Algorithm Portfolio

Daniel Cussen

Skylinesort

Skylinesort is a next-generation sorting algorithm. It beats the state of the art in many common scenarios. It sorts naturals $m \in [0, 2^n[$ for some $n \in \mathbb{N}$. Restricting the elements to naturals allows skylinesort to take advantage of random access in addition to just comparisons. By using both techniques, it is faster than algorithms that only use one of the two.

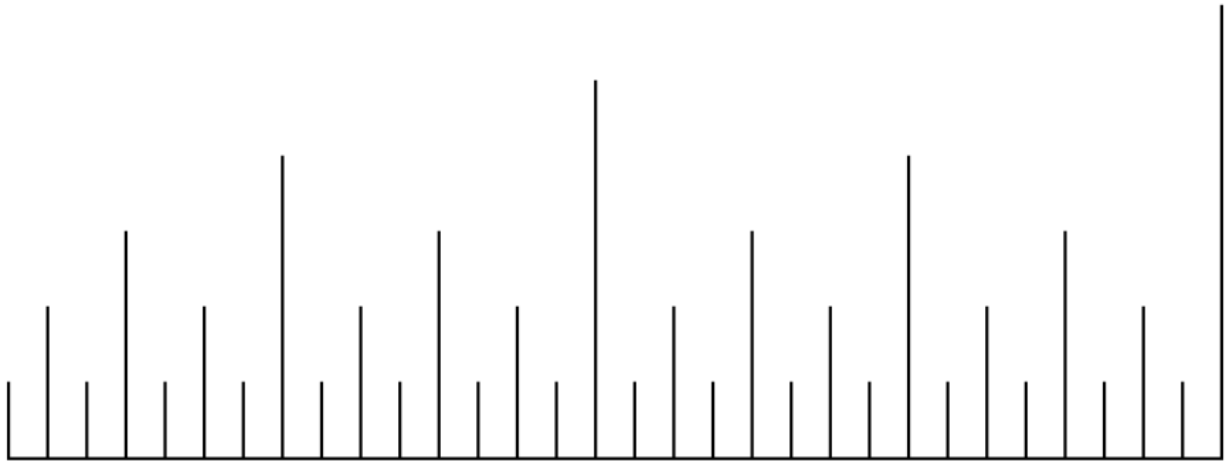

Figure 1: The skylinesort bracket for sorting 5-bit integers. The height of each line is a function of its index $i, i \in [0, 32[$. The bracket determines which other number a number being scattered must be compared with next.

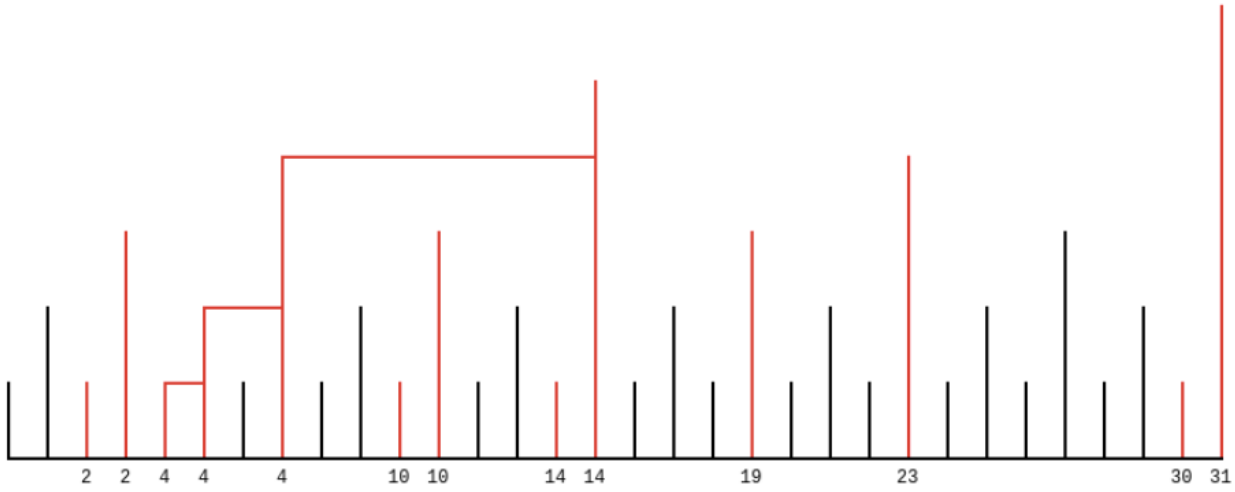

Figure 2: Scatter phase. While an element is being scattered, as the value 4 is in this example, that value keeps rising to the top of the current bracket line and drawing a line to the right from there to determine its next opponent.

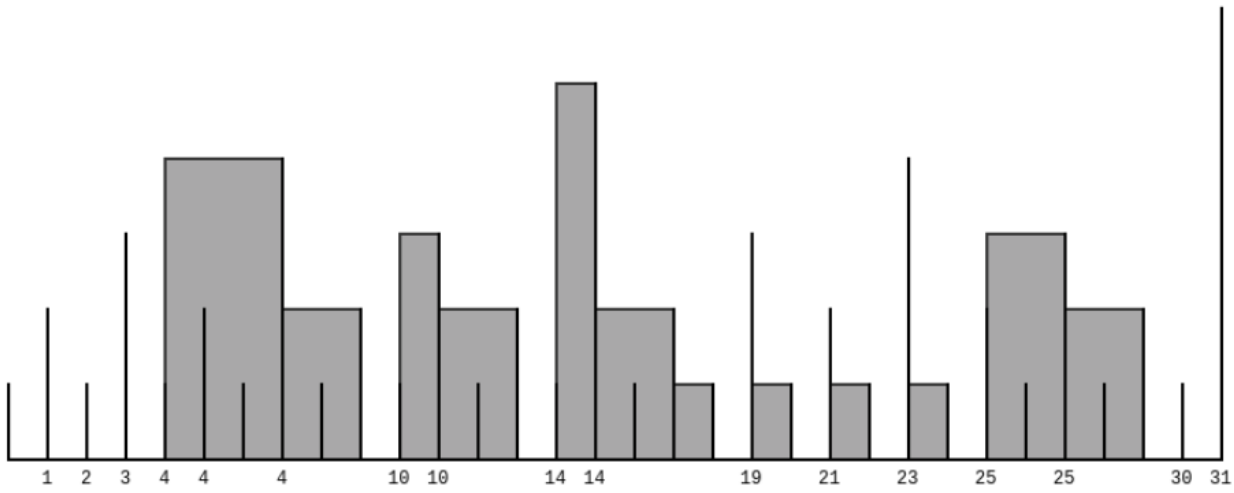

Figure 3: Skylinesort's namesake diagram, in which the gray rectangles resemble buildings in a city skyline, after completing the gather phase on the example input.

Data which caches frequently, repetitive data, and sparse data all benefit from skylinesort. It performs at its worst compared to quicksort and timsort when sorting uniformly distributed random numbers because of cache misses. It performs in $O(n)$ on sorted data, and $O(kn)$ on reversed data (k being the number of bits in each element). The average case is unknown. It is not equal to the best or worst cases, however; it is somewhere in between.

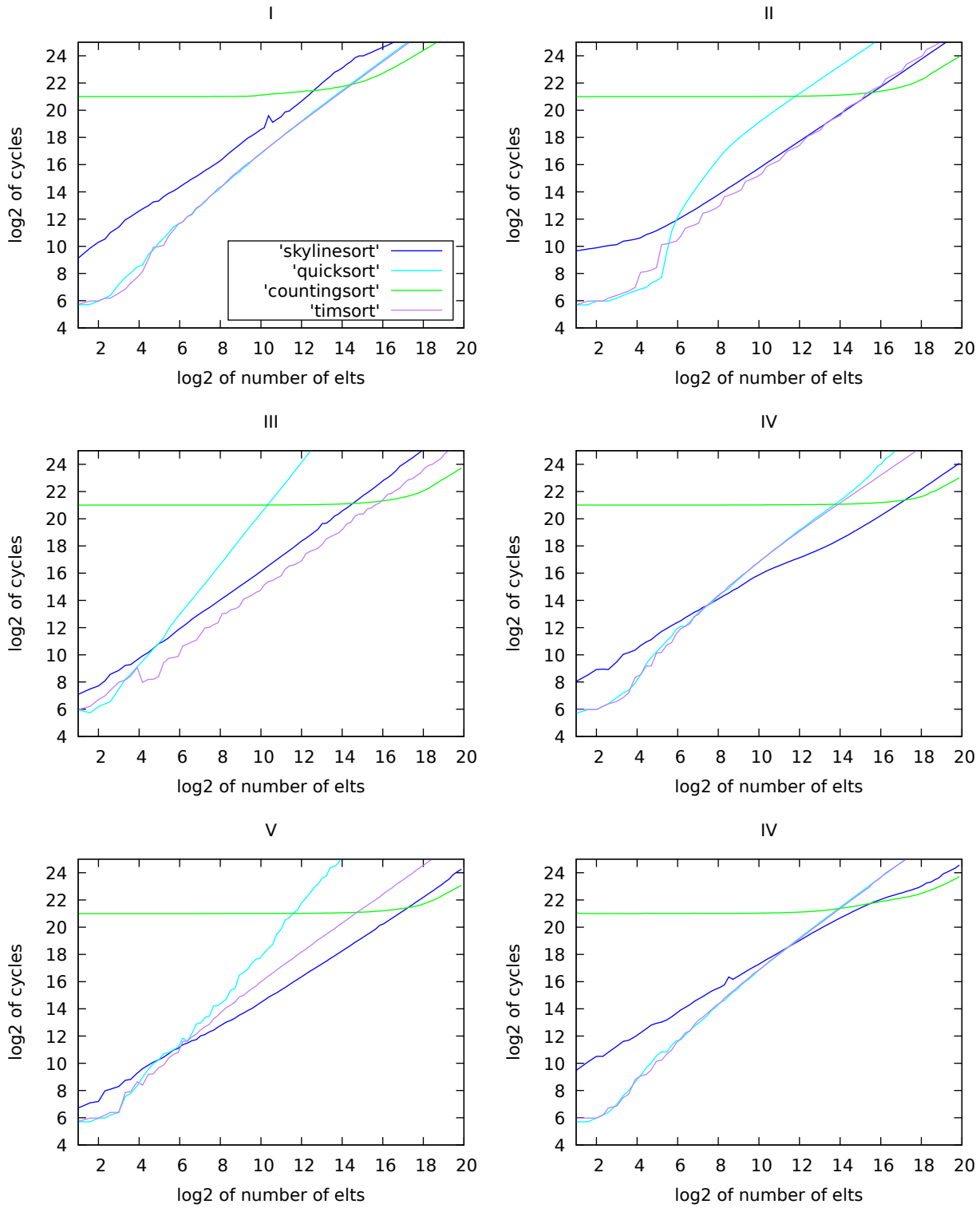

Figure 4: Speed tests comparing skylinesort with state-of-the-art sorting algorithms. Lower is better. I. Uniform random. II. Linear ascending. III. Linear descending. IV. Square roots random. V. Hyperbolic random. VI. Zip codes as presented in US Census data.

Catsort

Catsort also sorts integers, but faster than skylinesort. While its average case has also proved difficult to find, its worst case is quite good. Its scatter phase is both fast and allows for an optimal gather phase. The gather phase is the loop,

```
while(i>0){
  i=aux[i];
  v[j++]=i;
}
```

where `aux` is the auxiliary vector, in which gathering takes place, `v` is the output vector, `i` is initialized to `aux[0]`, and `j` is initialized to 0. The critical part is the second line, `i=aux[i]`. This means that in the auxiliary vector every nonzero element will lead to the next.

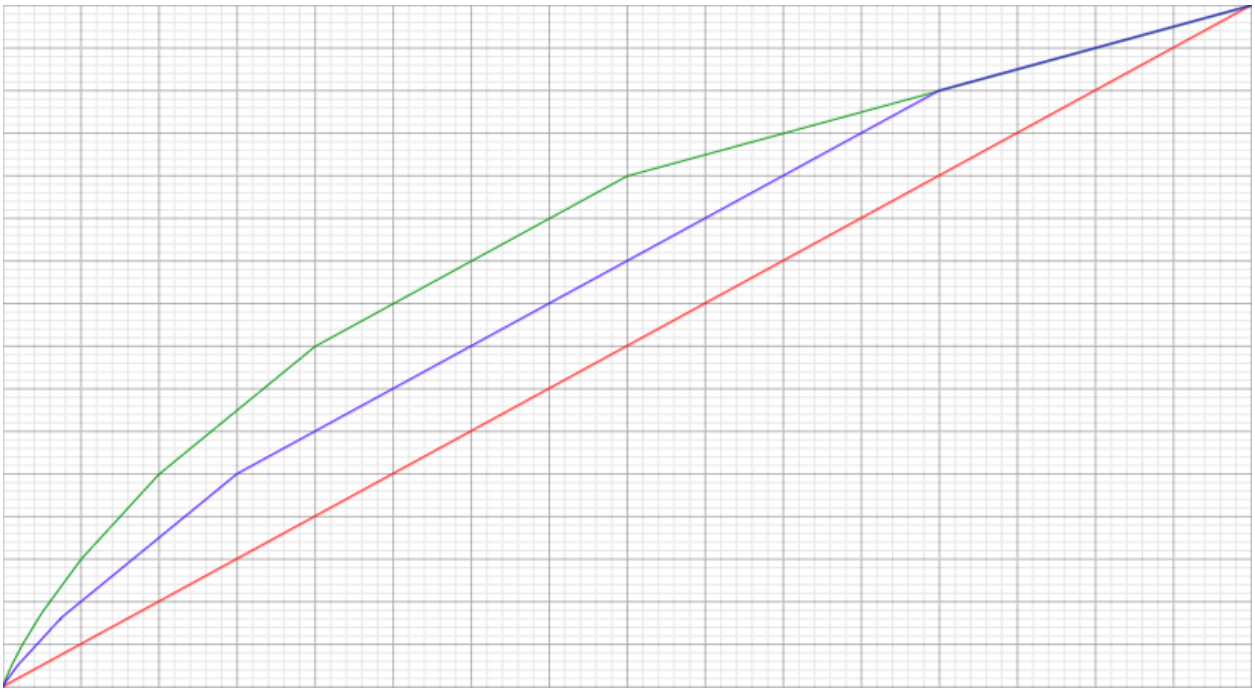

Figure 5: Random access performance component: 1-bit version (green) vs. 2-bit version (blue) vs. best case for all versions (red, linear).

Guesser 1.0

In mathematics, there is a technique known as "guess-and-check," and it is considered an undesirable habit very young mathematicians develop for solving problems when learning algebra. It largely relies on luck in this setting, but it doesn't have to. If there was a way to check all possible functions in some orderly fashion, the process would be deterministic and an attractive option. First, it would be guaranteed to find the right answer, if one exists, eventually. Second, depending on the order in which it evaluated functions, it could be made to always find the simplest answer first, which has the added benefit of being the smallest and usually the most efficient to compute. Finally, even in situations where no simple answer exists, it rules out the existence of such a simple answer in a rigorous way because it operated by exhaustion.

That is what the Guesser 1.0 is. It is an engine for generating functions of a given complexity—out of a given set of basic functions called primitives—exhaustively, and checking them all to see if they map a set of inputs into a corresponding desired output.

What is the complexity of a function though? The actual definition is part of the secret sauce but suffice to say it is based on the number of symbols that constitute a function, it is objective, and it is a natural number.

When all the functions up to a given complexity have been tried and erred, the existence of a simple explanation can be ruled out. The guessing process either produces the correct function that maps the inputs to the output, or proves by exhaustion that no simple-enough function exists.

Example 1

Let us take an example. Suppose we need to find the relationship between two input variables, $\{x, y\}$, and an output variable z. Given the three data points in the table below, what is the function f where $f(x, y) \to z$? Note that there's infinitely many functions f that fit the data, but we want to find a simple one because of Occam's Razor: the simplest answer is usually the best.

x	y	z
2	1	2
3	2	8
5	3	9

The only source of bias in applying the guesser to the problem will be deciding which primitive operators to use to solve the problem. For this example, we include addition, multiplication, subtraction, incrementing, modulo, integer part square root ($isqrt(n) = \lfloor\sqrt{n}\rfloor$), and the remainder of the integer part of the square root ($modsqrt(n) = n - \lfloor\sqrt{n}\rfloor^2$).

After running the guesser on the x, y, z, and the above primitives, the result is equal to

$$f(x, y) = modsqrt(x^2y + xy).$$

No simpler f exists.

Example 2

For another example, take the following subroutine of skylinesort, the one that updates x based on drawing a line to the left from its position:

```
x=((x+1)&x)-1;
```

This function was found by applying the guesser on a table of values. Here were used the values of the starting and ending points of drawing lines to the left.

Finding this function by hand would have taken days and ultimately would have meant to do exactly the same process as the guesser, but at human speed. In addition, performing this work by exhaustion added value in that it ruled out the existence of an even faster and simpler function that could do the same work.

The smaller function speeds up the gather phase by enough to make skylinesort up to 12.9% faster overall, and more on architectures with the appropriate assembly instructions.

Guesser 2.0

Guesser 1.0 tops out at evaluating 38,000 functions per second. Guesser 2.0 exceeds 1,000,000 functions per second. Beside that, the search space is much reduced, as shown in the graph below. This means that when searching for a function like the definition of the bell-curve,

$$\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

which has a complexity of about 20, Guesser 2.0 would be over a billion times faster than Guesser 1.0. For an idea of what that speedup means in human terms, it is like the relative difference between the blink of an eye and a human lifetime.

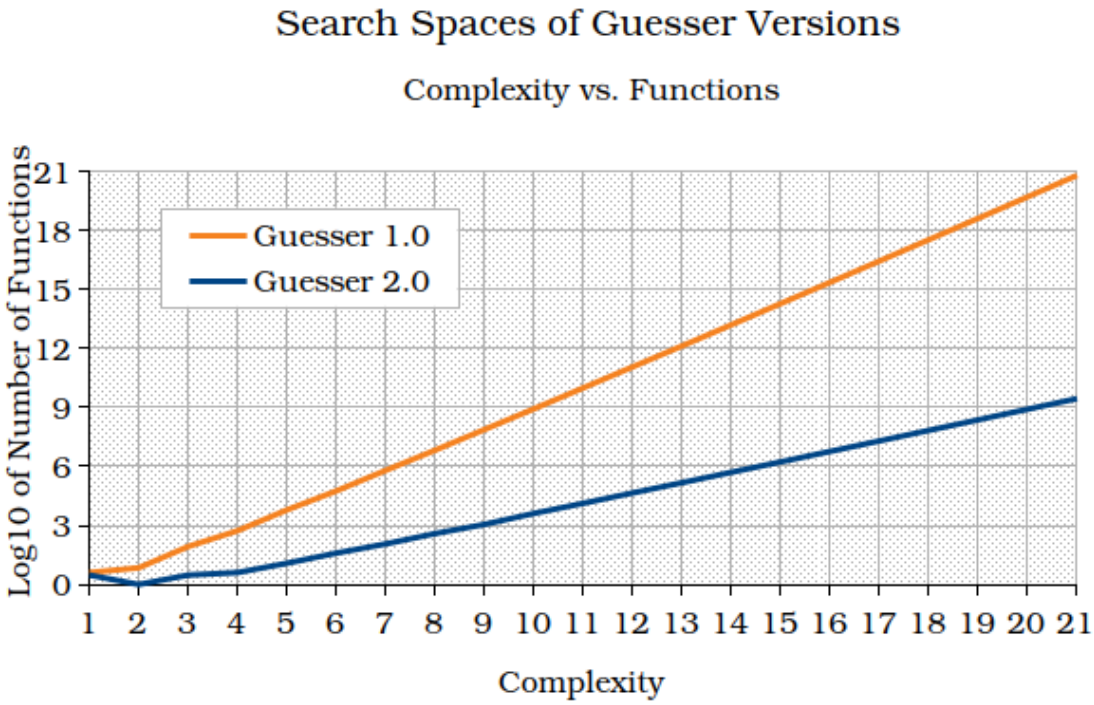

Figure 6: Complexity vs. number of functions of that complexity for each guesser. Y axis logarithmic. Guesser 2.0 searches through roughly the square root of the number of functions while still checking search spaces completely.

19766: Asynchronous Virus

At 18 bits, 19766, or x04D36 in hexadecimal, appears to be both the smallest and fastest known virus. At an equivalent of 9 base pairs, it is smaller, in terms of its information, than any other DNA or computer virus. It is even smaller than a single pixel's color on modern monitors. Its lifecycle has been measured at just about 9 nanoseconds on average, so cursory research indicates it is also the fastest. But it is safe: it only works on a chip called GA144, which is a matrix of 144 small computers. In this chip, 19766 hops from computer to computer as they communicate through their shared ports.

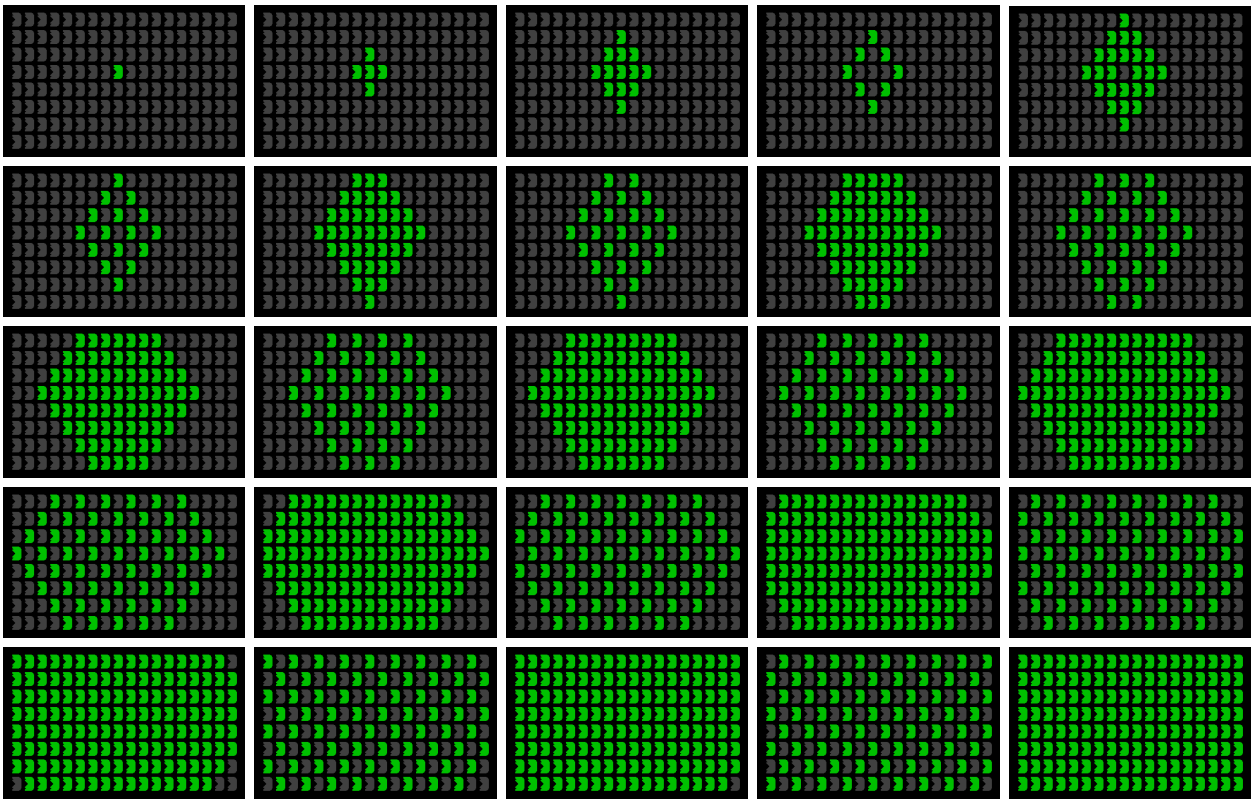

Figure 7: Infection of chip by 19766. It reaches all corners of the chip in 117.4 nanoseconds in this scenario. Each of the 25 rectangles is a still of the chip, and within these large rectangles, each of the 144 notched rectangles is a computer: green are active and gray inactive.