June 26, 2014 / Mike Bostock

Visualizing Algorithms

> *The power of the unaided mind is highly overrated… The real powers come from devising external aids that enhance cognitive abilities.* —Donald Norman

Algorithms are a fascinating use case for visualization. To visualize an algorithm, we don't merely fit data to a chart; there is no primary dataset. Instead there are logical rules that describe behavior. This may be why algorithm visualizations are so unusual, as designers experiment with novel forms to better communicate. This is reason enough to study them.

But algorithms are also a reminder that visualization is more than a tool for finding patterns in data. Visualization leverages the human visual system to augment human intellect: we can use it to better understand these important abstract processes, and perhaps other things, too.

This is an adaption of my talk at Eyeo 2014. A video of the talk is available on Vimeo. (Thanks, Eyeo folks!)

Sampling

Before I can explain the first algorithm, I first need to explain the problem it addresses.

Van Gogh's *The Starry Night*

Light — electromagnetic radiation — the light emanating from this screen, traveling through the air, focused by your lens and projected onto the retina — is a continuous signal. To be perceived, we must reduce light to discrete impulses by measuring its intensity and frequency distribution at different points in space.

This reduction process is called *sampling*, and it is essential to vision. You can think of it as a painter applying discrete strokes of color to form an image (particularly in Pointillism or Divisionism). Sampling is further a core concern of computer graphics; for example, to rasterize a 3D scene by raytracing, we must determine where to shoot rays. Even resizing an image requires sampling.

Sampling is made difficult by competing goals. On the one hand, samples should be evenly distributed so there are no gaps. But we must also avoid repeating, regular patterns, which cause aliasing. This is why you shouldn't wear a finely-striped shirt on camera: the stripes resonate with the grid of pixels in the camera's sensor and cause Moiré patterns.

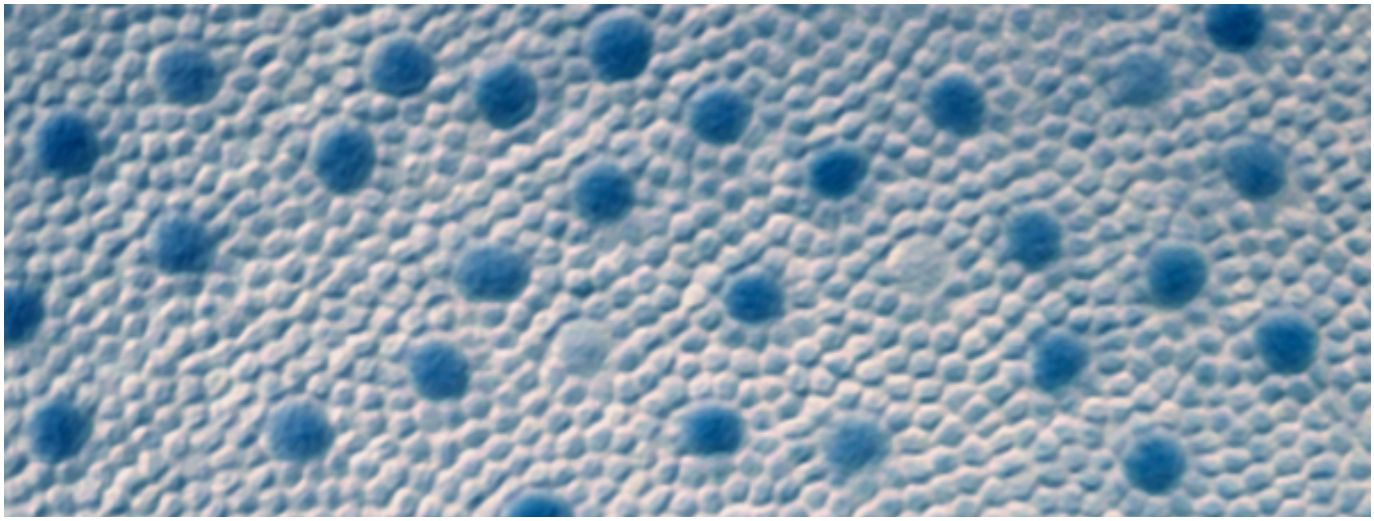
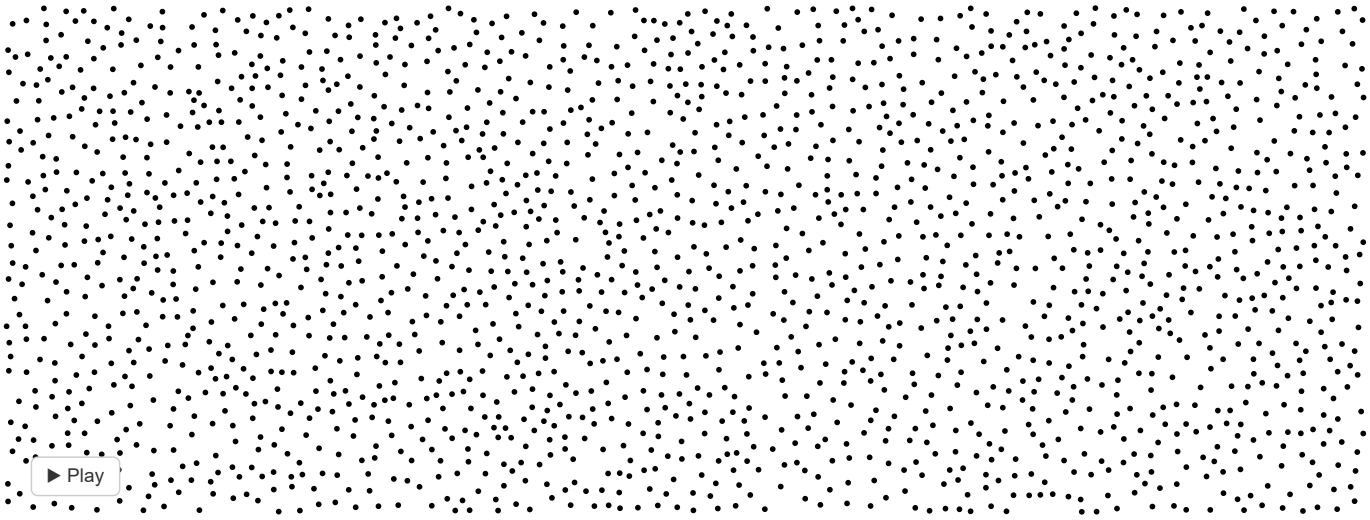


Photo: retinalmicroscopy.com

This micrograph is of the human retina's periphery. The larger *cone* cells detect color, while the smaller *rod* cells improve low-light vision.

The human retina has a beautiful solution to sampling in its placement of photoreceptor cells. The cells cover the retina densely and evenly (with the exception of the blind spot over the optic nerve), and yet the cells' relative positions are irregular. This is called a Poisson-disc distribution because it maintains a minimum distance between cells, avoiding occlusion and thus wasted photoreceptors.

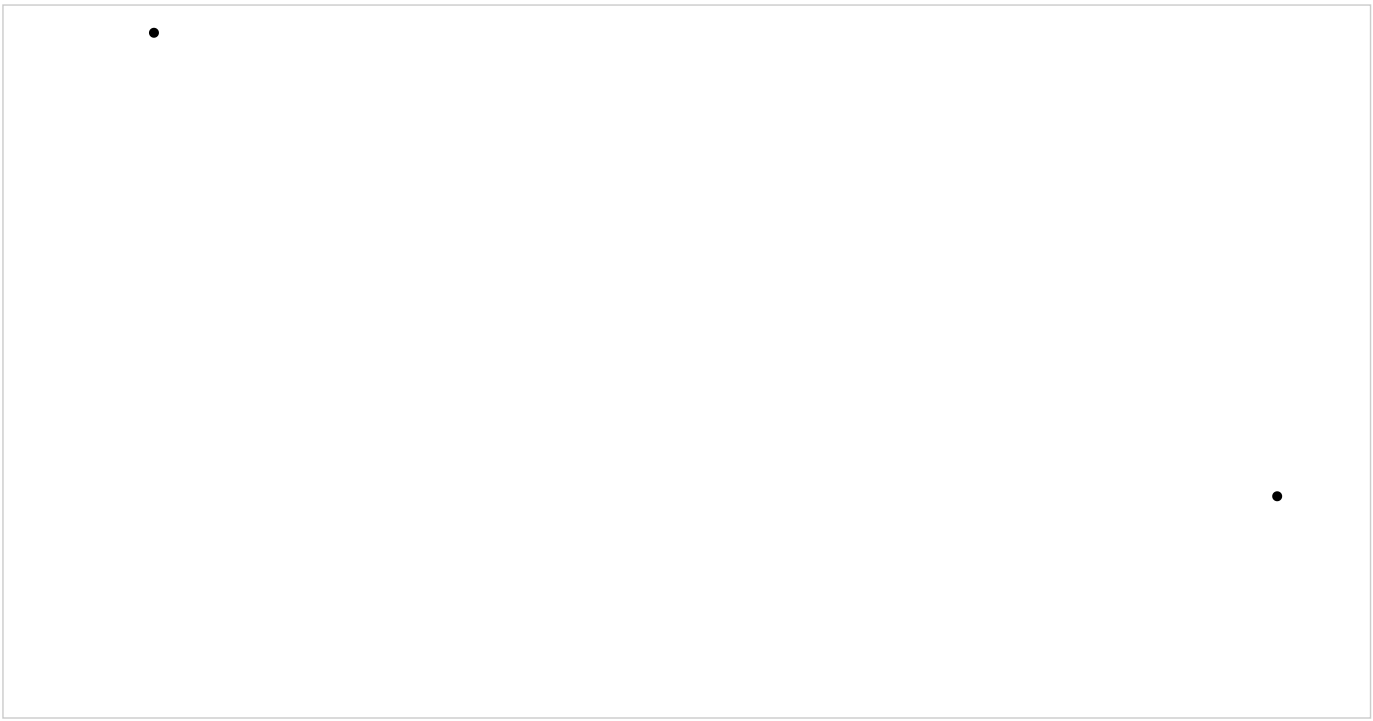
Unfortunately, creating a Poisson-disc distribution is hard. (More on that in a bit.) So here's a simple approximation known as Mitchell's best-candidate algorithm.



Best-candidate

You can see from these dots that best-candidate sampling produces a pleasing random distribution. It's not without flaws: there are too many samples in some areas (oversampling), and not enough in other areas (undersampling). But it's reasonably good, and just as important, easy to implement.

Here's how it works:



Best-candidate

For each new sample, the best-candidate algorithm generates a fixed number of candidates, shown in gray. (Here, that number is 10.) Each candidate is chosen uniformly from the sampling area.

The *best candidate*, shown in red, is the one that is farthest away from all previous samples, shown in black. The distance from each candidate to the closest sample is shown by the associated line and circle: notice that there are no other samples inside the gray or red circles. After all candidates are created and distances measured, the best candidate becomes the new sample, and the remaining candidates are discarded.

Now here's the code:

```
function sample() {
  var bestCandidate, bestDistance = 0;
  for (var i = 0; i < numCandidates; ++i) {
    var c = [Math.random() * width, Math.random() * height],
        d = distance(findClosest(samples, c), c);
    if (d > bestDistance) {
      bestDistance = d;
      bestCandidate = c;
    }
  }
  return bestCandidate;
}
```

As I explained the algorithm above, I will let the code stand on its own. (And the purpose of this essay is to let you study code through visualization, besides.) But I will clarify a few details:

The external `numCandidates` defines the number of candidates to create per sample. This parameter lets you trade-off speed with quality. The lower the number of candidates, the faster it runs. Conversely, the higher the number of candidates, the better the sampling quality.

The `distance` function is simple geometry:

```
function distance(a, b) {
  var dx = a[0] - b[0],
      dy = a[1] - b[1];
  return Math.sqrt(dx * dx + dy * dy);
}
```

You can omit the `sqrt` here, if you want, since it's a monotonic function and doesn't change the

determination of the best candidate.

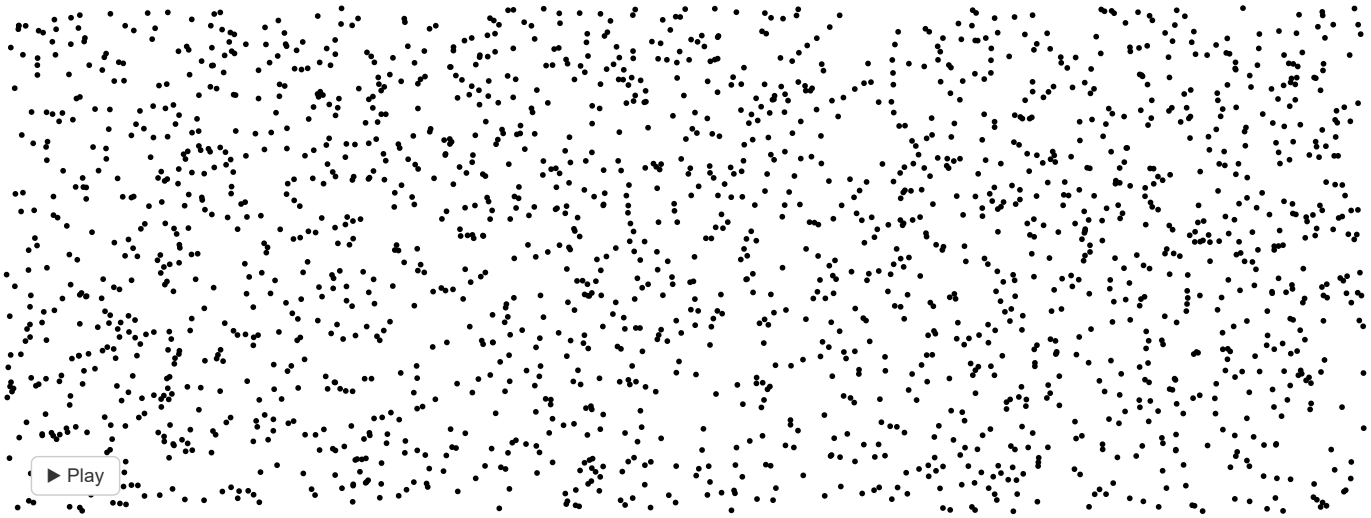
The `findClosest` function returns the closest sample to the current candidate. This can be done by brute force, iterating over every existing sample. Or you can accelerate the search, say by using a quadtree. Brute force is simple to implement but very slow (quadratic time, in O-notation). The accelerated approach is much faster, but more work to implement.

Speaking of trade-offs: when deciding whether to use an algorithm, we evaluate it not in a vacuum but against other approaches. And as a practical matter it is useful to weigh the complexity of implementation — how long it takes to implement, how difficult it is to maintain — against its performance and quality.

The simplest alternative is uniform random sampling:

```
function sample() {
  return [random() * width, random() * height];
}
```

It looks like this:



Uniform random

Uniform random is pretty terrible. There is both severe under- and oversampling: many samples are densely-packed, even overlapping, leading to large empty areas. (Uniform random sampling also represents the lower bound of quality for the best-candidate algorithm, as when the number of candidates per sample is set to one.)

Dots patterns are one way of showing sample pattern quality, but not the only way. For example, we can attempt to simulate vision under different sampling strategies by coloring an image according to the color of the closest sample. This is, in effect, a Voronoi diagram of the samples where each cell is colored by the associated sample.

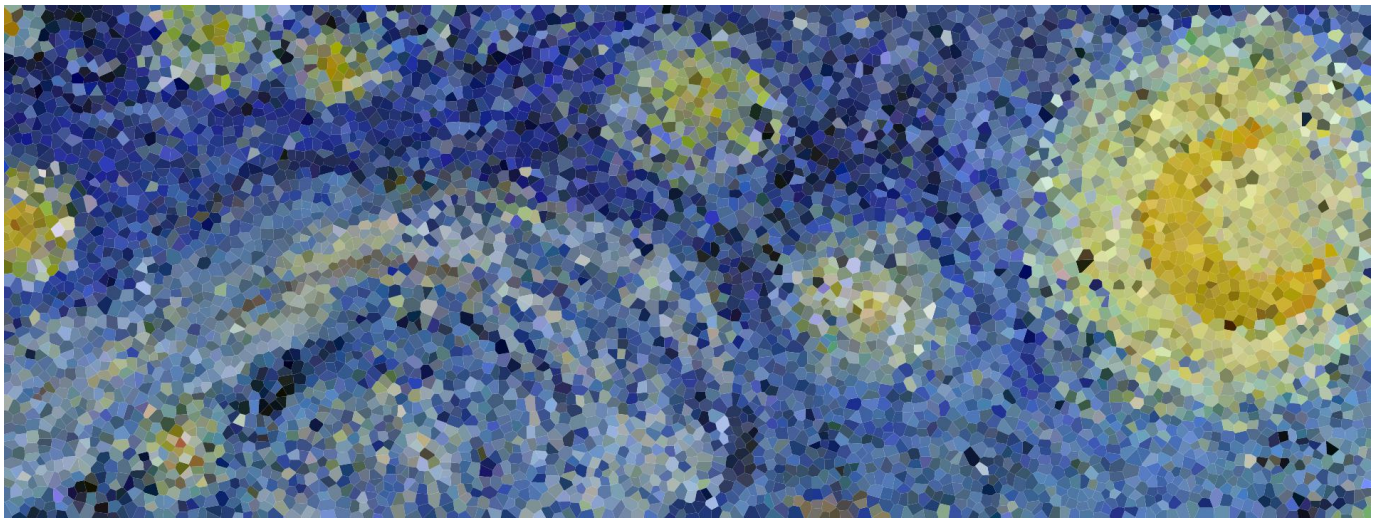
What does *The Starry Night* look like through 6,667 uniform random samples?

Uniform random

Hold down the mouse to compare to the original.

The lackluster quality of this approach is again apparent. The cells vary widely in size, as expected from the uneven sample distribution. Detail has been lost because densely-packed samples (small cells) are underutilized. Meanwhile, sparse samples (large cells) introduce noise by exaggerating rare colors, such as the pink star in the bottom-left.

Now observe best-candidate sampling:



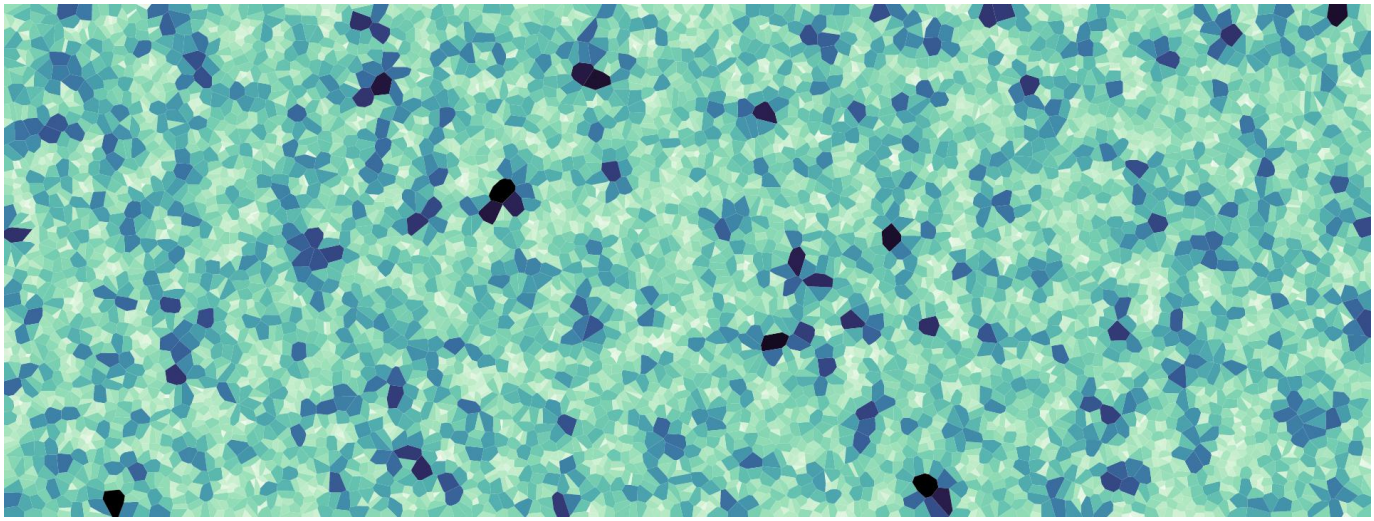
Best-candidate

Hold down the mouse to compare to the original.

Much better! Cells are more consistently sized, though still randomly placed. Despite the quantity of samples (6,667) remaining constant, there is substantially more detail and less noise thanks to their even distribution. If you squint, you can almost make out the original brush strokes.

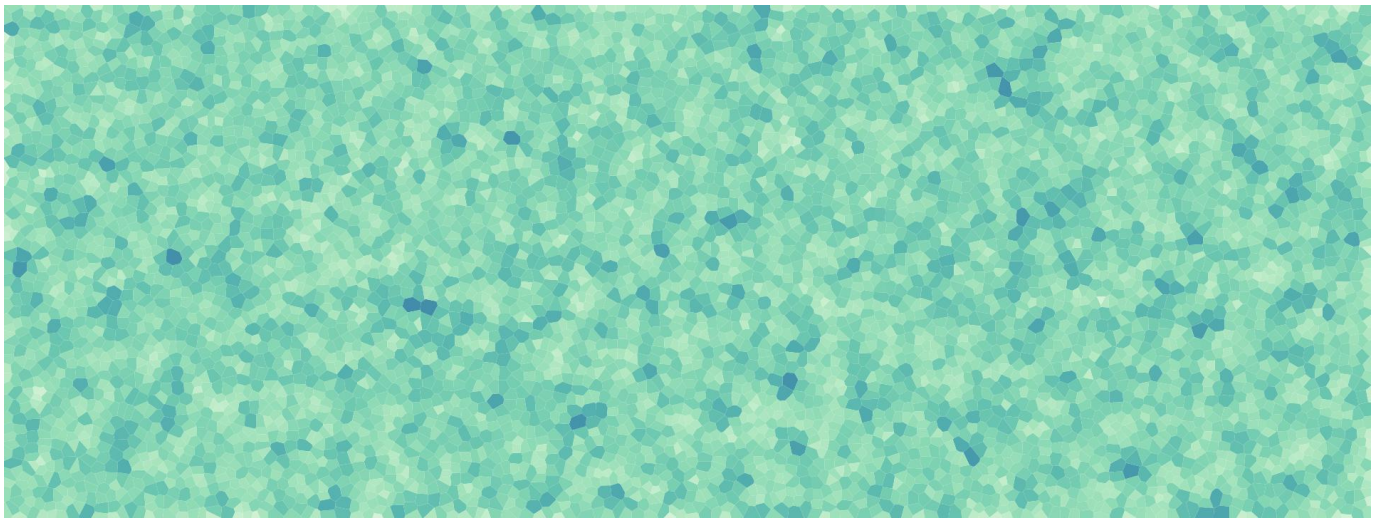
We can use Voronoi diagrams to study sample distributions more directly by coloring each cell according to its area. Darker cells are larger, indicating sparse sampling; lighter cells are smaller, indicating dense sampling. The optimal pattern has nearly-uniform color while retaining irregular sample positions. (A histogram showing cell area distribution would also be nice, but the Voronoi has the advantage that it shows sample position simultaneously.)

Here are the same 6,667 uniform random samples:



Uniform random

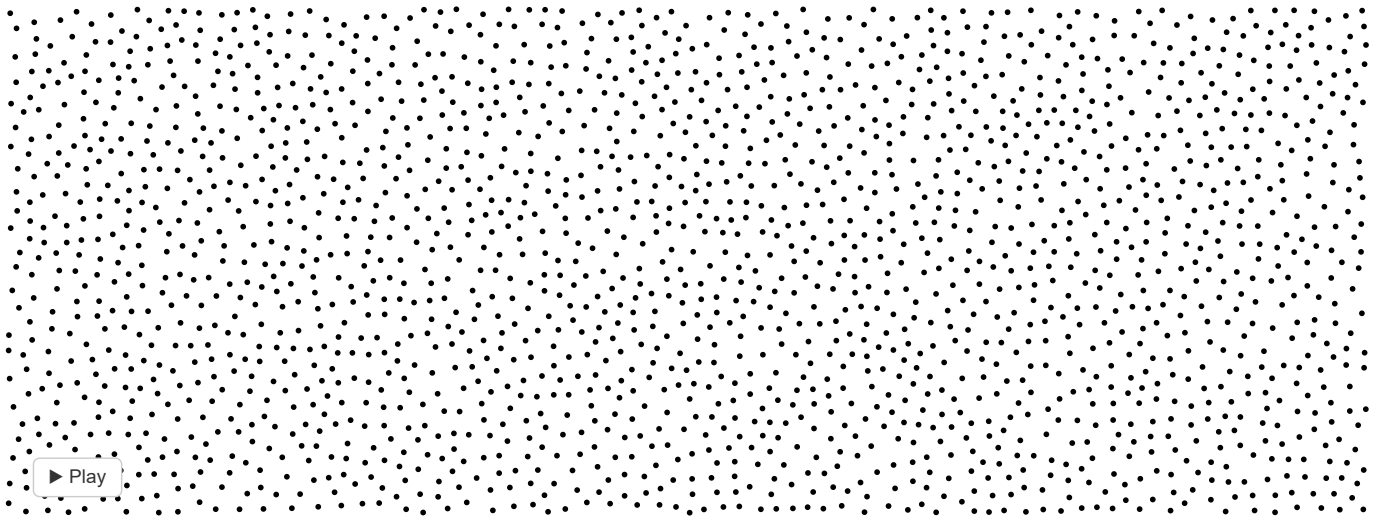
The black spots — large gaps between samples — would be localized deficiencies in vision due to undersampling. The same number of best-candidate samples exhibits much less variation in cell area, and thus more consistent coloration:



Best-candidate

Can we do better than best-candidate? Yes! Not only can we produce a *better* sample distribution with a different algorithm, but this algorithm is *faster* (linear time). It's at least as easy to implement as best-candidate. And this algorithm even scales to arbitrary dimensions.

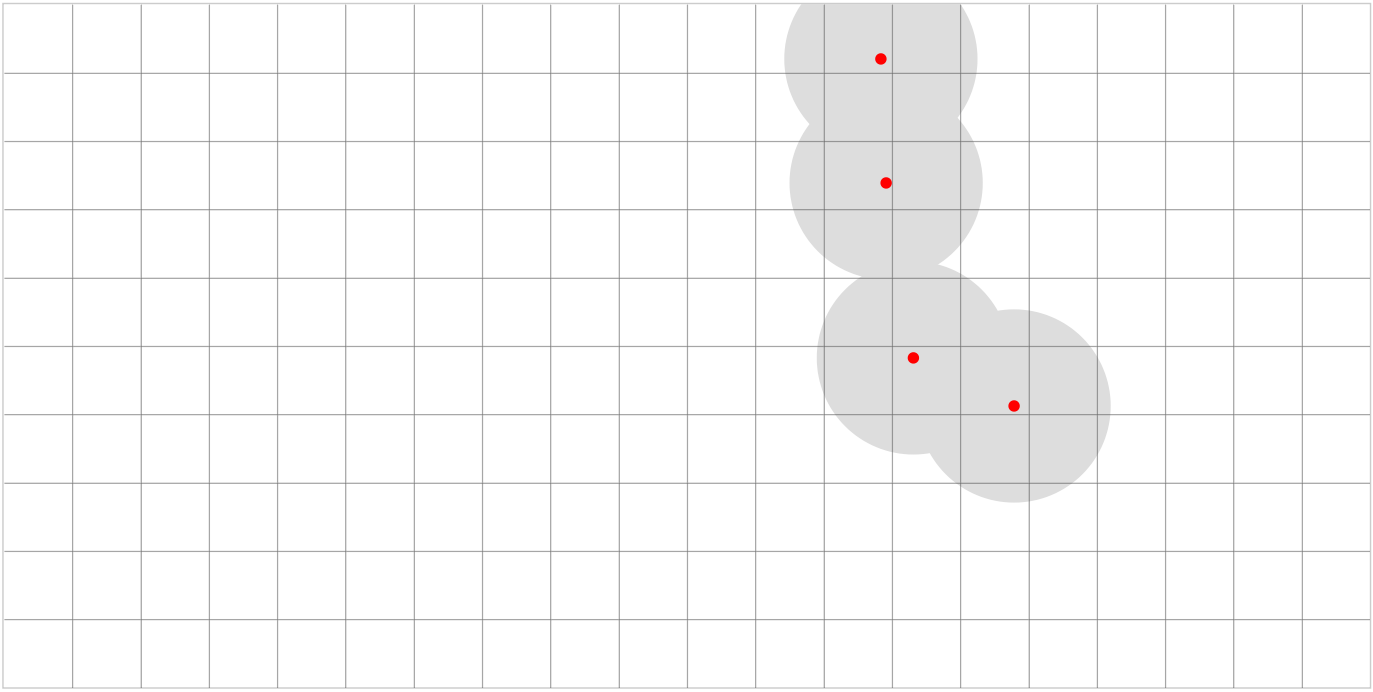
This wonder is called Bridson's algorithm for Poisson-disc sampling, and it looks like this:



Poisson-disc

This algorithm functions visibly differently than the other two: it builds incrementally from existing samples, rather than scattering new samples randomly throughout the sample area. This gives its progression a quasi-biological appearance, like cells dividing in a petri dish. Notice, too, that no samples are too close to each other; this is the minimum-distance constraint that defines a Poisson-disc distribution, enforced by the algorithm.

Here's how it works:



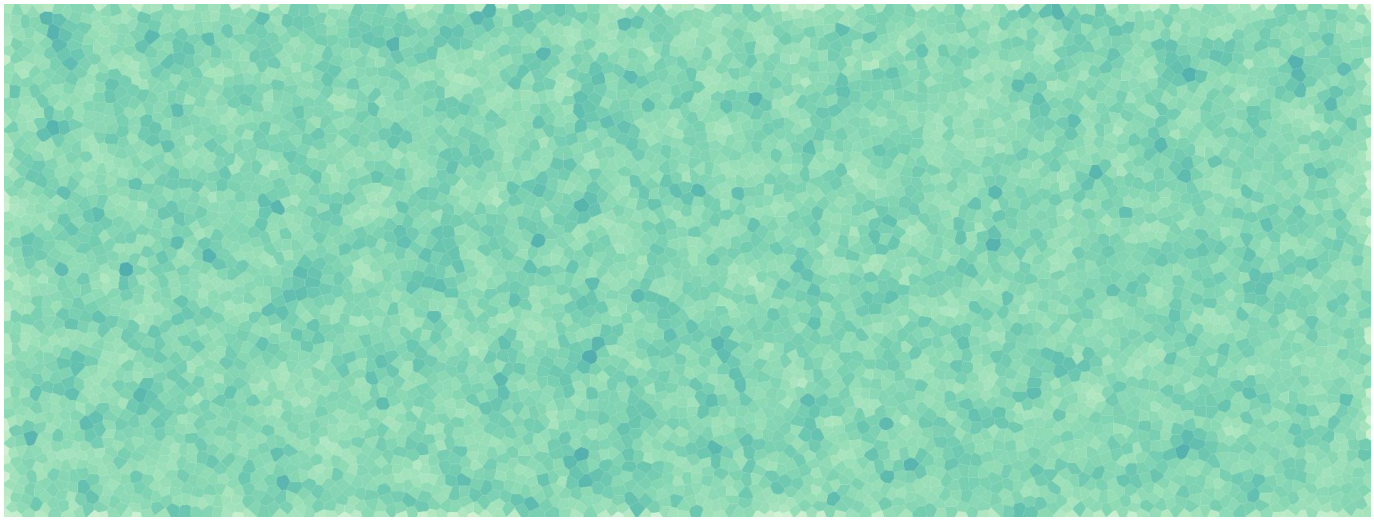
Poisson-disc

Red dots represent "active" samples. At each iteration, one is selected randomly from the set of all active samples. Then, some number of candidate samples (shown as hollow black dots) are randomly generated within an annulus surrounding the selected sample. The annulus extends from radius r to $2r$, where r is the minimum-allowable distance between samples.

Candidate samples within distance r from an existing sample are rejected; this "exclusion zone" is shown in gray, along with a black line connecting the rejected candidate to the nearby existing sample. A grid accelerates the distance check for each candidate. The grid size $r/\sqrt{2}$ ensures each cell can contain at most one sample, and only a fixed number of neighboring cells need to be checked.

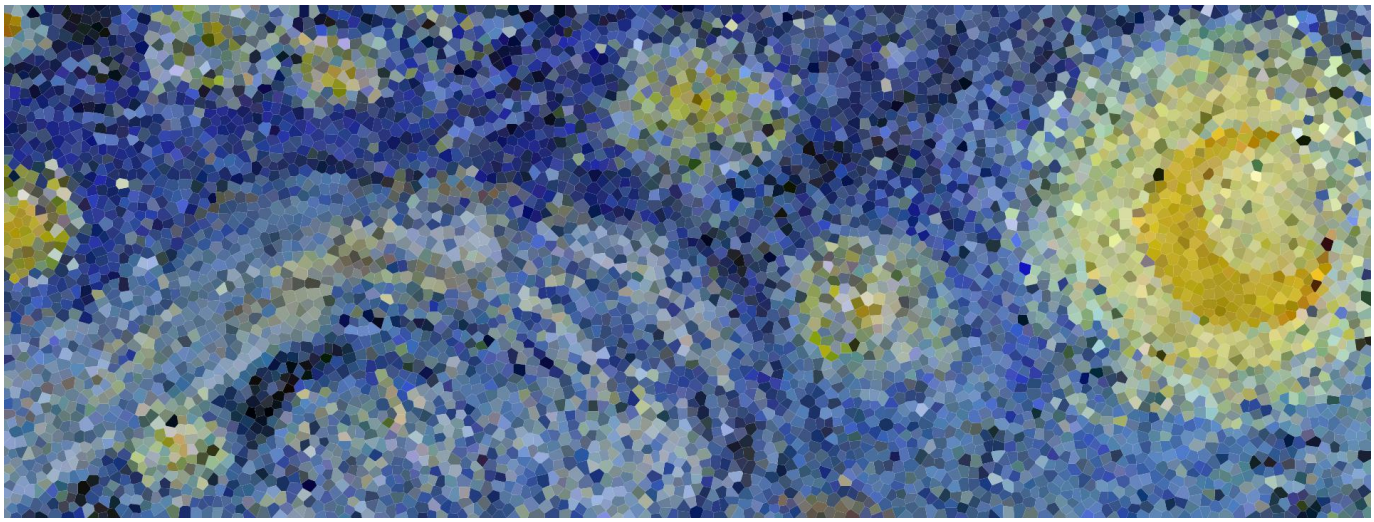
If a candidate is acceptable, it is added as a new sample, and a new active sample is randomly selected. If none of the candidates are acceptable, the selected active sample is marked as inactive (changing from red to black). When no samples remain active, the algorithm terminates.

The area-as-color Voronoi diagram shows Poisson-disc sampling's improvement over best-candidate, with no dark-blue or light-yellow cells:



Poisson-disc

The Starry Night under Poisson-disc sampling retains the greatest amount of detail and the least noise. It is reminiscent of a beautiful Roman mosaic:



Poisson-disc

Hold down the mouse to compare to the original.

Now that you've seen a few examples, let's briefly consider why to visualize algorithms.

Entertaining - I find watching algorithms endlessly fascinating, even mesmerizing. Particularly so when randomness is involved. And while this may seem a weak justification, don't underestimate the value of joy! Further, while these visualizations can be engaging even without understanding the underlying algorithm, grasping the importance of the algorithm can give a deeper appreciation.

Teaching - Did you find the code or the animation more helpful? What about pseudocode — that euphemism for code that won't compile? While formal description has its place in unambiguous documentation, visualization can make intuitive understanding more accessible.

Debugging - Have you ever implemented an algorithm based on formal description? It can be hard! Being able to see what your code is doing can boost productivity. Visualization does not supplant the need for tests, but tests are useful primarily for detecting failure and not explaining it. Visualization can also discover unexpected behavior in your implementation, even when the output looks correct. (See Bret Victor's Learnable Programming and Inventing on Principle for excellent related work.)

Learning - Even if you just want to learn for yourself, visualization can be a great way to gain deep understanding. Teaching is one of the most effective ways of learning, and implementing a visualization is like teaching yourself. I find it easier to remember an algorithm intuitively, having seen it, than to memorize code where I am bound to forget small but essential details.

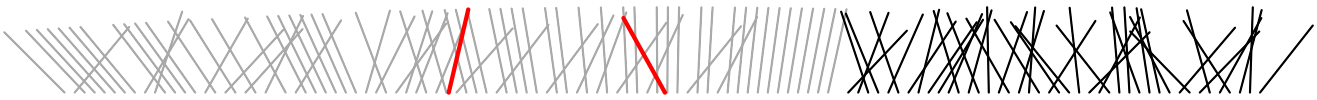
Shuffling

Shuffling is the process of rearranging an array of elements randomly. For example, you might shuffle a deck of cards before dealing a poker game. A good shuffling algorithm is unbiased, where every ordering is equally likely.

The Fisher–Yates shuffle is an optimal shuffling algorithm. Not only is it unbiased, but it runs in linear time, uses constant space, and is easy to implement.

```
function shuffle(array) {
  var n = array.length, t, i;
  while (n) {
    i = Math.random() * n-- | 0; // 0 ≤ i < n
    t = array[n];
    array[n] = array[i];
    array[i] = t;
  }
  return array;
}
```

Above is the code, and below is a visual explanation:



Each line represents a number. Small numbers lean left and large numbers lean right. (Note that you can shuffle an array of anything — not just numbers — but this visual encoding is useful for showing the order of elements. It is inspired by Robert Sedgwick's sorting visualizations in *Algorithms in C*.)

For a more detailed explanation of this algorithm, see my post on the Fisher–Yates shuffle.

The algorithm splits the array into two parts: the right side of the array (in black) is the shuffled section, while the left side of the array (in gray) contains elements remaining to be shuffled. At each step it picks a random element from the left and moves it to the right, thereby expanding the shuffled section by one. The original order on the left does not need to be preserved, so to make room for the new element in the shuffled section, the algorithm can simply swap the element into place. Eventually all elements are shuffled, and the algorithm terminates.

If Fisher–Yates is a good algorithm, what does a bad algorithm look like? Here's one:

```
// DON'T DO THIS!
function shuffle(array) {
  return array.sort(function(a, b) {
    return Math.random() - .5; // ಠ_ಠ
  });
}
```

This approach uses sorting to shuffle by specifying a random comparator function. A comparator defines the order of elements. It takes arguments *a* and *b* — two elements from the array to compare — and returns a value less than zero if *a* is less than *b*, a value greater than zero if *a* is greater than *b*, or zero if *a* and *b* are equal. The comparator is invoked repeatedly during sorting.

Here the comparator returns a random number between -.5 and +.5. The assumption is that this defines a random order, so sorting will jumble the elements randomly and perform a good shuffle.

If you don't specify a comparator to array.sort, elements are ordered lexicographically.

Unfortunately, this assumption is flawed. A random pairwise order (for any two elements) does not establish a random order for a set of elements. A comparator must obey *transitivity*: if $a > b$ and $b > c$,

then $a > c$. But the random comparator returns a random value, violating transitivity and causing the behavior of array.sort to be undefined! You might get lucky, or you might not.

How bad is it? We can try to answer this question by visualizing the output:

▶ Play

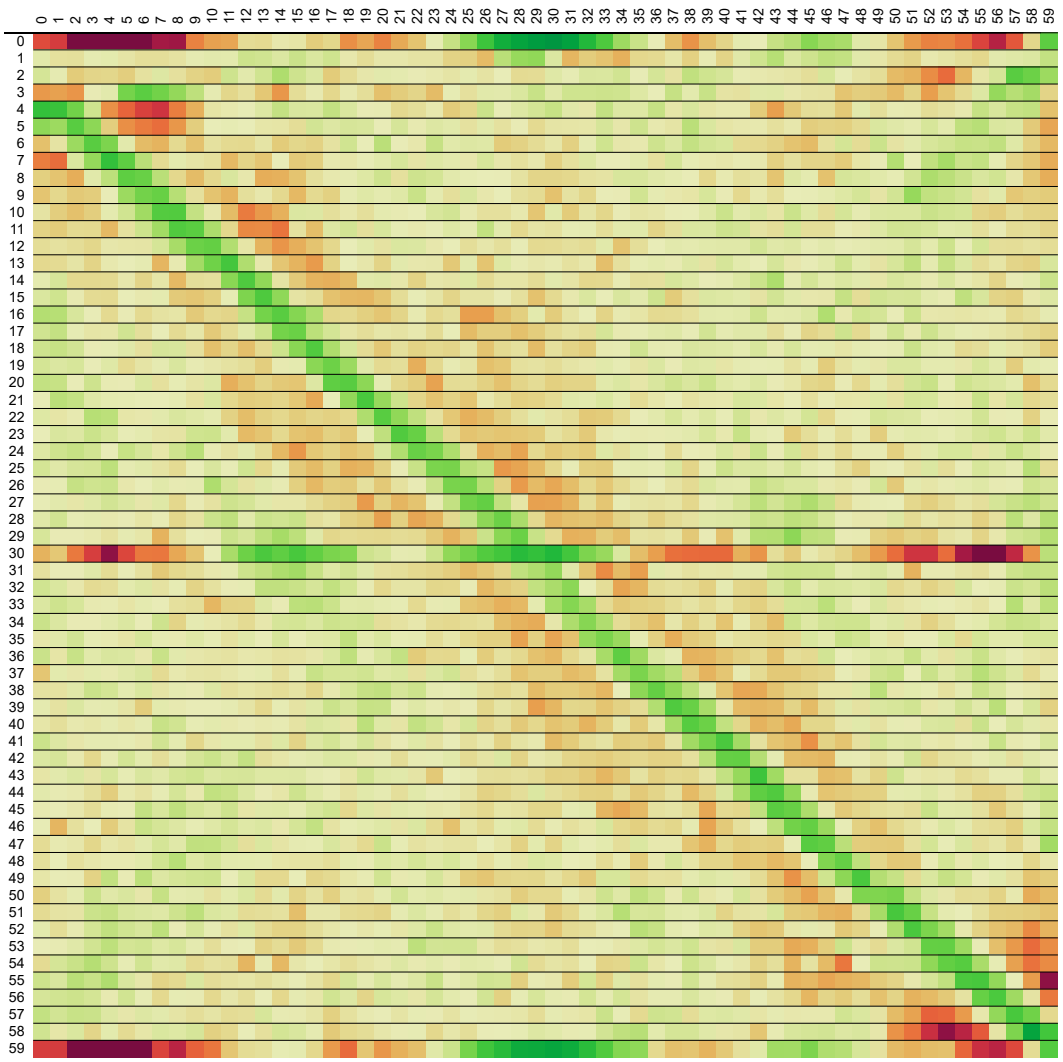
This may look random, so you might be tempted to conclude that random comparator shuffle is adequate, and dismiss concerns of bias as pedantic. But looks can be misleading! There are many things that appear random to the human eye but are substantially non-random.

Another reason this algorithm is bad is that sorting takes $O(n \lg n)$ time, making it significantly slower than Fisher–Yates which takes $O(n)$. But speed is less damning than bias.

This deception demonstrates that visualization is not a magic wand. Showing a single run of the algorithm does not effectively assess the quality of its randomness. We must instead carefully design a visualization that addresses the specific question at hand: what is the algorithm's bias?

To show bias, we must first define it. One definition is based on the probability that an array element at index i prior to shuffling will be at index j after shuffling. If the algorithm is unbiased, every element has equal probability of ending up at every index, and thus the probability for all i and j is the same: $1/n$, where n is the number of elements.

Computing these probabilities analytically is difficult, since it depends on knowing the exact sorting algorithm used. But computing them empirically is easy: we simply shuffle thousands of times and count the number of occurrences of element i at index j. An effective display for this matrix of probabilities is a matrix diagram:

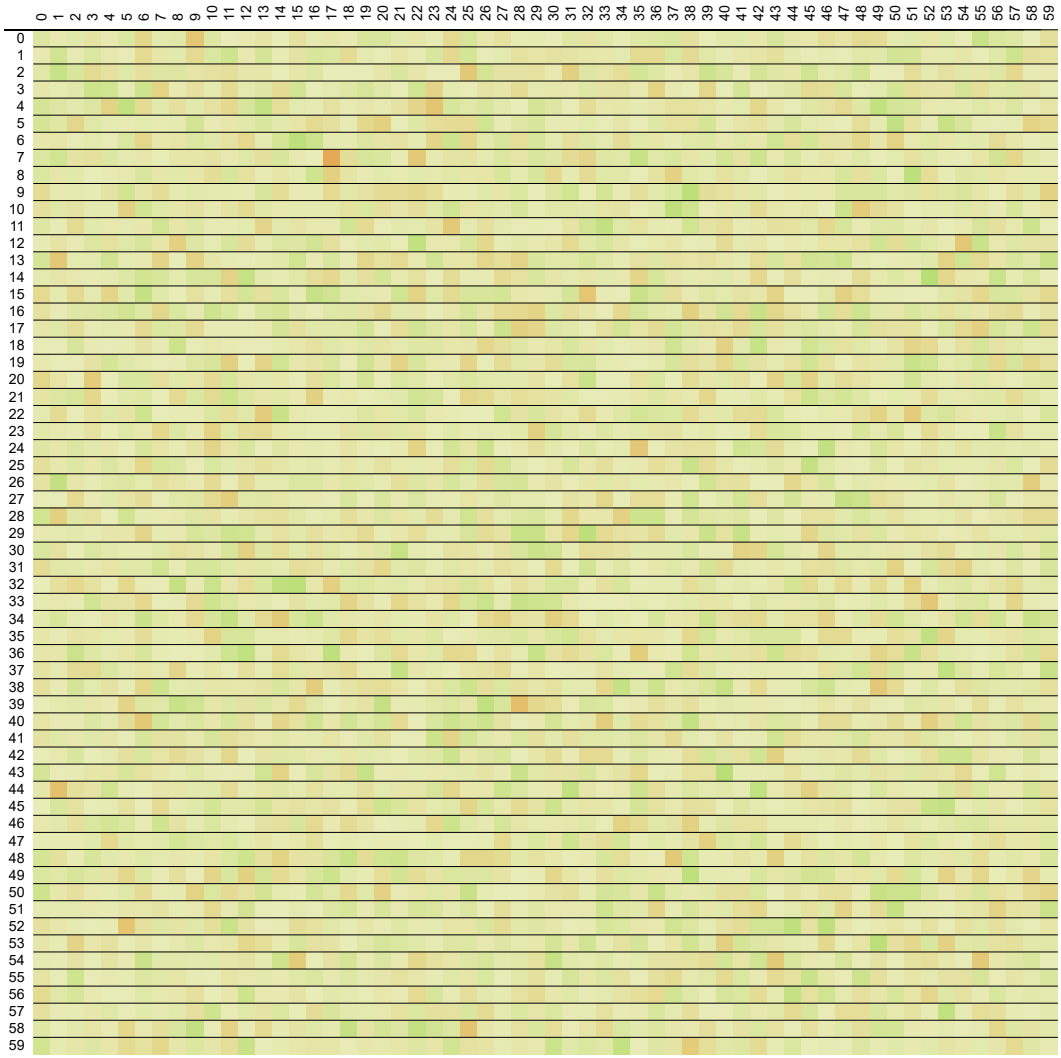


SHUFFLE BIAS
column = index before shuffle
row = index after shuffle
green = positive bias
red = negative bias

The column (horizontal position) of the matrix represents the index of the element prior to shuffling, while the row (vertical position) represents the index of the element after shuffling. Color encodes probability: green cells indicate *positive bias*, where the element occurs more frequently than we would expect for an unbiased algorithm; likewise red cells indicate *negative bias*, where it occurs less frequently than expected.

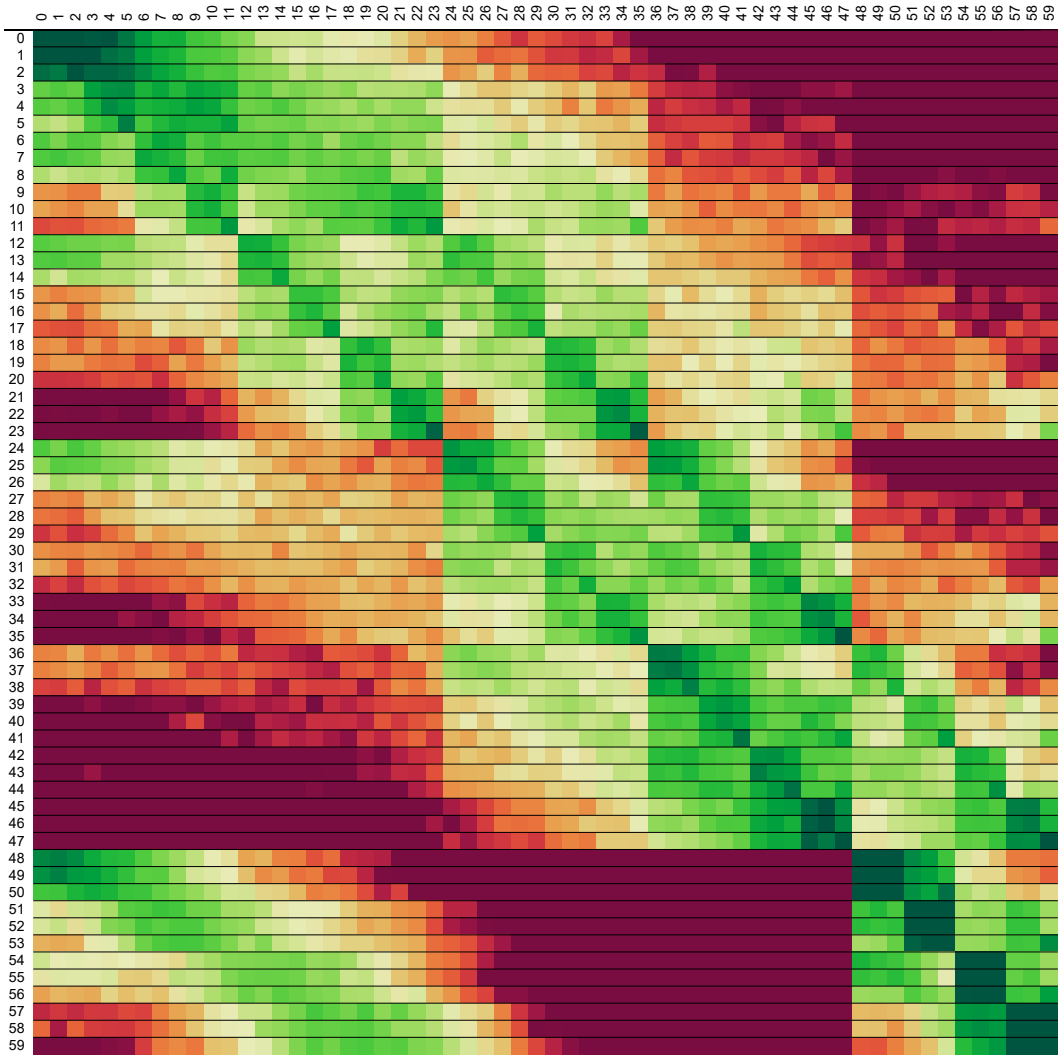
Random comparator shuffle in Chrome, shown above, is surprisingly mediocre. Parts of the array are only weakly biased. However, it exhibits a strong positive bias below the diagonal, which indicates a tendency to push elements from index *i* to *i+1* or *i+2*. There is also strange behavior for the first, middle and last row, which might be a consequence of Chrome using median-of-three quicksort.

The unbiased Fisher–Yates algorithm looks like this:



No patterns are visible in this matrix, other than a small amount of noise due to empirical measurement. (That noise could be reduced if desired by taking additional measurements.)

The behavior of random comparator shuffle is heavily dependent on your browser. Different browsers use different sorting algorithms, and different sorting algorithms behave very differently with (broken) random comparators. Here's random comparator shuffle on Firefox:



For an interactive version of these matrix diagrams to test alternative shuffling strategies, see Will It Shuffle?

This is egregiously biased! The resulting array is often barely shuffled, as shown by the strong green diagonal in this matrix. This does not mean that Chrome's sort is somehow "better" than Firefox's; it simply means you should never use random comparator shuffle. Random comparators are fundamentally broken.

Sorting

Sorting is the inverse of shuffling: it creates order from disorder, rather than *vice versa*. This makes sorting a harder problem with diverse solutions designed for different trade-offs and constraints.

One of the most well-known sorting algorithms is quicksort.

▶ Play

Quicksort

Quicksort first partitions the array into two parts by picking a pivot. The left part contains all elements less than the pivot, while the right part contains all elements greater than the pivot. After the array is partitioned, quicksort recurses into the left and right parts. When a part contains only a single element, recursion stops.

The partition operation makes a single pass over the active part of the array. Similar to how the Fisher–Yates shuffle incrementally builds the shuffled section by swapping elements, the partition operation builds the lesser (left) and greater (right) parts of the subarray incrementally. As each element is visited, if it is less than the pivot it is swapped into the lesser part; if it is greater than the pivot the partition operation moves on to the next element.

Here's the code:

```
function quicksort(array, left, right) {
  if (left < right - 1) {
    var pivot = left + right >> 1;
    pivot = partition(array, left, right, pivot);
    quicksort(array, left, pivot);
    quicksort(array, pivot + 1, right);
  }
}

function partition(array, left, right, pivot) {
  var pivotValue = array[pivot];
  swap(array, pivot, --right);
  for (var i = left; i < right; ++i) {
    if (array[i] < pivotValue) {
      swap(array, i, left++);
    }
  }
  swap(array, left, right);
  return left;
}
```

There are many variations of quicksort. The one shown above is one of the simplest — and slowest. This variation is useful for teaching, but in practice more elaborate implementations are used for better performance.

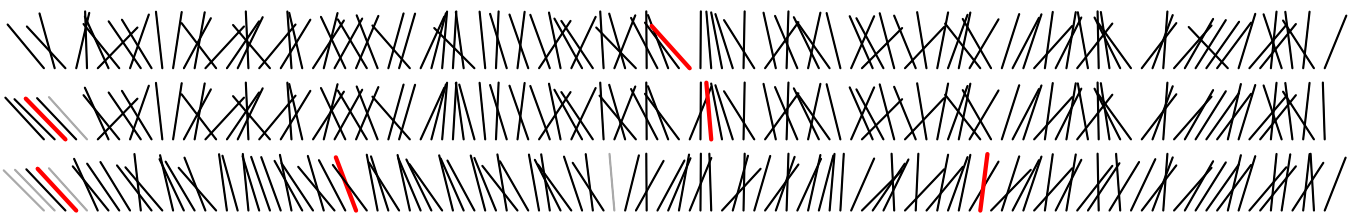
A common improvement is "median-of-three" pivot selection, where the median of the first, middle and last elements is used as the pivot. This tends to choose a pivot closer to the true median, resulting in similarly-sized left and right parts and shallower recursion. Another optimization is switching from quicksort to insertion sort for small parts of the array, which can be faster due to the overhead of function calls.

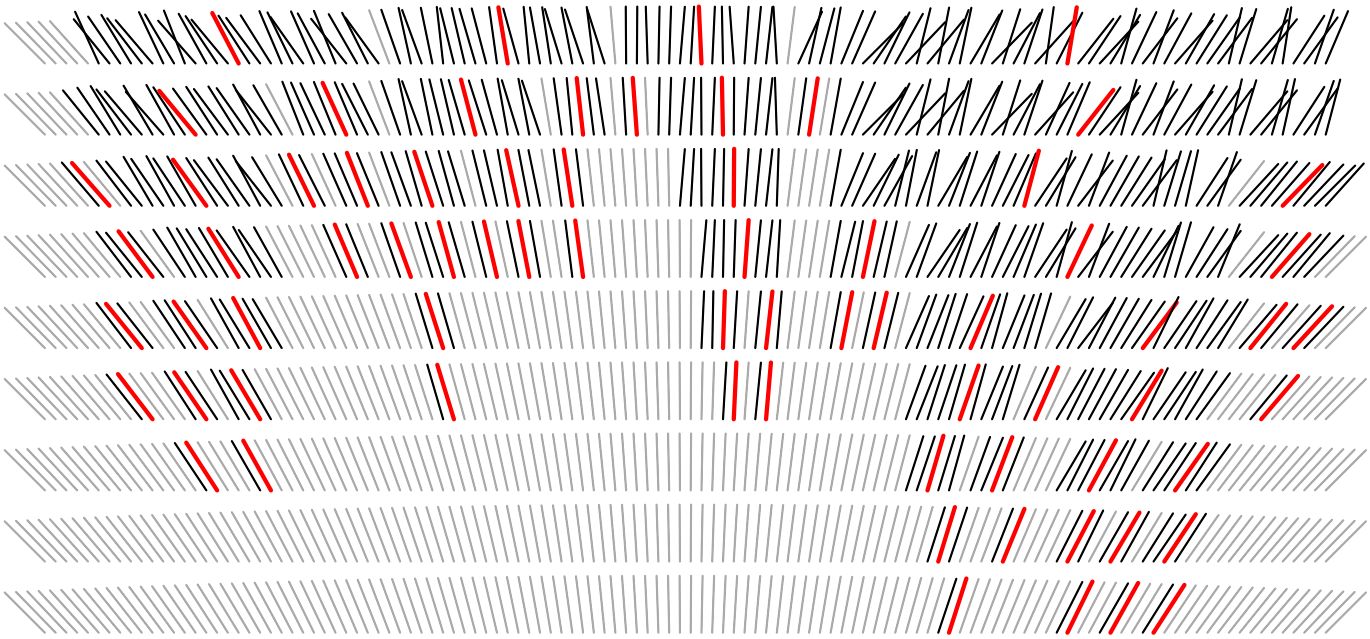
A particularly clever variation is Yaroslavskiy's dual-pivot quicksort, which partitions the array into three parts rather than two. This is the default sorting algorithm in Java and Dart.

The sort and shuffle animations above have the nice property that time is mapped to time: we can simply watch how the algorithm proceeds. But while intuitive, animation can be frustrating to watch, especially if we want to focus on an occasional weirdness in the algorithm's behavior. Animations also rely heavily on our memory to observe patterns in behavior. While animations are improved by controls to pause and scrub time, static displays that show everything at once can be even more effective. The eye scans faster than the hand.

A simple way of turning an animation into a static display is to pick key frames from the animation and display those sequentially, like a comic strip. If we then remove redundant information across key frames, we use space more efficiently. A denser display may require more study to understand, but is faster to scan since the eye travels less.

Below, each row shows the state of the array prior to recursion. The first row is the initial state of the array, the second row is the array after the first partition operation, the third row is after the first partition's left and right parts are again partitioned, *etc.* In effect, this is breadth-first quicksort, where the partition operation on both left and right proceeds in parallel.

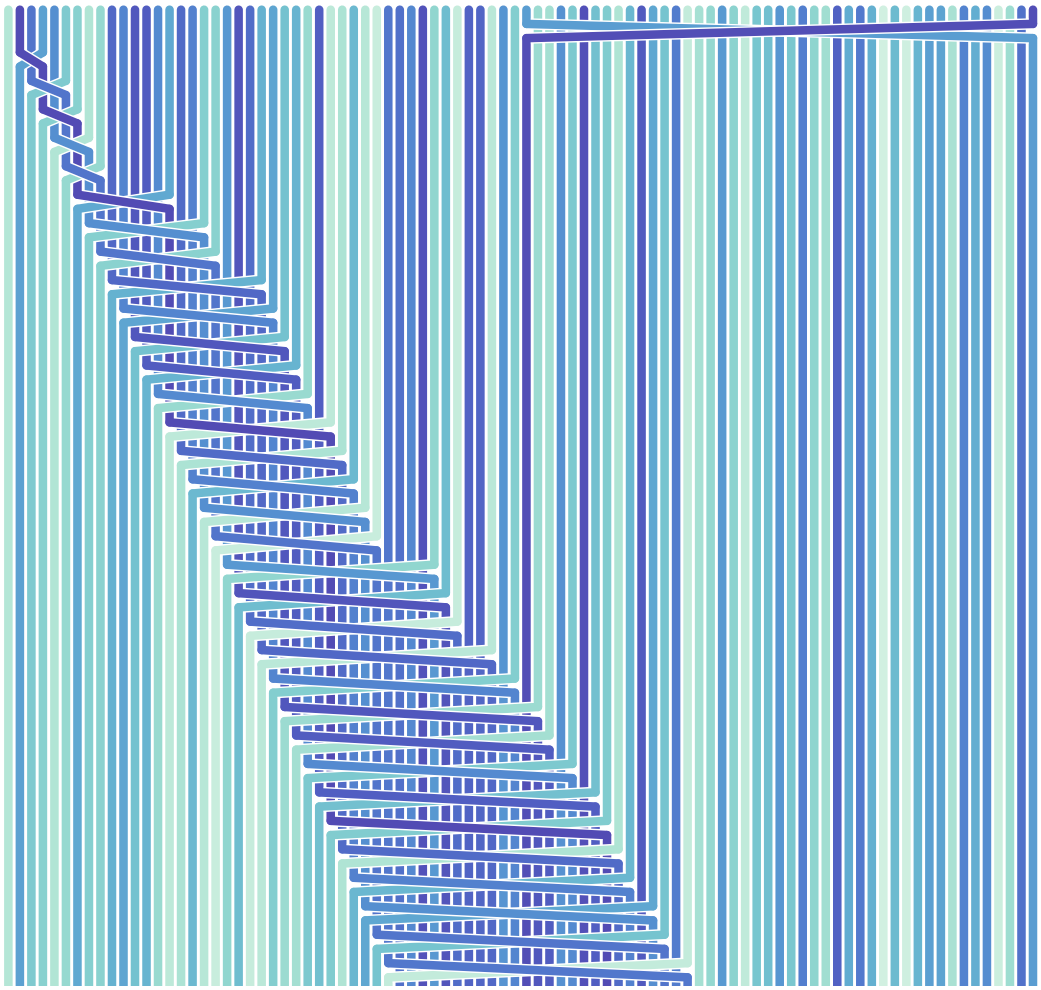




Quicksort

As before, the pivots for each partition operation are highlighted in red. Notice that the pivots turn gray at the next level of recursion: after the partition operation completes, the associated pivot is in its final, sorted position. The total depth of the display — the maximum depth of recursion — gives a sense of how efficiently quicksort performed. It depends heavily on input and pivot choice.

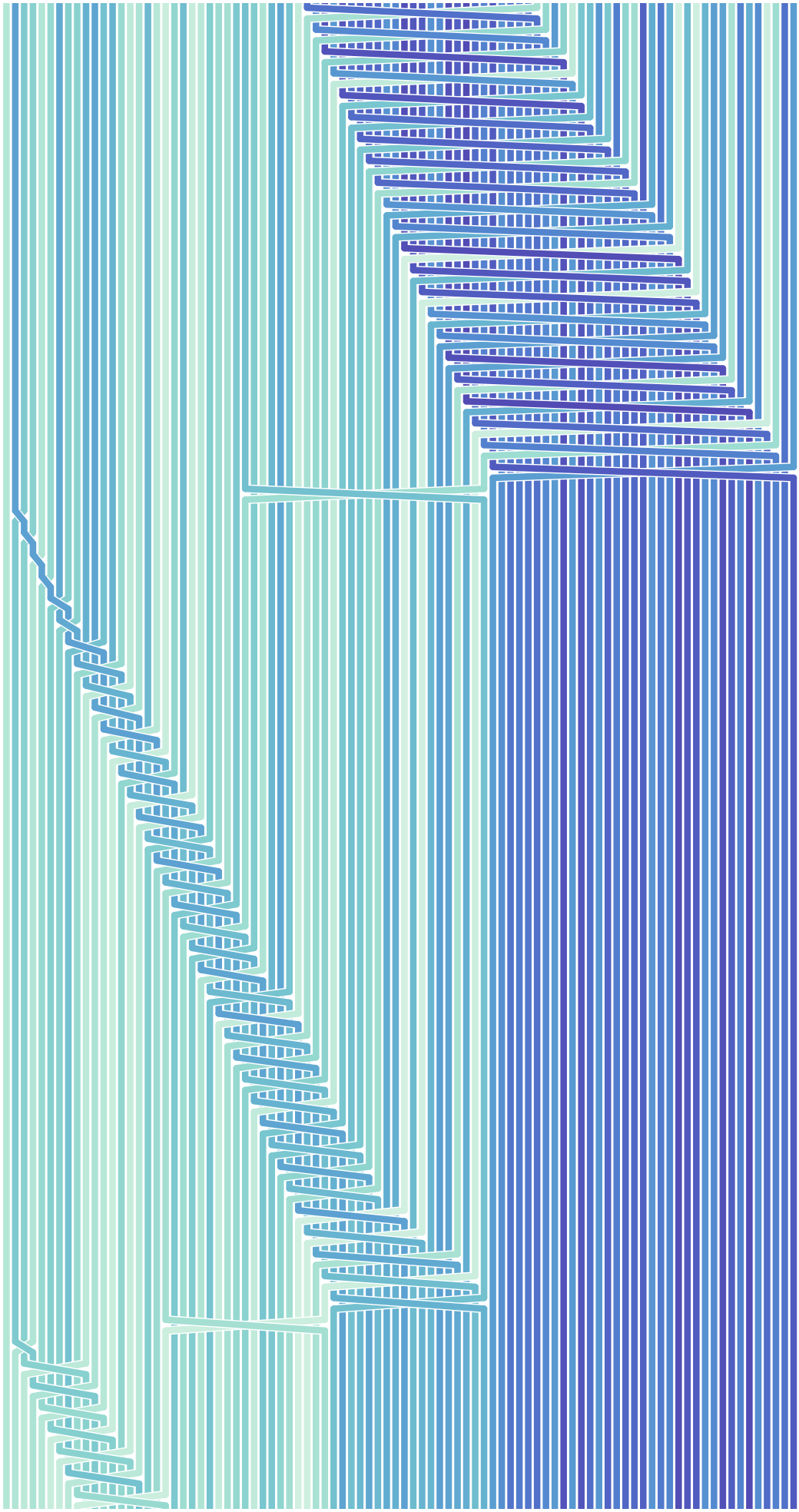
Another static display of quicksort, less dense but perhaps easier to read, represents each element as a colored thread and shows each sequential swap. (This form is inspired by Aldo Cortesi's sorting visualizations.) Smaller values are lighter, and larger values are darker.



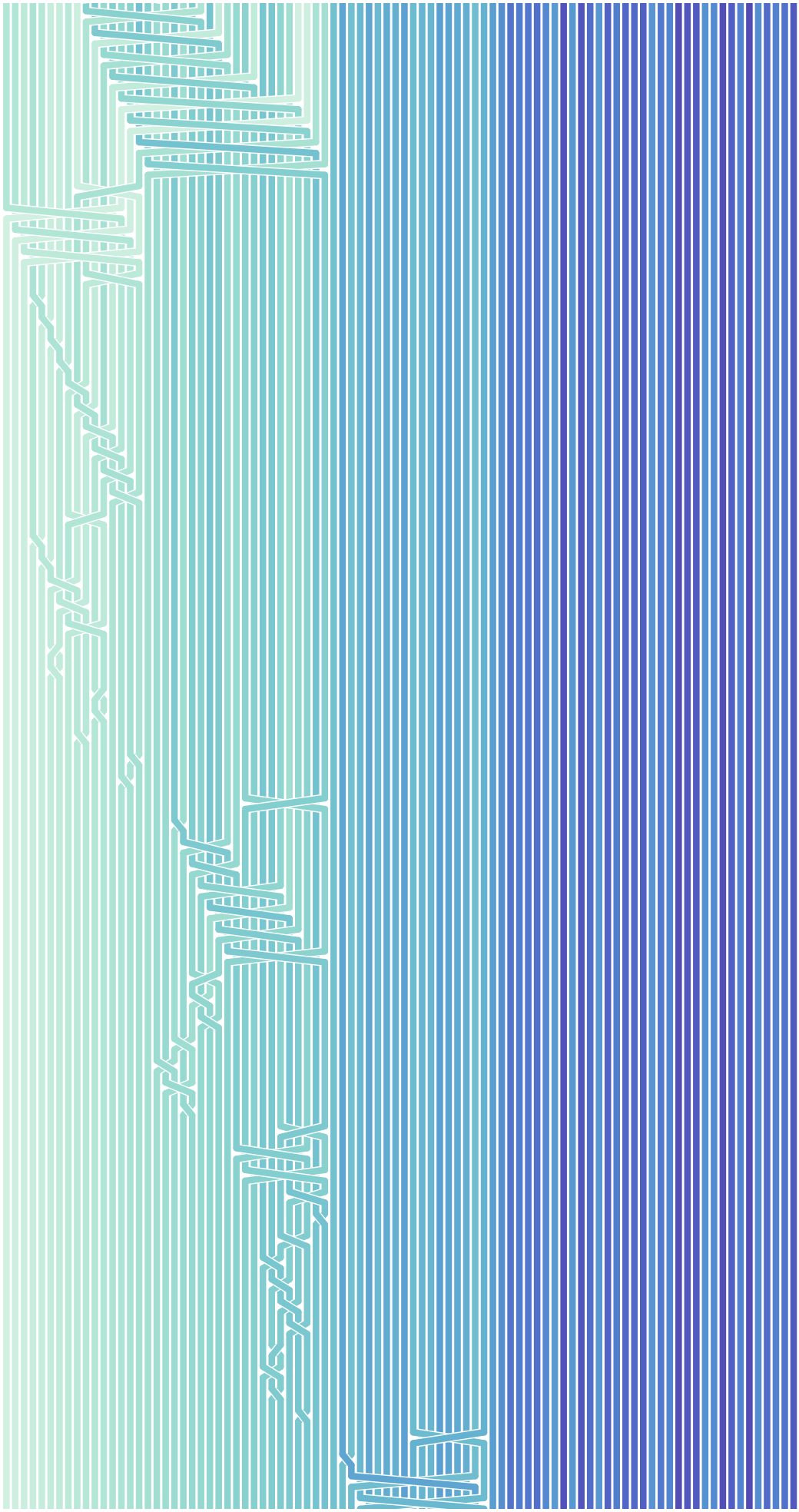
At the start of each partition, the pivot is moved to the end (the right) of the active subarray.

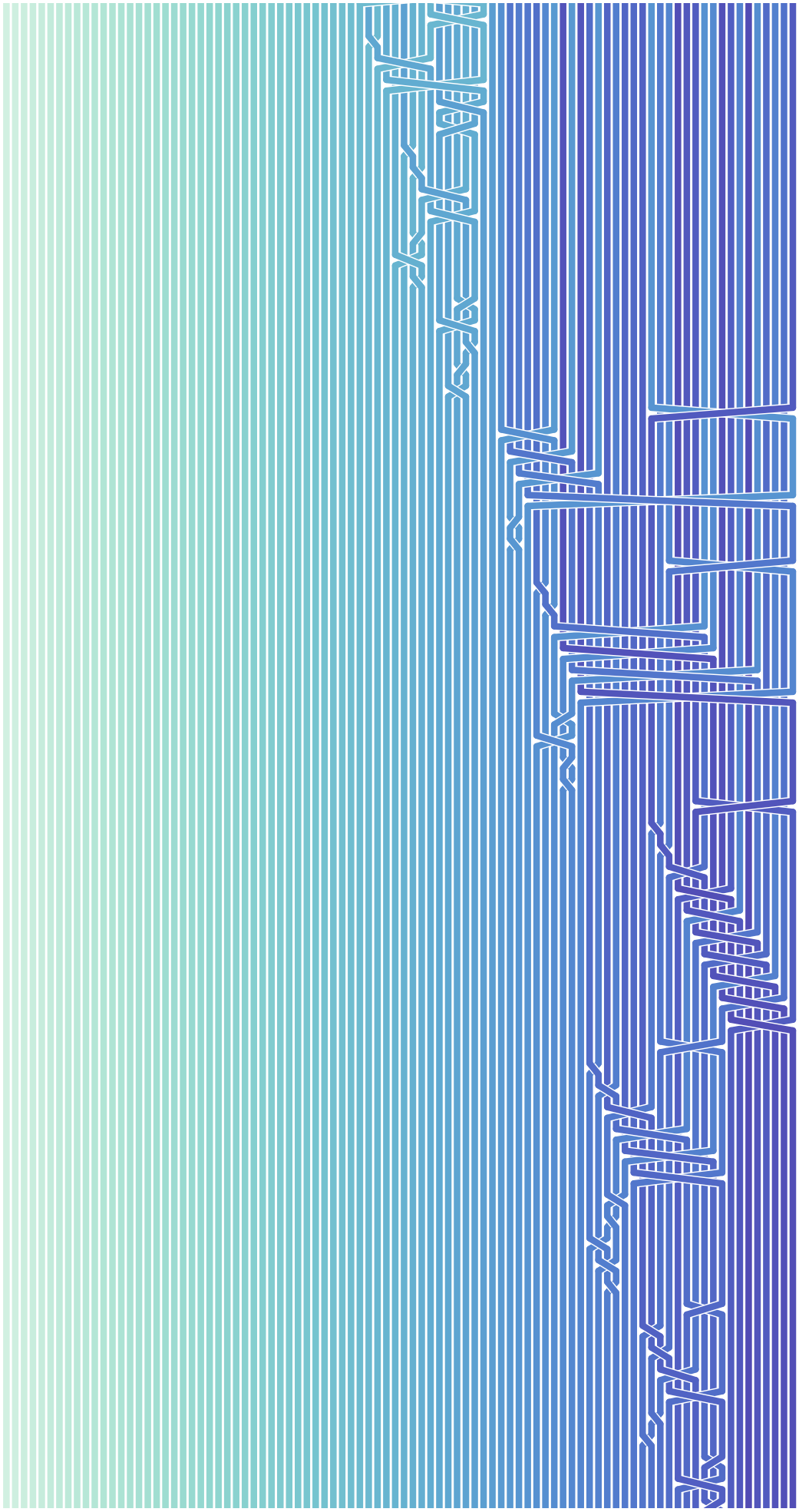
Partitioning then proceeds from left to right. At each step, a new element is added either of the set of *lesser* values (in which case a swap occurs) or to the set of *greater* values (in which case no swap occurs).

When a swap occurs, the left-most value greater than the pivot is moved to the right, so as to make room on the left for the new lesser value. Thus, notice that in all swap operations, only values *darker* than the pivot move right, and only values *lighter* than the pivot move left.

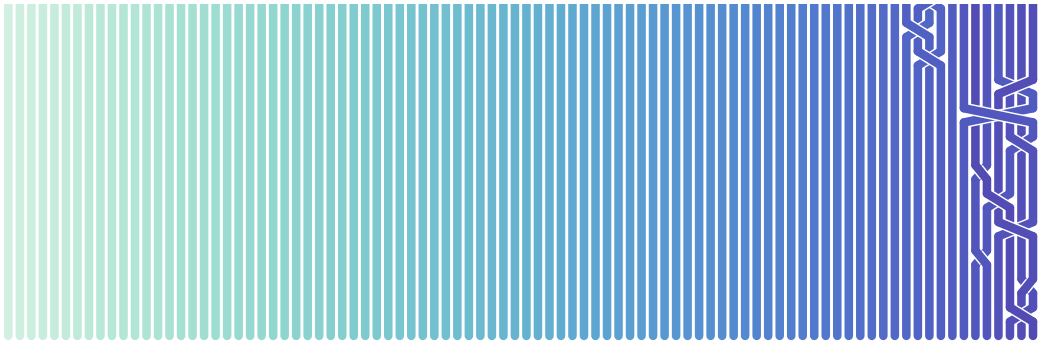


When the partition operation has visited all elements in the array, the pivot is placed in its final position between the two parts. Then, the algorithm recurses into the left part, followed by the right part (far below).





This visualization doesn't show the state of the stack, so it can appear to jump around arbitrarily due to the nature of recursion. Still, you can typically see when a partition operation finishes due to the characteristic movement of the pivot to the end of the active subarray.



Quicksort

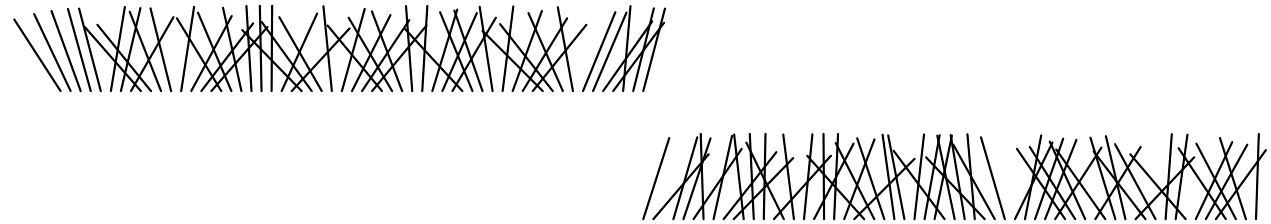
You've now seen three different visual representations of the same algorithm: an animation, a dense static display, and a sparse static display. Each form has strengths and weaknesses. Animations are fun to watch, but static visualizations allow close inspection without being rushed. Sparse displays may be easier to understand, but dense displays show the "macro" view of the algorithm's behavior in addition to its details.

Before we move on, let's contrast quicksort with another well-known sorting algorithm: mergesort.

```
function mergesort(array) {
  var n = array.length, a0 = array, a1 = new Array(n);
  for (var m = 1; m < n; m <<= 1) {
    for (var i = 0; i < n; i += m << 1) {
      var left = i,
          right = Math.min(i + m, n),
          end = Math.min(i + (m << 1), n);
      merge(a0, a1, left, right, end);
    }
    i = a0, a0 = a1, a1 = i;
  }
  if (array === a1) {
    for (var i = 0; i < n; ++i) {
      array[i] = a0[i];
    }
  }
}

function merge(a0, a1, left, right, end) {
  for (var i0 = left, i1 = right; left < end; ++left) {
    if (i0 < right && (i1 >= end || a0[i0] <= a0[i1])) {
      a1[left] = a0[i0++];
    } else {
      a1[left] = a0[i1++];
    }
  }
}
```

Again, above is the code and below is an animation:



▶ Play

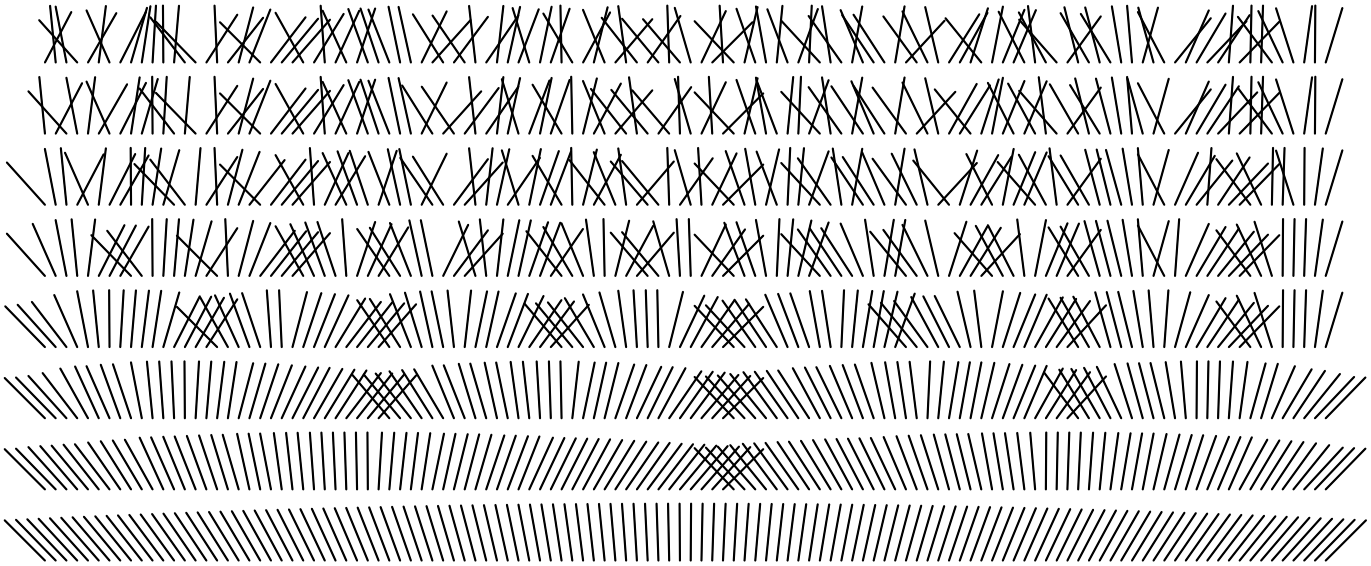
Mergesort

As you've likely surmised from either the code or the animation, mergesort takes a very different approach to sorting than quicksort. Unlike quicksort, which operates in-place by performing swaps, mergesort requires an extra copy of the array. This extra space is used to merge sorted subarrays,

combining the elements from pairs of subarrays while preserving order. Since mergesort performs copies instead of swaps, we must modify the animation accordingly (or risk misleading readers).

Mergesort works from the bottom-up. Initially, it merges subarrays of size one, since these are trivially sorted. Each adjacent subarray — at first, just a pair of elements — is merged into a sorted subarray of size two using the extra array. Then, each adjacent sorted subarray of size two is merged into a sorted subarray of size four. After each pass over the whole array, mergesort doubles the size of the sorted subarrays: eight, sixteen, and so on. Eventually, this doubling merges the entire array and the algorithm terminates.

Because mergesort performs repeated passes over the array rather than recursing like quicksort, and because each pass doubles the size of sorted subarrays regardless of input, it is easier to design a static display. We simply show the state of the array after each pass.



Mergesort

Let's again take a moment to consider what we've seen. The goal here is to study the behavior of an algorithm rather than a specific dataset. Yet there is still data, necessarily — the data is derived from the execution of the algorithm. And this means we can use the *type* of derived data to classify algorithm visualizations.

Level 0 / black box - The simplest class just shows the output. This does not explain the algorithm's operation, but it can still verify correctness. And by treating the algorithm as a black box, you can more easily compare outputs of different algorithms. Black box visualizations can also be combined with deeper analysis of output, such as the shuffle bias matrix diagram shown above.

Level 1 / gray box - Many algorithms (though not all) build up output incrementally. By visualizing the intermediate output as it develops, we start to see how the algorithm works. This explains more without introducing new abstraction, since the intermediate and final output share the same structure. Yet this type of visualization can raise more questions than it answers, since it offers no explanation as to why the algorithm does what it does.

Level 2 / white box - To answer "why" questions, white box visualizations expose the internal state of the algorithm in addition to its intermediate output. This type has the greatest potential to explain, but also the highest burden on the reader, as the meaning and purpose of internal state must be clearly described. There is a risk that the additional complexity will overwhelm the reader; layering information may make the graphic more accessible. Lastly, since internal state is highly-dependent on the specific algorithm, this type of visualization is often unsuitable for comparing algorithms.

There's also the practical matter of implementing algorithm visualizations. Typically you can't just run code as-is; you must instrument it to capture state for visualization. (View source on this page for examples.) You may even need to interleave execution with visualization, which is particularly challenging for recursive algorithms that capture state on the stack. Language parsers such as

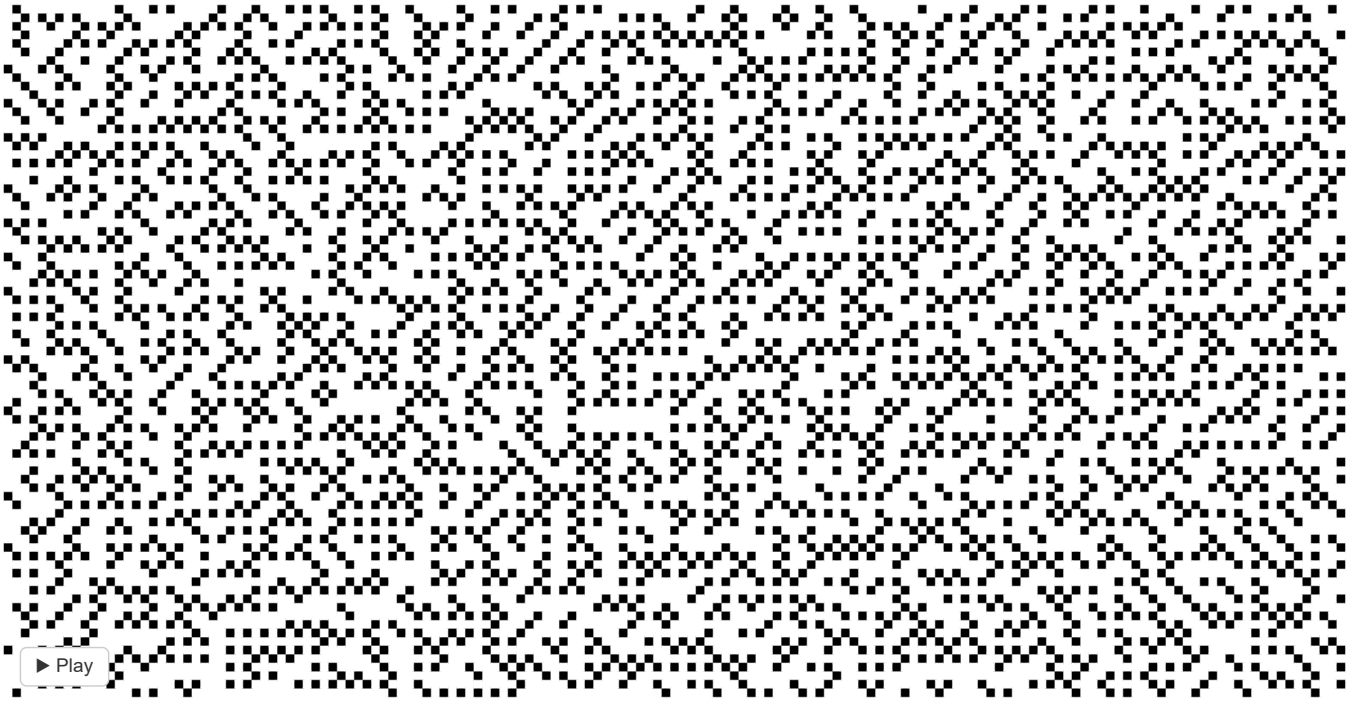
Esprima may facilitate algorithm visualization through code instrumentation, cleanly separating execution code from visualization code.

Maze Generation

The last problem we'll look at is maze generation. All algorithms in this section generate a spanning tree of a two-dimensional rectangular grid. This means there are no loops and there is a unique path from the root in the bottom-left corner to every other cell in the maze.

I apologize for the esoteric subject — I don't know enough to say why these algorithms are useful beyond simple games, and possibly something about electrical networks. But even so, they are fascinating from a visualization perspective because they solve the same, highly-constrained problem in wildly-different ways.

And they're just fun to watch.

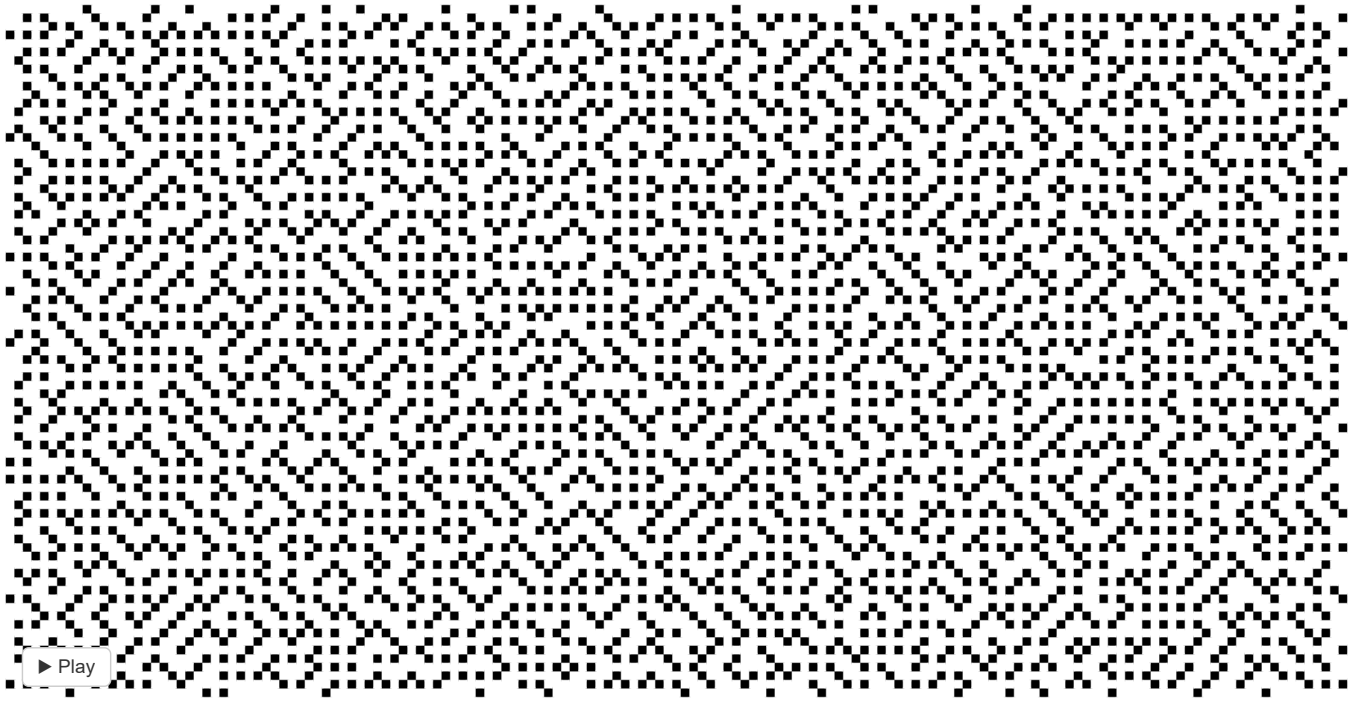


Random traversal

The random traversal algorithm initializes the first cell of the maze in the bottom-left corner. The algorithm then tracks all possible ways by which the maze could be extended (shown in red). At each step, one of these possible extensions is picked randomly, and the maze is extended as long as this does not reconnect it with another part of the maze.

Like Bridon's Poisson-disc sampling algorithm, random traversal maintains a frontier and randomly selects from that frontier to expand. Both algorithms thus appear to grow organically, like a fungus.

Randomized depth-first traversal follows a very different pattern:



Randomized depth-first traversal

Rather than picking a new random passage each time, this algorithm always extends the deepest passage — the one with the longest path back to the root — in a random direction. Thus, randomized depth-first traversal only branches when the current path dead-ends into an earlier part of the maze. To continue, it backtracks until it can start a new branch. This snake-like exploration leads to mazes with significantly fewer branches and much longer, winding passages.

Prim's algorithm constructs a minimum spanning tree, a spanning tree of a graph with weighted edges with the lowest total weight. This algorithm can be used to construct a random spanning tree by initializing edge weights randomly:

▶ Play

Randomized Prim's

At each step, Prim's algorithm extends the maze using the lowest-weighted edge (potential direction) connected to the existing maze. If this edge would form a loop, it is discarded and the next-lowest-weighted edge is considered.

Prim's algorithm is commonly implemented using a heap, which is an efficient data structure for prioritizing elements. When a new cell is added to the maze, connected edges (shown in red) are added to the heap. Despite edges being added in arbitrary order, the heap allows the lowest-weighted edge to be quickly removed.

Lastly, a most unusual specimen:

Wilson's

Wilson's algorithm uses loop-erased random walks to generate a uniform spanning tree — an unbiased sample of all possible spanning trees. The other maze generation algorithms we have seen lack this beautiful mathematical property.

The algorithm initializes the maze with an arbitrary starting cell. Then, a new cell is added to the maze, initiating a random walk (shown in red). The random walk continues until it reconnects with the existing maze (shown in white). However, if the random walk intersects itself, the resulting loop is erased before the random walk continues.

Initially, the algorithm can be frustratingly slow to watch, as early random walks are unlikely to reconnect with the small existing maze. As the maze grows, random walks become more likely to collide with the maze and the algorithm accelerates dramatically.

These four maze generation algorithms work very differently. And yet, when the animations end, the resulting mazes are difficult to distinguish from each other. The animations are useful for showing how the algorithm works, but fail to reveal the resulting tree structure.

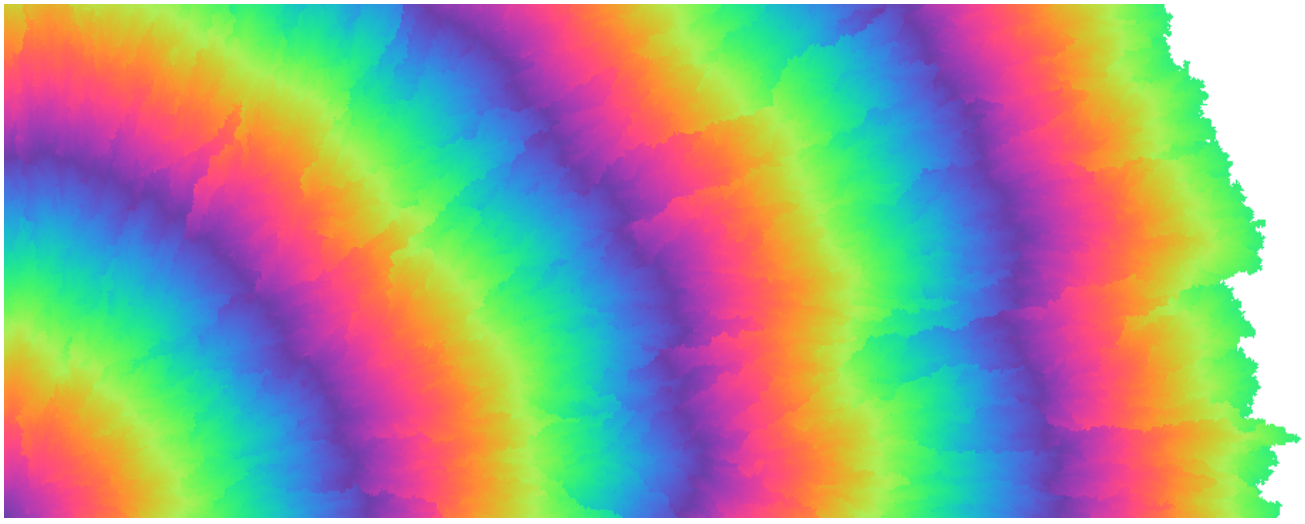
A way to show structure, rather than process, is to flood the maze with color:

▶ Play

Random traversal

Color encodes tree depth — the length of the path back to the root in the bottom-left corner. The color scale cycles as you get deeper into the tree; this is occasionally misleading when a deep path circles back adjacent to a shallow one, but the higher contrast allows better differentiation of local structure. (This is not a convential rainbow color scale, which is nominally considered harmful, but a cubehelix rainbow with improved perceptual properties.)

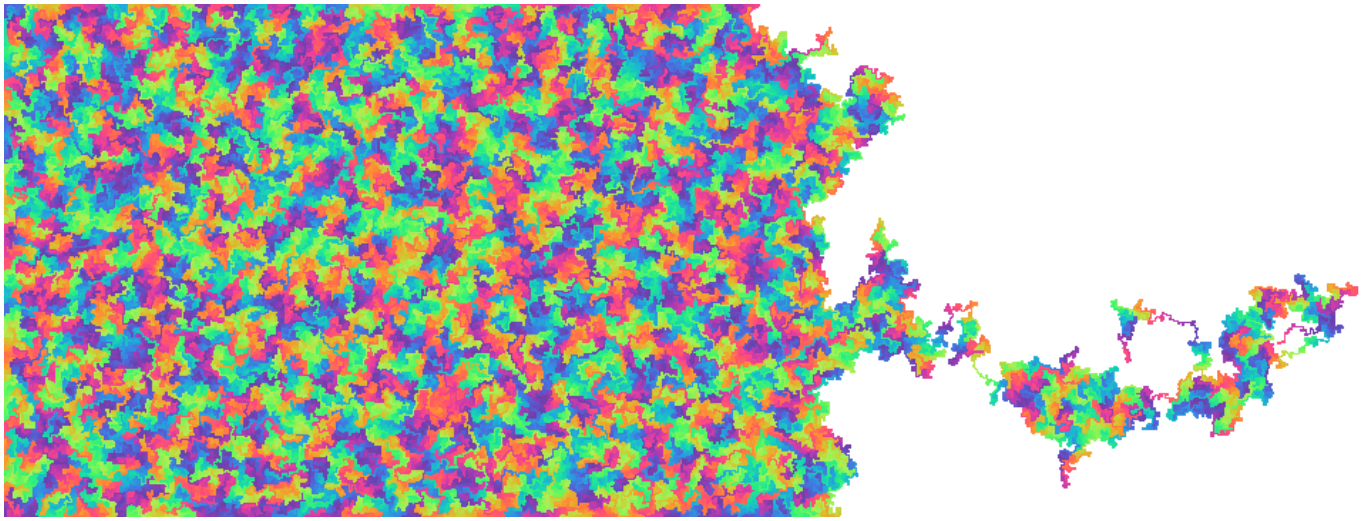
We can further emphasize the structure of the maze by subtracting the walls, reducing visual noise. Below, each pixel represents a path through the maze. As above, paths are colored by depth and color floods deeper into the maze over time.



Random traversal

Concentric circles of color, like a tie-dye shirt, reveal that random traversal produces many branching paths. Yet the shape of each path is not particularly interesting, as it tends to go in a straight line back to the root. Because random traversal extends the maze by picking randomly from the frontier, paths are never given much freedom to meander — they end up colliding with the growing frontier and terminate due to the restriction on loops.

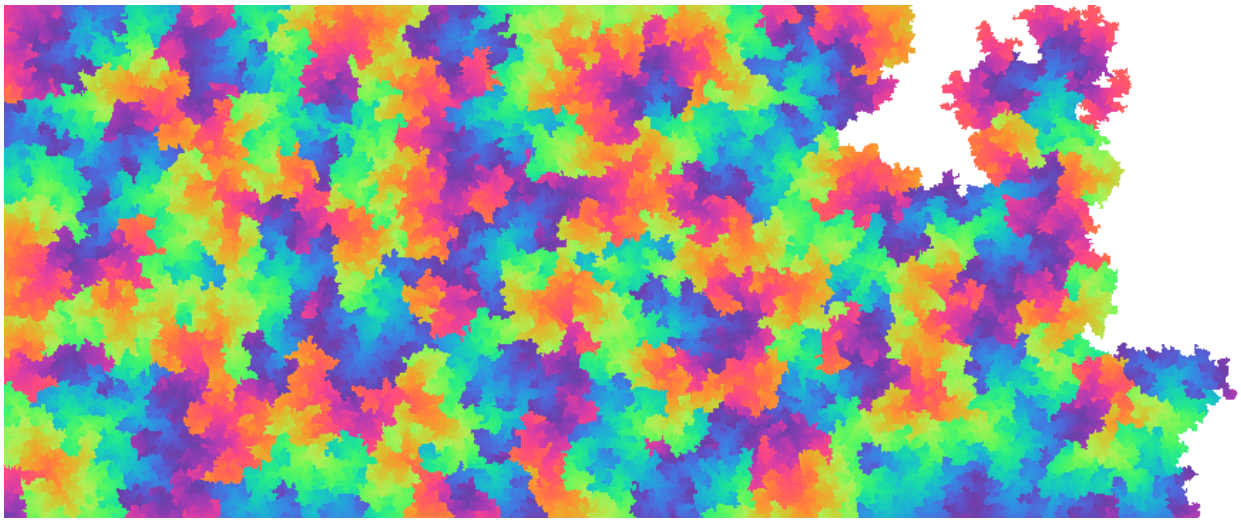
Randomized depth-first traversal, on the other hand, is *all* about the meander:



Randomized depth-first traversal

This animation proceeds at fifty times the speed of the previous one. This speed-up is necessary because randomized depth-first traversal mazes are much, much deeper than random traversal mazes due to limited branching. You can see that typically there is only one, and rarely more than a few, active branches at any particular depth.

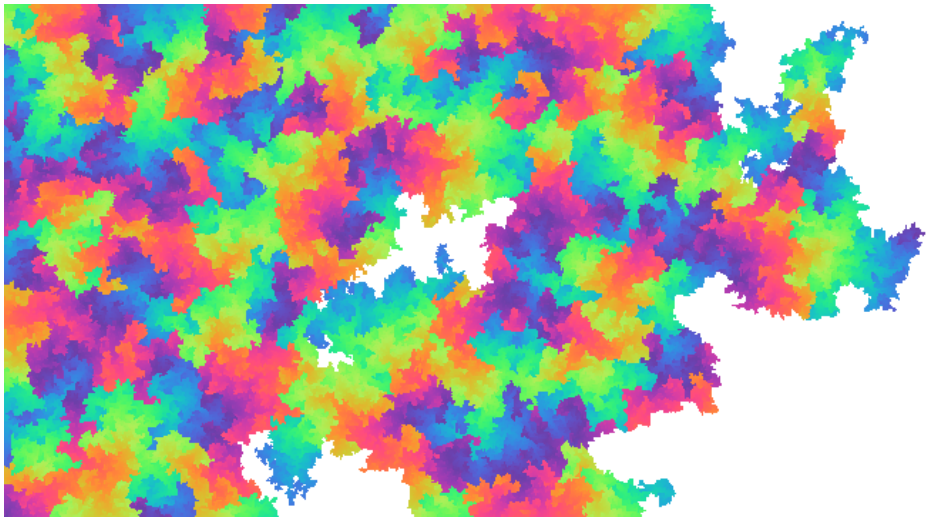
Now Prim's algorithm on a random graph:



Randomized Prim's

This is more interesting! The simultaneously-expanding florets of color reveal substantial branching, and there is more complex global structure than random traversal.

Wilson's algorithm, despite operating very differently, seems to produce very similar results:



Wilson's

Just because they *look* the same does not mean they *are*. Despite appearances, Prim's algorithm on a randomly-weighted graph does not produce a uniform spanning tree (as far as I know — proving this is outside my area of expertise). Visualization can sometimes mislead due to human error. An earlier version of the Prim's color flood had a bug where the color scale rotated twice as fast as intended; this suggested that Prim's and Wilson's algorithms produced very different trees, when in fact they appear much more similar than different.

Since these mazes are spanning trees, we can also use specialized tree visualizations to show structure. To illustrate the duality between maze and tree, here the passages (shown in white) of a maze generated by Wilson's algorithm are gradually transformed into a tidy tree layout. As with the other animations, it proceeds by depth, starting with the root and descending to the leaves:

▶ Play

Wilson's

For comparison, again we see how randomized depth-first traversal produces trees with long passages and little branching.

Randomized depth-first traversal

Both trees have the same number of nodes (3,239) and are scaled to fit in the same area (960×500 pixels). This hides an important difference: at this size, randomized depth-first traversal typically produces a tree two-to-five times deeper than Wilson's algorithm. The tree depths above are 250 and 870, respectively. In the larger 480,000-node mazes used for color flooding, randomized depth-first traversal produces a tree that is 10-20 times deeper!

Using Vision to Think

This essay has focused on algorithms. Yet the techniques discussed here apply to a broader space of problems: mathematical formulas, dynamical systems, processes, *etc.* Basically, anywhere there is code that needs understanding.

Shan Carter, Archie Tse and I recently built a new rent *vs.* buy calculator; powering the calculator is a couple hundred lines of code to compute the total cost of renting or buying a home. It's a simplistic model, but more complicated than fits in your head. The calculator takes about twenty input parameters (such as purchase price and mortgage rate) and considers opportunity costs on investments, inflation, marginal tax rates, and a variety of other factors.

The goal of the calculator is to help you decide whether you should buy or rent a home. If the total cost of buying is cheaper, you should buy. Otherwise, you should rent.

Except, it's not that simple.

To output an accurate answer, the calculator needs accurate inputs. While some inputs are well-known (such as the length of your mortgage), others are difficult or impossible to predict. No one can say exactly how the stock market will perform, how much a *specific* home will appreciate or depreciate, or how the renting market will change over time.

We can make educated guesses at each variable — for example, looking at Case–Shiller data. But if the calculator is a black box, then readers can't see how sensitive their answer is to small changes.

To fix this, we need to do more than output a single number. We need to show *how the underlying system works*. The new calculator therefore charts every variable and lets you quickly explore any variable's effect by adjusting the associated slider.

The slope of the chart shows the associated variable's importance: the greater the slope, the more the decision depends on that variable. Since variables interact with each other, changing a variable may change the slope of other charts.

This design lets you inspect many aspects of the system. For example, should you make a large down payment? Yes, if the down payment rate slopes down; or no, if the down payment rate slopes up, as with a higher investment return rate. This suggests that the optimal loan size depends on the difference between the opportunity cost on the down payment (money not invested) and the interest cost on the mortgage.

So, why visualize algorithms? Why visualize anything? To leverage the human visual system to improve understanding. Or more simply, to use vision to think.

Is It Better to Buy or Rent?

Related Work

I mentioned Aldo Cortesi's sorting visualizations earlier. (I also like Cortesi's visualizations of malware entropy.) Others abound, including: sorting.at, sorting-algorithms.com, and Aaron Dufour's Sorting Visualizer, which lets you plug in your own algorithm. YouTube user andrut's audibilizations are interesting. Robert Sedgwick has published several new editions of *Algorithms* since I took his class, and his latest uses traditional bars rather than angled lines.

Amit Patel explores "visual and interactive ways of explaining math and computer algorithms." The articles on 2D visibility, polygonal map generation and pathfinding are particularly great. Nicky Case published another lovely explanation of 2D visibility and shadow effects. I am heavily-indebted to Jamis Buck for his curation of maze generation algorithms. Christopher Wellons' GPU-based path finding implementation uses cellular automata — another fascinating subject. David Mimno gave a talk on visualization for models and algorithms at OpenVis 2014 that was an inspiration for this work. And like many, I have long been inspired by Bret Victor, especially Inventing on Principle and Up and Down the Ladder of Abstraction.

Jason Davies has made numerous illustrations of mathematical concepts and algorithms. Some of my favorites are: Lloyd's Relaxation, Coalescing Soap Bubbles, Biham-Middleton-Levine Traffic Model, Collatz Graph, Random Points on a Sphere, Bloom Filters, Animated Bézier Curves, Animated Trigonometry, Proof of Pythagoras' Theorem, and Morley's Trisector Theorem. Pierre Guilleminot's Fourier series explanation is great, as are Lucas V. Barbosa's Fourier transform time and frequency domains and an explanation of Simpson's paradox by Lewis Lehe & Victor Powell; also see Powell's animations of the central limit theorem and conditional probabilities. Steven Wittens makes mind-expanding visualizations of mathematical concepts in three dimensions, such as Julia fractals.

In my own work, I've used visualization to explain topology inference (including a visual debugger), D3's selections and the Fisher–Yates shuffle. There are more standalone visualizations on my bl.ocks. If you have suggestions for interesting visualizations, or any other feedback, please contact me on Twitter.

Thank you for reading!

June 26, 2014 / Mike Bostock