2},\frac{3}{2})$

$(\frac{3}{2},\frac{5}{2})$ $(\frac{5}{2},\frac{5}{2})$.

The lines connecting two intersections, or dividing two cells, of which there are $2(n)(n-1)$, will be **gridlines**. They generate their indeces in much the same way. It is notable that, for $j,k \in \mathbf{N} \mid j < n, k \leq n$, vertical lines are located at $(j+\frac{1}{2},k)$, and for $j,k \in \mathbf{N} \mid j \leq n, k < n$, horizontal lines are located at $(j,k+\frac{1}{2})$.

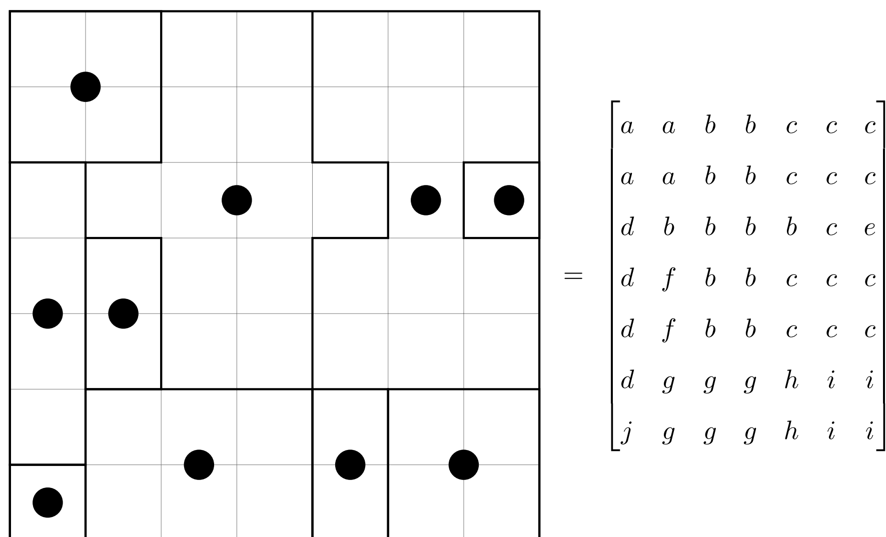
1.3 GALAXIES

An unsolved Galaxies puzzle bears some number of **centroids**. These can be on cells, intersections, or gridlines. Thus, for an nxn grid there are $(n^2)+(n-1)^2+2(n)(n-1) = (2n-1)^2$ possible centroid locations. Here is a sample unsolved galaxies puzzle:

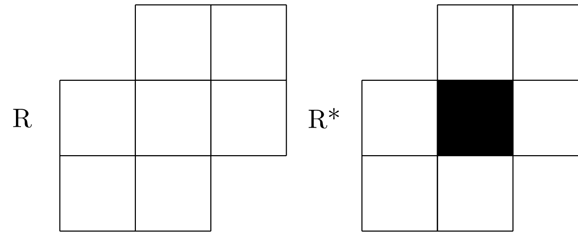


To solve this, you must create a full **partition** of this grid. A partition means you divide a thing into parts. You will fill in some of the gridlines, so that cell falls within a distinct region of other cells, fenced off by "on" gridlines. We will frequently notate this as an nxn matrix, with a value $\in \mathbf{N}$ for each cell as determined by which region it falls into, a process illustrated below. Each region must be 180° rotationally symmetric about a centroid, which means that if you spun it halfway around while holding it in place at the centroid, it would be the same shape, taking up the same cells.

Below is the solution to the puzzle, as well as its representation in matrix notation.

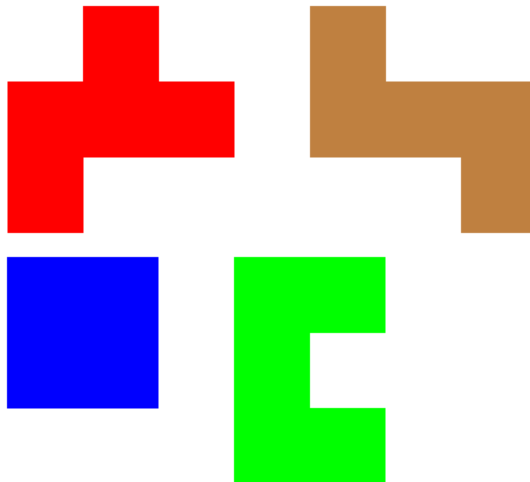


Now, partitioning a grid is very easy, even if you're making sure there's only one centroid per region. To truly solve a Galaxies puzzle, each region of the partition must be 180° **rotationally symmetric** about a centroid, using every centroid. In a region R that is rotationally symmetric about a centroid c at (x, y), there are an even number of cells within the region not housing the centroid, R^*.



There are three possible cases for the size of the region housing the centroid, which will be written as $(|R| - |R^*|) = h$. If the centroid sits on a cell, it touches one cell only, and $h = 1$. If the centroid is on a gridline, it touches two, and $h = 2$, and in the same way on an intersection $h = 4$. Further, for every cell $[g \ @ \ (a, b) \in R^*]$, there exists a cell $[h \ @ \ (c, d) \in R^*]$, for which $(\frac{a+c}{2}, \frac{b+d}{2}) = (x, y)$. We will use this as the definition of rotational symmetry. We will notate half of the number of cells R^* of a symmetric region has with λ.

The concept is easier to understand visually. Two of the below shapes are symmetric, two are not.



The symmetric shapes are blue and brown. You could manually find the centroid, and affirm a correspondence for both, but for now just imagine rotating them 180°. In this case, λ of the blue square is zero, and λ of the brown zigzag

is 2.

In the context of grids, which is distinct from the case earlier, with integers, we define a partition as a assignment of cells into non-empty **regions**, in such a way that every cell is included in exactly one region. In addition, within 2D space, groups are **connected**, meaning there exists a path crossing adjacent cells fully contained within the region, for every pair of cells within the region. Each region is surrounded by gridlines which are **on**, which we will call **walls**, but between cells of the same region they are **off**, or, **lawns**. This nitty-gritty definition makes it seem unappealing, pedantic, but the intuition behind this is not very difficult to understand, and the process of solving doesn't really *feel* like math. We just need to think of them like this if we want to count, categorize, generalize, or make them.

The goal of this project is to be able to generate a lot of these with an algorithm. To begin with, we'll need to figure out how many ways there are to partition a nxn grid, or at least figure out how they develop, acknowledging that the sequence grows exponentially, and somehow get a whole bunch of partitions. Then, we'll need to somehow make sure all its regions are rotationally symmetric, and find our centroids. Then we'll have Galaxies! This project will be accomplished in Python, and we will go through each algorithm manually, at close having some nice puzzles.

2 nx1

2.1 THE FIRST DIMENSION

How would we do this for an nx1 grid? Everything is 180° rotationally symmetric here, so we'll ignore that for now. A helpful jumping off point for

categorizing 1-dimensional grids is to begin by thinking of integers.

A **partition** of an integer n is a list of integers $[a_1, ..., a_m] \mid \sum_{i=1}^{m} a_i = n$. Figuring out how many partitions there are for large values of n is very hard when order doesn't matter, when partitions like [2,1] and [1,2] are equivalent. In galaxies, order does matter. This is clear when dealing with 2-dimensional puzzles: if regions get rearranged, the puzzle becomes clearly different, and possibly invalid. In sum,

$\neq$.

Let's count the first few partitions manually.

1 can be [1].

2 can be [2], or [1, 1].

3 can be [3], [1, 2], [2, 1], [1, 1, 1] (Pictured below).

4 can be [4], [1, 3], [3, 1], [2, 2], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 1, 1].

1, 2, 4, 8. The simplest function that could model this is 2^{n-1}. Let's prove that this is true by drawing a bijection partitions of integers to partitions of nx1 grids, and counting them explicitly.

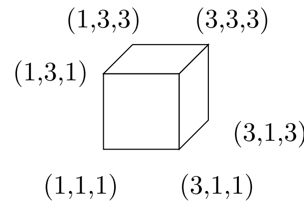
The partitions of the 3x1 grid look like

There are 2 gridlines in the grid for all of those partitions. For any $n \in N$, there will be $n - 1$ gridlines in the nx1 grid. Each of these gridlines can be turned on or off, meaning there are two possible states for each one. Each is totally independent of the others, so we can count all of them by multiplying 2 by itself $n - 1$ times, or, calculating 2^{n-1}. Thus, each list of numbers adding up

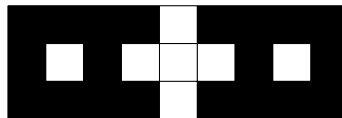
to n can be represented with a binary string of length $n-1$, with each 1 meaning that the previous region has concluded. So, for any sequence of size n consisting of zeroes and ones (these will be henceforth referred to as **bit strings**), we can generate a solved 1-dimensional galaxy of size $n-1$, converting zeroes to lawns, and ones to walls.

2.2 AN ASIDE ON ROTATIONAL SYMMETRY

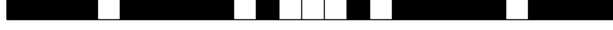
A lot of what will follow deals with general grid partitions, ignoring the symmetry requirement. The reason for this is that 180° rotational symmetry generalizes across dimensions, and can be accounted for quite easily. Let's take a regular old cube, like



Suppose this cube exists on three different 3x3 grids, which have been stacked on top of each other. We could fan the the three grids out methodically, preserving the order of the cells, and represent the cube on a 2-dimensional 9x3 grid, like



This form is 180° rotationally symmetric, although not connected and not containing its center. We can condense this form further, into one dimension, by a similar process of rearranging, onto a 27x1 grid.



Instead of indeces denoted as tuples, like (x, y) it now makes more sense to have integer indices, calculated for a newly 1-dimensional $[c \,@\, (x, y)]$ like $i_{2D\rightarrow 1D}(c) = (n * y) + x$. We know that this shape used to be rotationally symmetric. So, we know:

$S\ R \in G \mid [c(R) \,@\, (a, b)]$

Suppose we have a region in a valid galaxy such that the center of the region is located at the index (a, b).

$\forall\ [j \,@\, (x, y)] \in R^*,$

For every cell j that is located at the index (x, y) in the part of the region that does not touch the center,

$\exists\ [k \,@\, (v, w)] \in R^* \mid \frac{x+v}{2} = x, \frac{y+w}{2} = b$

there exists a cell k, also in the part of the region that does not touch the center, for which the tuple composed of the average of the x-values and the average of the y-values between the two cells is the center.

The method of turning indeces to integers between 1 and n^2 by defining the index of c as $i(c) = (n * y) + x$ gives a unique index to all cells, but let's arbitrarily define a cell's "outdex" solely based on its region, and relationship with its centroid. We'll use a semicolon instead of the at sign. An outdex is calculated for $x \in R$ by $i(x) - i(c(R))$. The below region is marked with outdexes.

	-3	-2
-1	0	1
2	3	

$\exists\ R \in G \mid [i(R) \ @\ (a,b),\ o(R)\ ;\ 0]$

With that in mind, let's suppose that there exists a region in a valid galaxy such that the center of the region is located at the index $i = (a,b)$, and outdex $o = 0$.

$\exists\ [u\ @\ (x,y),\ u\ ;\ q] \mid (x-\alpha, y-\beta) = (a,b).\ q = n\beta + \alpha$

Let there then also exist a cell u at index (x,y) such that x minus some value α is a and y minus some value β is b. The outdex of u is gonna be $n\beta+\alpha$.

$\exists\ [v\ @\ (s,t),\ v\ ;\ p] \mid \frac{s+x}{2} = a,\ \frac{t+y}{2} = b.$

Because we know R is rotationally symmetric, we know that there exists a cell v at index (s,t) and outdex p such that the two indeces average out to the center.

$\frac{s+a+\alpha}{2} = a, \frac{t+b+\beta}{2} = b \to (s,t) = (a-\alpha, b-\beta).$

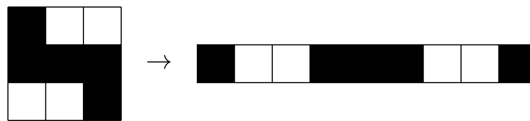
From a while ago, we know that $(x,y) = (a+\alpha, b+\beta)$, so we can substitute that in and quickly solve for $s = (a-\alpha)$ and $t = b-\beta$. With our formula from earlier for calculating outdex, we can conclude from this that

$p = n(-\beta) + (-\alpha) = -q.$

So, for a region to be 180° rotationally symmetric, its one-dimensional condensed form must be palindromic, or, the same forward and backward. The below algorithm generalizes beyond forms with binary states, by this same logic.

```python
def is_it_rotationally_symmetric(bitstring):
    gnirtstib = bitstring[::-1]
    #[::-1] reverses a string in python
    if gnirtstib == bitstring:
        return True
    else:
        return False

# print(is_it_rotationally_symmetric("100101001")) returns True
# print(is_it_rotationally_symmetric("100101101")) returns False
```

The converse is not true, as regions whose 1-dimensional forms have no gaps are always palindromic, even when not symmetric. Nevertheless, this correspondence, and understanding, will help us enormously as we go on.



3 2x2

3.1 THE SECOND DIMENSION

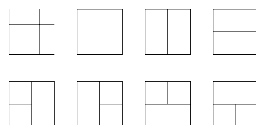
This is more difficult. Connection of regions becomes a bigger issue. The best $2^{\#\,gridlines}$ can be is a remedial upper bound on the total number of partitions. There's trivially one valid partition for a 1*1 grid, but potentially $2^{2n(n-1)} = 16$ partitions for a $2 * 2$. They are detailed on the next page, with the eight valid galaxies on top, then the valid partitions. The final four are merely arrangements, similar to this one.

1	2
3	4

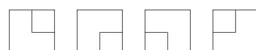
The reason that we do not count this and other arrangements of its ilk as partitions is that they are inherent contradictions. Consider a wall between two cells as a designator that ”these two cells do not share a group”, and a lawn as a designator that ”these two cells do share a group”.

Without loss of generality, the above arrangement is stating that both “3 in group with 1 in group with 2 in group with 4 $\rightarrow$ 3 in group with 4” and “3 not in group with 4”, and is contradicting itself. Thus, we have:

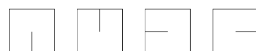
3.1.1 GALAXIES



3.1.2 PARTITIONS



3.1.3 SILLY SQUARES



With the remedial geometric intuition that the above argument holds for larger grids (and the acknowledgement that a validly partitioned grid with n^2 cells can be appreciated as $(n-1)^2$ validly partitioned $2x2$ grids) and not much else in mind, we can posit

The Simple Grid Conjecture: A grid is partitioned when none of its

intersections touch only one wall.

We will attempt to prove this later.

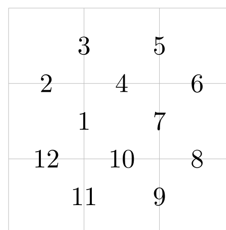
$$\begin{bmatrix} a & a \\ a & a \end{bmatrix} \begin{bmatrix} a & a \\ a & b \end{bmatrix} \begin{bmatrix} a & a \\ b & a \end{bmatrix} \begin{bmatrix} a & a \\ b & b \end{bmatrix} \begin{bmatrix} a & a \\ b & c \end{bmatrix} \begin{bmatrix} a & b \\ a & a \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ a & b \end{bmatrix} \begin{bmatrix} a & b \\ a & c \end{bmatrix} \begin{bmatrix} a & b \\ b & b \end{bmatrix} \begin{bmatrix} a & b \\ c & b \end{bmatrix} \begin{bmatrix} a & b \\ c & c \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

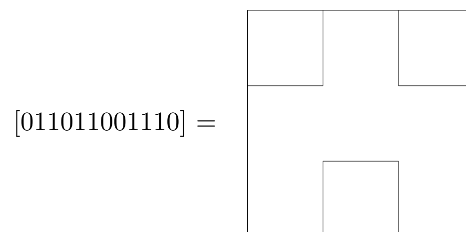
4 3x3

4.1 METHOD A

There are 12 gridlines on a 3x3 grid. This means that there cannot be more than 4096 partitions of such a grid. Let's put numbers on our gridlines like so:



We can let the i'th element of a given 12-element bit string represent the gridline in the above diagram marked as i. Here is an example.



Let's just make a list of every 12-bit string.

```
figaro = []
for i in range(4096):
    s = bin(i)[2:]
    while len(s) != 12:
        s = "0" + s
    figaro.append(s)

print("There are " + str(len(figaro)) + " elements of this list right
    now.")
# There are 4096 elements of this list right now.
print("The number '151' in base-10 translates to " + bin(151)[2:] + " in
    binary.")
# The number '151' in base-10 translates to 10010111 in binary.
print("The 152nd element of this list containing zero is " + figaro[151]
    + ".")
# The 152nd element of this list containing zero is 000010010111.
```

This algorithm tells us all possible $3 * 3$ grid partitions, as well a bunch of binary strings that map to invalid partitions, violating the Simple Grid Conjecture, and leaving an intersection connected to only one gridline. Let's get rid of those, checking that [1,2,3,4], [4,5,6,7], [7,8,9,10], and [10,11,12,1] all do not sum to 1.

```
for i in range(len(figaro)):
    grid = figaro[i]
    for j in range(4):
        if (int(grid[(3 * j) + 0]) + int(grid[(3 * j) + 1]) +
             int(grid[(3 * j) + 2]) + int(grid[((3 * j) + 3) % 12]) == 1):
            figaro[i] = "blunt"
```

```python
treebytree = []
for i in range(4096):
    if figaro[i] != "blunt":
        treebytree.append(figaro[i])

print("There are " + str(len(treebytree)) + " elements of this list
    right now.")
```

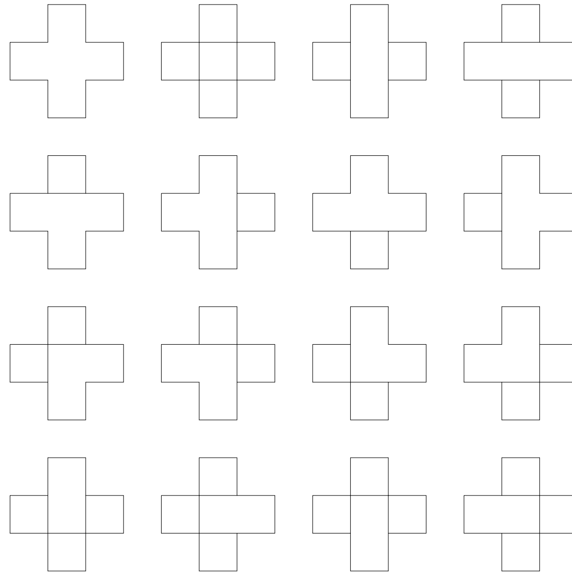
Above, we iterate through each of the 4096 strings, checking that the slices all do not add to one. If at least one does, the string is removed from the list. This is because the corresponding grid violates the Simple Grid Conjecture. The list *treebytree* contains every possible partition. The code's output is:

```
There are 1442 elements of this list right now.
```

4.2 METHOD B

This total can also be found by hand.

Suppose a five-cell polyomino, with four cells entirely surrounding one center cell. There are four total interior gridlines to this shape, allowing for only 16 possible partitions, which are as follows.



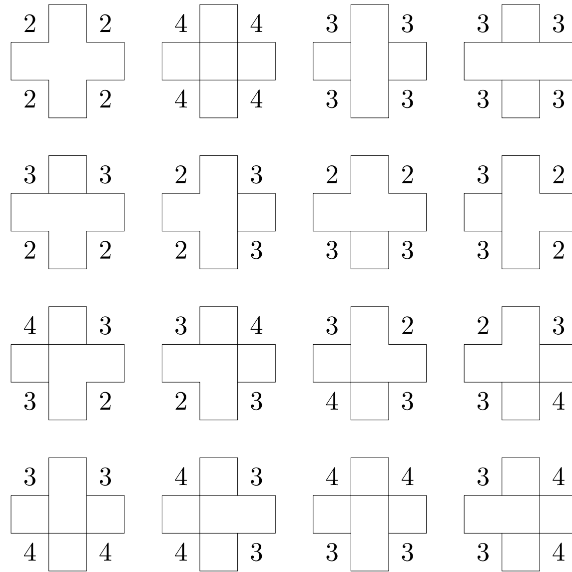
Any corner cell (which has not yet been counted) has three consequential neighbors: the two cells adjacent to it, and the center cell. Each of these three-cell regions can be in one of three important states:

Case 2. Entirely Grouped (One Group, 2 Lawns)

Case 3. Partially Separated / Partially Grouped (Two Groups, 1 Wall, 1 Lawn)

Case 4. Entirely Separated (Three Groups, 2 Walls)

A corner cell whose neighborhood (defined as a standard Moore neighborhood, all cells orthogonally or diagonally adjacent to a given cell) aligns to Case 2 will have 2 possible groupings - with the group it is surrounded by, or alone. Case 3 implies 3 possible groupings, with the group horizontally adjacent, vertically adjacent, and alone. Case 4 implies 4 possible groupings, with the group horizontally adjacent, vertically adjacent, alone, and the fourth choice of merging the two groups it is adjacent to. Thus, we can fill in the above diagram with the numbers that their cases align to.



We must multiply together the outcomes of the corner spaces to get the total amount of 3x3 partitions of a given cross formation. Once we add all 16 cross formation counts together, We will have the total number of possible 3x3 partitions, if we assume a grid region can only be invalid by violating the Simple Grid Conjecture. Thus:

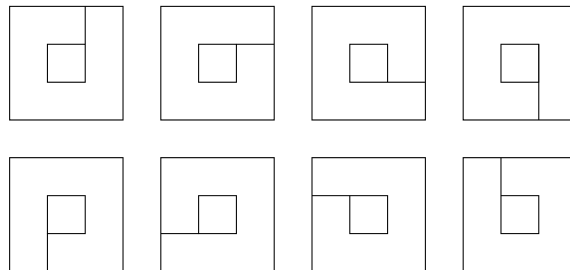
$$(16) + (256) + ((2) * (81)) + ((4) * (36)) + ((4) * (72)) + ((4) * (144)) = 1,442$$

4.3 TOROIDALITY

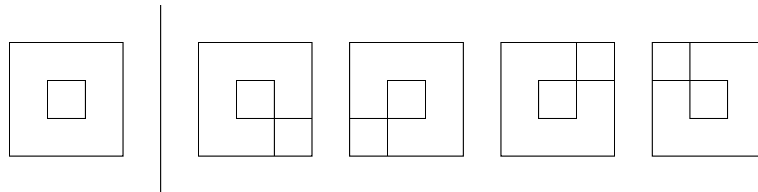
To show n=3 $\rightarrow$ 1442, we would need to prove the Simple Grid Conjecture holds not just ”if” no intersections are lonely, but ”if and only if”. I will not do this.

Envision the smallest grid region that could exist on both sides of a gridline. If its height is 2 or less, it will have to bend back on itself, causing a ”lonely intersection”, violating the conjecture. Same goes for width. Thus, the smallest possible group that could border itself whilst bearing no lonely intersections will

have height and width 3. This region is pictured below, along with its seven like-minded associates.



Subtracting these off from the main total gives 1434 grid partitions for $n = 3$. We will be calling regions that are symmetric but do not contain their centers **toroidal**. This means that they completely surround some natural number of interior cells of a separate group. There exist valid toroidal groups, five in a 3x3 grid, below. One, the one on the left, is symmetric.

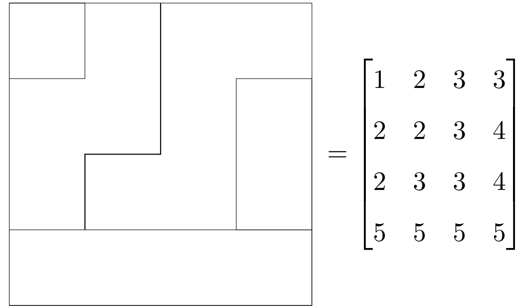


These grids are fine, but they cannot be notated as proper Galaxies, as they do not include their center. They count as valid partitions, so they're cool for now. This leaves **1434 = 2*3*239** partitions of a 3x3 grid.

5 4x4

5.1 ANOTHER DULL OUTDEX PROOF

Suppose a valid galaxy of length and width n, that is, a partitioned grid into $180°$ rotationally symmetric regions. Suppose we notate it by least-to-greatest group membership, creating a matrix of integers instead of letters. To draw a specific example without loss of generality,



We can think of a partitioned region as an ordered list of indeces, that forms a subset of the list of all possible tuples of integers from 1 to n. If we are given a region, in this list format, that we know is rotationally symmetric, finding the centroid is as simple as taking the mean of both sides of the tuples independently.

Let's call a general region, or list of tuples, R, and let R be rotationally symmetric. Recall that each cell $[g \; @ \; (a,b)] \in R^*$ which is centered around centroid $[c \; @ \; (x,y)] \in R$ has a "mirror image" $[h \; @ \; (c,d)] \in R*$ for which $(\frac{a+c}{2}, \frac{b+d}{2}) = (x,y)$. Lets split R into three lists, s_1, c_0, s_2, where c_0 contains $R - R^*$, s_1 contains those remaining cells with negative outdexes, and s_2 contains those cells with positive outdexes. Let $c_0 = R - R^*$, a list of one, two, or four tuples $[(a_1,b_1), ...(a_m,b_m)]$ for which $(\frac{\sum_{n=1}^{m} a_n}{m}, \frac{\sum_{n=1}^{m} b_n}{m}) = (x,y)$.

We learned a while ago that a cell $[g \ @ \ (x+\alpha, y+\beta)] \in R^*$ means that there must also be a cell $[h \ @ \ (x-\alpha, y-\beta)] \in R^*$. Thus, g and h are in separate groups, and $|s_1| = |s_2| = \lambda(R)$. Further, if the combined outdex of $s_1 = w$, for some w, then the outdex of $s_2 = -w$, and the mean of all elements has an outdex of zero, which means it is at the center of the region.

Further, if we suppose g is the k'th term in s_1, then there are $k-1$ terms with positive outdexes before g in s_1, and $\lambda(R) - k$ terms after. Because each of these corresponds to a term of similar outdex, but different sign, the reverse is true for s_2. That is, h is the $(\lambda(R) - k)$'th term in s_2, and there are $k-1$ terms after it. This will influence the filter algorithm.

5.2 n=4, COMPUTER

Let's put numbers to the intersections on the $4*4$ grid. There's 24 total gridlines, as shown below.

1 21 5
2 3 6 7
4 9 8
22 10 11 23
13 12 17
14 15 18 19
16 24 20

Gridlines 1-20 can be toggled to lawns and walls in 12^5 ways. This is because [1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16], [17,18,19,20] all have 12 possible partitions (by our 2x2 section), and none of the 12^5 combinations violate the Simple Grid Conjecture immediately. Some leave lonely intersections with one remaining unassigned gridline, or empty intersections but for the unassigned. In both situations, the unassigned gridline has only one possible case. If there

are two or three walls at its intersection, there are two possible cases. With this, we can write some Python code.

```
qs = [[0,0,0,0], [0,1,0,1], [1,0,1,0], [1,1,1,1], [0,0,1,1], [0,1,1,0],
    [1,1,0,0,], [1,0,0,1], [1,1,1,0], [1,1,0,1], [1,0,1,1], [0,1,1,1]]
#All 2x2 grids


fours = []
order = [2,5,8,
        3,9,12,
        7,10,16,
        11,14,17]
#The final gridlines' neighbors, organized.
for i in range(12 ** 5):
    proxylist = [qs[i % 12] + qs[int(np.floor(i / 12)) % 12] +
        qs[int(np.floor(i / 144)) % 12] + qs[int(np.floor(i / 1728)) %
        12] + qs[int(np.floor(i / 20736))]]

    #all unique lists of five elements of q. There are 12^5 total.

    for j in range(4):
        b = len(proxylist)

        if (proxylist[0][order[(3*j)]] + proxylist[0][order[(3*j)+1]] +
            proxylist[0][(3*j)+2]) == 0:
            for k in range(b):
                proxylist[k].append(0)

        #When all neighboring gridlines are lawns, the last remaining
            gridline
        #must also be a lawn.
```

```
    elif (proxylist[0][order[(3*j)]] + proxylist[0][order[(3*j)+1]]
        + proxylist[0][(3*j)+2]) == 1:
        for k in range(b):
            proxylist[k].append(1)

    #When one neighboring gridline is a wall, the last remaining
        gridline
    #must also be a wall.

    else:
        b = len(proxylist)
        for k in range(b):
            newlist = proxylist[k] + [0]
            proxylist[k].append(1)
            proxylist.append(newlist)

    #In any other case, the gridline could be either a lawn or a
        wall.

fours += proxylist
```

The final list has length $1,632,816 (2x2x2x2x3x3x17x23x29)$. All of these are possible partitions, but the list might contain invalid regions which border themselves. Instead of combing through this data and scaling this tactic to our general case, we will try a totally new algorithm, keeping this number and tactic in mind.

6 nxn

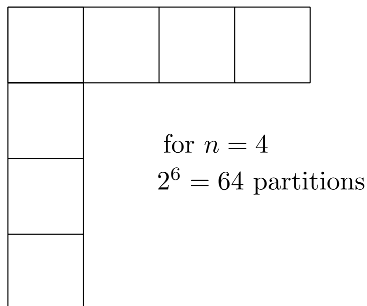
6.1 INSPIRATION

Let's examine the $2 * 2$ case more closely.

There is one way to partition the $1 * 1$ grid.

If we add a block on the side, we can either have one group or two. On the bottom, there are similarly two options. Four total, for a three-cell region, as follows.

Let's go back to the n*1 case, and modify it slightly. Picture a line of cells that's $2n - 1$ cells long, as opposed to n. This region has 2^{2n-2} valid partitions, as does the below one, with a bend.



These partitions are very easy for a computer to generate, by modifying our old brute force algorithm. We'll write a function that takes any natural number n, which represents the integer side length of the grid. We want to make a list composed of all partitions of the region formed by the leftmost

vertical nx1 region and the upmost horizontal region, total size of both 2n - 1, and, by what we determined in the first dimension, total partitions $= 2^{2n-2}$. So we can cleanly iterate, we'll do it by going first across the top row, then down the leftmost column. This is what order it will go in for $n = 4$:

1	5	6	7
2			
3			
4			

This iterates through every number from 0 to 2^{2n-2} in binary. This is still 1-dimensional in terms of how we count it, despite it being a bit long, because all interior cells have two adjacent neighbors, and both exterior cells only have one.

```
def topline(n):
    edge = []
    for i in range(2**(n-1)):
        bitstringofi = bin(i)[2:]
        while len(bitstringofi) < (n-1):
            bitstringofi = "0" + bitstringofi
        frame = [1]
        for j in range(n-1):
            if bitstringofi[j] == "0":
                frame.append(frame[-1])
# If zero, lawn.
            else:
                frame.append(frame[-1] + 1)
# If one, wall
        edge.append(frame)
```

```
    return(edge)

topline(3)
# [[1, 1, 1], [1, 1, 2], [1, 2, 2], [1, 2, 3]]

def firstvert(n, top):
    edges = []
    toprow = top
    frame = []
    for h in range(2**(n-1)):
        for i in range(2**(n-1)):
            frame += toprow[h]
            bitstringofi = bin(i)[2:]
            while len(bitstringofi) < (n-1):
                bitstringofi = "0" + bitstringofi
            if bitstringofi[0] == "0":
                frame.append(1)
            elif bitstringofi[0] == "1":
                frame.append(frame[-1] + 1)
            for j in range(n-2):
                if bitstringofi[j+1] == "0":
                    frame.append(frame[-1])
                    # If zero, lawn.
                elif bitstringofi[j+1] == "1":
                    frame.append(max(frame) + 1)
                    # If one, wall
            edges.append(frame)
            frame = []
    return(edges)

firstvert(3, topline(3))
```

```
#[[1, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 1, 2, 2], [1, 1, 1, 2, 3],
# [1, 1, 2, 1, 1], [1, 1, 2, 1, 3], [1, 1, 2, 3, 3], [1, 1, 2, 3, 4],
# [1, 2, 2, 1, 1], [1, 2, 2, 1, 3], [1, 2, 2, 3, 3], [1, 2, 2, 3, 4],
# [1, 2, 3, 1, 1], [1, 2, 3, 1, 4], [1, 2, 3, 4, 4], [1, 2, 3, 4, 5]]
```

Here, we get all bit strings of length $2^{2(n-1)}$, with Python's convenient "bin()" function. Then, we cycle through the "frame", setting elements to the same or different groups by the index of the bit string. This creates all possible "frames" of grid partitions, although we might change them later. To see why, we must go back to the $2 * 2$s.

There are two possible groups that the cell we place at a could belong to, while maintaining a valid partition of the entire grid: With the group, and without, like so.

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \rightarrow$$

```
    if (nw == ne) and (nw == sw):
        return([nw, (nw+1)])
```

Cells b and c have the same spread of cases. With the group above, with the group to the left, or totally alone. We can treat them the same way in our algorithm.

$$\begin{bmatrix}1 & 1\\2 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\2 & 2\end{bmatrix}\begin{bmatrix}1 & 1\\2 & 3\end{bmatrix}\begin{bmatrix}1 & 2\\1 & 1\end{bmatrix}\begin{bmatrix}1 & 2\\1 & 2\end{bmatrix}\begin{bmatrix}1 & 2\\1 & 3\end{bmatrix}\rightarrow$$

```
if (nw == ne) or (ne == sw):
    return([ne, sw, (max([ne, sw]) + 1)])
```

Now, in the case of d, we've got four possible partitions: the three available to b and c, and to merge b and c into one group. We can just return 0 in the last case.

$$\begin{bmatrix}1 & 2\\3 & 2\end{bmatrix}\begin{bmatrix}1 & 2\\3 & 3\end{bmatrix}\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}1 & 2\\2 & 2\end{bmatrix}\rightarrow$$

```
else:
    return([ne, sw, (max([ne, sw]) + 1), 0])
```

To merge, we need access to the whole grid, not just the 2x2 section we're analyzing. This is a darn shame: it will increase our function's runtime dramatically. This is why there's no nice table at the end of the section: doing this billions of times requires more computational resources than I can muster. If we want to truly know the number of partitions, theoretically, we must check it. On seeing 0, we call merge, once we have access to the full list, and set all values in either group to the minimum index. To make sure we don't get our numbering system off, we must subtract one from all indeces above the one whose group we removed.

```
def merge(gridsofar, ne, sw):
    mn = min(ne, sw)
    mx = max(ne, sw)
    newgrid = []
    for i in range(len(gridsofar)):
        if gridsofar[i] == mx:
            newgrid.append(mn)
        elif gridsofar[i] < mx:
            newgrid.append(gridsofar[i] - 1)
        else:
            newgrid.append(gridsofar[i])
    return(newgrid)
```

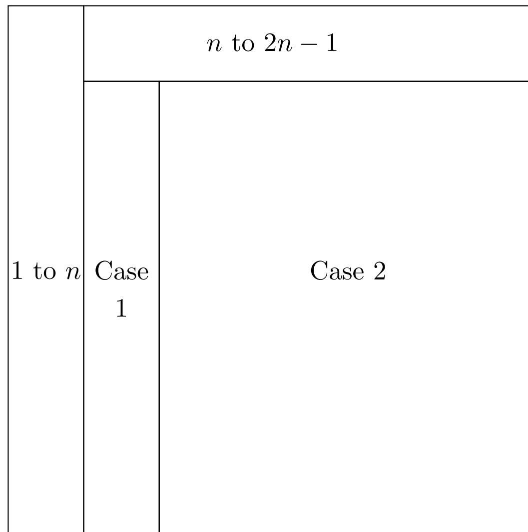
Now that all cases have been accounted for, if we know the group membership of a cell c's northwest, northeast, and southwest neighbors, we can figure out all possible partitions for that region including c within that base partition.

```
def se(nw, ne, sw):
    if (nw == ne) and (ne == sw):
        return([nw, (nw + 1)])
    elif (nw == ne) or (ne == sw):
        return([ne, sw, (max([ne, sw]) + 1)])
    else:
        return([sw, ne, (max([ne, sw]) + 1), 0])
```

This puts all pieces together, taking in a northeast, northwest, and southeast group index, and returning the list of all possible partitions including the southeastern cell. Considering our "frame" holds the entire rest of the grid to its southeast, if we can figure out the right way to call se() $(n-1)^2$ times, we'll have all partitions of a given size.

6.2 VISUALIZING OUR INDECES

Keep in mind that computers start counting at zero, so the index of the first element of a list here is gonna be zero. The framer algorithm, from earlier, returns all possible top rows and leftmost columns to valid galaxies in matrix notation. Here's what it leaves us with:



Okay, let's look at what we have to recur. The two cases are split, based on whether they border the first vertical, for practical and uninteresting reasons.

```
def partition_counter(n):
    frames = framer(n)
    for j in range(n-1):
        dummy = []
        for k in range(len(frames)):
            frame = frames[k]
            stock = se(frame[?], frame[?], frame[?])
            for l in range(len(stock)):
                if stock[l] != 0:
                    z = (frame + [stock[l]])
```

```
                dummy.append(z)
            else:
                frame2 = merge(frame, frame[?], frame[?])
                dummy.append(frame2)
    frames = dummy
    for i in range(n-2):
        dummy = []
        for k in range(len(frames)):
            frame = frames[k]
            stock = se(frame[?], frame[?], frame[?])
            for l in range(len(stock)):
                if stock[l] != 0:
                    z = (frame + [stock[l]])
                    dummy.append(z)
                else:
                    frame2 = merge(frame, frame[?], frame[?])
                    dummy.append(frame2)
        frames = dummy
return(len(frames))
```

On any given loop, we know k and j, which correspond to the index of the cell we are filling in plus two, or $k, j \to [c \text{ @ } (k+2, j+2)]$. At this point, the cell we are filling in has an index on the frame that we can easily find by computing "len(frame)". The cell to its west, which we want to put into "stock" in place of "frame[sw]", was the last input cell, "len(frame) - 1". However, this only holds in the case that $k \neq 0$, and otherwise the southwestern cell is at $n - j - 2$. We must split the code into two cases. The northwestern cell works very similarly, being n away in the second part, otherwise, at index $n-j-1$. The northeastern is reliably $n-1$ away. We can clean a lot of this, due to python's handy negative wraparound indexing.

```
def partition_counter(n):
    frames = firstrow(n,firstcol(n))
    dummy = []

    for c in range(n-1):
        for k in range(len(frames)):
            frame = frames[k]
            firstcell = se(frame[c], frame[1-n], frame[1+c], max(frame))
            for l in range(len(firstcell)):
                if firstcell[l] != 0:
                    z = (frame + [firstcell[l]])
                    dummy.append(z)
                else:
                    frame2 = merge(frame, frame[1-n], frame[1+c])
                    dummy.append(frame2)

        frames = dummy
        dummy = []

        for m in range(n-2):
            for k in range(len(frames)):
                frame = frames[k]
                nextcell = se(frame[-n], frame[-n+1], frame[-1],
                     max(frame))
                for l in range(len(nextcell)):
                    if nextcell[l] != 0:
                        z = (frame + [nextcell[l]])
                        dummy.append(z)
                    else:
                        z = merge(frame, frame[-n+1], frame[-1])
```

```python
                    dummy.append(z)

            frames = dummy
            dummy = []
    return (len(frames))


# TABLE OF VALUES
# partition_counter(1) - 1
# partition_counter(2) - 12
# partition_counter(3) - 1434
# partition_counter(4) - 1665869 [4.7 SECONDS TO RUN]

# If https://oeis.org/A145835 is to be believed something is
# incorrect. Also, I don't think it's possible to get this done for
# n > 5. Let's just play galaxies now.

partition_counter(3)
```

We can use what we've done to make an endless supply of galaxies. We don't need to iterate through every conceivable galaxy of size n to have some playable ones, we can just call for a random list of 100,000 at each size of n that we want. Arbitrarily, let's say $n = 7, 11, 15$, as those are the sizes I find the most fun.

7 GALAXIES

7.1 GENERATING STANDARD PARTITIONS

We know how to count galaxies: it will surely be a snap to generate some radom ones, by using the existing code. Let's grab some sample partitions. Note that the "least to greatest" notation goes along the length of the frame.

```
def random_partition_of_size(n):
    frames = framer(n)
    frame = frames[np.random.randint(0, len(frames))]
    print(frame)
    for j in range(n-1):
        frame = frames[k]
        stock = se(frame[n - j - 1], frame[(len(frame) - (n+1))],
            frame[n - j - 2])
        q = np.random.randint(0, len(stock))
        if stock[q] != 0:
            frame.append(stock[q])
        elif stock[q] == 0:
            frame = merge(frame, frame[(len(frame) - (n+1))], frame[n - j
                - 2])
        for i in range(n-2):
            stock = se(frame[(len(frame) - n)], frame[(len(frame) -
                (n+1))], frame[(len(frame) - 1)])
            q = np.random.randint(0, len(stock))
            if stock[q] != 0:
                frame.append(stock[q])
            elif stock[q] == 0:
                frame = merge(frame, frame[(len(frame) - (n+1))],
                    frame[(len(frame) - 1)])
```

```
    return([frames[np.random.randint(0,len(frames))],
            frames[np.random.randint(0,len(frames))],
            frames[np.random.randint(0,len(frames))]])
```

Now, my main problem with this code is that it takes four seconds to generate three partitions for $n = 4$. Nevertheless, let's look at what we got.

```
              OUTPUTS


[[1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 2, 6, 5, 1, 6, 6],
[1, 1, 2, 2, 3, 3, 3, 3, 1, 1, 3, 1, 5, 1, 1, 1],
[1, 2, 3, 3, 1, 1, 5, 1, 5, 5, 1, 1, 5, 1, 6, 6]]


            DEFRAMED (MANUALLY)


 PARTITION1:   PARTITION2:   PARTITION3:
[3, 4, 4, 5]  [2, 3, 3, 3]  [3, 1, 1, 5]
[2, 5, 5, 5]  [2, 3, 1, 1]  [3, 1, 5, 5]
[2, 2, 6, 5]  [1, 3, 1, 5]  [2, 1, 1, 5]
[1, 1, 6, 6]  [1, 1, 1, 1]  [1, 1, 6, 6]
```

Pretty good! We'd want to change this so that it doesn't calculate every partition for a given n, but only a certain amount. Then, we'd want to deframe them, ideally in-house, and re-index the groups. Hmm. Maybe we'd be better off starting from scratch.

Let's say we have an empty grid, which we want to partition row by row. Our first step should be checking the first cell in the row, to make sure it's not empty, assigning it to be the first member of a new region if so.

```
def identity (dg, n, c):
```

```python
    if (dg[n*c] == 0):
        dg[n*c] = max(dg) + 1
        return dg
    else:
        return dg
```

We can set the vertical gridlines along the row to lawns or walls randomly.

```python
def vertical (dg, n, c):
    for j in range(n-1):
        if np.random.randint(1,4) != 1:
            if dg[(n*c) + j + 1] == 0:
                dg[(n*c) + j + 1] = max(dg) + 1
        else:
            if dg[(n*c) + j + 1] == 0:
                dg[(n*c) + j + 1] = dg[(n*c) + j]
    return dg
```

We can then assign rows of horizontal gridlines, only acting if we find a lawn.

```python
def horizontal (dg, n, c):
    for j in range(n):
        if np.random.randint(1,4) != 1:
            dg[(n*c) + j + n] = dg[(n*c) + j]
    return dg
```

So now all we need is a managerial mtehod, to call those in the correct order.

```python
def randomizer(n, a):
    frames = []
    for j in range(a):
        frame = [0] * (n*n)
        for i in range(n-1):
            identity(frame, n, i)
            vertical(frame, n, i)
            horizontal(frame, n, i)
        identity(frame, n, n-1)
        vertical(frame, n, n-1)
        frames.append(frame)
    return frames
```

That's much better! Instead of a minute to do a 15x15, it cranks out a big batch in an instant! We've got access to a pretty important additional variable here that I've glossed over: the ratio of walls to lawns. The line "if np.random.randint(1,4) != 1", which occurs in both vertical() and horizontal(), is what decides the ratio. Let's try to figure out how changing this impacts the expected value of the maximum of a partition. It'd be nice to be able to predict this, as enjoyable galaxies usually hover around $3n$ distinct regions, this number varying somewhat based on play style and grid size.

The average number of lawns as intended by the random number generator in the vertical method will be written as $l_{(v)}$. The current code for vertical, which draws a random number from [1,2,3] and creates a wall in $\frac{2}{3}$ of cases, thus, $w_{(v)} = \frac{2}{3} \mid \frac{l}{w}_{(v)} = \frac{1}{2}$. If we only consider the first n terms, as designated by the vertical() method, there should be $w_{(v)}n + 1$ groups on average. Horizontal works differently, not inflicting a binary status, but exclusively growing groups. Thus, it does not positively impact the number of groups at all. It does, however, remove $n * l_{(h)}$ potential cells from what vertical can change. Thus, the expected

amount of new groups to come from an iteration of vertical() after the first is $w_{(v)}(w_{(h)}n)+1$. We run this n times, so the total amount of expected groups is $w_{(v)}n+1+(n-1)(w_{(v)}(w_{(h)}n)+1) = n(w_{(v)}+1+w_{(v)}w_{(h)}(n-1))$.

We want 3n groups, give it a buffer of maybe $\frac{n}{2}$ on both sides. So, we want $3n = n(w_{(v)}+1+w_{(v)}w_{(h)}(n-1))$. With some algebra, we can show that in this case, $\frac{2}{w_v}-1 = w_h(n-1)$. This is actually pretty good news. It means that we can graph this (with a slider for n). I'm looking for $w_v = w_h$, out of laziness. It appears that this occurs at $w = 0.5$ @ $n = 7$, $w = 0.4$ @ $n = 11$, and, bucking what I was hoping would be a great pattern, $w = 0.3439$ @ $n = 15$. So we know what we're after!

```
def symmetricizer(frames):
    symmetrix = []
    for i in range(len(frames)):
        frame = frames[i]
        m = max(frame)
        for j in range(m):
            region = []
            for k in range(len(frame)):
                if frame[k] == j:
                     region.append(k)
            a = np.mean(region)
            while len(region) > 1:
                if ((region[0] + region[-1]) / 2) == a:
                    region.pop(0)
                    region.pop()
                else:
                    frame = [0]
                    break
        if frame != [0]:
```

```
        symmetrix.append(frame)
    return symmetrix
```

This'll take in a list of lists, and get rid of the ones with asymmetric regions, and check for toroidality too. I would start making the centroid-finder now, but it seems we've hit a wall set by God: as n grows, the proportion of symmetric partitions to asymmetric partitions approaches zero. I got some stuff for 7*7s that passed all my tests, but "symmetricizer(randomizer(11,100000))" returned nothing. We must change tactics!

7.2 REVISED ENGINE

Let's start by thinking of this differently. We don't want to partition row by row, we should be using a method that allows interesting symmetries to evolve naturally. Our groups should be different sizes every time, but predictably random. Our new first step will be to create a gaussian distribution, assigning intended sizes to regions.

```
import numpy as np

def gaussian(n):

    c = np.random.randint(np.ceil(n - (n ** 0.5)), np.ceil(n + (n **
        0.5)))
    #We are creating an n*n grid with around c total regions, and c
        centroids.
    #Those numbers don't matter too much right now.

    stage = n ** 2
    mu = stage / c  + 1
```

```
    sigma = (mu/2 - 0.3)
    gaussian = [0]
    #We will decide the intended sizes of our regions via normal
        distribution,
    #where the mean is the amount of cells divided by the amount of
        regions (times
    #1.3 bc that worked better in practice), and sigma = n^2/7 (same
        deal).


    gaussian = np.random.normal(mu, abs(sigma), int(c*1.5))

    return(gaussian)
```

A this function returns a list of about 1.5*n random numbers generated according to a random distribution with ($\mu \approx n+1, \sigma = \frac{\mu}{2} - 0.3$). I fiddled with these numbers a lot, and found these to give the best results. Here it is in action:

```
gaussian(7) =
array([13.17771841, 7.21201705, 6.52547926, 7.93499612, 6.5501925 ,
    8.18675391, 9.95702839, 8.1603684 , 6.76672675, 5.40133441,
    6.45472034, 7.5216323 , 3.84928569])
```

Now we can start on the bulk of our algorithm.

```
def starsystem(n):

    normal = gaussian(n)
    realestate = []
    for i in range(n):
```

```
    for j in range(n):
        realestate.append([float(i), float(j)])
```

Our normal distribution will by design have a sum larger than n^2, but our grid (which is called "realestate" here) won't have room for everyone. We want this.

```
regions = []
for k in range(len(normal)):
    if len(realestate) == 0:
            return regions
    r_1 = np.random.randint(len(realestate))
    unit = [realestate[r_1]]
    cx = unit[0][0]
    cy = unit[0][1]
    realestate.remove(unit[0])
    size = int(normal[k] - 1)
```

Right now, the center is some random, available cell in realestate. We've gotta always make sure to edit realestate and size, so no regions overlap (inherent to partitions).

```
    if size%4 == 0:
        if ([cx+1, cy+1] in realestate) and ([cx, cy+1] in
            realestate) and ([cx+1, cy] in realestate):
            unit.append([cx+1, cy+1])
            realestate.remove([cx+1, cy+1])
            unit.append([cx, cy+1])
            realestate.remove([cx, cy+1])
            unit.append([cx+1, cy])
            realestate.remove([cx+1, cy])
```

```
            cx += 0.5
            cy += 0.5
            size -= 3
    elif size%2 == 1:
        if [cx+1, cy] in realestate:
            unit.append([cx+1, cy])
            realestate.remove([cx+1, cy])
            cx += 0.5
            size -= 1
```

We can also have a region with a centroid at a gridline or intersection. We do this based on the region size's modulus 2 and 4, which nicely ensures that at this point our size is an even number.

```
for l in range(int(np.floor(size/2))):
            if len(realestate) == 0:
                return regions
            options = get_neighbors(unit, realestate, cx, cy)
```

Now we've got a list of all elements we could grow into, although we need to figure out the mirror image of whatever we choose, whatever it is is definetely available. It is easy for a region to have no further options before it's reached its intended size: this is why our gaussian was larger than intended. Now, we must write "getneighbors".

```
def get_neighbors(unit, realestate, cx, cy):
    options = []
    for i in range(len(unit)):
        x = unit[i][0]
        y = unit[i][1]
        if (not ([x+1,y] in unit)) and ([(2*cx)-x+1,(2*cy)-y] in
```

```
            realestate) and ([(2*cx)-x-1,(2*cy)-y] in realestate):
            options.append([unit[i][0]+1, unit[i][1]])
        if (not ([x, y+1] in unit) and ([x, y+1] in realestate) and
            ([(2*cx)-x, (2*cy)-y-1] in realestate)):
            options.append([unit[i][0], unit[i][1]+1])
    return options
```

This function takes in:

- unit - the region so far

- realestate - all remaining available cells in the grid

- cx, cy - the coordinates of the center of the region

To figure out all possible ways the region could grow while remaining symmetric, we go through every cell in the region, and check if

- a. the cell to its right/up is not yet part of the region. We don't need to check left/down, as part b handles that.

- b. if (a = True), True if both cell [c @ (x,y)] and [d @ (2cx - x, 2cy - y)] are available.

The function returns all cells found by part a. We can use that to find its symmetric "mirror image" once we get back to the larger function, so we don't need to return the one we checked in part b, since we know it's fine. Now we can finish "starsystem(n)"

```
            if len(options) > 0:
                r_2 = (1+len(realestate)) % len(options)
                mirror = options[r_2]
                rirrom = [(2*cx) - mirror[0], (2*cy) - mirror[1]]
                if (mirror in realestate) and (rirrom in realestate):
```

```
                unit.append(mirror)
                realestate.remove(mirror)
                unit.append(rirrom)
                realestate.remove(rirrom)
        regions.append(unit)
    return(regions)
```

If we have options, "(l+len(realestate)) % len(options)" pseudorandomly picks one, then we append it and its mirror image to the region. If we don't, we try again. Once the size iterator gets to zero, we return whatever region we have. We can't guarantee that it'll be the size we wanted (real estate's out of control these days), but we can promise it's symmetric.

At close, this should be mostly partitioned. Maybe a couple blank spots here and there. Let's fill in the rest.

```
def fillinthe(rest, n):

    dummy = []
    for i in range(n):
        for j in range(n):
            dummy.append([float(i),float(j)])
    #realestate by a different name.

    for i in range(len(rest)):
        region = rest[i]
        for j in range(len(region)):
            dummy.remove(region[j])

    #We've got everything that was left over.
```

```
while len(dummy) > 0:
    x = dummy[0][0]
    y = dummy[0][1]
    if ([x,y+1] in dummy) and ([x+1,y] in dummy) and ([x+1,y+1] in
        dummy):
        rest.append([[x,y],[x,y+1],[x+1,y],[x+1,y+1]])
        dummy.remove([x,y])
        dummy.remove([x+1,y])
        dummy.remove([x,y+1])
        dummy.remove([x+1,y+1])
    elif ([x,y+1] in dummy) and ([x-1,y] in dummy) and ([x-1,y+1] in
        dummy):
        rest.append([[x,y],[x,y+1],[x-1,y],[x-1,y+1]])
        dummy.remove([x+1,y])
        dummy.remove([x,y-1])
        dummy.remove([x+1,y-1])
        dummy.remove([x,y])
    elif ([x,y-1] in dummy) and ([x+1,y] in dummy) and ([x+1,y-1] in
        dummy):
        rest.append([[x,y],[x,y-1],[x+1,y],[x+1,y-1]])
        dummy.remove([x,y])
        dummy.remove([x+1,y])
        dummy.remove([x,y-1])
        dummy.remove([x+1,y-1])
    elif ([x,y-1] in dummy) and ([x-1,y] in dummy) and ([x-1,y-1] in
        dummy):
        rest.append([[x,y],[x,y-1],[x-1,y],[x-1,y-1]])
        dummy.remove([x,y])
        dummy.remove([x-1,y])
        dummy.remove([x,y-1])
        dummy.remove([x-1,y-1])
```

```
        elif [x,y+1] in dummy:
            rest.append([[x,y],[x,y+1]])
            dummy.remove([x,y])
            dummy.remove([x,y+1])
        elif [x,y-1] in dummy:
            rest.append([[x,y],[x,y-1]])
            dummy.remove([x,y])
            dummy.remove([x,y-1])
        elif [x+1,y] in dummy:
            rest.append([[x,y],[x+1,y]])
            dummy.remove([x,y])
            dummy.remove([x+1,y])
        elif [x-1,y] in dummy:
            rest.append([[x,y],[x-1,y]])
            dummy.remove([x,y])
            dummy.remove([x-1,y])
        else:
            rest.append([[x,y]])
            dummy.remove([x,y])
    return(rest)
```

This big glob of code groups stuff together as either 1, 2, or 4-cell regions. Our existing ones take up a lot of space, and are pretty big, so it's nice to have some that will definitely be small. Now that our grid is totally partitioned, all we need is two functions: one to determine centroids, and one to render our galaxies on paper, using the tikzpackage LaTex extension that all my diagrams were made in. They're pretty boilerplate, and are as follows.

```
def centralizer(regions):
    centroids = []
    for i in range(len(regions)):
```

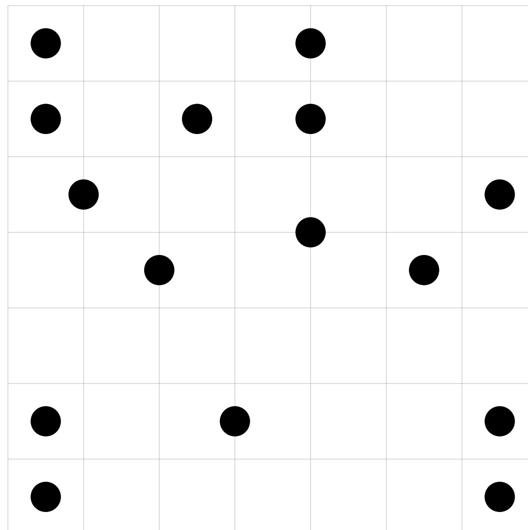
```
        region = regions[i]
        mx = 0
        my = 0
        s = len(region)
        if s > 1:
            for j in range(s):
                mx += region[j][0]
                my += region[j][1]
            centroids.append([mx/s,my/s])
        else:
            centroids.append(region[0])
    return(centroids)

def tikzgenerator(n):
    g = fillinthe(starsystem(n),n)
    c = centralizer(g)
    print("\ " + "begin{center}")
    print("\ " + "begin{tikzpicture}")
    print("\ " + f"draw[step=1cm,lightgray,very thin] (0,0) grid
        ({n},{n});")
    for i in range(len(c)):
        print("\ " + f"fill[color=black] {c[i][0]+0.5,c[i][1]+0.5}
            circle (0.2);")
    print("\ " + "end{tikzpicture}")
    print("\ " + "end{center}")
    return
for i in range(20):
    tikzgenerator(7)
for i in range(20):
    tikzgenerator(10)
for i in range(20):
```

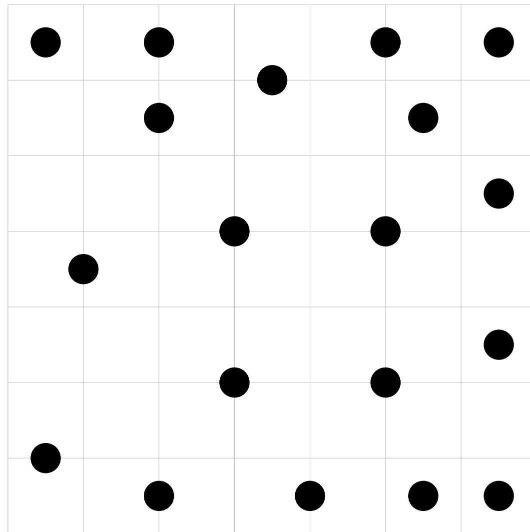
```
tikzgenerator(15)
```

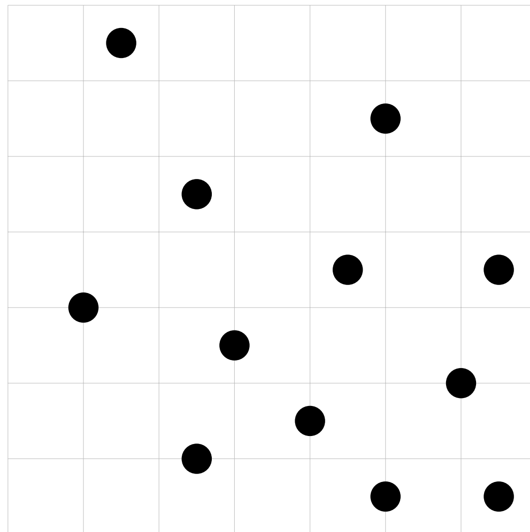
With this, I can leave you with a whole bunch of galaxies. If you need more, run the above code for whatever n you want. Happy solving!

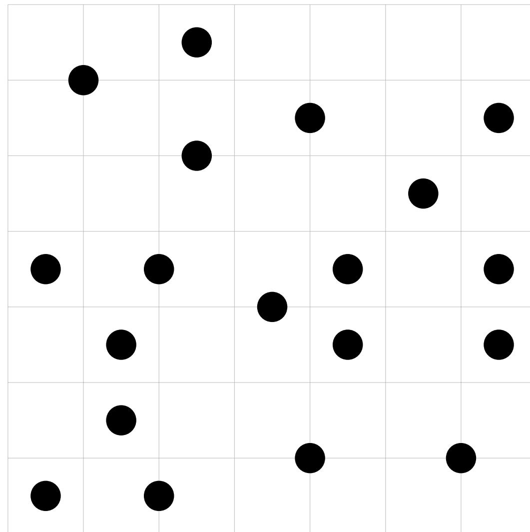
7.3 PUZZLES

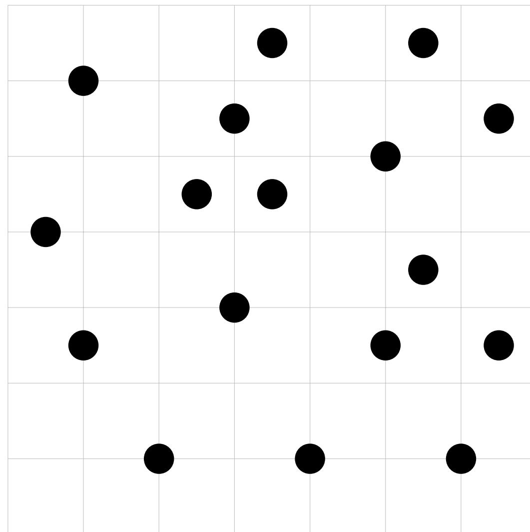
7.3.1 n=7

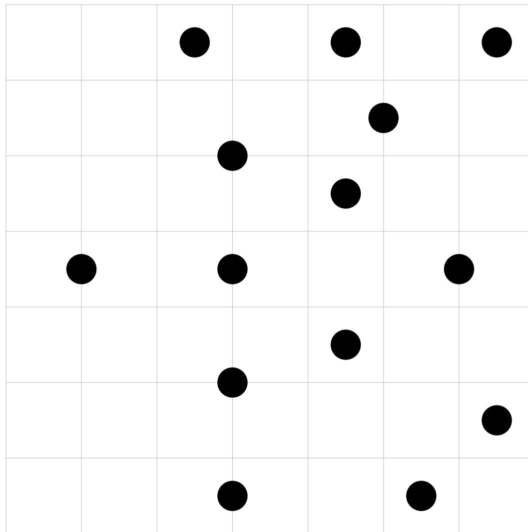


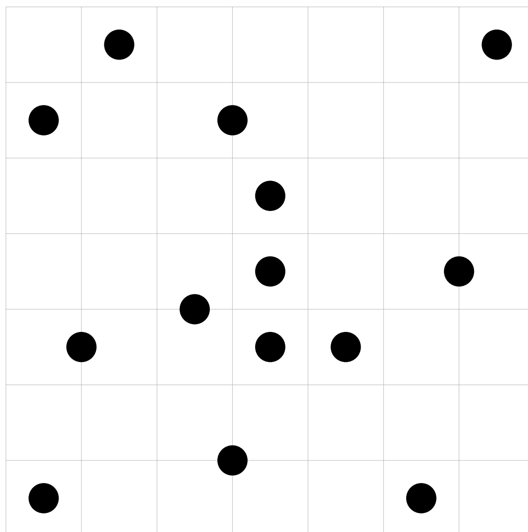


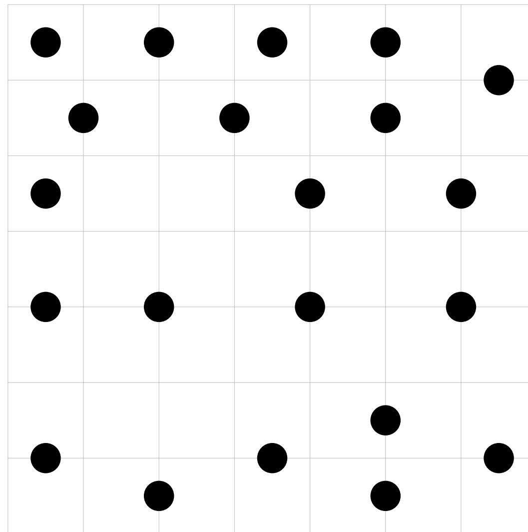


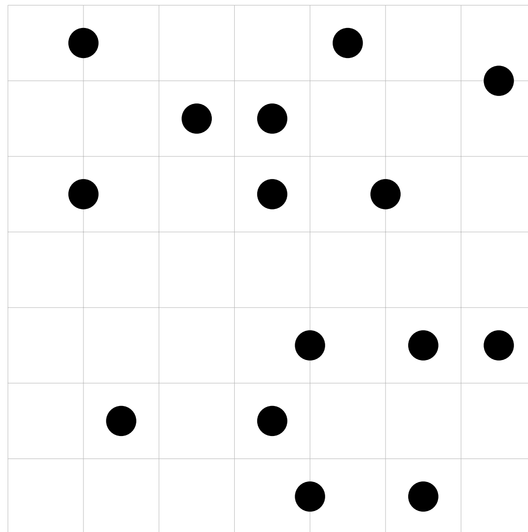


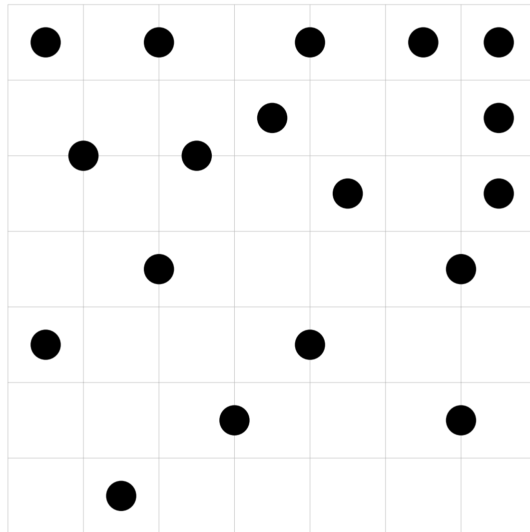


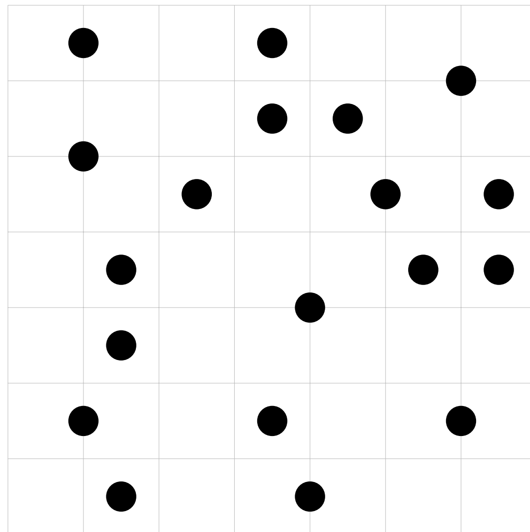


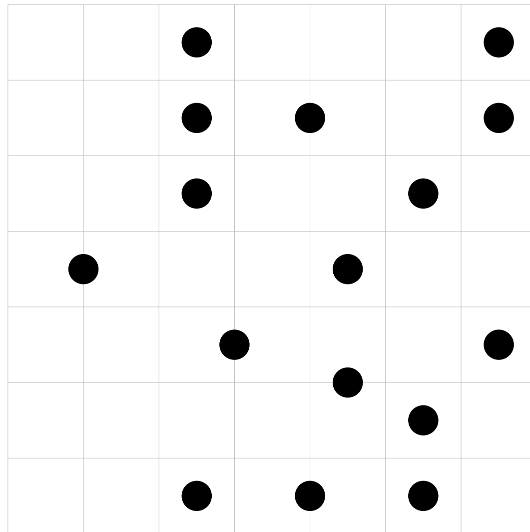


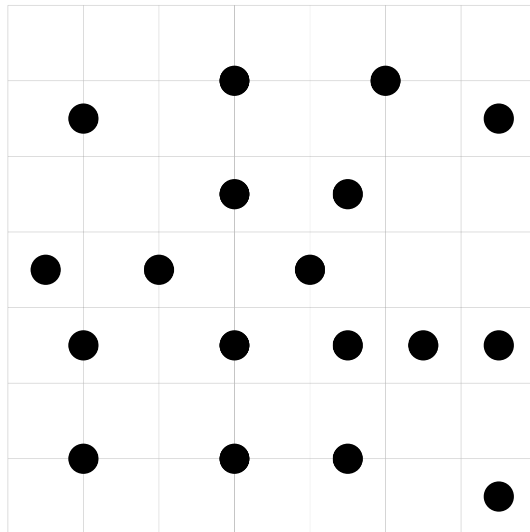


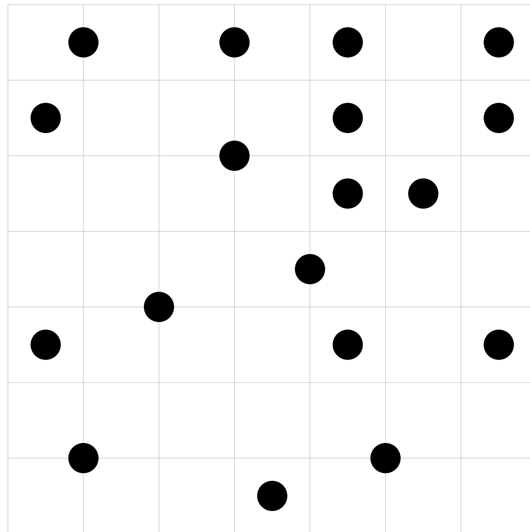


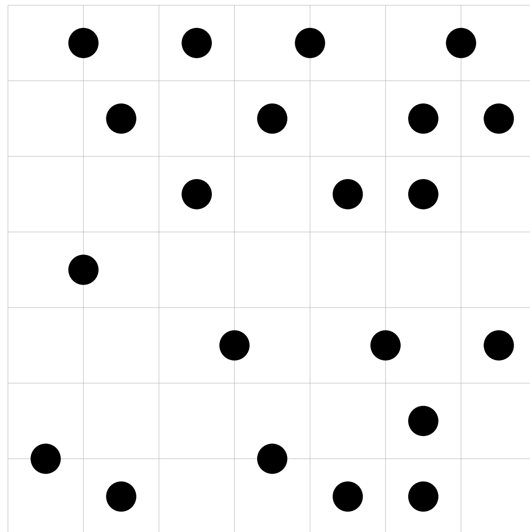




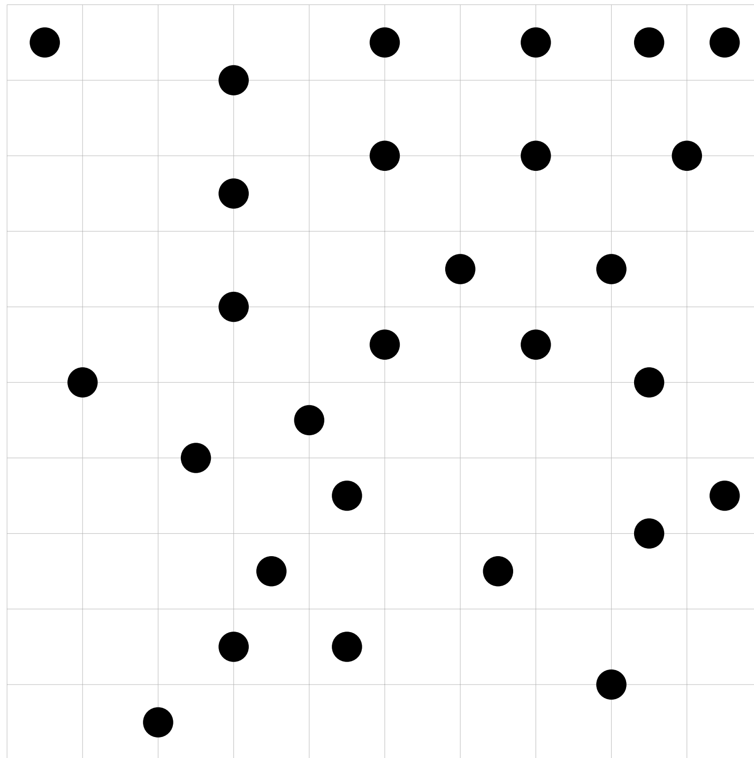


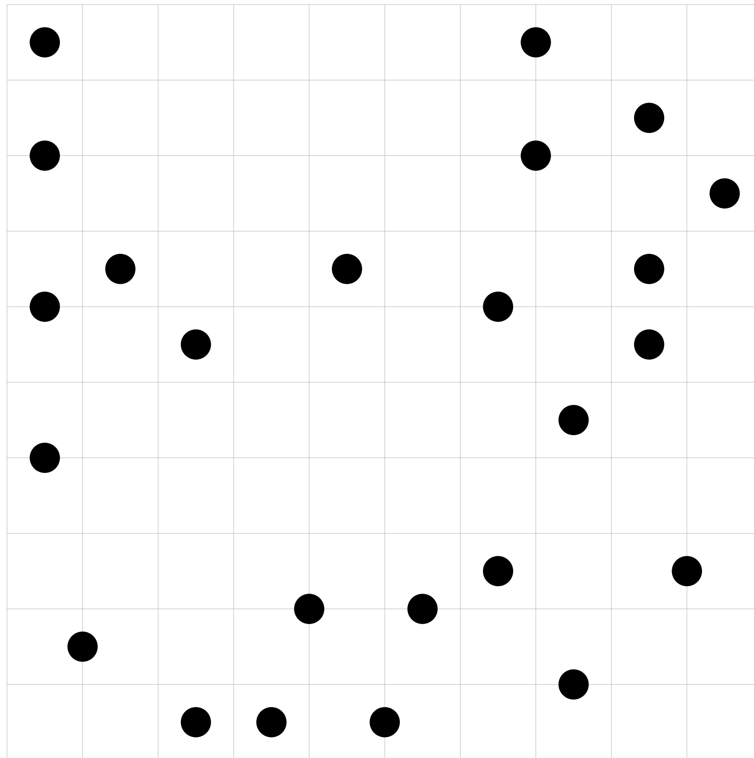


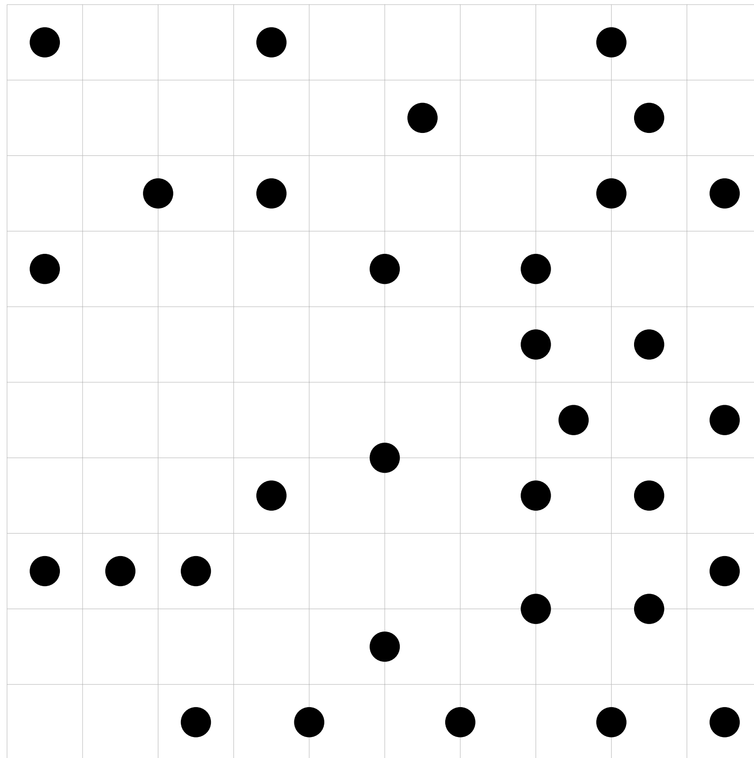


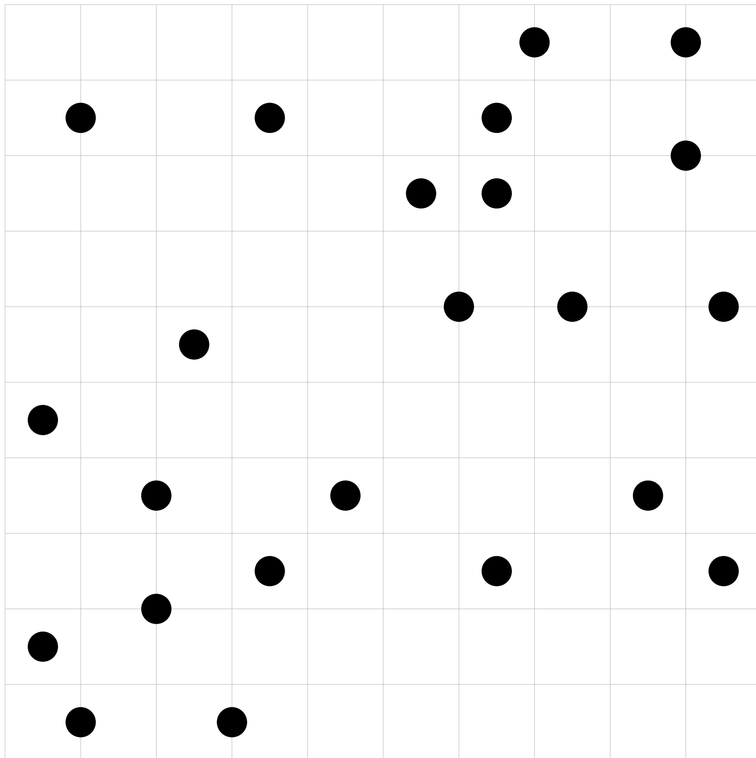


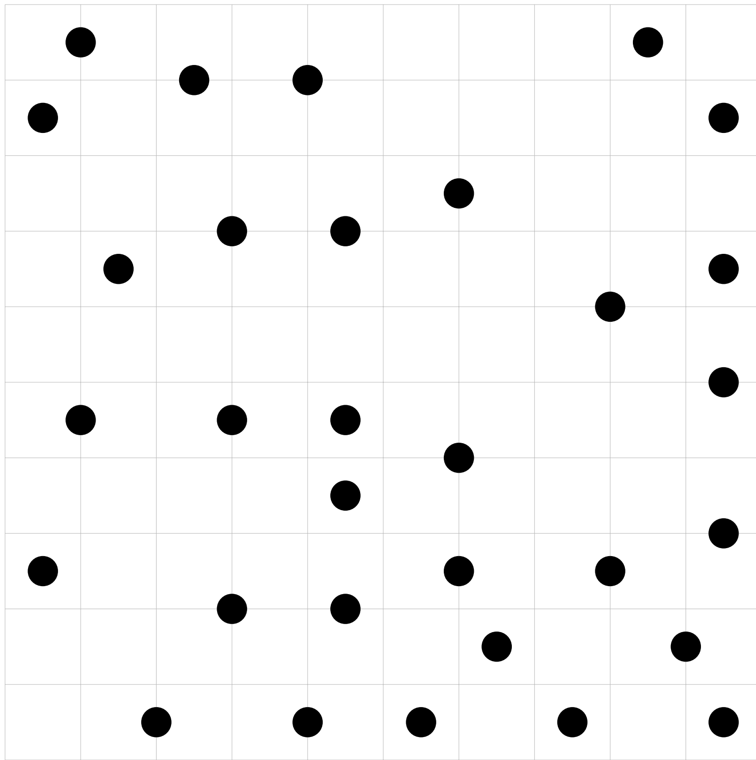
7.3.2 n=10

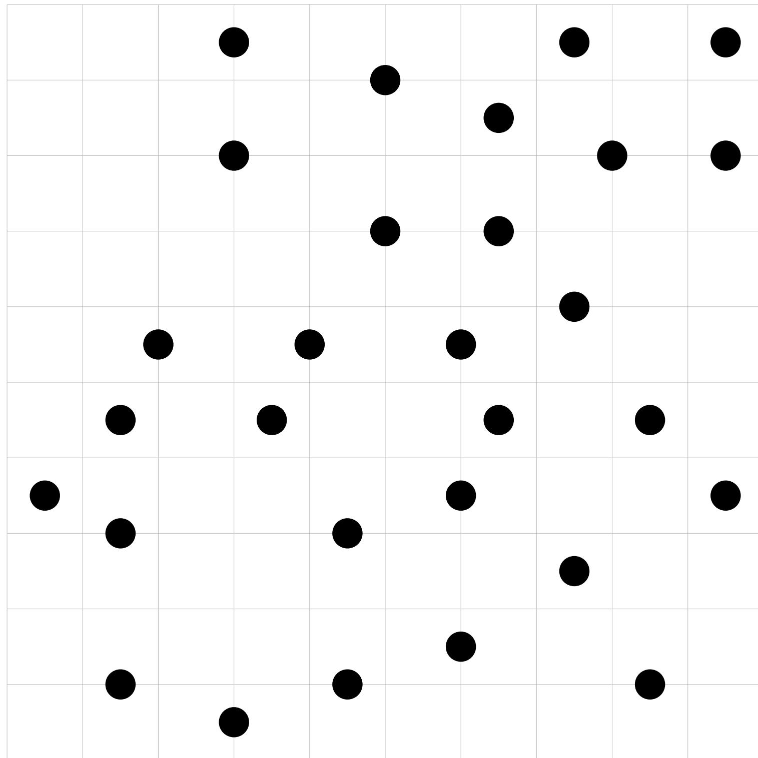


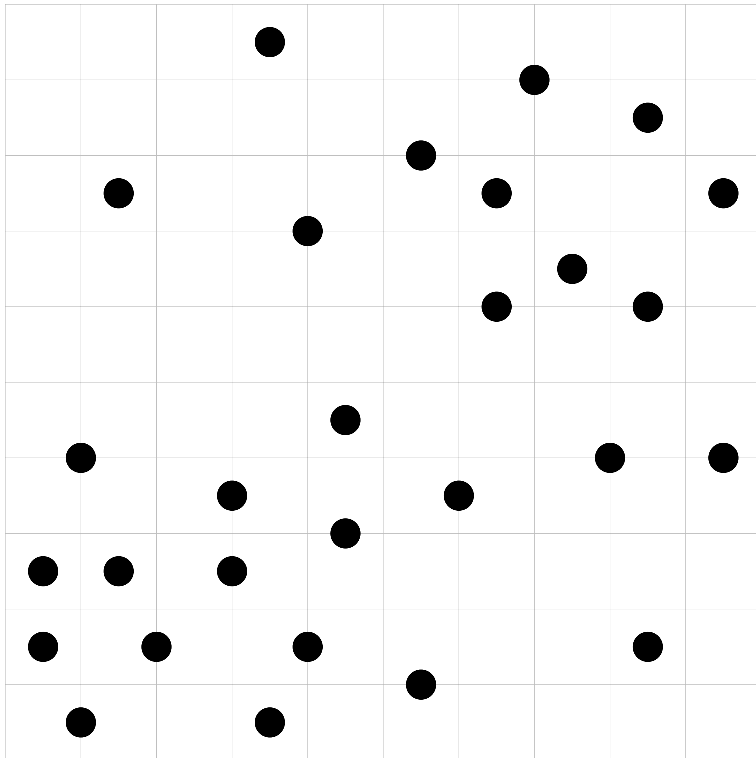


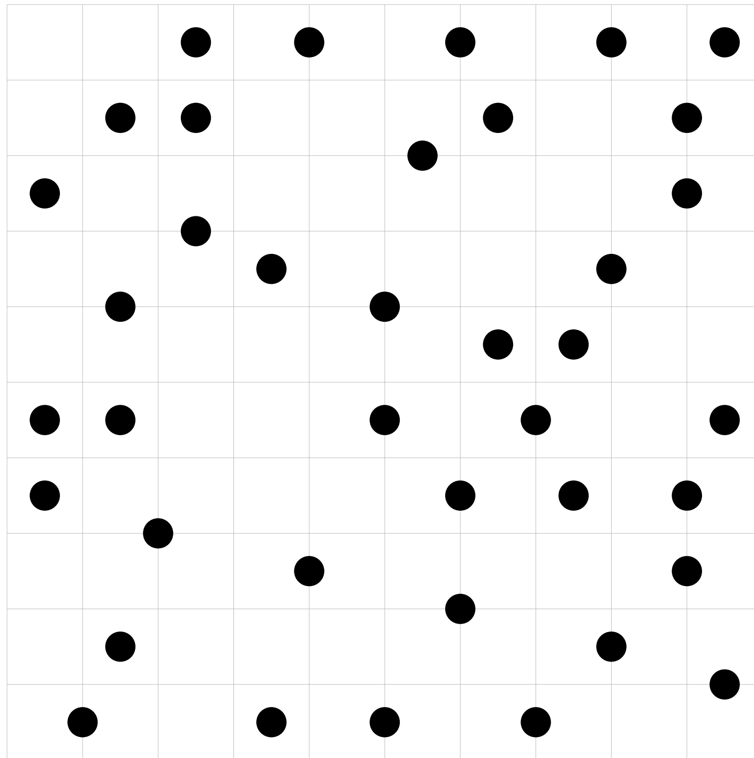


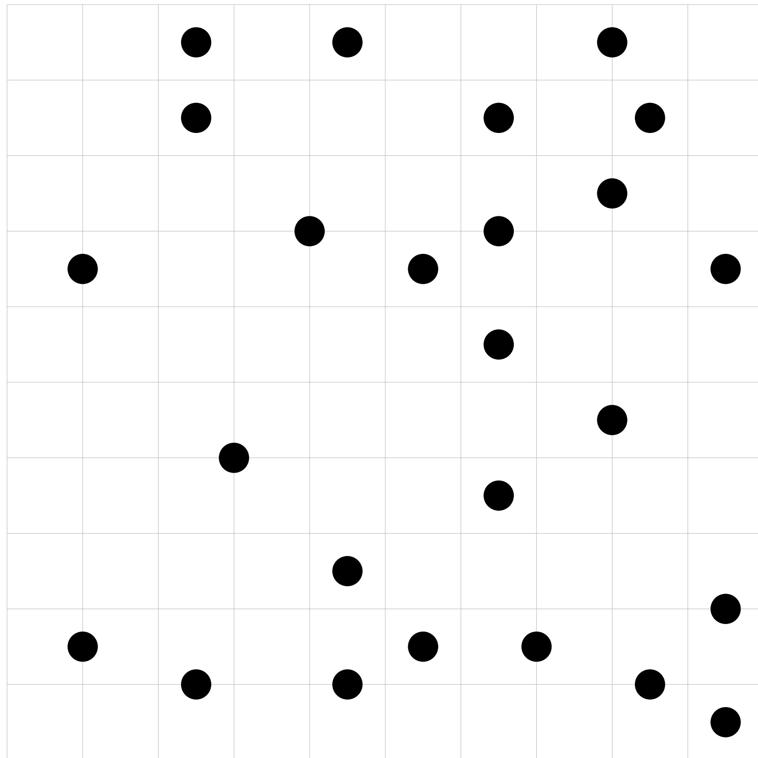


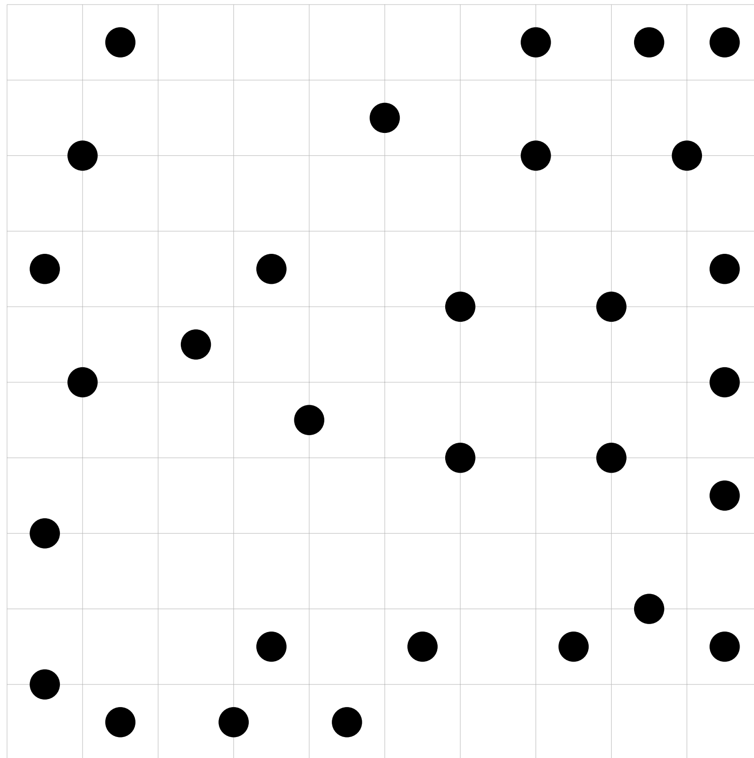


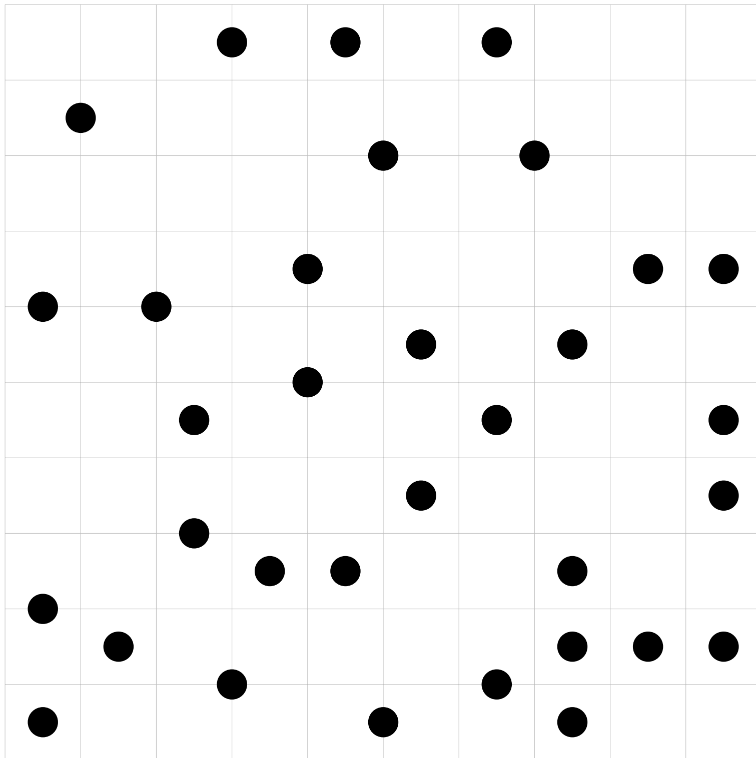


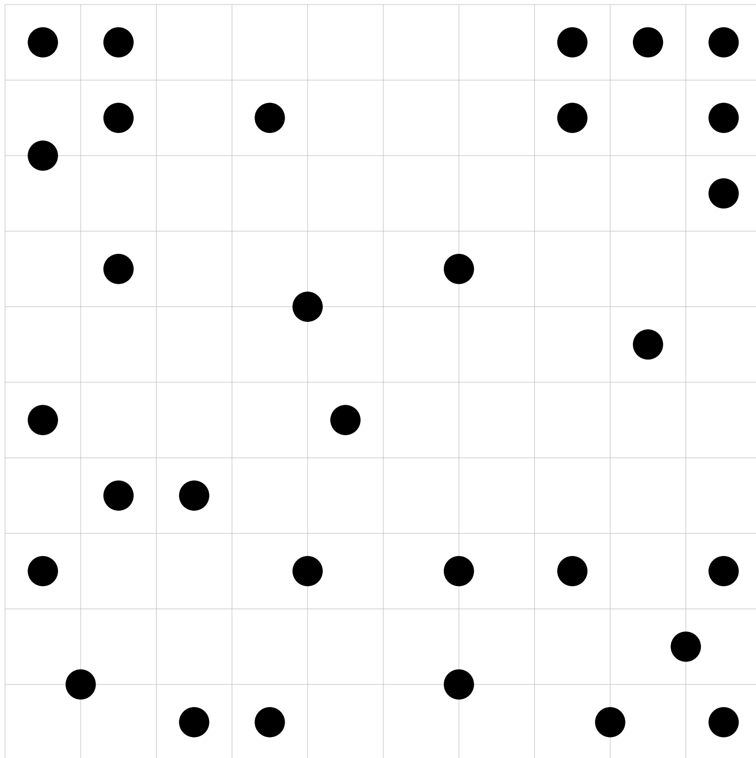


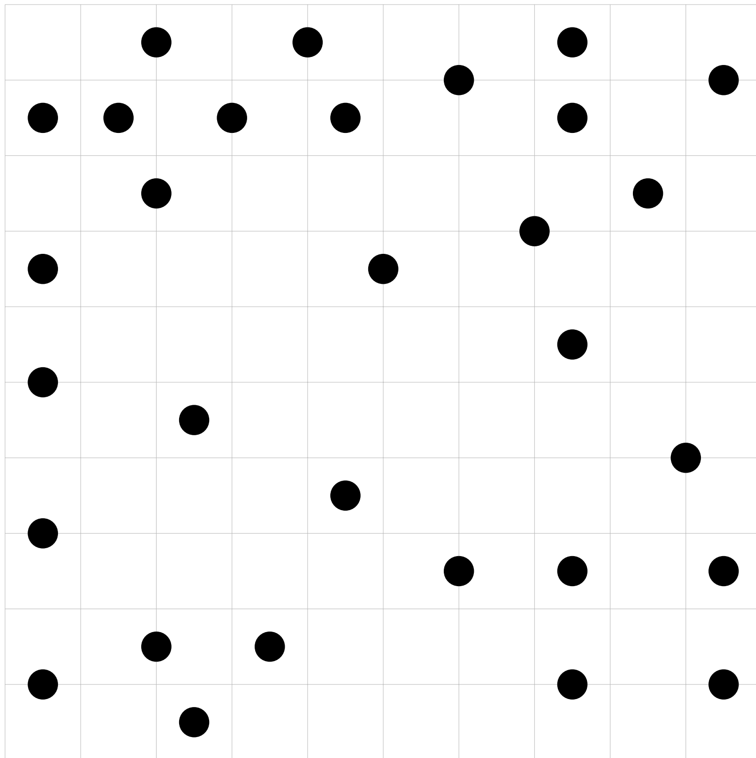


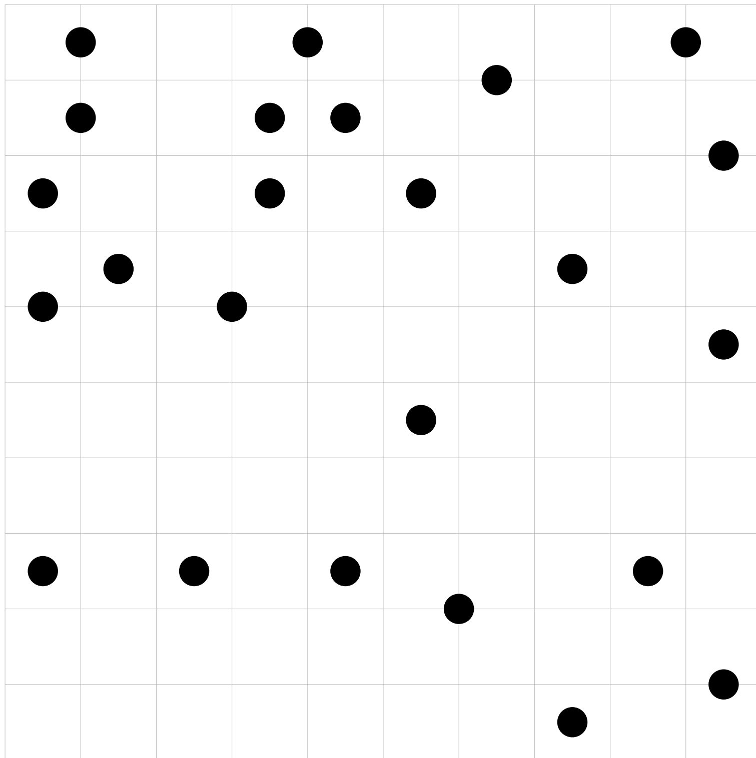


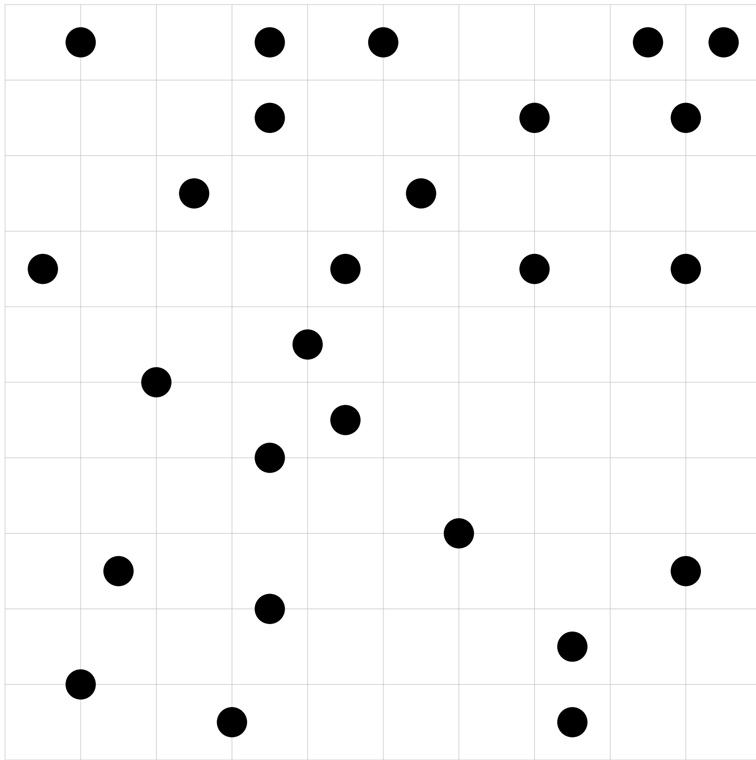


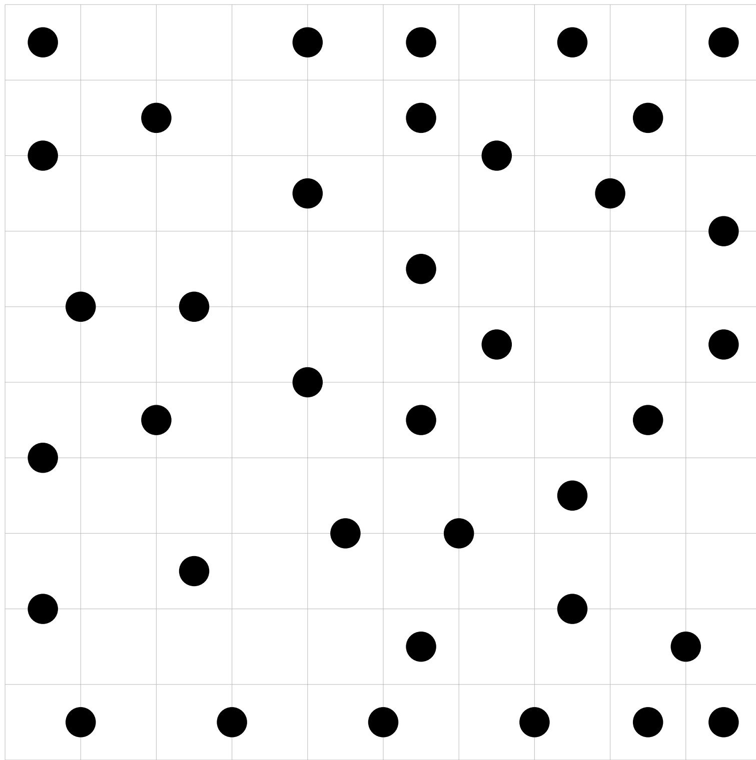


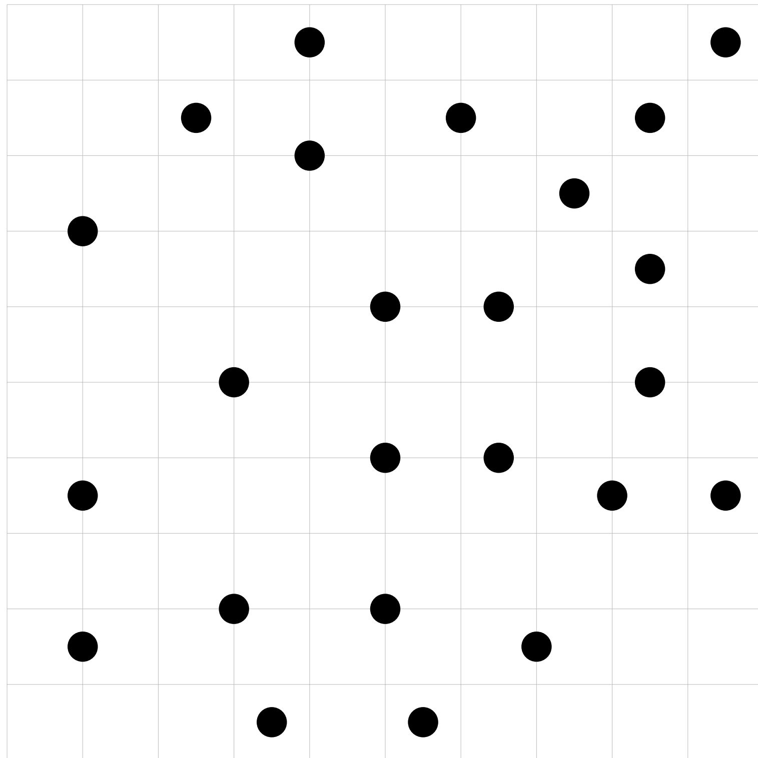


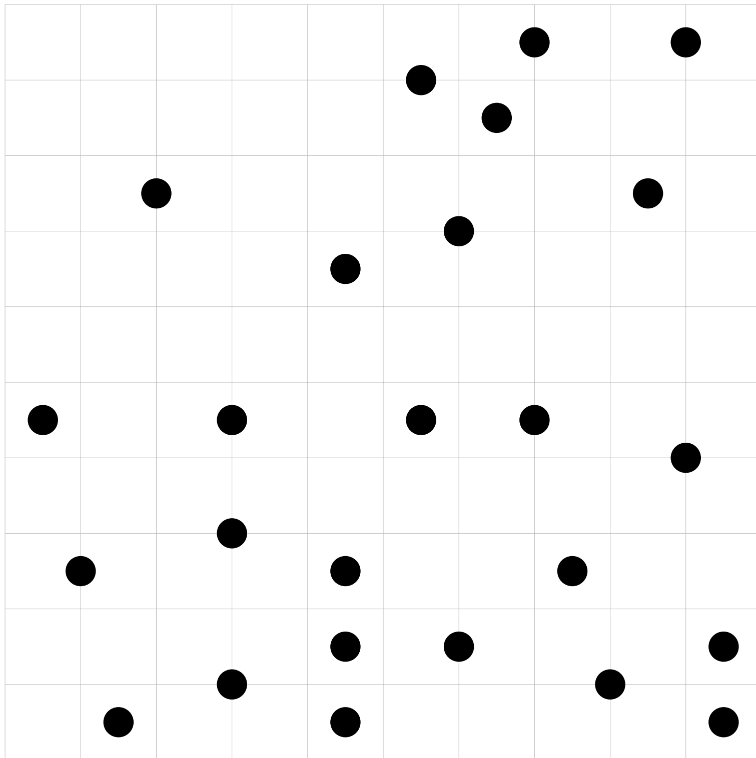


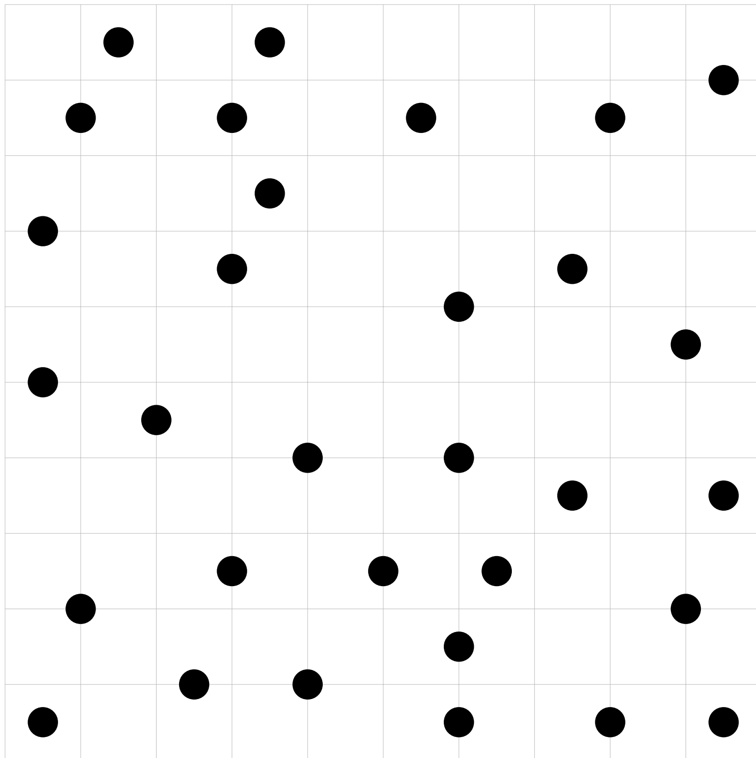


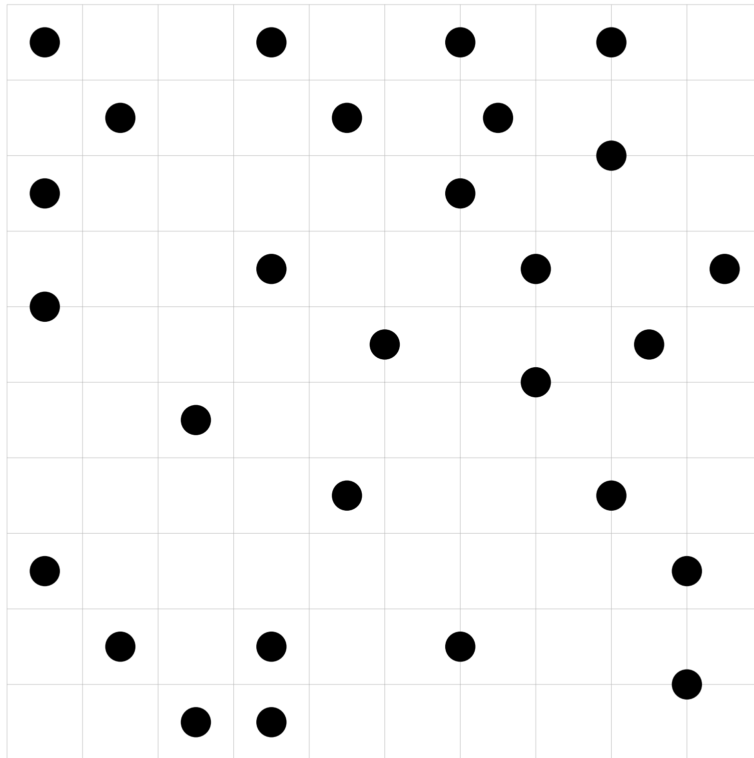




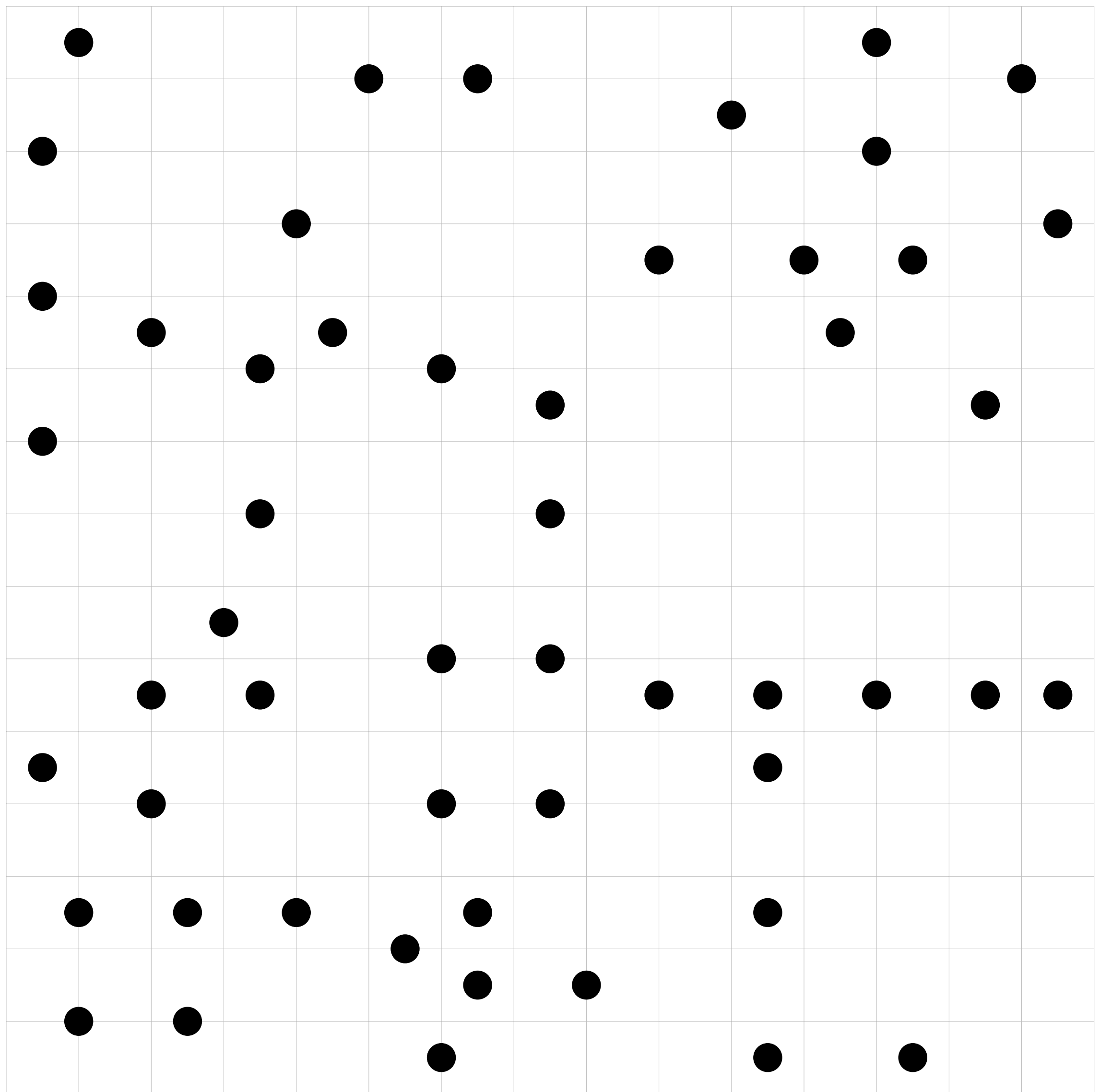


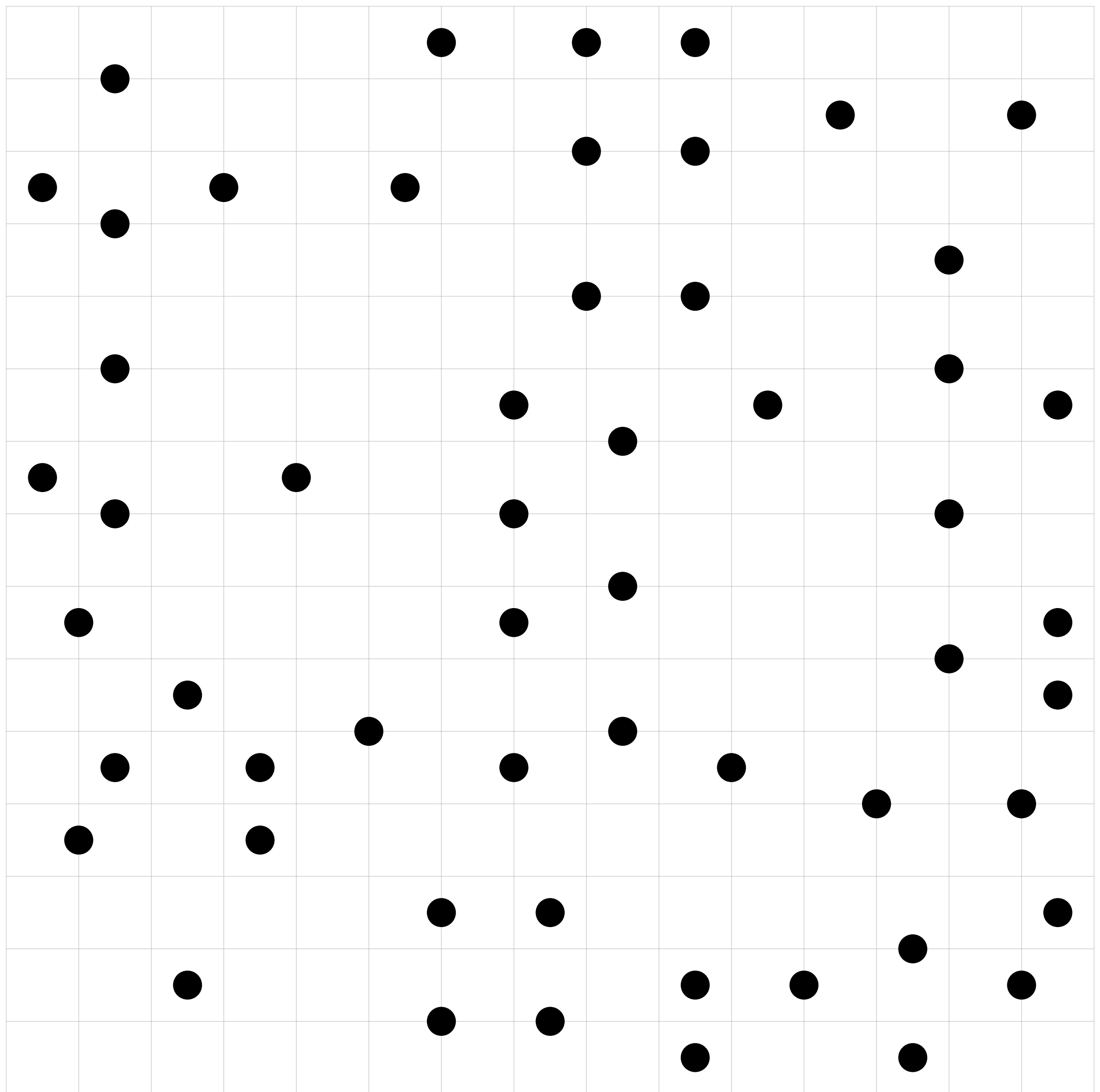


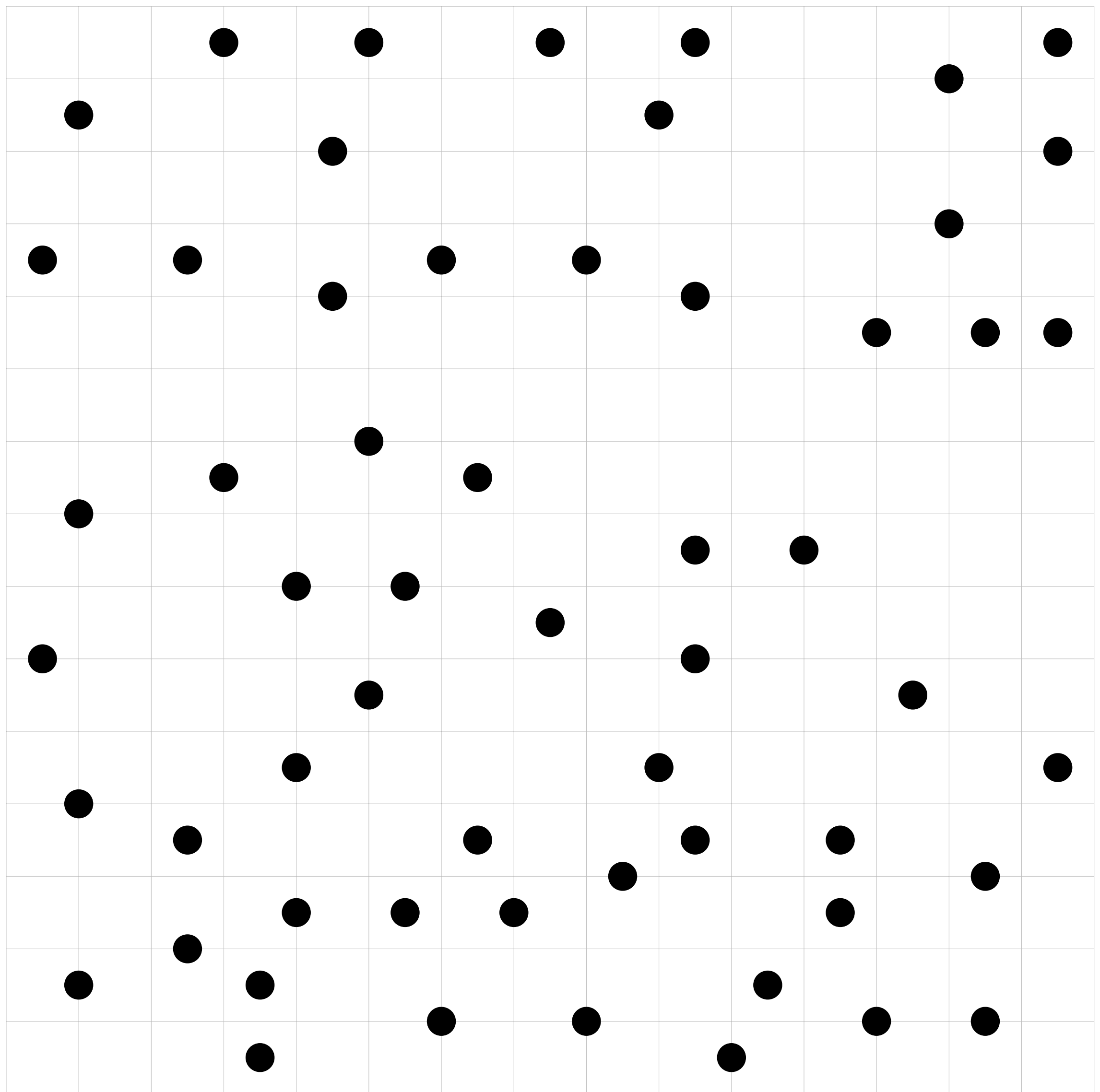


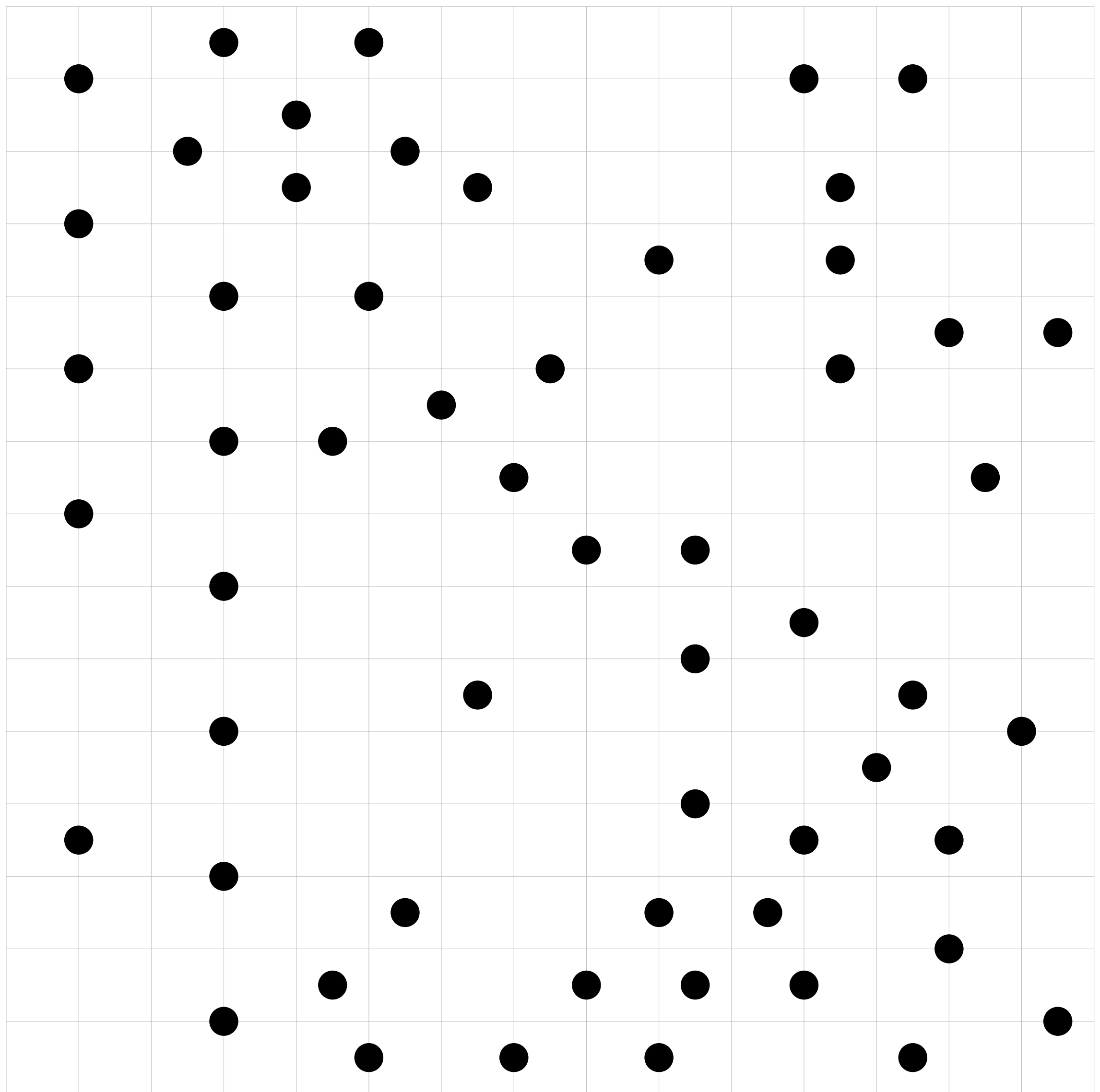


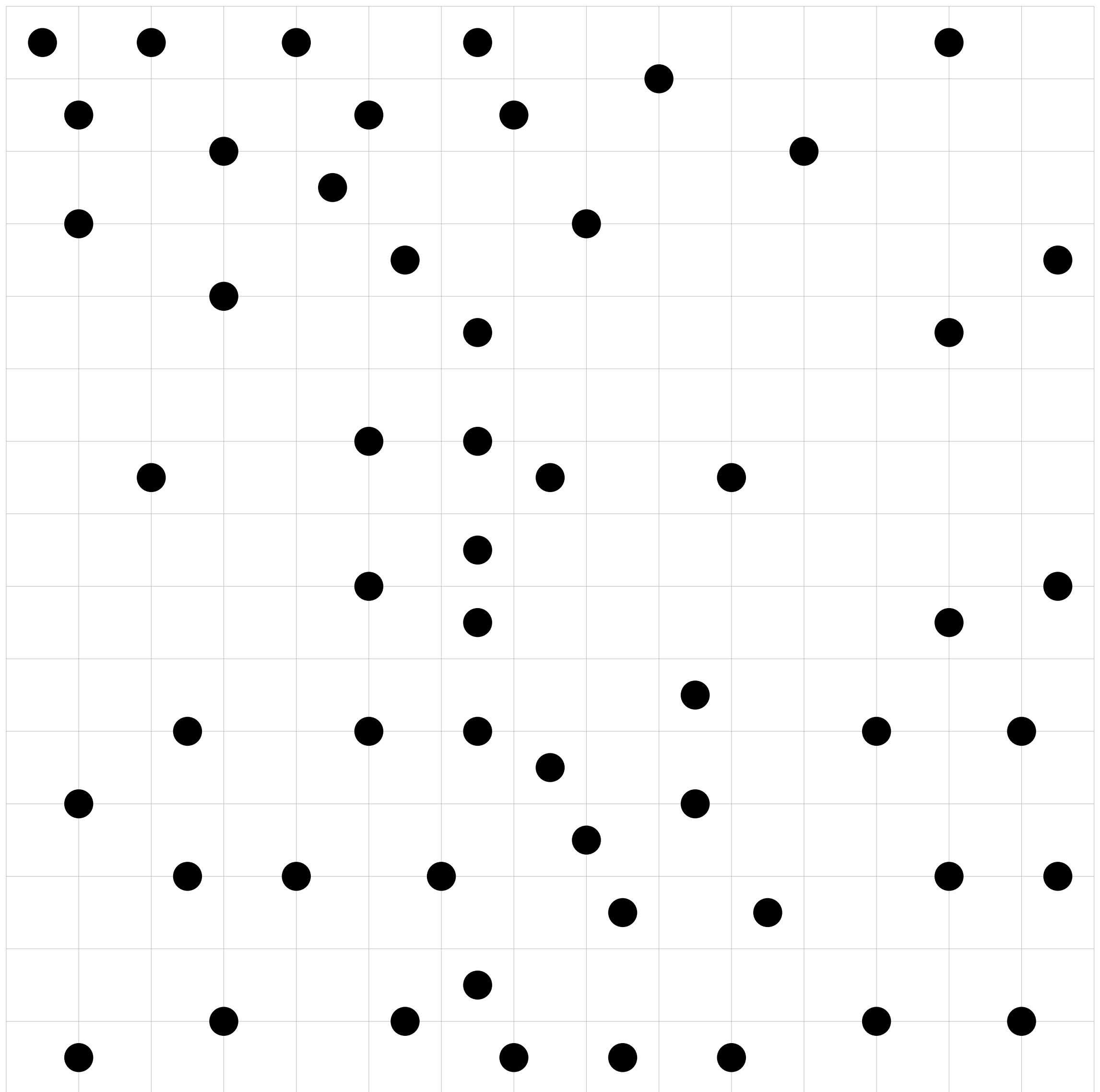
7.3.3 n=15

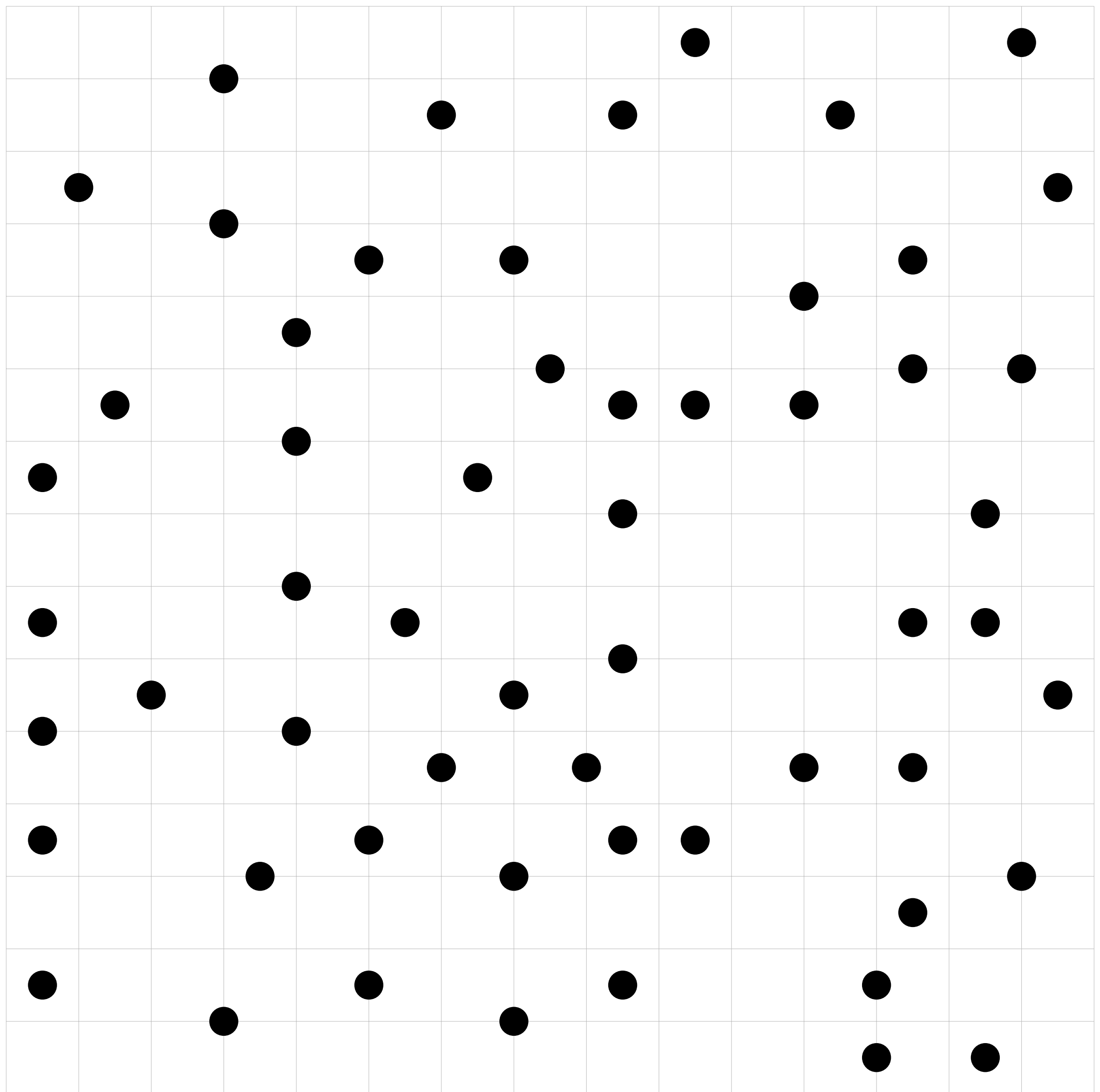


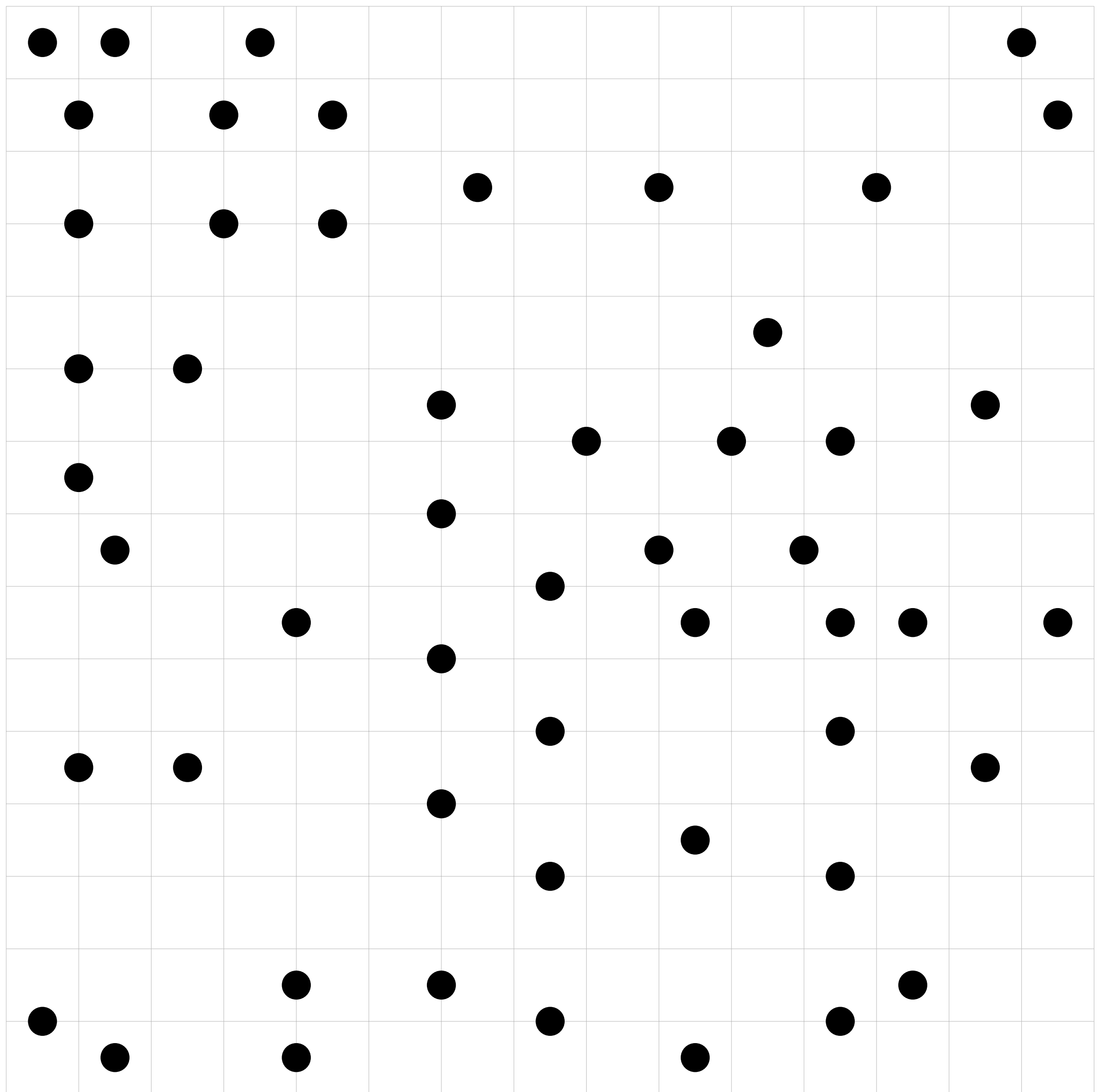


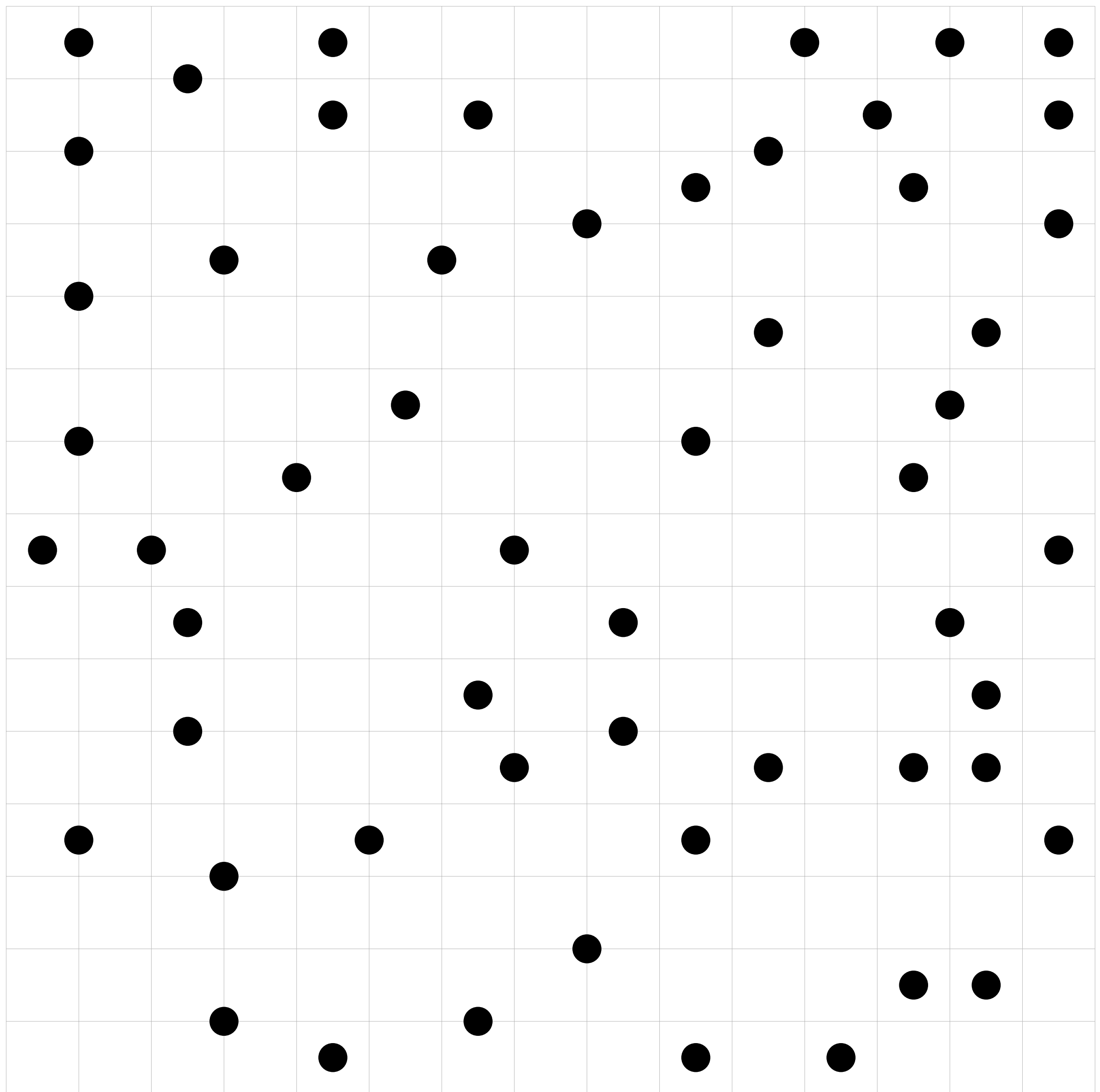


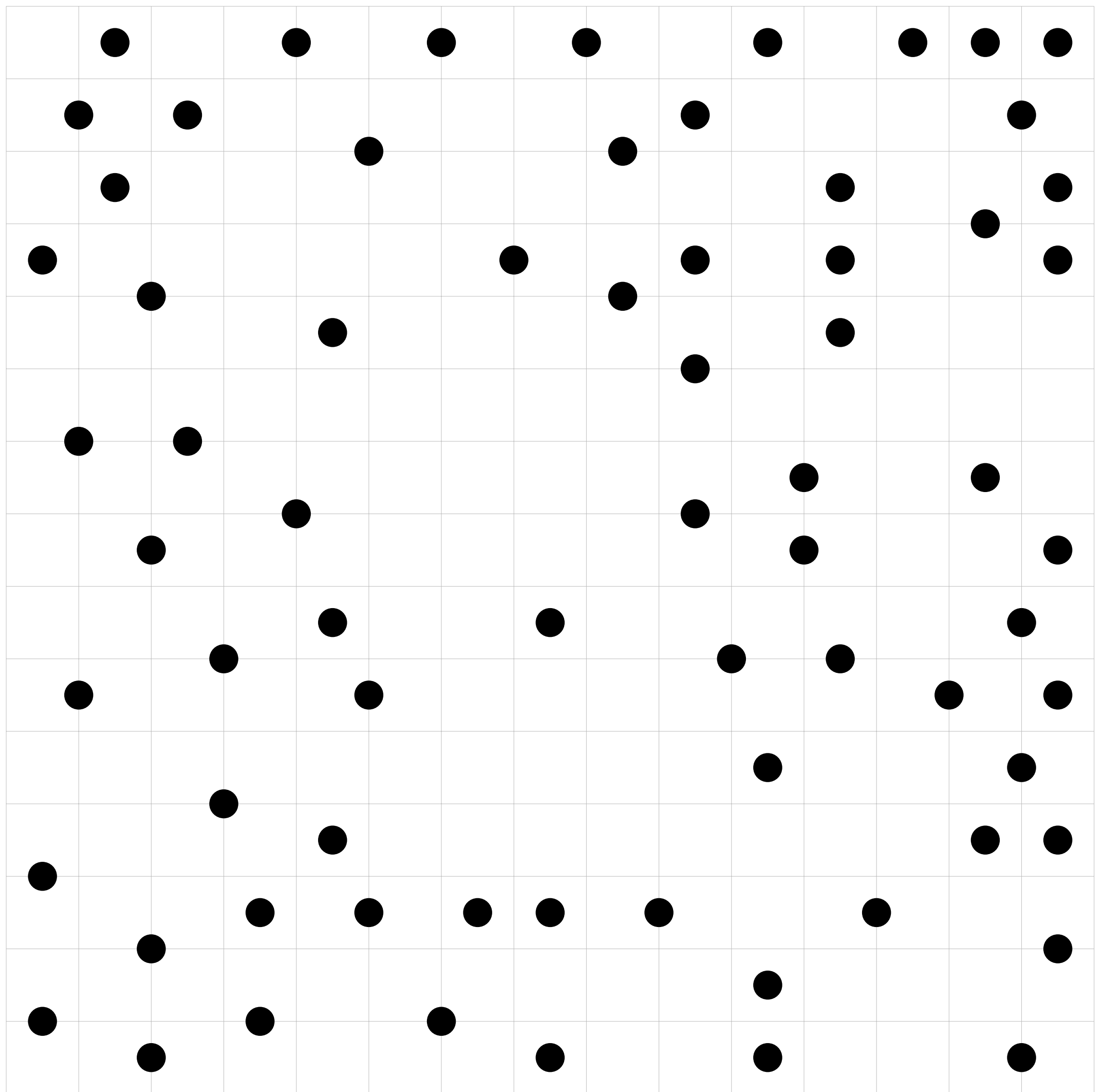


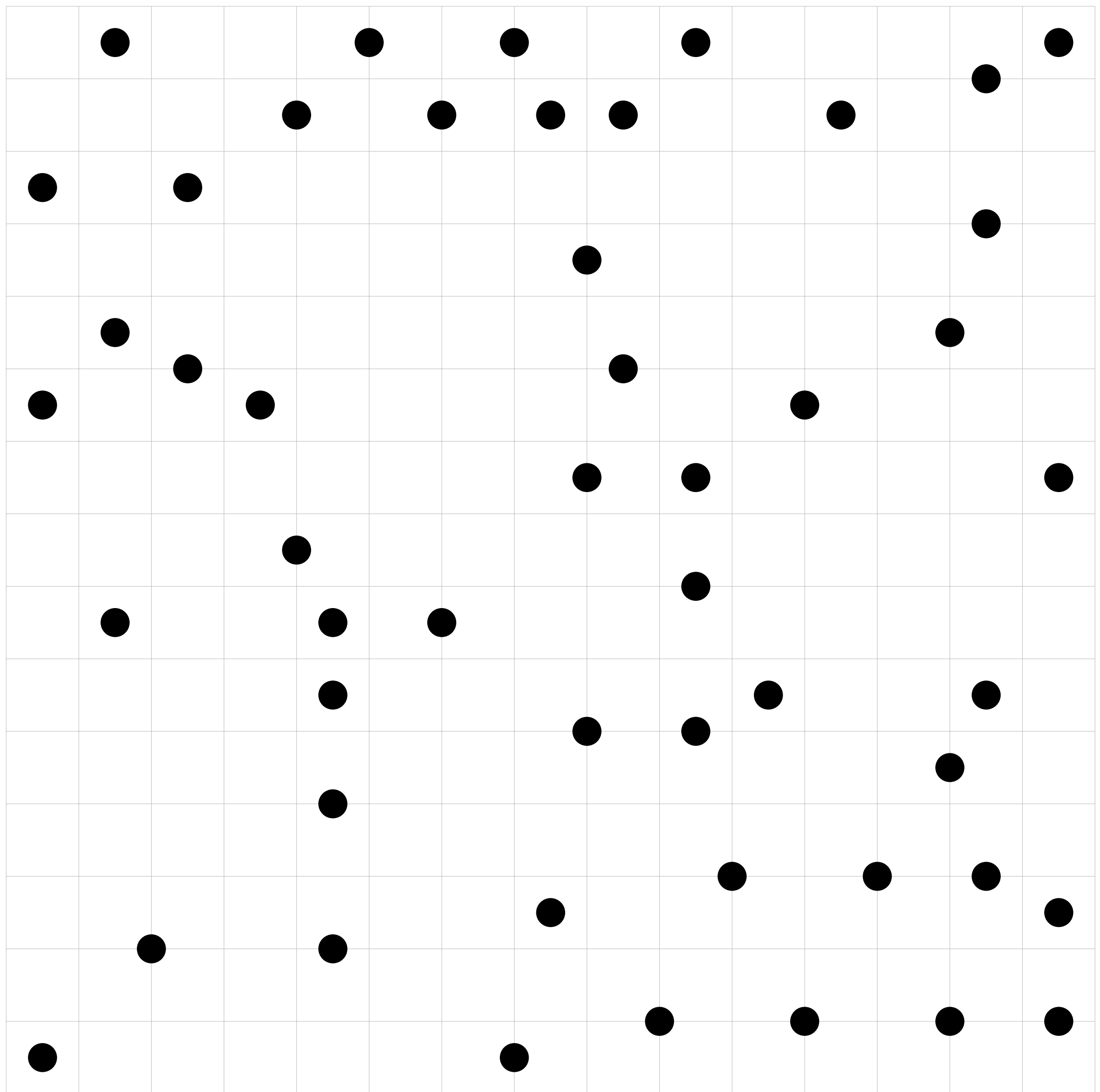


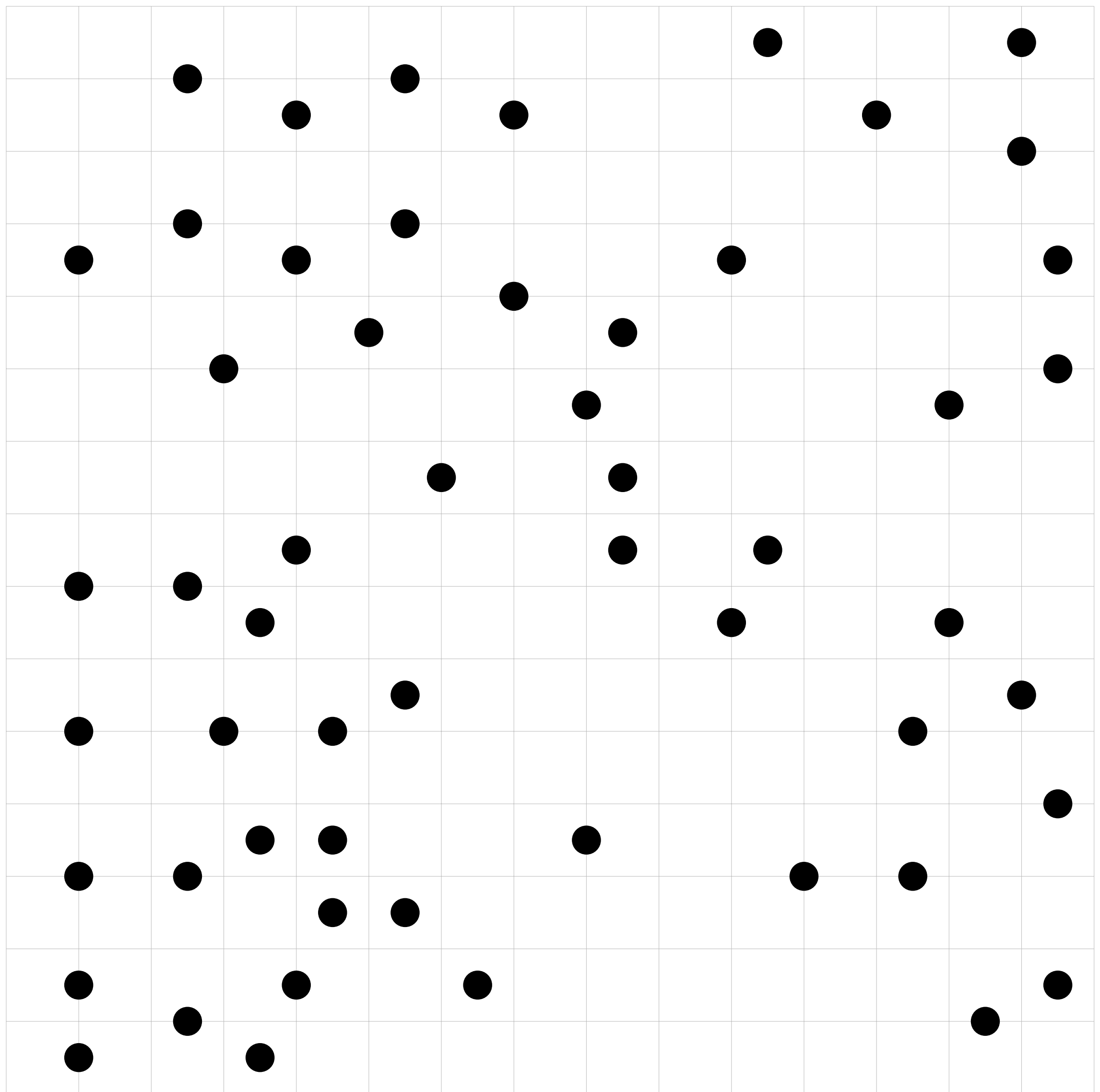


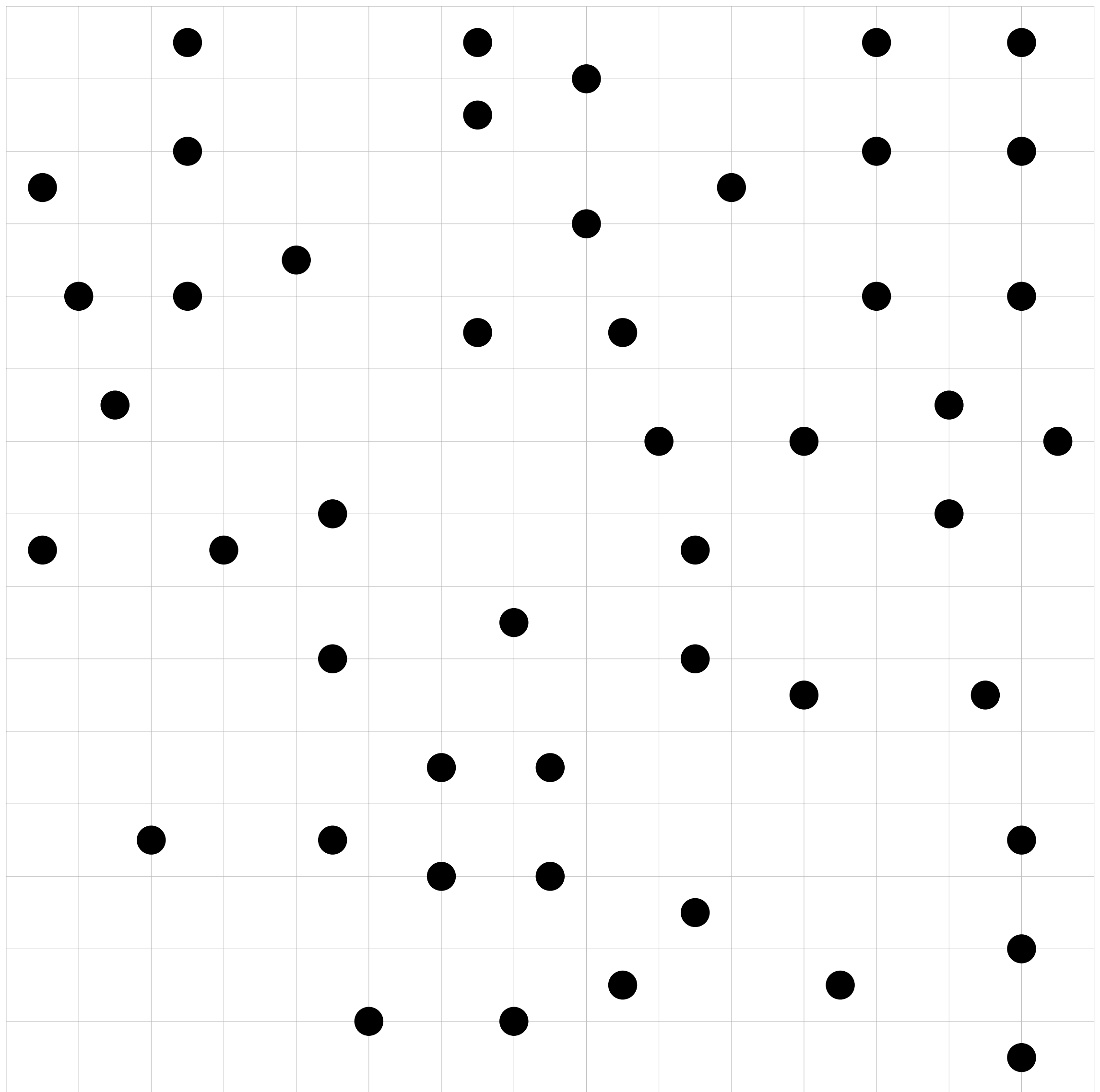


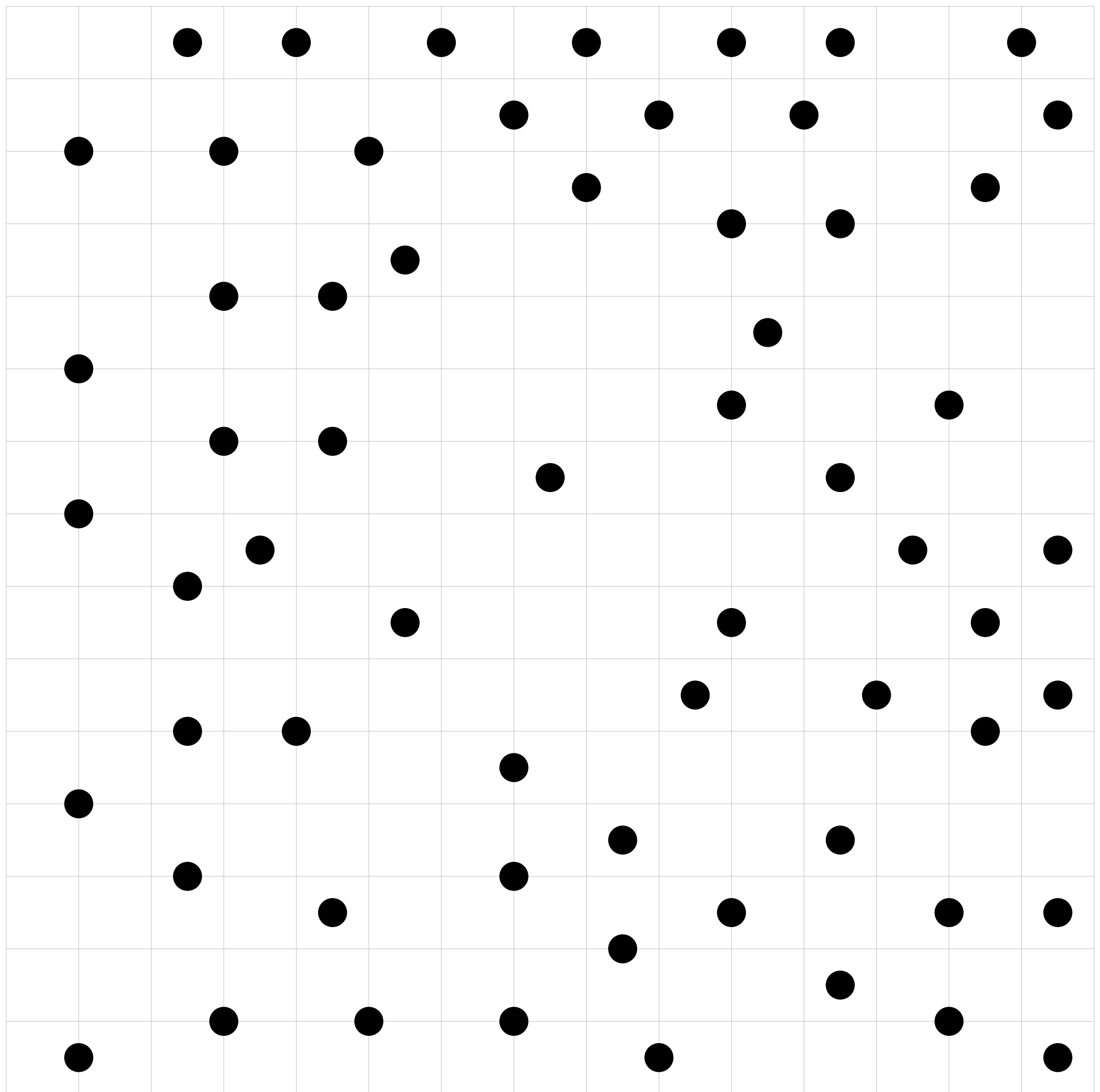


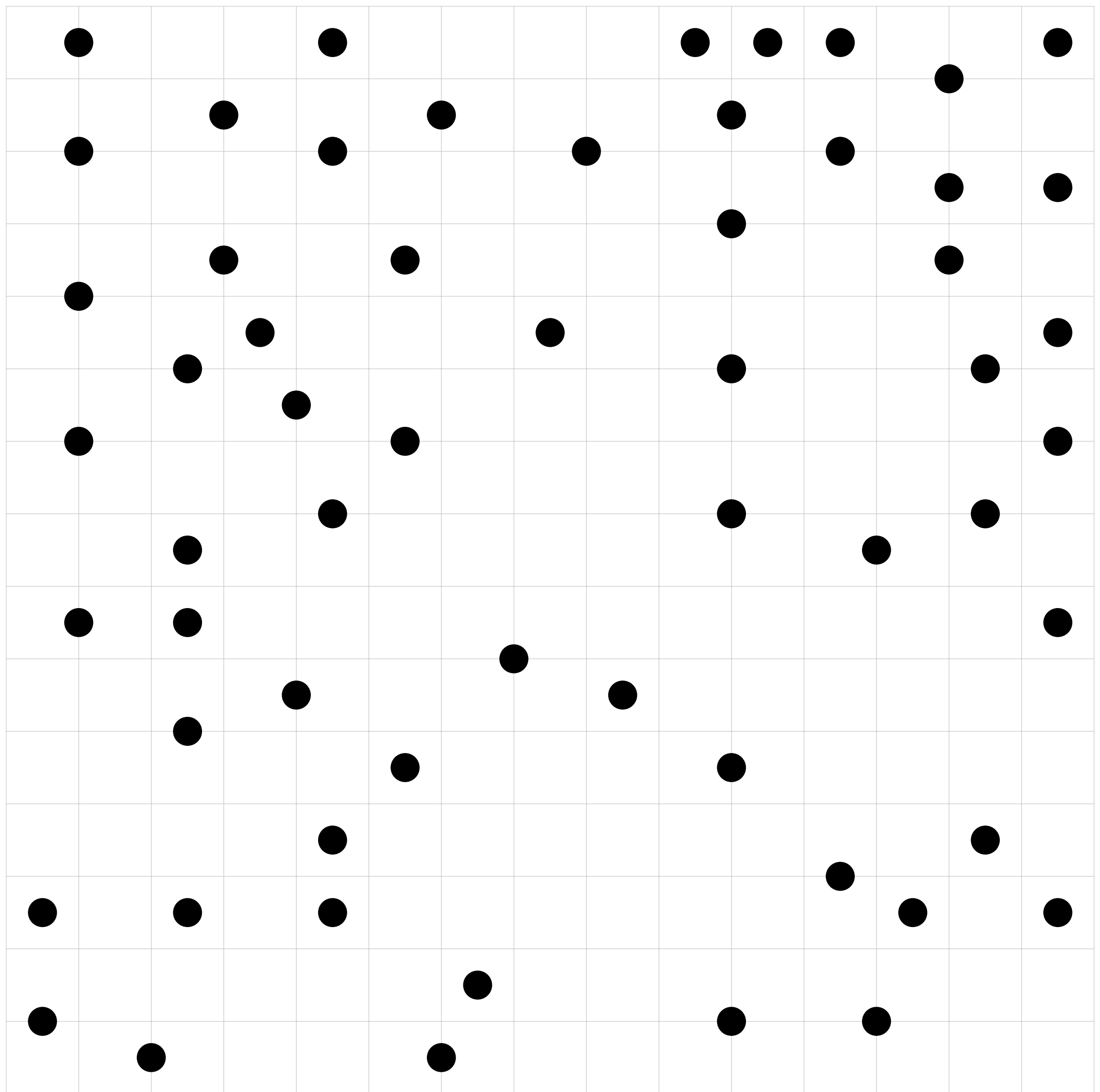


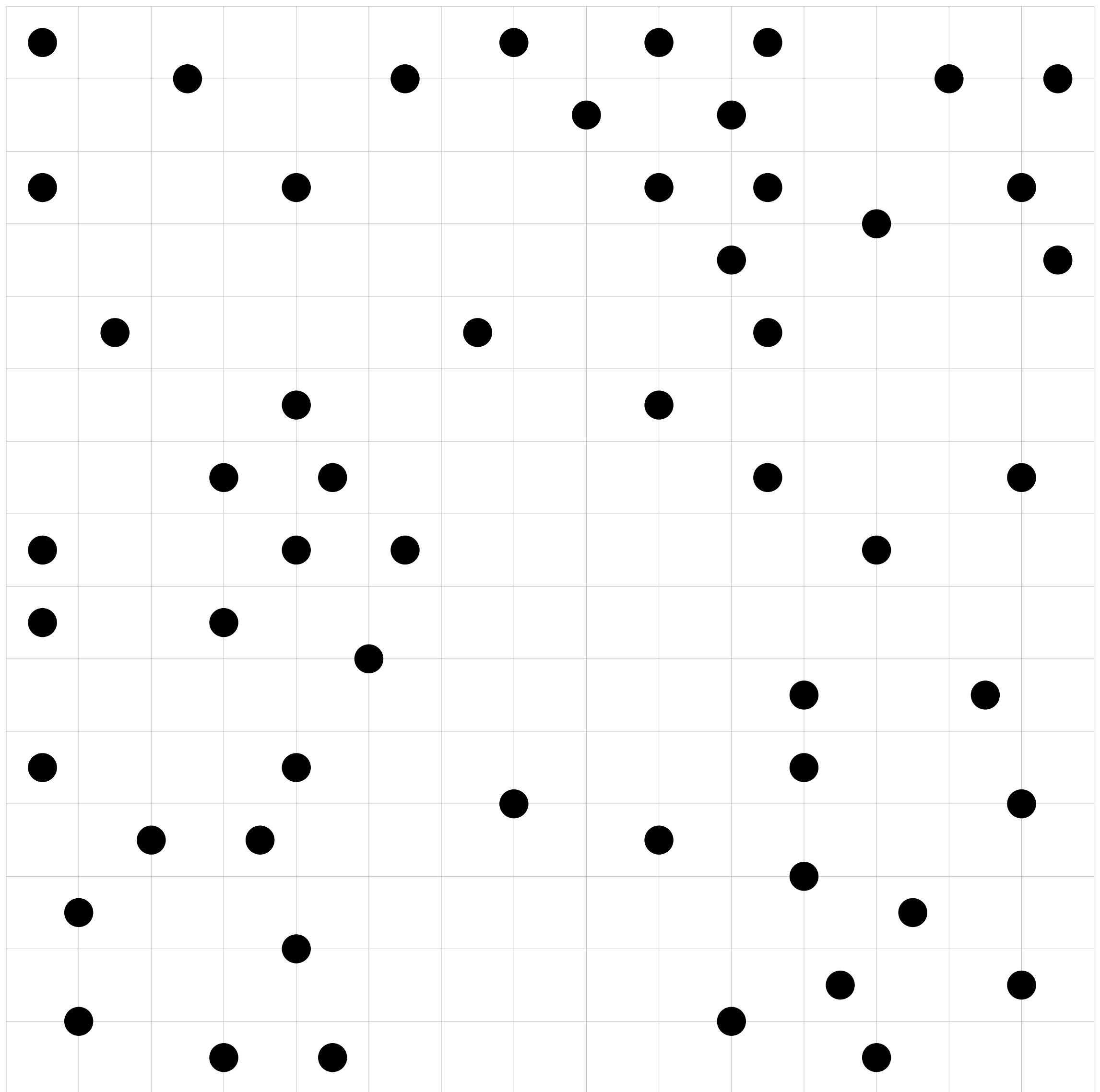


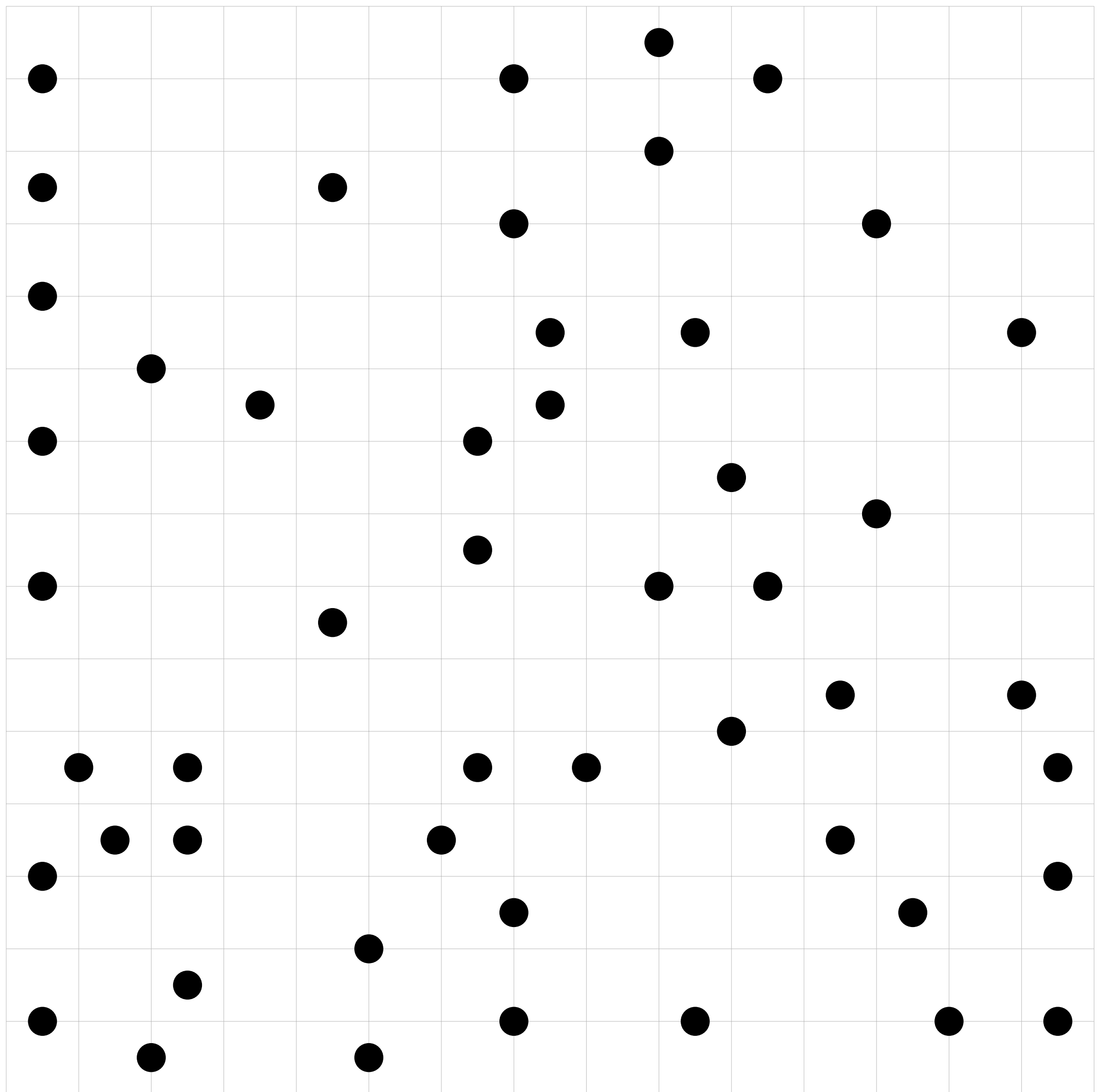


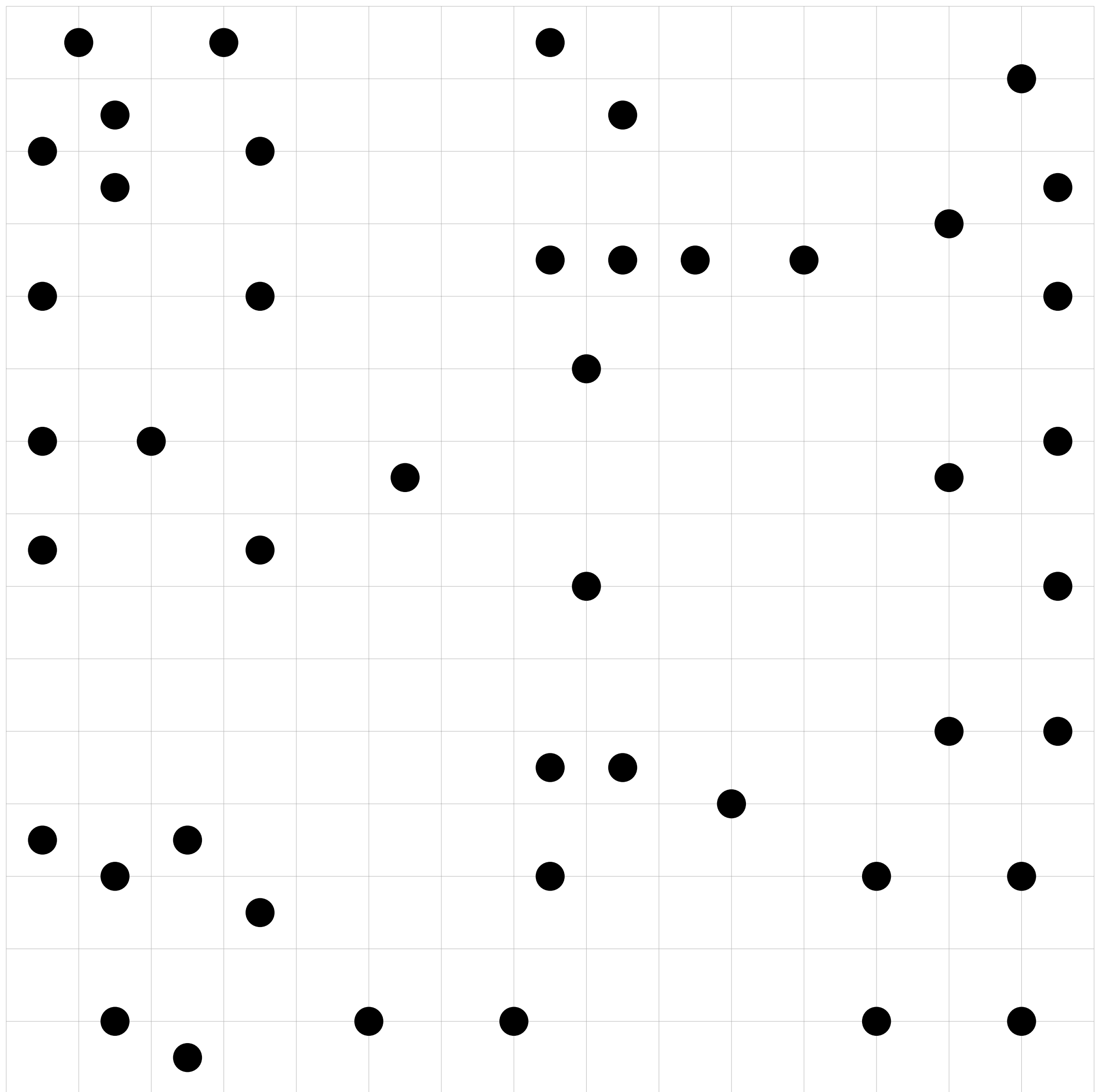


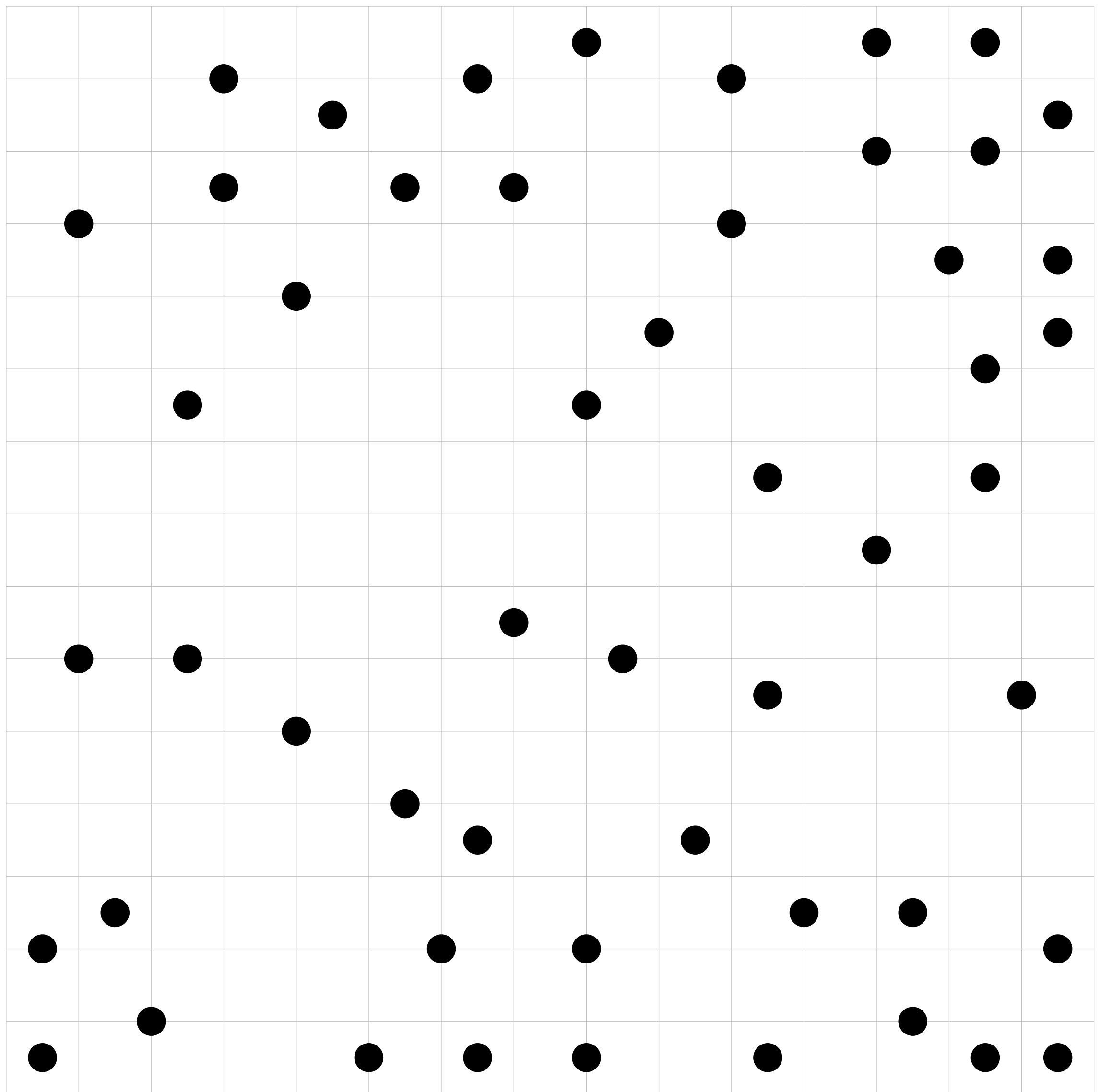


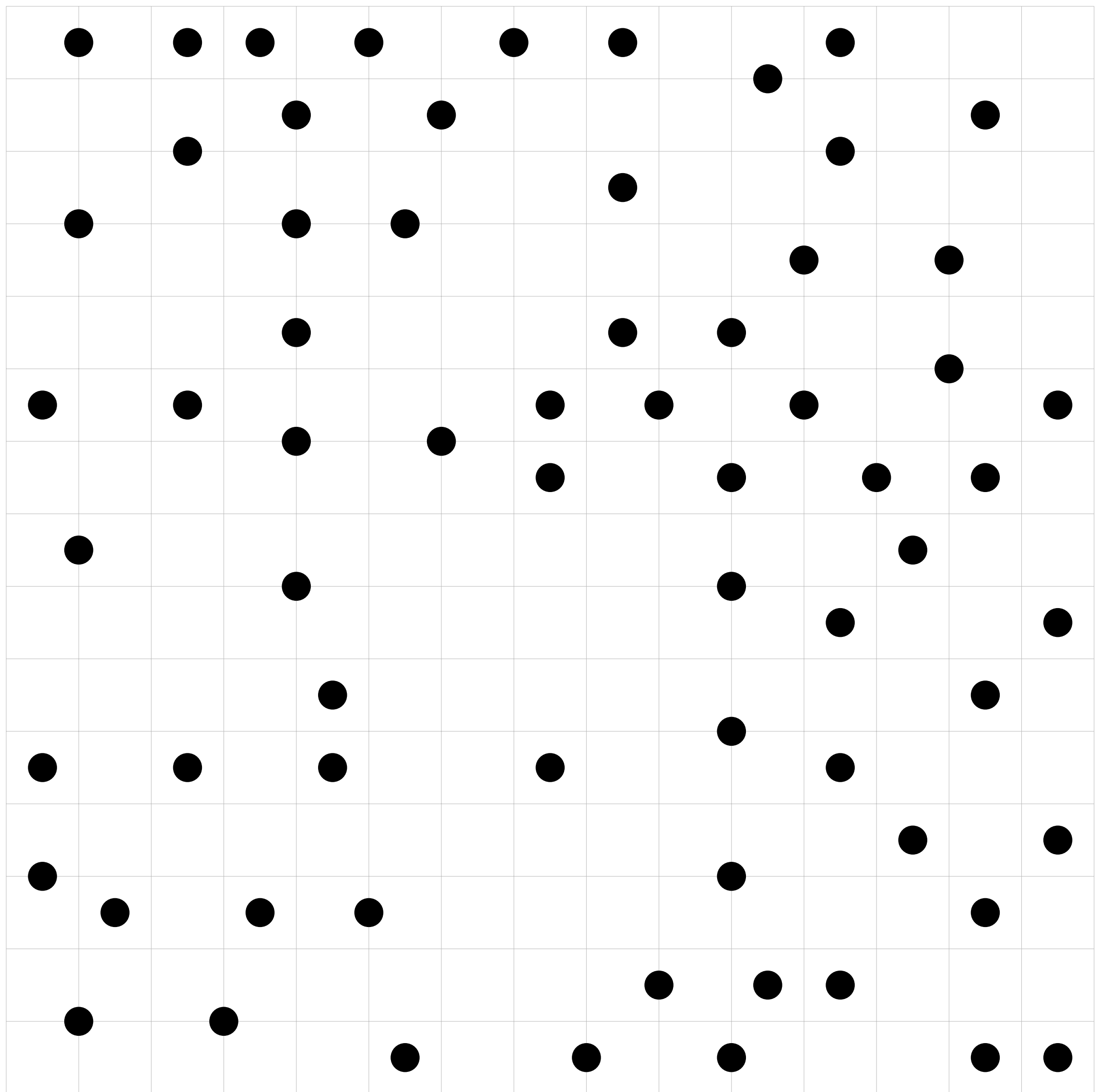


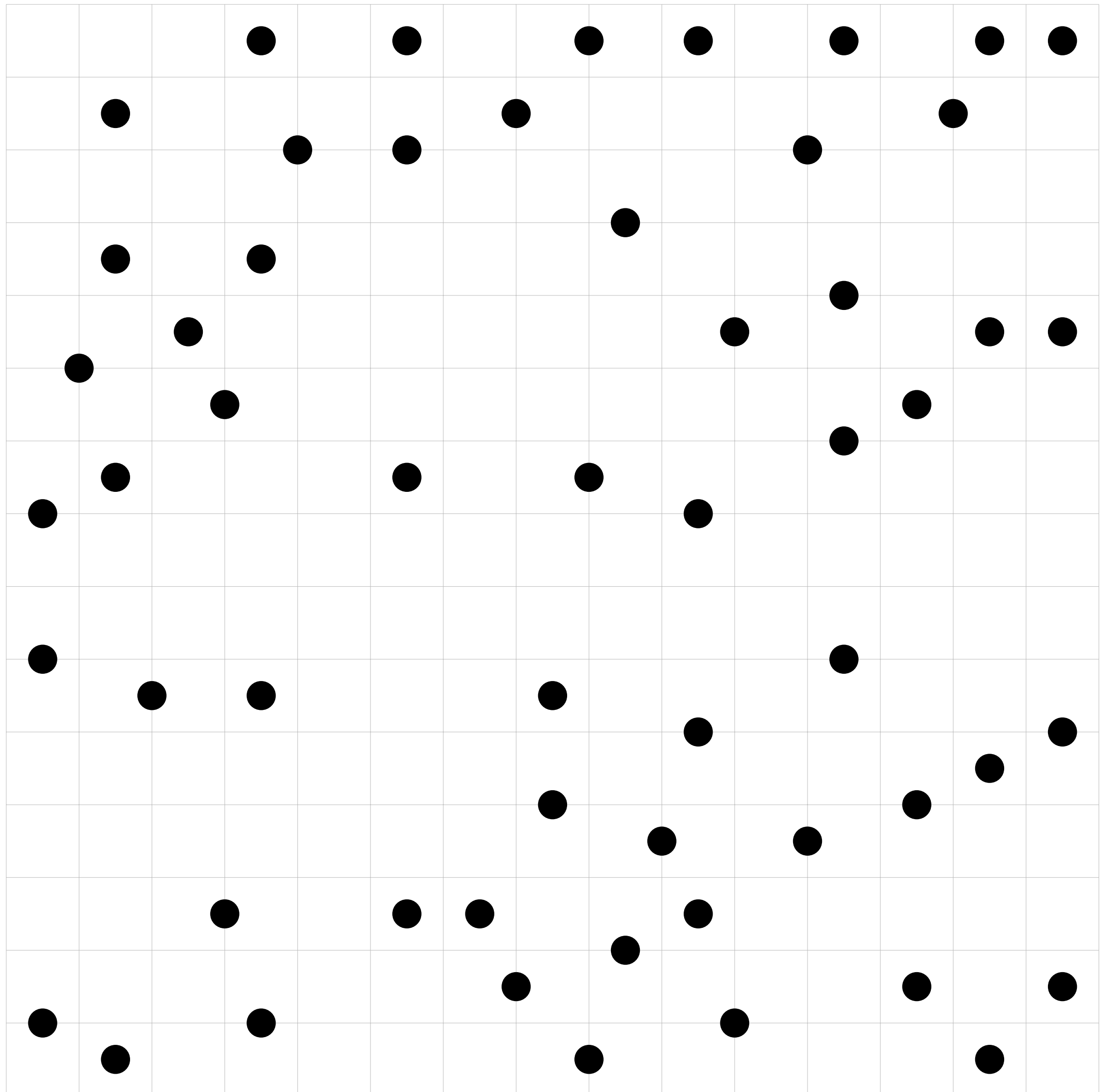












8 SOURCES

1. https://content.time.com/time/arts/article/0,8599,1205307,00.html

2. https://content.time.com/time/specials/packages/article /0,28804,1975813_1975838_1976198,00.html

3. https://www.nikoli.co.jp/en/puzzles/tentai_show/

4. https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/galaxies.html