EXCERPTED FROM

STEPHEN WOLFRAM
A NEW KIND OF SCIENCE

NOTES FOR CHAPTER 10:

Processes of Perception and Analysis

Processes of Perception and Analysis

Defining the Notion of Randomness

■ **Page 554 · Algorithmic information theory.** A description of a piece of data can always be thought of as some kind of program for reproducing the data. So if one could find the shortest program that works then this must correspond to the shortest possible description of the data—and in algorithmic information theory if this is no shorter than the data itself then the data is considered to be algorithmically random.

How long the shortest program is for a given piece of data will in general depend on what system is supposed to run the program. But in a sense the program will on the whole be as short as possible if the system is universal (see page 642). And between any two universal systems programs can differ in length by at most a constant: for one can always just add a fixed interpreter program to the programs for one system in order to make them run on the other system.

As mentioned in the main text, any data generated by a simple program can by definition never be algorithmically random. And so even though algorithmic randomness is often considered in theoretical discussions (see note below) it cannot be directly relevant to the kind of randomness we see in so many systems in this book—or, I believe, in nature.

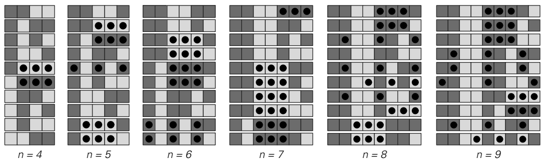
If one considers all 2^n possible sequences (say of 0's and 1's) of length n then it is straightforward to see that most of them must be more or less algorithmically random. For in order to have enough programs to generate all 2^n sequences most of the programs one uses must themselves be close to length n. (In practice there are subtleties associated with the encoding of programs that make this hold only for sufficiently large n.) But even though one knows that almost all long sequences must be algorithmically random, it turns out to be undecidable in general whether any particular sequence is algorithmically random. For in general one can give no upper limit to how much computational effort one might have to expend in order to find out whether any given short program—after any number of steps—will generate the sequence one wants.

But even though one can never expect to construct them explicitly, one can still give formal descriptions of sequences that are algorithmically random. An example due to Gregory Chaitin is the digits of the fraction Ω of initial conditions for which a universal system halts (essentially a compressed version—with various subtleties about limits—of the sequence from page 1127 giving the outcome for each initial condition). As emphasized by Chaitin, it is possible to ask questions purely in arithmetic (say about sequences of values of a parameter that yield infinite numbers of solutions to an integer equation) whose answers would correspond to algorithmically random sequences. (See page 786.)

As a reduced analog of algorithmic information theory one can for example ask what the simplest cellular automaton rule is that will generate a given sequence if started from a single black cell. Page 1186 gives some results, and suggests that sequences which require more complicated cellular automaton rules do tend to look to us more complicated and more random.

■ **History.** Randomness and unpredictability were discussed as general notions in antiquity in connection both with questions of free will (see page 1135) and games of chance. When probability theory emerged in the mid-1600s it implicitly assumed sequences random in the sense of having limiting frequencies following its predictions. By the 1800s there was extensive debate about this, but in the early 1900s with the advent of statistical mechanics and measure theory the use of ensembles (see page 1020) turned discussions of probability away from issues of randomness in individual sequences. With the development of statistical hypothesis testing in the early 1900s various tests for randomness were proposed (see page 1084). Sometimes these were claimed to have some kind of general significance, but mostly they were just viewed as simple practical methods. In many fields outside of statistics, however, the idea persisted even to the 1990s that block frequencies (or flat frequency spectra) were somehow the only ultimate tests for randomness. In 1909 Emile Borel had formulated the notion of normal numbers (see page 912) whose infinite digit sequences contain all blocks with equal frequency. And in the 1920s Richard von Mises—attempting to capture the observed lack of systematically successful gambling schemes—suggested that randomness for individual infinite sequences could be defined in general by requiring that "collectives" consisting of elements appearing at positions specified by any procedure should show equal frequencies. To disallow procedures say specially set up to pick out all the infinite number of 1's in a sequence Alonzo Church in 1940 suggested that only procedures corresponding to finite computations be considered. (Compare page 1021 on coarse-graining in thermodynamics.) Starting in the late 1940s the development of information theory began to suggest connections between randomness and inability to compress data, but emphasis on $p\,Log[p]$ measures of information content (see page 1071) reinforced the idea that block frequencies are the only real criterion for randomness. In the early 1960s, however, the notion of algorithmic randomness (see note above) was introduced by Gregory Chaitin, Andrei Kolmogorov and Ray Solomonoff. And unlike earlier proposals the consequences of this definition seemed to show remarkable consistency (in 1966 for example Per Martin-Löf proved that in effect it covered all possible statistical tests)—so that by the early 1990s it had become generally accepted as the appropriate ultimate definition of randomness. In the 1980s, however, work on cryptography had led to the study of some slightly weaker definitions of randomness based on inability to do cryptanalysis or make predictions with polynomial-time computations (see page 1089). But quite what the relationship of any of these definitions might be to natural science or everyday experience was never much discussed. Note that definitions of randomness given in dictionaries tend to emphasize lack of aim or purpose, in effect following the common legal approach of looking at underlying intentions (or say at physical construction of dice) rather than trying to tell if things are random from their observed behavior.

■ **Inevitable regularities and Ramsey theory.** One might have thought that there could be no meaningful type of regularity that would be present in all possible data of a given kind. But through the development since the late 1920s of Ramsey theory it has become clear that this is not the case. As one example, consider looking for runs of m equally spaced squares of the same color embedded in sequences of black and white squares of length n. The pictures below show results with $m = 3$ for various n. For $n < 9$ there are always some sequences in which no runs of length 3 exist. But it turns out that for $n \geq 9$ every single possible sequence contains at least one run of length 3. For any m the same is true for sufficiently large n; it is known that $m = 4$ requires $n \geq 35$ and $m = 5$ requires $n \geq 178$. (In problems like this the analog of n often grows extremely rapidly with m.) If one has a sufficiently long sequence, therefore, just knowing that a run of equally spaced identical elements exists in it does not narrow down at all what the sequence actually is, and can so cannot ultimately be considered a useful regularity.



(Compare pattern-avoiding sequences on page 944.)

Defining Complexity

■ **Page 557 · History.** There have been terms for complexity in everyday language since antiquity. But the idea of treating complexity as a coherent scientific concept potentially amenable to explicit definition is quite new: indeed this became popular only in the late 1980s—in part as a result of my own efforts. That what one would usually call complexity can be present in mathematical systems was for example already noted in the 1890s by Henri Poincaré in connection with the three-body problem (see page 972). And in the 1920s the issue of quantifying the complexity of simple mathematical formulas had come up in work on assessing statistical models (compare page 1083). By the 1940s general comments about biological, social and occasionally other systems being characterized by high complexity were common, particularly in connection with the cybernetics movement. Most often complexity seems to have been thought of as associated with the presence of large numbers of components with different types or behavior, and typically also with the presence of extensive interconnections or interdependencies. But occasionally—especially in some areas of social science—complexity was instead thought of as being characterized by somehow going beyond what human minds can handle. In the 1950s there was some discussion in pure mathematics of notions of complexity associated variously with sizes of axioms for logical theories, and with numbers of ways to satisfy such axioms. The development of information theory in the late 1940s—followed by the discovery of the structure of DNA in 1953—led to the idea that perhaps complexity might be related to information content. And when the notion of algorithmic information content as the length of a shortest program (see page 1067) emerged in the 1960s it was suggested that this might be an appropriate definition for complexity. Several other definitions used in specific fields in the 1960s and 1970s were also based on sizes of descriptions: examples were optimal orders of models in systems theory, lengths of logic expressions for circuit and program design, and numbers of factors in Krohn-Rhodes decompositions of semigroups. Beginning in the 1970s computational complexity theory took a somewhat different direction, defining what it called complexity in terms of resources needed to perform computational tasks. Starting in the 1980s with the rise of complex systems research (see page 862) it was considered important by many physicists to find a definition that would provide some kind of numerical measure of complexity. It was noted that both very ordered and very disordered systems normally seem to be of low complexity, and much was made of the observation that systems on the border between these extremes—particularly class 4 cellular automata—seem to have higher complexity. In addition, the presence of some kind of hierarchy was often taken to indicate higher complexity, as was evidence of computational capabilities. It was also usually assumed that living systems should have the highest complexity—perhaps as a result of their long evolutionary history. And this made informal definitions of complexity often include all sorts of detailed features of life (see page 1178). One attempt at an abstract definition was what Charles Bennett called logical depth: the number of computational steps needed to reproduce something from its shortest description. Many simpler definitions of complexity were proposed in the 1980s. Quite a few were based just on changing $p_i\, Log[p_i]$ in the definition of entropy to a quantity vanishing for both ordered and disordered p_i. Many others were based on looking at correlations and mutual information measures—and using the fact that in a system with many interdependent and potentially hierarchical parts this should go on changing as one looks at more and more. Some were based purely on fractal dimensions or dimensions associated with strange attractors. Following my 1984 study of minimal sizes of finite automata capable of reproducing states in cellular automaton evolution (see page 276) a whole series of definitions were developed based on minimal sizes of descriptions in terms of deterministic and probabilistic finite automata (see page 1084). In general it is possible to imagine setting up all sorts of definitions for quantities that one chooses to call complexity. But what is most relevant for my purposes in this book is instead to find ways to capture everyday notions of complexity—and then to see how systems can produce these. (Note that since the 1980s there has been interest in finding measures of complexity that instead for example allow maintainability and robustness of software and management systems to be assessed. Sometimes these have been based on observations of humans trying to understand or verify systems, but more often they have just been based for example on simple properties of networks that define the flow of control or data—or in some cases on the length of documentation needed.) (The kind of complexity discussed here has nothing directly to do with complex numbers such as $\sqrt{-1}$ introduced into mathematics since the 1600s.)

Data Compression

▪ **Practicalities.** Data compression is important in making maximal use of limited information storage and transmission capabilities. One might think that as such capabilities increase, data compression would become less relevant. But so far this has not been the case, since the volume of data always seems to increase more rapidly than capabilities for storing and transmitting it. In the future, compression is always likely to remain relevant when there are physical constraints—such as transmission by electromagnetic radiation that is not spatially localized.

▪ **History.** Morse code, invented in 1838 for use in telegraphy, is an early example of data compression based on using shorter codewords for letters such as "e" and "t" that are more common in English. Modern work on data compression began in the late 1940s with the development of information theory. In 1949 Claude Shannon and Robert Fano devised a systematic way to assign codewords based on probabilities of blocks. An optimal method for doing this was then found by David Huffman in 1951. Early implementations were typically done in hardware, with specific choices of codewords being made as compromises between compression and error correction. In the mid-1970s, the idea emerged of dynamically updating codewords for Huffman encoding, based on the actual data encountered. And in the late 1970s, with online storage of text files becoming common, software compression programs began to be developed, almost all based on adaptive Huffman coding. In 1977 Abraham Lempel and Jacob Ziv suggested the basic idea of pointer-based encoding. In the mid-1980s, following work by Terry Welch, the so-called LZW algorithm rapidly became the method of choice for most general-purpose compression systems. It was used in programs such as PKZIP, as well as in hardware devices such as modems. In the late 1980s, digital images became more common, and standards for compressing them emerged. In the early 1990s, lossy compression methods (to be discussed in the next section) also began to be widely used. Current image compression standards include: FAX CCITT 3 (run-length encoding, with codewords determined by Huffman coding from a definite distribution of run lengths); GIF (LZW); JPEG (lossy discrete cosine transform, then Huffman or arithmetic coding); BMP (run-length encoding, etc.); TIFF (FAX, JPEG, GIF, etc.). Typical compression ratios currently achieved for text are around 3:1, for line diagrams and text images around 3:1, and for photographic images around 2:1 lossless, and 20:1 lossy. (For sound compression see page 1080.)

■ **Page 560 · Number representations.** The sequence of 1's and 0's representing a number *n* are obtained as follows:

(a) *Unary.* *Table[0, {n}]*. (Not self-delimited.)

(b) *Ordinary base 2.* *IntegerDigits[n, 2]*. (Not self-delimited.)

(c) *Length prefixed.* Starting with an ordinary base 2 digit sequence, one prepends a unary specification of its length, then a specification of that length specification, and so on:

```
(Flatten[{Sign[-Range[1 - Length[#], 0]], #}] &)[
  Map[Rest, IntegerDigits[Rest[Reverse[NestWhileList[
            Floor[Log[2, #]] &, n + 1, # > 1 &]]], 2]]]
```

(d) *Binary-coded base 3.* One takes base 3 representation, then converts each digit to a pair of base 2 digits, handling the beginning and end of the sequence in a special way.

```
Flatten[IntegerDigits[
  Append[2 - With[{w = Floor[Log[3, 2 n]]},
     IntegerDigits[n - (3^(w+1) - 1)/2, 3, w]], 3], 2, 2]]
```

(e) *Fibonacci encoding.* Instead of decomposing a number into a sum of powers of an integer base, one decomposes it into a sum of Fibonacci numbers (see page 902). This decomposition becomes unique when one requires that no pair of 1's appear together.

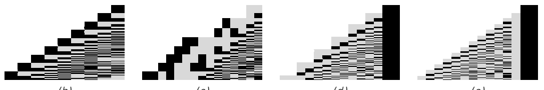
```
Apply[Take, RealDigits[(N[#, N[Log[10, #] + 3]] &)[
     n √5 /GoldenRatio^2 + 1/2], GoldenRatio]]
```

The representations of all the first *Fibonacci[n] - 1* numbers can be obtained from (the version in the main text has *Rest[RotateLeft[Join[#, {0, 1}]]] &* applied)

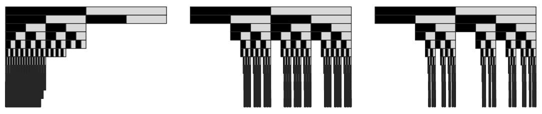
```
Apply[Join, Map[Last,
  NestList[{#[[2]], Join[Map[Join[{1, 0}, Rest[#]] &, #[[2]]],
      Map[Join[{1, 0}, #] &, #[[1]]]]} &, {{}, {{1}}}, n - 3]]]
```

■ **Lengths of representations.** (a) *n*, (b) *Floor[Log[2, n] + 1]*, (c) *Tr[FixedPointList[Max[0, Ceiling[Log[2, #]]] &, n + 2]] - n - 3*, (d) *2 Ceiling[Log[3, 2 n + 1]]*, (e) *Floor[Log[GoldenRatio, $\sqrt{5}$ (n + 1/2)]]*. Large *n* approximations: (a) *n*, (b) *Log[2, n]*, (c) *Log[2, n] + Log[2, Log[2, n]] + ...*, (d) *2 Log[3, n]*, (e) *Log[GoldenRatio, n]*.

Shown on a logarithmic scale, representations (b) through (e) (given here for numbers 1 through 500) all grow roughly linearly:



■ **Completeness.** If one successively reads 0's and 1's from an infinite sequence then the representations (c), (d) and (e) have the property that eventually one will always accumulate a valid representation for some number or another. The pictures below show which sequences of 0's and 1's correspond to complete numbers in these representations. Every vertical column is a possible sequence of 0's and 1's, and the column is shown to terminate when a complete number is obtained.



With an infinite random sequence of 0's and 1's, different number representations yield different distributions of sizes of numbers. Representation (b), for example, is more weighted towards large numbers, while (c) is more weighted towards small numbers. Maximal compression for a sequence of numbers with a particular distribution of sizes is obtained by choosing a representation that yields a matching such distribution. (See also page 949.)

■ **Practical computing.** Numbers used for arithmetic in practical computing are usually assumed to have a fixed length of, say, 32 bits, and thus do not need to be self-delimiting. In *Mathematica*, where integers can be of essentially any size, a representation closer to (b) above is used.

■ **Page 561 · Run-length encoding.** Data can be converted to run lengths by *Map[Length, Split[data]]*. Each number is then replaced by its representation.

With completely random input, the output will on average be longer by a factor *Sum[$2^{-(n+1)}$ r[n], {n, 1, ∞}]* where *r[n]* is the length of the representation for *n*. For the Fibonacci encoding used in the main text, this factor is approximately 1.41028. (In base 2 this number has 1's essentially at positions *Fibonacci[n]*; as discussed on page 914, the number is transcendental.)

■ **Page 563 · Huffman coding.** From a list *p* of probabilities for blocks, the list of codewords can be generated using

```
Map[Drop[Last[#], -1] &, Sort[
   Flatten[MapIndexed[Rule, FixedPoint[Replace[Sort[#],
         {{p0_, i0_}, {p1_, i1_}, pi___} → {{p0 + p1, {i0, i1}},
           pi}] &, MapIndexed[List, p]][[1, 2]], {-1}]]]] - 1
```

Given the list of codewords *c*, the sequence of blocks that occur in encoded data *d* can be uniquely reconstructed using

```
First[{{}, d} //. MapIndexed[
      {{r___}, Flatten[{#1, s___}]} → {{r, #2[[1]]}, {s}} &, c]]
```

Note that the encoded data can consist of any sequence of 0's and 1's. If all 2^b possible blocks of length *b* occur with equal probability, then the Huffman codewords will consist of blocks equivalent to the original ones. In an opposite extreme, blocks with probabilities 1/2, 1/4, 1/8, ... will yield codewords of lengths 1, 2, 3, ...

In practical applications, Huffman coding is sometimes extended to allow the choice of codewords to be updated dynamically as more data is read.

■ **Maximal block compression.** If one has data that consists of a long sequence of blocks, each of length *b*, and each independently chosen with probability *p[i]* to be of type *i*, then as argued by Claude Shannon in the late 1940s, it turns out that the minimum number of base 2 bits needed on average to represent each block in such a sequence is *h = -Sum[p[i] Log[2, p[i]], {i, 2^b}]*. If all blocks occur with an equal probability of 2^{-b}, then *h* takes on its maximum possible value of *b*. If only one block occurs with nonzero probability then *h == 0*. Following Shannon, the quantity *h* (whose form is analogous to entropy in physics, as discussed on page 1020) is often referred to as "information content". This name, however, is very misleading. For certainly *h* does not in general give the length of the shortest possible description of the data; all it does is to give the shortest length of description that is obtained by treating successive blocks as if they occur with independent probabilities. With this assumption one then finds that maximal compression occurs if a block of probability *p[i]* is represented by a codeword of length *-Log[2, p[i]]*. Huffman coding with a large number of codewords will approach this if all the *p[i]* are powers of 1/2. (The self-delimiting of codewords leads to deviations for small numbers of codewords.) For *p[i]* that are not powers of 1/2, non-integer length codewords would be required. The method of arithmetic coding provides an alternative in which the output does not consist of separate codewords concatenated together. (Compare algorithmic information content discussed on pages 554 and 1067.)

■ **Arithmetic coding.** Consider dividing the interval from 0 to 1 into a succession of bins, with each bin having a width equal to the probability for some sequence of blocks to occur. The idea of arithmetic coding is to represent each such bin by the digit sequence of the shortest number within the bin—after trailing zeros have been dropped. For any sequence *s* this can be done using

```
Module[{c, m = 0},
 Map[c[#] = {m, m += Count[s, #]/Length[s]} &, Union[s]];
 Function[x, (First[RealDigits[2^# Ceiling[2^-# Min[x]],
     2, -#, -1]] &)[Floor[Log[2, Max[x] - Min[x]]]]][
  Fold[(Max[#1] - Min[#1]) c[#2] + Min[#1] &, {0, 1}, s]]]
```

Huffman coding of a sequence containing a single 0 block together with *n* 1 blocks will yield output of length about *n*; arithmetic coding will yield length about *Log[n]*. Compression in arithmetic coding still relies, however, on unequal block probabilities, just like in Huffman coding. Originally suggested in the early 1960s, arithmetic coding reemerged in the late 1980s when high-speed floating-point computation became common, and is occasionally used in practice.

■ **Page 565 · Pointer-based encoding.** One can encode a list of data *d* by generating pointers to the longest and most recent copies of each subsequence of length at least *b* using

```
PEncode[d_, b_ : 4] := Module[{i, a, u, v},
   i = 2; a = {First[d]}; While[i ≤ Length[d], {u, v} =
      Last[Sort[Table[{MatchLength[d, i, j], j}, {j, i - 1}]]];
    If[u ≥ b, AppendTo[a, p[i - v, u]]; i += u,
      AppendTo[a, d[[i]]]; i++]]; a]

MatchLength[d_, i_, j_] := With[{m = Length[d] - i}, Catch[
   Do[If[d[[i + k]] =!= d[[j + k]], Throw[k]], {k, 0, m}]; m + 1]]
```

The process of encoding can be made considerably faster by keeping a dictionary of previously encountered subsequences. One can reproduce the original data using

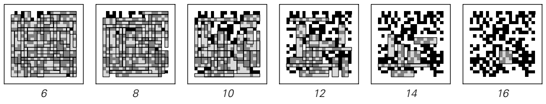
```
PDecode[a_] := Module[{d = Flatten[
    a /. p[j_, r_] :→ Table[p[j], {r}]]}, Flatten[MapIndexed[
   If[Head[#1] === p, d[[#2]] = d[[#2 - First[#1]]], #1] &, d]]]
```

To get a representation purely in terms of 0 and 1, one can use a self-delimiting representation for each integer that appears. (Knowing the explicit representation one could then determine whether each block would be shorter if encoded literally or using a pointer.) The encoded version of a purely repetitive sequence of length *n* has a length that grows like *Log[n]*. The encoded version of a purely nested sequence grows like *Log[n]*^{2}. The encoded version of a sufficiently random sequence grows like *n* (with the specific encoding used in the text, the length is about *2 n*). Note that any sequence of 0's and 1's corresponds to the beginning of the encoding for some sequence or another.

It is possible to construct sequences whose encoded versions grow roughly like fractional powers of *n*. An example is the sequence *Table[Append[Table[0, {r}], 1], {r, s}]* whose encoded version grows like $\sqrt{n}$ *Log[n]*. Cyclic tag systems often seem to produce sequences whose encoded versions grow like fractional powers of n. Sequences produced by concatenation sequences are not typically compressed by pointer encoding.

With completely random input, the probability that the length b subsequence which begins at element n is a repeat of a previous subsequence is roughly $1-(1-2^{-b})^{n-1}$. The overall fraction of a length n input that consists of repeats of length at least b is greater than $1-2^b/n$ and is essentially

$$1 - Sum[(1-2^{-b})^i\, Product[1 + (1-2^{-b})^j - (1-2^{-b-1})^j, \{j, i-b+1, i-1\}], \{i, b, n-b\}]/(n-2b+1)$$



■ **LZW algorithms.** Practical implementations of pointer-based encoding can maintain only a limited dictionary of possible repeats. Various schemes exist for optimizing the construction, storage and rewriting of such dictionaries.

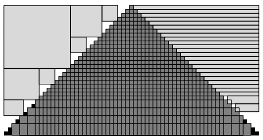
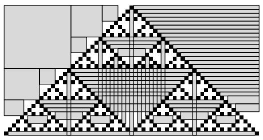
■ **Page 568 · Recursive subdivision.** In one dimension, encoding can be done using

```
Subdivide[a_] := Flatten[
    If[Length[a] == 2, a, If[Apply[SameQ, a], {1, First[a]},
        {0, Map[Subdivide, Partition[a, Length[a]/2]]}]]]
```

In n dimensions, it can be done using

```
Subdivide[a_, n_] := With[{s = Table[1, {n}]}, Flatten[
    If[Dimensions[a] == 2 s, a, If[Apply[SameQ, Flatten[a]],
        {1, First[Flatten[a]]}, {0, Map[Subdivide[#, n] &,
            Partition[a, 1/2 Length[a] s], {n}]}]]]]
```

■ **2D run-length encoding.** A simple way to generalize run-length encoding to two dimensions is to scan data one row after another, always finding the largest rectangle of uniform color that starts at each particular point. The pictures below show regions with an area of more than 10 cells found in this way. The presence of so many thin and overlapping regions prevents good compression.

2D run-length encoding can also be done by scanning the data according to a more complicated space-filling curve, of the kind discussed on page 893.

Irreversible Data Compression

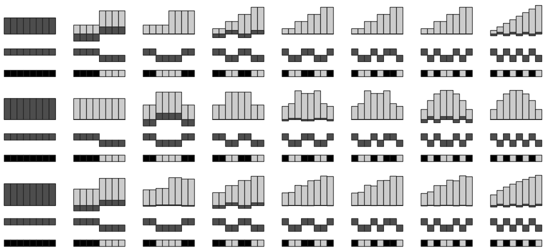
■ **History.** The idea of creating sounds by adding together pure tones goes back to antiquity. At a mathematical level, following work by Joseph Fourier around 1810 it became clear by the mid-1800s how any sufficiently smooth function could be decomposed into sums of sine waves with frequencies corresponding to successive integers. Early telephony and sound recording in the late 1800s already used the idea of compressing sounds by dropping high- and low-frequency components. From the early days of television in the 1950s, some attempts were made to do similar kinds of compression for images. Serious efforts in this direction were not made, however, until digital storage and processing of images became common in the late 1980s.

■ **Orthogonal bases.** The defining feature of a set of basic forms is that it is complete, in the sense that any piece of data can be built up by adding the basic forms with appropriate weights. Most sets of basic forms used in practice also have the feature of being orthogonal, which turns out to make it particularly easy to work out the weights for a given piece of data. In 1D, a basic form $a[[i]]$ is just a list. Orthogonality is then the property that $a[[i]] . a[[j]] == 0$ for all $i \neq j$. And when this property holds, the weights are given essentially just by $data . a$.

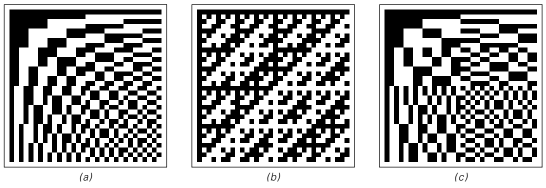
The concept of orthogonal bases was historically worked out first in the considerably more difficult case of continuous functions. Here a typical orthogonality property is $Integrate[f[r, x] f[s, x], \{x, 0, 1\}] == KroneckerDelta[r, s]$. As discovered by Joseph Fourier around 1810, this is satisfied for basis functions such as $Sin[2 n \pi x]/\sqrt{2}$.

■ **Page 573 · Walsh transforms.** The basic forms shown in the main text are 2D Walsh functions—represented as ± 1 matrices. Each collection of such functions can be obtained from lists of vectors representing 1D Walsh functions by using `Outer[Outer[Times, ##] &, b, b, 1, 1]`, or equivalently `Map[Transpose, Map[# b &, b, {2}]]`.

The pictures below show how 1D arrays of data values can be built up by adding together 1D Walsh functions. At each step the Walsh function used is given underneath the array of values obtained so far.



The components of the vectors for 1D Walsh functions can be ordered in many ways. The pictures below show the complete matrices of basis vectors obtained with three common orderings.



The matrices for size $n = 2^s$ can be obtained from

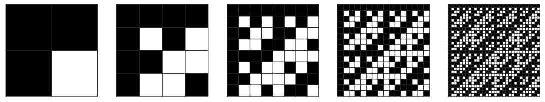
```
Nest[Apply[Join, f[{Map[Flatten[Map[{#, #} &, #]] &, #],
    Map[Flatten[Map[{#, -#} &, #]] &, g[#]]}]] &, {{1}}, s]
```

with (a) *f = Identity*, *g = Reverse*, (b) *f = Transpose*, *g = Identity*, and (c) *f = g = Identity*. (a) is used in the main text. Known as sequency order, it has the property that each row involves one more change of color than the previous row. (b) is known as natural or Hadamard order. It exhibits a nested structure, and can be obtained as in the pictures below from the evolution of a 2D substitution system, or equivalently from a Kronecker product as in

```
Nest[Flatten2D[Map[# {{1, 1}, {1, -1}} &, #, {2}]] &, {{1}}, s]
```

with

```
Flatten2D[a_] :=
  Apply[Join, Apply[Join, Map[Transpose, a], {2}]]
```



(c) is known as dyadic or Paley order. It is related to (a) by Gray code reordering of the rows, and to (b) by reordering according to (see page 905)

```
BitReverseOrder[a_] :=
  With[{n = Length[a]}, a[[Map[FromDigits[Reverse[#], 2] &,
      IntegerDigits[Range[0, n - 1], 2, Log[2, n]]] + 1]]]
```

It is also given by

```
Array[Apply[Times, (-1)^(IntegerDigits[#1, 2, s]
        Reverse[IntegerDigits[#2, 2, s]])] &, 2^{s, s}, 0]
```

where (b) is obtained simply by dropping the *Reverse*.

Walsh functions can correspond to nested sequences. The function at position *2/3 (1 + 4^(-(Floor[s/2] + 1/2))) 2^s* in basis (a), for example, is exactly the Thue-Morse sequence (with 0 replaced by -1) from page 83.

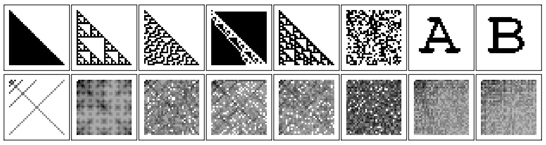
Given the matrix *m* of basis vectors, the Walsh transform is simply *data . m*. Direct evaluation of this for length *n* takes n^2 steps. However, the nested structure of *m* in natural order allows evaluation in only about *n Log[n]* steps using

```
Nest[Flatten[Transpose[Partition[#, 2] . {{1, 1}, {1, -1}}]] &,
  data, Log[2, Length[data]]]
```

This procedure is similar to the fast Fourier transform discussed below. Transforms of 2D data are equivalent to 1D transforms of flattened data.

Walsh functions were used by electrical engineers such as Frank Fowle in the 1890s to find transpositions of wires that minimized crosstalk; they were introduced into mathematics by Joseph Walsh in 1923. Raymond Paley introduced the dyadic basis in 1932. Mathematical connections with harmonic analysis of discrete groups were investigated from the late 1940s. In the 1960s, Walsh transforms became fairly widespread in discrete signal and image processing.

■ **Page 575 · Walsh spectra.** The arrays of absolute values of weights of basic forms for successive images are as follows:

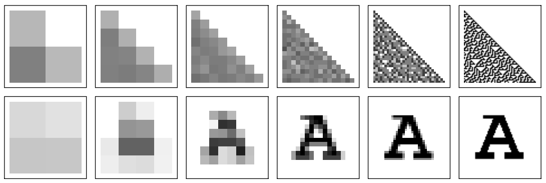


■ **Hadamard matrices.** Hadamard matrices are *n×n* matrices with elements -1 and +1, whose rows are orthogonal, so that *m . Transpose[m] == n IdentityMatrix[n]*. The matrices used in Walsh transforms are special cases with $n = 2^s$. There are thought to be Hadamard matrices with every size *n = 4 k* (and for *n > 2* no other sizes are possible); the number of distinct such matrices for each *k* up to 7 is 1, 1, 1, 5, 3, 60, 487. The so-called Paley family of Hadamard matrices for *n = 4 k = p + 1* with *p* prime are given by

```
PadLeft[Array[JacobiSymbol[#2 - #1, n - 1] &, {n, n} - 1] -
    IdentityMatrix[n - 1], {n, n}, 1]
```

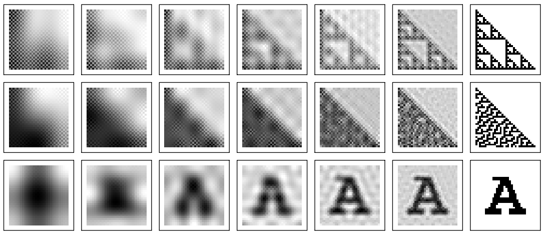
Originally introduced by Jacques Hadamard in 1893 as the matrices with elements *Abs[a] ≤ 1* which attain the maximal possible determinant $\pm n^{n/2}$, Hadamard matrices appear in various combinatorial problems, particularly design of exhaustive combinations of experiments and Reed-Muller error-correcting codes.

■ **Image averaging.** Walsh functions yield significantly better compression than simple successive averaging of 2×2 blocks of cells, as shown below.



■ **Practical image compression.** Two basic phenomena contribute to our ability to compress images in practice. First, that typical images of relevance tend to be far from random—indeed they often involve quite limited numbers of distinct objects. And second, that many fine details of images go unnoticed by the human visual system (see the next section).

■ **Fourier transforms.** In a typical Fourier transform, one uses basic forms such as *Exp[ⅈ π r x/n]* with *r* running from 1 to *n*. The weights associated with these forms can be found using *Fourier*, and given these weights the original data can also be reconstructed using *InverseFourier*. The pictures below show what happens in such a so-called discrete cosine transform when different fractions of the weights are kept, and others are effectively set to zero. High-frequency wiggles associated with the so-called Gibbs phenomenon are typical near edges.



Fourier[data] can be thought of as multiplication by the $n \times n$ matrix *Array[Exp[2 π ⅈ #1 #2/n] &, {n, n}, 0]*. Applying *BitReverseOrder* to this matrix yields a matrix which has an essentially nested form, and for size $n = 2^s$ can be obtained from

```
Nest[With[{c = BitReverseOrder[Range[0, Length[#] - 1]/
     Length[#]]}, Flatten2D[MapIndexed[#1 {{1, 1},
      {1, -1}(-1)^c[[Last[#2]]]} &, #, {2}]]] &, {{1}}, s]
```

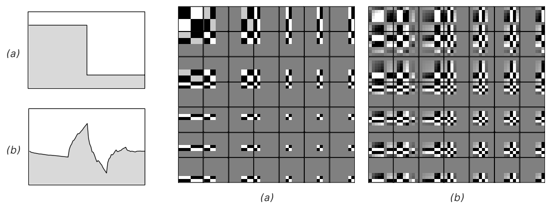
Using this structure, one obtains the so-called fast Fourier transform which operates in *n Log[n]* steps and is given by

```
With[{n = Length[data]}, Fold[Flatten[Map[With[
    {k = Length[#]/2}, {{1, 1}, {1, -1}} . {Take[#, k], Drop[
      #, k](-1)^(Range[0, k - 1]/k)}] &, Partition[##]]] &,
  BitReverseOrder[data], 2^Range[Log[2, n]]]/√n ]
```

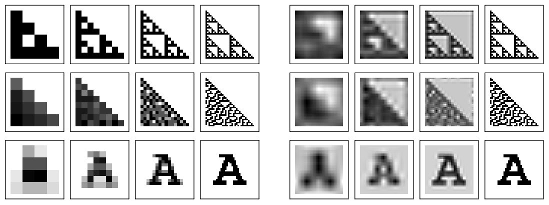
(See also page 1080.)

■ **JPEG compression.** In common use since the early 1990s JPEG compression works by first assigning color values to definite bins, then applying a discrete Fourier cosine transform, then applying Huffman encoding to the resulting weights. The "quality" of the image is determined by how many weights are kept; a typical default quality factor, used say by *Export* in *Mathematica*, is 75.

■ **Wavelets.** Each basic form in an ordinary Walsh or Fourier transform has nonzero elements spread throughout. With wavelets the elements are more localized. As noted in the late 1980s basic forms can be set up by scaling and translating just a single appropriately chosen underlying shape. The (a) Haar and (b) Daubechies wavelets *ψ[x]* shown below both have the property that the basic forms $2^{m/2}\ \psi[2^m x - n]$ (whose 2D analogs are shown as on page 573) are orthogonal for every different *m* and *n*.



The pictures below show images built up by keeping successively more of these basic forms. Sharp edges have fewer wiggles than with Fourier transforms.



■ **Sound compression.** See page 1080.

Visual Perception

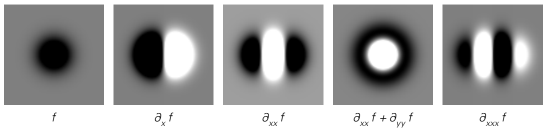
■ **Color vision.** The three types of color-sensitive cone cells on the human retina each have definite response curves as a function of wavelength. The perceived color of light with a given wavelength distribution is basically determined by the three numbers obtained by integrating these responses. For any wavelength distribution it turns out that if one scales these numbers to add up to one, then the chromaticity values obtained must lie within a certain region. Mixing *n* specific colors in different proportions allows one to reach any point in an *n*-cornered polytope. For $n = 3$ this polytope comes close to filling the region of all possible colors, but for no *n* can it completely fill it—which is why practical displays and printing processes can produce only limited ranges of colors.

An important observation, related to the fact that limitations in color ranges are usually not too troublesome, is that the perceived colors of objects stay more or less constant even when viewed in very different lighting, corresponding to very different wavelength distributions. In recent years it has become clear that the origin of this phenomenon is that beyond the original cone cells, most color-sensitive cells in our visual system respond not to absolute color levels, but instead to differences in color levels at slightly different positions. (Responses to nearby relative values rather than absolute values seem to be common in many forms of human perception.)

The fact that white light is a mixture of colors was noticed by Isaac Newton in 1704, and it became clear in the course of the 1700s that three primaries could reproduce most colors. Thomas Young suggested in 1802 that there might be three types of color receptors in the eye, but it was not until 1959 that these were actually identified—though on the basis of perceptual experiments, parametrizations of color space were already well established by the 1930s. While humans and primates normally have three types of cone cells, it has been found that other mammals normally have two, while birds, reptiles and fishes typically have between 3 and 5.

▪ **Nerve cells.** In the retina and the brain, nerve cells typically have an irregular tree-like structure, with between a few and a few thousand dendrites carrying input signals, and one or more axons carrying output signals. Nerve cells can respond on timescales of order milliseconds to changes in their inputs by changing their rate of generating output electrical spikes. As has been believed since the 1940s, most often nerve cells seem to operate at least roughly by effectively adding up their inputs with various positive or negative weights, then going into an excited state if the result exceeds some threshold. The weights seem to be determined by detailed properties of the synapses between nerve cells. Their values can presumably change to reflect certain aspects of the activity of the cell, thus forming a basis for memory (see page 1102). In organisms with a total of only a few thousand nerve cells, each individual cell typically has definite connections and a definite function. But in humans with perhaps 100 billion nerve cells, the physical connections seem quite haphazard, and most nerve cells probably develop their function as a result of building up weights associated with their actual pattern of behavior, either spontaneous or in response to external stimuli.

▪ **The visual system.** Connected to the 100 million or so light-sensitive photoreceptor cells on the retina are roughly two layers of nerve cells, with various kinds of cross-connections, out of which come the million fibers that form the optic nerve. After essentially one stop, most of these go to the primary visual cortex at the back of the brain, which itself contains more than 100 million nerve cells. Physical connections between nerve cells have usually been difficult to map. But starting in the 1950s it became possible to record electrical activity in single cells, and from this the discovery was made that many cells respond to rather specific visual stimuli. In the retina, most common are center-surround cells, which respond when there is a higher level of light in the center of a roughly circular region and a lower level outside, or vice versa. In the first few layers of the visual cortex about half the cells respond to elongated versions of similar stimuli, while others seem sensitive to various forms of change or motion. In the fovea at the center of the retina, a single center-surround cell seems to get input from just a few nearby photoreceptors. In successive layers of the visual cortex cells seem to get input from progressively larger regions. There is a very direct mapping of positions on the retina to regions in the visual cortex. But within each region there are different cells responding to stimuli at different angles, as well as to stimuli from different eyes. Cells with particular kinds of responses are usually found to be arranged in labyrinthine patterns very much like those shown on page 427. And no doubt the processes which produce these patterns during the development of the organism can be idealized by simple 2D cellular automata. Quite what determines the pattern of illumination to which a given cell will respond is not yet clear, although there is some evidence that it is the result of adaptation associated with various kinds of test inputs. Since the late 1970s, it has been common to assume that the response of a cell can be modelled by derivatives of Gaussians such as those shown below, or perhaps by Gabor functions given by products of trigonometric functions and Gaussians. Experiments have determined responses to these and other specific stimuli, but inevitably no experiment can find all the stimuli to which a cell is sensitive.

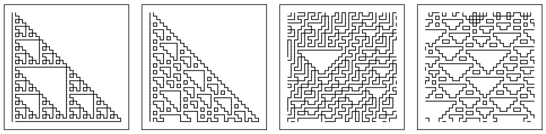


The visual systems of a number of specific higher and lower organisms have now been studied, and despite a few differences (such as cross-connections being behind the photoreceptors on the retinas of octopuses and squids, but in front in most higher animals), the same general features are usually seen. In lower organisms, there tend to be fewer layers of cells, with individual cells more specialized to particular visual stimuli of relevance to the organism.

▪ **Feedback.** Most of the lowest levels of visual processing seem to involve only signals going successively from one layer in the eye or brain to the next. But presumably there is at least some feedback to previous layers, yielding in effect iteration of rules like the ones used in the main text. The resulting evolution process is likely to have attractors, potentially explaining the fact that in images such as "Magic Eye" random dot stereograms features can pop out after several seconds or minutes of scrutiny, even without any conscious effort.

▪ **Scale invariance.** In a first approximation our recognition of objects does not seem to be much affected by overall size or overall light level. For light level—as with color constancy—this is presumably achieved by responding only to differences between levels at different positions. Probably the same effect contributes to scale invariance by emphasizing only edges and corners. And if one is looking at objects like letters, it helps that one has learned them at many different sizes. But also similar cells most likely receive inputs from regions with a range of different sizes on the retina—making even unfamiliar textures seem the same over at least a certain range of scales. When viewed at a normal reading distance of 12 inches each square in the picture on page 578 covers a region about 5 cells across on the retina. With good lighting and good eyesight the textures in the picture can still be distinguished at a distance of 5 feet, where each square covers only one cell. But if the picture is enlarged by a factor of 3 or more then at normal reading distance it can become difficult to distinguish the textures—perhaps because the squares cover regions larger than the templates used at the lowest levels in our visual system.

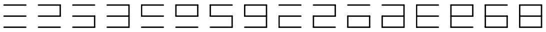
▪ **History.** Ever since antiquity the visual arts have yielded practical schemes and sometimes also fairly abstract frameworks for determining what features of images will have what impact. In fact, even in prehistoric times it seems to have been known, for example, that edges are often sufficient to communicate visual forms, as in the pictures below.



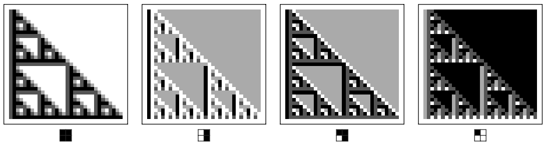
Visual perception has been used for centuries as an example in philosophical discussions about the nature of experience. Traditional mathematical methods began to be applied to it in the second half of the 1800s, particularly through the development of psychophysics. Studies of visual illusions around the end of the 1800s raised many questions that were not readily amenable to numerical measurement or traditional mathematical analysis, and this led in part to the Gestalt approach to psychology which attempted to formulate various global principles of visual perception.

In the 1940s and 1950s, the idea emerged that visual images might be processed using arrays of simple elements. At a largely theoretical level, this led to the perceptron model of the visual system as a network of idealized neurons. And at a practical level it also led to many systems for image processing (see below), based essentially on simple cellular automata (see page 928). Such systems were widely used by the end of the 1960s, especially in aerial reconnaissance and biomedical applications.

Attempts to characterize human abilities to perceive texture appear to have started in earnest with the work of Bela Julesz around 1962. At first it was thought that the visual system might be sensitive only to the overall autocorrelation of an image, given by the probability that randomly selected points have the same color. But within a few years it became clear that images could be constructed—notably with systems equivalent to additive cellular automata (see below)—that had the same autocorrelations but looked completely different. Julesz then suggested that discrimination between textures might be based on the presence of "textons", loosely defined as localized regions like those shown below with some set of distinct geometrical or topological properties.



In the 1970s, two approaches to vision developed. One was largely an outgrowth of work in artificial intelligence, and concentrated mostly on trying to use traditional mathematics to characterize fairly high-level perception of objects and their geometrical properties. The other, emphasized particularly by David Marr, concentrated on lower-level processes, mostly based on simple models of the responses of single nerve cells, and very often effectively applying *ListConvolve* with simple kernels, as in the pictures below.



In the 1980s, approaches based on neural networks capable of learning became popular, and attempts were made in the context of computational neuroscience to create models combining higher- and lower-level aspects of visual perception.

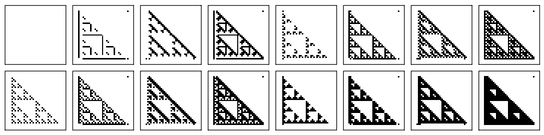
The basic idea that early stages of visual perception involve extraction of local features has been fairly clear since the 1950s, and researchers from a variety of fields have invented and reinvented implementations of this idea many times. But mainly through a desire to use traditional mathematics, these implementations have tended to be implicitly restricted to using elements with various linearity properties—typically leading to rather unconvincing results. My model is closer to what is often done in practical image processing, and apparently to how actual nerve cells work, and in effect assumes highly nonlinear elements.

■ **Page 581 · Implementation.** The exact matches for a template σ in data containing elements 0 and 1 can be obtained from

```
Sign[ListCorrelate[2 σ - 1, data] - Count[σ, 1, 2]] + 1
```

■ **Testing the model.** Although it is difficult to get good systematic data, the many examples I have tried indicate that the levels of discrimination between textures that we achieve with our visual system agree remarkably well with those suggested by my simple model. A practical issue that arises is that if one repeatedly tries experiments with the same set of textures, then after a while one learns to discriminate these particular textures better. Shifting successive rows or even just making an overall rotation seems, however, to avoid this effect.

■ **Related models.** Rather than requiring particular templates to be matched, one can consider applying arbitrary cellular automaton rules. The pictures below show results from a single step of the 16 even-numbered totalistic 5-neighbor rules. The results are surprisingly easy to interpret in terms of feature extraction.



■ **Image processing.** The release of programs like Photoshop in the late 1980s made image processing operations such as smoothing, sharpening and edge detection widely available on general-purpose computers. Most of these operations are just done by applying *ListConvolve* with simple kernels. (Even before computers, such convolutions could be done using the fact that diffraction of light effectively performs Fourier transforms.) Ever since the 1960s all sorts of schemes for nonlinear processing of images have been discussed and used in particular communities. An example originally popular in the earth and environmental sciences is so-called mathematical morphology, based on "dilation" of data consisting of 0's and 1's with a "structuring element" σ according to *Sign[ListConvolve[σ, data, 1, 0]]* (as well as the dual operation of "erosion"). Most schemes like this can ultimately be thought of as picking out templates or applying simple cellular automaton rules.

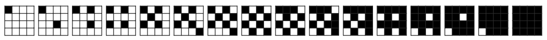
■ **Real textures.** The textures I consider in the main text are all based on arrays of discrete black and white squares. One can also consider textures associated, say, with surface roughness of physical objects. Models of these are often needed for realistic computer graphics. Common approaches are to assume that the surfaces are random with some frequency spectrum, or can be generated as fractals using substitution systems with random parameters. In recent times, models based on wavelets have also been used.

■ **Statistical methods.** Even though they do not appear to correspond to how the human visual system works, statistical methods are often used in trying to discriminate textures automatically. Correlations, conditional entropies and fractal dimensions are commonly computed. Often it is assumed that different parts of a texture are statistically independent, so that the texture can be characterized by probabilities for local patterns, as in a so-called Markov random field or generalized autoregressive moving average (ARMA) process.

■ **Camouflage.** On both animals and military vehicles it is often important to have patterns that cannot be distinguished from a background by the visual systems of predators. And in most cases this is presumably best achieved by avoiding differences in densities of certain local features. Note that in a related situation almost any fairly random overlaid pattern containing many local features can successfully be used to mask the contents of a paper envelope.

■ **Halftoning.** In printed books like this one, gray levels are usually obtained by printing small dots of black with varying sizes. On displays consisting of fixed arrays of pixels, gray levels must be obtained by having only a certain density of pixels be black. One way to achieve this is to break the array into $2^n \times 2^n$ blocks, then successively to fill in pixels in each block until the appropriate gray level is reached, as in the pictures below, in an order given for example by

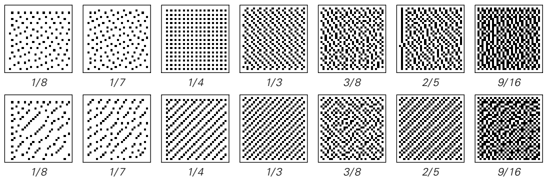
```
Nest[
  Flatten2D[{{4# + 0, 4# + 2}, {4# + 3, 4# + 1}}] &, {{0}}, n]
```



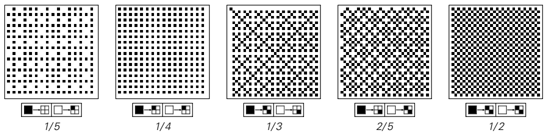
An alternative to this so-called ordered dither approach is the Floyd-Steinberg or error-diffusion method invented in 1976. This scans sequentially, accumulating and spreading total gray level in the data, then generating a black pixel whenever a threshold is exceeded. The method can be implemented using

```
Module[{a = Flatten[data], r, s},
 {r, s} = Dimensions[data]; Partition[Do[
   a[[i + {1, s - 1, s, s + 1}]] += m (a[[i]] - If[a[[i]] < 1/2, 0, 1]),
   {i, r s - s - 1}]; Map[If[# < 1/2, 0, 1] &, a], s]]
```

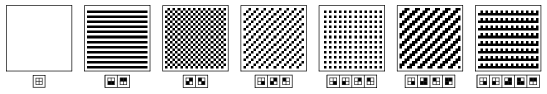
In its original version $m = \{7, 3, 5, 1\}/16$, as in the first row of pictures below. But even with $m = \{1, 0, 1, 0\}/2$ the method generates fairly random patterns, as in the second row below. (Note that significantly different results can be obtained if different boundary conditions are used for each row.)



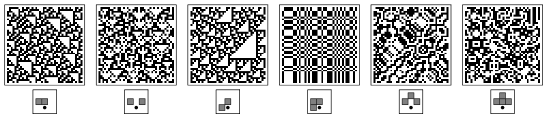
To give the best impression of uniform gray, one must in general minimize features detected by the human visual system. One simple way to do this appears to be to use nested patterns like the ones below.



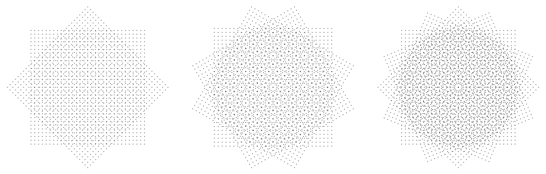
■ **Generating textures.** As discussed on page 217, it is in general difficult to find 2D patterns which at all points match some definite set of templates. With 2×2 templates, there turn out to be just 7 minimal such patterns, shown below. Constructing patterns in which templates occur with definite densities is also difficult, although randomized iterative schemes allow some approximation to be obtained.



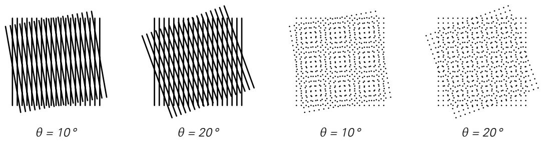
One-dimensional cellular automata are especially convenient generators of distinctive textures. Indeed, as was noticed around 1980, generalizations of additive rules involving cells in different relative locations can produce textures with similar statistics, but different visual appearance, as shown below. (All the examples shown turn out to correspond to ordinary, sequential and reversible cellular automata seen elsewhere in this book.) (See also page 1018.)



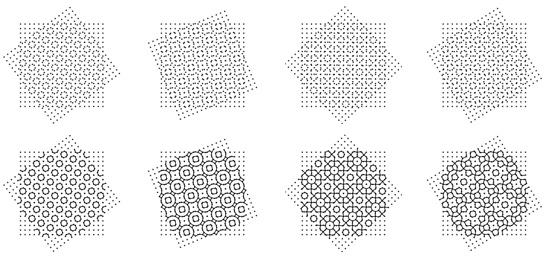
■ **Moire patterns.** The pictures below show moire patterns formed by superimposing grids of points at different angles. Our visual system does not immediately perceive the grids, but instead mainly picks up features formed from local arrangements of dots. The second picture below is similar to patterns of halftone screens visible in 4-color printing under a magnifying glass.



In the first two pictures below, bands with spacing $1/2\ Csc[\theta/2]$ are visible wherever lines cross. In the second two pictures there is also an apparent repetitive pattern with approximately the same repetition period.

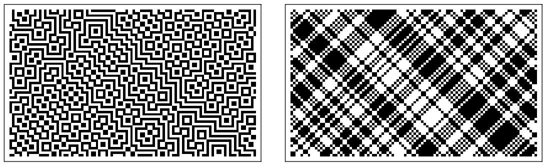


The patterns are exactly repetitive only when $Tan[\theta] == u/v$, where u and v are elements of a primitive Pythagorean triple (so that u, v and $Sqrt[u^2 + v^2]$ are all integers, and $GCD[u, v] == 1$). This occurs when $u = r^2 - s^2$, $v = 2\,r\,s$ (see page 945), and in this case the minimum displacement that leaves the whole pattern unchanged is $\{s, r\}$.



The second row of pictures illustrates what happens if points closer than distance $1/\sqrt{2}$ are joined. The results appear to capture at least some of the features picked out by our visual system.

■ **Perception and presentation.** In writing this book it has been a great challenge to find graphical representations that make the behavior of systems as clear as possible for the purposes of human visual perception. Even small changes in representation can greatly affect what properties are noticed. As a simple example, the pictures below are identical, except for the fact that the colors of cells on alternate rows have been reversed.



Auditory Perception

■ **Sounds.** The human auditory system is sensitive to sound at frequencies between about 20 Hz and 20 kHz. Middle A on a piano typically corresponds to a frequency of 440 Hz. Each octave represents a change in frequency by a factor of two. In western music there are normally 12 notes identified within an octave. These differ in frequency by successive factors of roughly $2^{1/12}$—with different temperament schemes using different rational approximations to powers of this quantity.

The perceived character of a sound seems to depend most on the frequencies it contains, but also to be somewhat affected by the way its intensity ramps up with time, as well as the way frequencies change during this ramp up. Many musical instruments produce sound by vibrating strings or air in cylindrical or conical tubes, and in these cases, there is one main frequency, together with roughly equally spaced overtones. In percussion instruments, the spectrum of frequencies is usually much more complicated. In speech, vowels and voiced consonants tend to be characterized by the lowest two or three frequencies of the mouth. In nature, processes such as fluid turbulence and fracture yield a broad spectrum of frequencies. In speech, letters like "s" also yield broad spectra, presumably because they involve fluid turbulence.

Any sound can be specified by giving its amplitude or waveform as a function of time. *Sin[ω t]* corresponds to a pure tone. Other simple mathematical functions can also yield distinctive sounds. FM synthesis functions such as *Sin[ω (t + a Sin[b t])]* can be made to sound somewhat like various musical instruments, and indeed were widely used in early synthesizers.

■ **Auditory system.** Sound is detected by the motion it causes in hair cells in the cochlea of the inner ear. When vibrations of a particular frequency enter the cochlea an active process involving hair cells causes the vibrations to be concentrated at a certain distance down the cochlea. To a good approximation this distance is proportional to the logarithm of the frequency, and going up one octave in frequency corresponds to moving roughly 3.5 mm. Of the 12,000 or so hair cells in the cochlea most seem to be involved mainly with mechanical issues; about 3500 seem to produce outgoing signals. These are collected by about 30,000 nerve fibers which go down the auditory nerve and after several stops reach the auditory cortex. Different nerve cells seem to have rates of firing which are set up to reflect both sound intensity, and below perhaps 300 Hz, actual amplitude peaks in the sound waveform. Much as in both the visual and tactile systems, there seems to be a fairly direct mapping from position on the cochlea to position in the auditory cortex. In animals such as bats it is known that specific nerve cells respond to particular kinds of frequency changes. But in primates, for example, little is known about exactly what features are extracted in the auditory cortex.

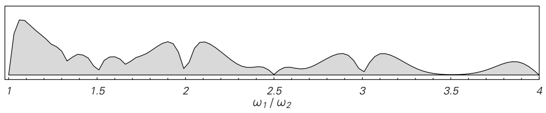
The fact that there are a million nerve fibers going from the eye to the brain, but only about 30,000 going from the ear to the brain means that while it takes several million bits per second to transmit video of acceptable quality, a few tens of thousands of bits are adequate for audio (NTSC television is 5 MHz; audio CDs 22 kHz; telephone 8 kHz). Presumably related is also the fact that it is typically much easier to make realistic sound effects than realistic visual ones.

■ **Chords.** Two pure tones played together exhibit beats at the difference of their frequencies—a consequence of the fact that

$$Sin[\omega_1\, t] + Sin[\omega_2\, t] == 2\, Sin[1/2\, (\omega_1 + \omega_2)\, t]\, Cos[1/2\, (\omega_1 - \omega_2)\, t]$$

With $\omega \simeq 500\,Hz$, one can explicitly hear the time variation of the beats if their frequency is below about 15 Hz, and the result is quite pleasant. But between 15 Hz and about 60 Hz, the sound tends to be rather grating—possibly because this frequency range conflicts with that used for signals in the auditory nerve.

In music it is usually thought that chords consisting of tones with frequencies whose ratios have small denominators (such as 3/2, corresponding to a perfect fifth) yield the most pleasing sounds. The mechanics of the ear imply that if two tones of reasonable amplitude are played together, progressively smaller additional signals will effectively be generated at frequencies $Abs[n_1\, \omega_1 \pm n_2\, \omega_2]$. The picture below shows the extent to which such frequencies tend to be in the range that yield grating effects. The minima at values of ω_2/ω_1 corresponding to rationals with small denominators may explain why such chords seem more pleasing. (See also page 917.)



■ **History.** The notion of musical notes and of concepts such as octaves goes back at least five thousand years. Around 550 BC the Pythagoreans identified various potential connections between numbers and the perception of sounds. And over the course of time a wide range of mathematical and aesthetic principles were suggested. But it was not until the 1800s, particularly with the work of Hermann Helmholtz, that the physical basis for the perception of sound began to be seriously investigated. Work on speech sounds by Alexander Graham Bell and others was related to the development of the telephone in the late 1800s. In the past few decades, with better experiments, particularly on the emission of sound by the ear, and with ideas and analysis from electrical engineers and physicists the basic behavior of at least the cochlea is becoming largely understood.

■ **Sonification.** Sound has occasionally been used as a means of understanding scientific data. In the 1950s and 1960s analog computers (and sometimes digital computers) routinely had sound output. And in the 1970s some discoveries about chaos in differential equations were made using such output. In experimental neuroscience sounds are also routinely used to monitor impulses in nerve cells.

■ **Implementation.** *ListPlay[data]* in *Mathematica* generates sound output by treating the elements of *data* as successive samples in the waveform of the sound, typically with a default sample rate of 8000 Hz.

■ **Time variation.** Many systems discussed in this book produce sounds with distinctive and sometimes pleasing time variation. Particularly dramatic are the concatenation systems discussed on page 913, as well as successive rows in nested patterns such as *Flatten[IntegerDigits[NestList[BitXor[#, 2 #] &, 1, 500], 2]]* and sequences based on numbers such as *Flatten[Table[If[GCD[i, j] == 0, 1, 0], {i, 1000}, {j, i}]]* (see page 613). The recursive sequences on page 130 yield sounds reminiscent of many natural systems.

■ **Musical scores.** Instead of taking a sequence to correspond directly to the waveform of a sound, one can consider it to give a musical score in which each element represents a note of a certain frequency, played for some specific short time. (One can avoid clicks by arranging the waveform to cross zero at both the beginning and end of each note.) With this setup my experience is that both repetitive and random sequences tend to seem quite monotonous and dull. But nested sequences I have found can quite often generate rather pleasing tunes. (One can either determine frequencies of notes directly from the values of elements, or, say, from cumulative sums of such values, or from heights in paths like those on page 892.) (See also page 869.)

■ **Recognizing repetition.** The curve of the function *Sin[x] + Sin[$\sqrt{2}$ x]* shown on page 146 looks complicated to the eye. But a sound with a corresponding waveform is recognized by the ear as consisting simply of two pure tones. However, if one uses the function to generate a score—say playing a note at the position of each peak—then no such simplicity can be recognized. And this fact is presumably why musical scores normally have notes only at integer multiples of some fixed time interval.

■ **Sound compression.** Sound compression has in practice mostly been applied to human speech. In typical voice coders (vocoders) 64k bits per second of digital data are obtained by sampling the original sound waveform 8000 times per second, and assigning one of 256 possible levels to each sample. (Since the 1960s, so-called mu-law companding has often been used, in which these levels are distributed exponentially in amplitude.) Encoding only differences between successive samples leads to perhaps a factor of 2 compression. Much more dramatic compression can be achieved by making an explicit model for speech sounds. Most common is to assume that within each phoneme-length chunk of a few tens of milliseconds the vocal tract acts like a linear filter excited either by pure tones or randomness. In so-called linear predictive coding (LPC) optimal parameters are found to make each sound sample be a linear combination of, say, 8 preceding samples. The residue from this procedure is then often fitted to a code book of possible forms, and the result is that intelligible speech can be obtained with as little as 3 kbps of data. Hardware implementations of LPC and related methods have been widespread since before the 1980s; software implementations are now becoming common. Music has in the past rarely been compressed, except insofar as it can be specified by a score. But recently the MP3 format associated with MPEG and largely based on LPC methods has begun to be used for compression of arbitrary sounds, and is increasingly applied to music.

■ **Page 586 · Spectra.** The spectra shown are given by *Abs[Fourier[data]]*, where the symmetrical second half of this list is dropped in the pictures. Also of relevance are intensity or power spectra, obtained as the square of these spectra. These are related to the autocorrelation function according to

```
Fourier[list]^2 ==
   Fourier[ListConvolve[list, list, {1, 1}]]/Sqrt[Length[list]]
```

(See also page 1074.)

■ **Spectra of substitution systems.** Questions that turn out to be related to spectra of substitution systems have arisen in various areas of pure mathematics since the late 1800s. In the 1980s, particularly following discoveries in iterated maps and quasicrystals, studies of such spectra were made in the context of number theory and dynamical systems theory. Some general principles were proposed, but a great many exceptions were always eventually found.

As suggested by the pictures in the main text, spectra such as (b) and (d) in the limit consist purely of discrete Dirac delta function peaks, while spectra such as (a) and (c) also contain essentially continuous parts. There seems to be no simple criterion for deciding from the rule what type of spectrum will be obtained. (In some cases it works to look at whether the limiting ratio of lengths on successive steps is a Pisot number.) One general result, however, is that all so-called Sturmian sequences *Round[(n + 1) a + b] - Round[n a + b]* with *a* an irrational number must yield discrete spectra. And as discussed on page 903, if *a* is a quadratic irrational, then such sequences can be generated by substitution systems.

For any substitution system the spectrum *ϕ[i][t, ω]* at step *t* from initial condition *i* is given by a linear recurrence relation in terms of the *ϕ[j][t - 1, ω]*. With *k* colors each giving a string of the same length *s* the recurrence relation is

```
Thread[Map[ϕ[#][t + 1, ω] &, Range[k] - 1] ==
  Apply[Plus, MapIndexed[Exp[ⅈ ω (Last[#2] - 1) s^t]
     ϕ[#1][t, ω] &, Range[k] - 1 /. rules, {-1}], {1}]/√s ]
```

Some specific properties of the examples shown include:

(a) *(Thue-Morse sequence)* The spectrum is essentially *Nest[Range[2 Length[#]] Join[#, Reverse[#]] &, {1}, t]*. The main peak is at position 1/3, and in the power spectrum this peak contains half of the total. The generating function for the sequence (with 0 replaced by -1) satisfies $f[z] == (1 - z) f[z^2]$, so that $f[z] == Product[1 - z^{2^n}, \{n, 0, \infty\}]$. (Z transform or generating function methods can be applied directly only for substitution systems with rules such as *{1 → list, 0 → 1 - list}*.) After *t* steps a continuous approximation to the spectrum is $Product[1 - Exp[2^s ⅈ \omega], \{s, t\}]$, which is an example of a type of product studied by Frigyes Riesz in 1918 in connection with questions about the convergence of trigonometric series. It is related to the product of sawtooth functions given by $Product[Abs[Mod[2^s \omega, 2, -1]], \{s, t\}]$. Peaks occur for values of *ω* such as 1/3 that are not well approximated by numbers of the form $a/2^b$ with small *a* and *b*.

(b) *(Fibonacci-related sequence)* This sequence is a Sturmian one. The maximum of the spectrum is at *Fibonacci[t]*. The spectrum is roughly like the markings on a ruler that is recursively divided into *{GoldenRatio, 1}* pieces.

(c) *(Cantor set)* In the limit, no single peak contains a nonzero fraction of the power spectrum. After *t* steps a continuous approximation to the spectrum is $Product[1 + Exp[3^s\, 2 ⅈ \omega], \{s, t\}]$.

(d) *(Period-doubling sequence)* The spectrum is $(2^\# - (-1)^\# \&)[1 + IntegerExponent[n, 2]]$, almost like the markings on a base 2 ruler.

(See also page 917.)

▪ **Flat spectra.** Any impulse sequence *Join[{1}, Table[0, {n}]]* will yield a flat spectrum. With odd *n* the same turns out to be true for sequences $Exp[2 \pi ⅈ Mod[Range[n]^2, n]/n]$—a fact used in the design of acoustic diffusers (see page 1183). For sequences involving only two distinct integers flat spectra are rare; with *±1* those equivalent to *{1, 1, 1, -1}* seem to be the only examples. ($\{r^2, r s, s^2, -r s\}$ works for any *r* and *s*, as do all lists obtained working modulo $x^n - 1$ from *p[x]/p[1/x]* where *p[x]* is any invertible polynomial.) If one ignores the first component of the spectrum the remainder is flat for a constant sequence, or for a random sequence in the limit of infinite length. It is also flat for maximal length LFSR sequences (see page 1084) and for sequences *JacobiSymbol[Range[0, p - 1], p]* with prime *p* (see page 870). By adding a suitable constant to each element one can then arrange in such cases for the whole spectrum to be flat. If *Mod[p, 4] == 1* *JacobiSymbol* sequences also satisfy *Fourier[list] == list*. Sequences of 0's and 1's that have the same property are *{1, 0, 1, 0}*, *{1, 0, 0, 1, 0, 0, 1, 0, 0}* or in general *Flatten[Table[{1, Table[0, {n - 1}]}, {n}]]*. If -1 is allowed, additional sequences such as *{0, 1, 0, -1, 0, -1, 0, 1}* are also possible. (See also pages 911.)

▪ **Nested vibrations.** With an assembly of springs arranged in a nested pattern simple initial excitations can yield motion that shows nested behavior in time. If the standard methodology of mechanics is followed, and the system is analyzed in terms of normal modes, then the spectrum of possible frequencies can look complicated, just as in the examples on page 586. (Similar considerations apply to the motion of quantum mechanical electrons in nested potentials.)

▪ **Page 587 · Random block sequences.** Analytical forms for all but the last spectrum are: 1, $u^2/(1 + 8 u^2)$, $1/(1 + 8 u^2)$, u^2, $(1 - 4 u^2)^2/(1 - 5 u^2 + 8 u^4)$, $u^2/(1 - 5 u^2 + 8 u^4)$, $u^2 + 1/36\, DiracDelta[\omega - 1/3]$, where $u = Cos[\pi \omega]$, and *ω* runs from *0* to *1/2* in each plot. Given a list of blocks such as *{{1, 1}, {0}}* each element of *Flatten[list]* can be thought of as a state in a finite automaton or a Markov process (see page 1084). The transitions between these states have probabilities given by *m[Map[Length, list]]* where

```
m[s_] := With[{q = FoldList[Plus, 0, s]}, ReplacePart[
   RotateRight[IdentityMatrix[Last[q]], {0, 1}], 1/Length[s],
   Flatten[Outer[List, Rest[q], Drop[q, -1] + 1], 1]]]
```

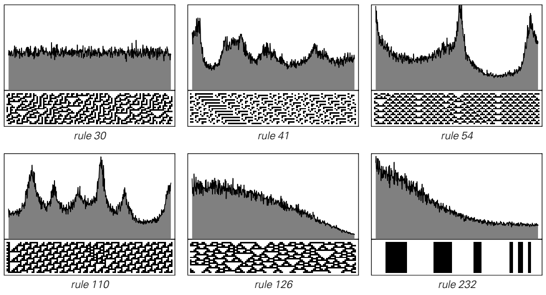
The average spectrum of sequences generated according to these probabilities can be obtained by computing the correlation function for elements a distance *r* apart

```
ξ[list_, r_] := With[{w = (# - Apply[Plus, #]/Length[#] &)[
        Flatten[list]]}, w . MatrixPower[
        m[Map[Length, list]], r] . w/Length[w]]
```

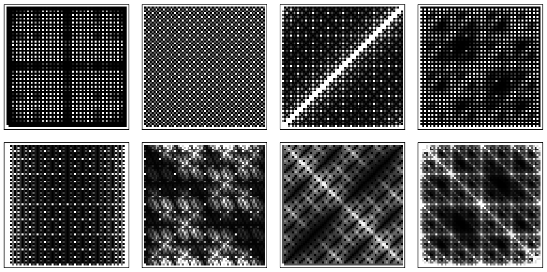
then forming *Sum[ξ[Abs[r]] Cos[2 π r ω], {r, -n/2, n/2}]* and taking the limit $n \to \infty$. If $\xi[r] = \lambda^r$ then the spectrum is $(1-\lambda^2)/(\lambda^2 - 2\lambda\, Cos[2\pi\omega] + 1) - 1$. For a random walk (see page 977) in which ± 1 occur with equal probability the spectrum is $Csc[\pi\omega]^2/2$, or roughly $1/\omega^2$.

The same basic setup also applies to spectra associated with linear filters and ARMA time series processes (see page 1083), in which elements in a sequence are generated from external random noise by forming linear combinations of the noise with definite configurations of elements in the sequence.

▪ **Spectra of cellular automata.** When cellular automata have non-trivial attractors as discussed in Chapter 6 the spectra of sequences obtained at particular steps can exhibit a variety of features, as shown below.



▪ **2D spectra.** The pictures below give the 2D Fourier transforms of the nested patterns shown on page 583.



▪ **Diffraction patterns.** X-ray diffraction patterns give Fourier transforms of the spatial arrangement of atoms in a material. For an ordinary crystal with atoms on a repetitive lattice, the patterns consist of a few isolated peaks. For quasicrystals with generalized Penrose tiling structures the patterns also contain a few large peaks, though as in example (b) on page 586 there are also a hierarchy of smaller peaks present. In general, materials with nested structures do not necessarily yield discrete diffraction patterns. In the early 1990s, experiments were done in which layers a few atoms thick of two different materials were deposited in a Thue-Morse sequence. The resulting object was found to yield X-ray diffraction patterns just like example (a) on page 586.

Statistical Analysis

▪ **History.** Some computations of odds for games of chance were already made in antiquity. Beginning around the 1200s increasingly elaborate results based on the combinatorial enumeration of possibilities were obtained by mystics and mathematicians, with systematically correct methods being developed in the mid-1600s and early 1700s. The idea of making inferences from sampled data arose in the mid-1600s in connection with estimating populations and developing precursors of life insurance. The method of averaging to correct for what were assumed to be random errors of observation began to be used, primarily in astronomy, in the mid-1700s, while least squares fitting and the notion of probability distributions became established around 1800. Probabilistic models based on random variations between individuals began to be used in biology in the mid-1800s, and many of the classical methods now used for statistical analysis were developed in the late 1800s and early 1900s in the context of agricultural research. In physics fundamentally probabilistic models were central to the introduction of statistical mechanics in the late 1800s and quantum mechanics in the early 1900s. Beginning as early as the 1700s, the foundations of statistical analysis have been vigorously debated, with a succession of fairly specific approaches being claimed as the only ones capable of drawing unbiased conclusions from data. The practical use of statistical analysis began to increase rapidly in the 1960s and 1970s, particularly among biological and social scientists, as computers became more widespread. All too often, however, inadequate amounts of data have ended up being subjected to elaborate statistical analyses whose results are then blindly assumed to represent definitive scientific conclusions. In the 1980s, at least in some fields, traditional statistical analysis began to become less popular, being replaced by more direct examination of data presented graphically by computer. In addition, in the 1990s, particularly in the context of consumer electronics devices, there has been an increasing emphasis on using statistical analysis to make decisions from data, and methods such as fuzzy logic and neural networks have become popular.

■ **Practical statistics.** The vast majority of statistical analysis is in practice done on continuous numerical data. And with surprising regularity it is assumed that random variations in such data follow a Gaussian distribution (see page 976). But while this may sometimes be true—perhaps as a consequence of the Central Limit Theorem—it is rarely checked, making it likely that many detailed inferences are wrong. So-called robust statistics uses for example medians rather than means as an attempt to downplay outlying data that does not follow a Gaussian distribution.

Classical statistical analysis mostly involves trying to use data to estimate parameters in specific probabilistic models. Non-parametric statistics and related methods often claim to derive conclusions without assuming particular models for data. But insofar as a conclusion relies on extrapolation beyond actual measured data it must inevitably in some way use a model for data that has not been measured.

■ **Time series.** Sequences of continuous numerical data are often known as time series, and starting in the 1960s standard models for them have consisted of linear recurrence relations or linear differential equations with random noise continually being added. The linearity of such models has allowed efficient methods for estimating their parameters to be developed, and these are widely used, under slightly different names, in control engineering and in business analysis. In recent years nonlinear models have also sometimes been considered, but typically their parameters are very difficult to estimate reliably. As discussed on page 919 it was already realized in the 1970s that even without external random noise nonlinear models could produce time series with seemingly random features. But confusion about the importance of sensitivity to initial conditions caused the kind of discoveries made in this book to be missed.

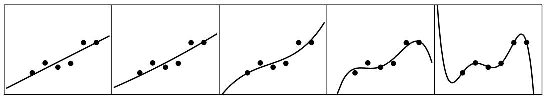
■ **Page 588 · Origin of probabilities.** Probabilities are normally assumed to enter for at least two reasons: (a) because of random variation between individuals, and (b) because of random errors in measurement. (a) is particularly common in the biological and social sciences; (b) in the physical sciences. In physics effects of statistical mechanics and quantum mechanics are also assumed to introduce probabilities. Probabilistic models for abstract mathematical systems have in the past been rare, though the results about randomness in this book may make them more common in the future.

■ **Probabilistic models.** A probabilistic model must associate with every sequence a probability that is a number between 0 and 1. This can be done either by giving an explicit procedure for taking sequences and finding probabilities, or by defining a process in which sequences are generated with appropriate probabilities. A typical example of the first approach is the Ising model for spin systems in which relative probabilities of sequences are found by multiplying together the results of applying a simple function to blocks of nearby elements in the sequence. Monte Carlo methods and probabilistic cellular automata provide examples of the second approach.

■ **Page 588 · Binomial distribution.** If black squares appear independently with probability p then the probability that m squares out of n are black is $\mathit{Binomial}[n, m]\, p^m\, (1-p)^{n-m}$.

■ **Page 589 · Estimation of parameters.** One way to estimate parameters in simple probabilistic models is to compute the mean and other moments of the data and then to work out what values of the parameters will reproduce these. More general is the maximum likelihood method in which one finds the values of the parameters which maximize the probability of generating the observed data from the model. (Least squares fits do this for models in which the data exhibits independent Gaussian variations.) Various modifications can be made involving for example weighting with a risk function before maximizing. If one starts with a priori probability distributions for all parameters, then Bayes's Theorem on conditional probabilities allows one to avoid the arbitrariness of methods such as maximum likelihood and explicitly to work out from the observed data what the probability is for each possible choice of parameters in the model. It is rare in practice, however, to be able to get convincing *a priori* probability distributions, although when there are physical or other reasons to expect entropy to be maximized the so-called maximum entropy method may be useful.

■ **Complexity of models.** The pictures at the top of the next page show least squares fits (found using *Fit* in *Mathematica*) to polynomials with progressively higher degrees and therefore progressively more parameters. Which fit should be considered best in any particular case must ultimately depend on external considerations. But since the 1980s there have been attempts to find general criteria, typically based on maximizing quantities such as $-\mathit{Log}[p] - d$ (the Akaike information criterion), where p is the probability that the observed data would be generated from a given model ($-\mathit{Log}[p]$ is proportional to variance in a least squares fit), and d is the number of parameters in the model.



■ **Page 590 · Markov processes.** The networks in the main text can be viewed as representing finite automata (see page 957) with probabilities associated with transitions between nodes or states. Given a vector of probabilities to be in each state, the evolution of the system corresponds to multiplication by the matrix of probabilities for each transition. (Compare the calculation of properties of substitution systems on page 890.) Markov processes first arose in the early 1900s and have been widely studied since the 1950s. In their first uses as models it was typically assumed that each state transition could explicitly be observed. But by the 1980s hidden Markov models were being studied, in which only some of the states or transitions could be distinguished by outside observations. Practical applications were made in speech understanding and text compression. And in the late 1980s, building on work of mine from 1984 (described on page 276), James Crutchfield made a study of such models in which he defined the complexity of a model to be equal to $-p\,Log[p]$ summed over all connections in the network. He argued that the best scientific model is one that minimizes this complexity—which with probabilities 0 and 1 is equivalent to minimizing the number of nodes in the network.

■ **Non-local processes.** It follows from the fact that any path in a finite network must always eventually return to a node where it has been before that any Markov process must be fundamentally local, in the sense that the probabilities it implies for what happens at a given point in a sequence must be independent of those for points sufficiently far away. But probabilistic models based on other underlying systems can yield sequences with long-range correlations. As an example, probabilistic neighbor-independent substitution systems can yield sequences with hierarchical structures that have approximate nesting. And since the mid-1990s such systems (usually characterized as random trees or random context-free languages) have sometimes been used in analyzing data that is expected to have grammatical structure of some kind.

■ **Page 594 · Block frequencies.** In any repetitive sequence the number of distinct blocks of length m must become constant with m for sufficiently large m. In a nested sequence the number must always continue increasing roughly linearly, and must be greater than m for every m. (The differences of successive numbers themselves form a nested sequence.) If exactly $m+1$ distinct blocks occur for every m, then the sequence must be of the so-called Sturmian type discussed on page 916, and the n^{th} element must be given by $Round[(n+1)a+b]-Round[n\,a+b]$, where a is an irrational number. Up to limited m nested sequences can contain all k^m possible blocks, and can do so with asymptotically equal frequencies. Pictures (b), (c) and (d) show the simplest cases where this occurs (for length 3 $\{1\to\{1, 1, 1, 0, 0, 0\}, 0\to\{1, 0\}\}$ also works). Linear feedback shift registers of the type used in picture (e) are discussed below. Concatenation sequences of the type used in picture (f) are discussed on page 913. In both cases equal frequencies of blocks are obtained only for sequences of length exactly 2^j.

■ **LFSR sequences.** Often referred to as pseudonoise or PN sequences, maximal length linear feedback shift register sequences have repetition period 2^n-1 and are generated by shift registers that go through all their possible states except the one consisting of all 0's, as discussed on page 974. Blocks in such sequences obtained from $Partition[list, n, 1]$ must all be distinct since they correspond to successive complete states of the shift register. This means that every block with length up to n (except all 0's) must occur with equal frequency. (Note that only a small fraction of all possible sequences with this property can be generated by LFSRs.) The regularity of PN sequences is revealed by looking at the autocorrelation $RotateLeft[(-1)^{list}, m] . (-1)^{list}$. This quantity is -1 for all nonzero m for PN sequences (so that all but the first component in $Abs[Fourier[(-1)^{list}]]^2$ are equal), but has mean 0 for truly random sequences. (Related sequences can be generated from $RealDigits[1/p, 2]$ as discussed on page 912.)

■ **Entropy estimates.** Fitting the number of distinct blocks of length b to the form $k^{h\,b}$ for large b the quantity h gives the so-called topological entropy of the system. The so-called measure entropy is given as discussed on page 959 by the limit of $-Sum[p_i\,Log[k, p_i], \{i, k^b\}]/b$ where the p_i are the probabilities for the blocks. Actually getting accurate estimates of such entropies is however often rather difficult, and typically upper bounds are ultimately all that can realistically be given. Note also that as discussed in the main text having maximal entropy does not by any means imply perfect randomness.

■ **Tests of randomness.** Statistical analysis has in practice been much more concerned with finding regularities in data than in testing for randomness. But over the course of the past century a variety of tests of randomness have been proposed, especially in the context of games of chance and their government regulation. Most often the tests are applied not directly to sequences of 0's and 1's, but instead say to numbers obtained from blocks of 8 elements. A typical collection of tests described by Donald Knuth in 1968 includes: (1) frequency or equidistribution test (possible elements should occur with equal frequency); (2) serial test (pairs of elements should be equally likely to be in descending and ascending order); (3) gap test (runs of elements all greater or less than some fixed value should have lengths that follow a binomial distribution); (4) poker test (blocks corresponding to possible poker hands should occur with appropriate frequencies); (5) coupon collector's test (runs before complete sets of values are found should have lengths that follow a definite distribution); (6) permutation test (in blocks of elements possible orderings of values should occur equally often); (7) runs up test (runs of monotonically increasing elements should have lengths that follow a definite distribution); (8) maximum-of-t test (maximum values in blocks of elements should follow a power-law distribution). With appropriate values of parameters, these tests in practice tend to be at least somewhat independent, although in principle, if sufficient data were available, they could all be subsumed into basic block frequency and run-length tests. Of the sequences on page 594, (a) through (d) as well as (f) fail every single one of the tests, (e) fails only the serial test, while (g) and (h) pass all the tests. (Failure is defined as a value that is as large or small as that obtained from the data occurring below a specified probability in the set of all possible sequences.) Widespread use of tests like these on pseudorandom generators (see page 974) began in the late 1970s, with discoveries of defects in common generators being announced every few years.

In the 1980s simulations in physics had begun to use pseudorandom generators to produce sequences with billions of elements, and by the late 1980s evidence had developed that a few common generators gave incorrect results in such cases as phase transition properties of the 3D Ising model and shapes of diffusion-limited aggregates. (These difficulties provided yet more support for my contention that models with intrinsic randomness are more reliable than those with external randomness.) In the 1990s various idealizations of physics simulations—based on random walks, correlation functions, localization of eigenstates, and so on—were used as tests of pseudorandom generators. These tests mostly seem simpler than those shown on page 597 obtained by running a cellular automaton rule on the data.

Over the years, essentially every proposed statistical test of randomness has been applied to the center column of rule 30. And occasionally people have told me that their tests have found deviations from randomness. But in every single case further investigation showed that the results were somehow incorrect. So as of now, the center column of rule 30 appears to pass every single proposed statistical test of randomness.

▪ **Difference tables.** See page 1091.

▪ **Randomized algorithms.** Whether a randomized algorithm gives correct answers can be viewed as a test of randomness for whatever supposedly random sequence is provided to it. But in most practical cases such tests are not particularly stringent; linear congruential generators, for example, almost always pass. (There are perhaps exceptions in VLSI testing.) And this is basically why it has so often proved possible to replace randomized algorithms by deterministic ones that are at least as efficient (see page 1192). An example is Monte Carlo integration, where what ultimately matters is uniform sampling of the integrand—which can usually be achieved better by quasi-random irrational number multiple (see page 903) or digit reversal (see page 905) sequences than by sequences one might consider more random.

Cryptography and Cryptanalysis

▪ **History.** Cryptography has been in use since antiquity, and has been a decisive factor in a remarkably large number of military and other campaigns. Typical of early systems was the substitution cipher of Julius Caesar, in which every letter was cyclically shifted in the alphabet by three positions, with A being replaced by D, B by E, and so on. Systems based on more arbitrary substitutions were in use by the 1300s. And while methods for their cryptanalysis were developed in the 1400s, such systems continued to see occasional serious use until the early 1900s. Ciphers of the type shown on page 599 were introduced in the 1500s, notably by Blaise de Vigenère; systematic methods for their cryptanalysis were developed in the mid-1800s and early 1900s. By the mid-1800s, however, codes based on books of translations for whole phrases were much more common than ciphers, probably because more sophisticated algorithms for ciphers were difficult to implement by hand. But in the 1920s electromechanical technology led to the development of rotor machines, in which an encrypting sequence with an extremely long period was generated by rotating a sequence of noncommensurate rotors. A notable achievement of cryptanalysis was the 1940 breaking of the German Enigma rotor machine using a mixture of statistical analysis and automatic enumeration of keys. Starting in the 1950s, electronic devices were the primary ones used for cryptography. Linear feedback shift registers and perhaps nonlinear ones seem to have been common, though little is publicly known about military cryptographic systems after World War II. In 1977 the U.S. government introduced the DES data encryption standard, and in the 1980s this became the dominant force in the growing field of commercial cryptography. DES takes 64-bit blocks of data and a 56-bit key, and applies 16 rounds of substitutions and permutations. The S-box that implements each substitution works much like a single step of a cellular automaton. No fast method of cryptanalysis for DES is publicly known, although by now for a single DES system an exhaustive search of keys has become feasible. Two major changes occurred in cryptography in the 1980s. First, cryptographic systems routinely began to be implemented in software rather than in special-purpose hardware, and thus became much more widely available. And second, following the introduction of public-key cryptography in 1975, the idea emerged of basing cryptography not on systems with complicated and seemingly arbitrary construction, but instead on systems derived from well-known mathematical problems. Initially several different problems were considered, but after a while the only ones to survive were those such as the RSA system discussed below based essentially on the problem of factoring integers. Present-day publicly available cryptographic systems are almost all based on variants of either DES (such as the IDEA system of PGP), linear feedback shift registers or RSA. My cellular automaton cryptographic system is one of the very few fundamentally different systems to have been introduced in recent years.

■ **Basic theory.** As was recognized in the 1920s the only way to make a completely secure cryptographic system is to use a so-called one-time pad and to have a key that is as long as the message, and is chosen completely at random separately for each message. As soon as there are a limited number of possible keys then in principle one can always try each of them in turn, looking in each case to see whether they imply an original message that is meaningful in the language in which the message is written. And as Claude Shannon argued in the 1940s, the length of message needed to be reasonably certain that only one key will satisfy this criterion is equal to the length of the key divided by the redundancy of the language in which the message is written—equal to about 0.5 for English (see below).

In a cryptographic system with keys of length n there will typically be a total of k^n possible keys. If one guesses a key it will normally take a time polynomial in n to check whether the key is correct, and thus the problem of cryptanalysis is in the class known in theoretical computer science as NP or non-deterministic polynomial time (see page 1142). It is suspected but not established that there exist at least some problems in NP that cannot be solved in polynomial time, potentially indicating that for an appropriate system it might be impossible to do cryptanalysis in any time polynomial in n. (See page 1089.)

■ **Text.** As the picture below illustrates, English text typically remains intelligible until about half its characters have been deleted, indicating that it has a redundancy of around 0.5. Most other languages have slightly higher redundancies, making documents in those languages slightly longer than their counterparts in English.

```
About half the letters in typical English text are redundant.
About half the letter- in typical Eng--sh text are redun-ant.
Abou- half the -etter- in ty-ical Eng--sh text are redun--nt.
Abou- half the -e-t--- i- ty-ical Eng--sh text are redun--nt.
Abou- half t-e -e-t--- -- ty-ical Eng--sh text ar- red-n--nt.
Abou- h-l- t-e -e----- -- ty-ical Eng--sh text ar- r-d-n--nt.
Abou- h-l- t-- -e----- -- ty-ical -ng--sh tex- ar- --d-n--nt.
Abou- h--- --- -e----- -- ty-ica- -ng---h tex- ar- --d-n--nt.
Abou- h--- --- -e----- -- ty---a- -ng---h te-- -r- --d-n--nt.
A-ou- h--- --- -e----- -- ty---a- -ng---- te-- -r- --d----n-.
--ou- ---- --- -e----- -- ty---a- -n----- te-- -r- -------n-.
--o-- ---- --- -e----- -- ty---a- ------- -e-- -r- ---------.
----- ---- --- -e----- -- t------ ------- -e-- --- ---------.
```

Redundancy can in principle be estimated by breaking text into blocks of length b, then looking for the limit of the entropy as $b \to \infty$ (see page 1084). Statistically uniform samples of text do not in practice, however, tend to be large enough to allow more than about $b = 6$ to be reached, and the presence of correlations (even though exponentially damped) between far-separated letters means that computed entropies usually decrease continually with b, making it difficult to estimate their limit (see page 1084). Note that particularly in computer languages higher redundancy is found if one takes account of grammatical structure.

■ **Page 599 · Cryptanalysis.** The so-called Vigenère cipher was thought for several centuries to be unbreakable. The idea of looking for repeats was introduced by Friedrich Kasiski in 1863. A statistical approach based on the fact that frequencies tend to be closer to uniform for longer keys was introduced by William Friedman in the 1920s. The methods described in the main text are fairly characteristic of the mixture between generality and detail that is typical in practical cryptanalysis.

■ **Page 600 · Linear feedback shift registers.** See notes on pages 974 and 1084. LFSR sequences are widely used in radio technology, particularly in the context of spread spectrum applications. Their purpose is usually to provide a way to distinguish or synchronize signals, and sometimes to provide a level of cryptographic security. In CDMA technology for cellular telephones, for example, data is overlaid on LFSR sequences, and sequences other than the one intended for a particular receiver seem like noise which can be ignored. As another example, the Global Positioning System (GPS) works by having 24 satellites each transmit maximal length sequences from different length 10 LFSRs. Position is deduced from the arrival times of signals, as determined by the relative phases of the LFSR sequences received. (GPS P-code apparently uses much longer LFSR sequences and repeats only every 267 days. Before May 2000 it was used to add unpredictable timing errors to ordinary GPS signals.)

▪ **LFSR cryptanalysis.** Given a sequence obtained from a length *n* LFSR (see page 975)

```
Nest[Mod[Append[#, Take[#, -n] . vec], 2] &, list, t]
```

the vector of taps *vec* can be deduced from

```
LinearSolve[Table[Take[seq, {i, i + n - 1}], {i, n}],
  Take[seq, {n + 1, 2 n}], Modulus → 2]
```

(An iterative algorithm in *n* taking about n^2 rather than n^3 steps was given by Elwyn Berlekamp and James Massey in 1968.) The same basic approach can be used to deduce the rule for an additive cellular automaton from vertical sequences.

▪ **Page 603 · Rule 30 cryptography.** Rule 30 is known to have many of the properties desirable for practical cryptography. It does not repeat with any short period or show any obvious structure for almost all keys. Small changes in keys typically leads to large changes in the encrypting sequence. The Boolean expressions which determine the encrypting sequence from the key rapidly become highly complex (see page 618). And furthermore the system can be implemented very efficiently, particularly in parallel hardware.

I originally studied rule 30 in the context of basic science, but I soon realized that it could serve as the basis for practical random sequence generation and cryptography, and I analyzed this extensively in 1985. (Most but not all of the results from my original paper are included in this book, together with various new results.) In 1985 and soon thereafter a number of people (notably Richard and Carl Feynman) tried to cryptanalyze rule 30, but without success. From the beginning, computations of spacetime entropies for rule 30 (see page 960) gave indications that for strong cryptography one should not sample all cells in a column, and in 1991 Willi Meier and Othmar Staffelbach described essentially the explicit cryptanalysis approach shown on page 601. Rule 30 has been widely used for random sequence generation, but for a variety of reasons I have not in the past much emphasized its applications in cryptography.

▪ **Properties of rule 30.** Rule 30 can be written in the form $p \veebar (q \vee r)$ (see page 869) and thus exhibits a kind of one-sided additivity on the left. This leads to some features that are desirable for cryptography (such as long repetition periods) and to some that are not (such as the sideways evolution of page 601). It implies that every block of length m that occurs at a particular step has exactly 4 immediate predecessor blocks of length $m + 2$ (see page 960). It also implies that all 2^t possible single columns of t cells can be generated from some initial condition. Not all 4^t pairs of adjacent columns can occur, however. There seems to be no simple characterization, say in terms of paths through networks, of which can, but for successive t the total numbers are

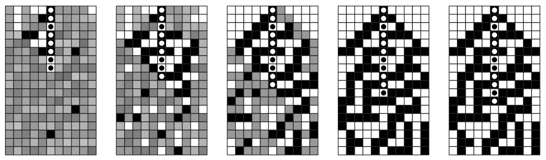
```
{4, 12, 32, 80, 200, 496, 1208, 2916, 6964, 16476, 38616,
  89844, 207544, 476596, 1089000, 2477236, 5615036}
```

or roughly 2.25^t.

Given two complete adjacent columns page 601 shows how all columns any distance to the left can be found. It turns out that this can be done even if the right-hand one of the two adjacent columns is not complete. So for example whenever there is a black cell in the left column it is irrelevant what appears in the right column. Note that the configuration of relevant cells can be repetitive only if the initial conditions were repetitive (see page 871).

In a cellular automaton of limited size n, any column must eventually repeat. There could be 2^n distinct possible columns; in practice, for successive n there are {2, 3, 7, 14, 30, 60, 101, 245, 497, 972, 1997, 3997}—within 2% of 2^n already for $n = 12$. This means that for the initial conditions to be determined uniquely, the number of cells that must be given in a column is almost exactly n, as illustrated in the pictures below. Many distinct columns correspond to starting at different points on a single cycle of states. The length of the longest cycle grows roughly like $2^{0.63 n}$ (see page 260). The complete cycle structure is illustrated on page 962. Most of the 2^n possible states have unique predecessors; for large n, about $2^{0.76 n}$ or $Root[\#^3 - \#^2 - 2 \&, 1]^n$ instead have 0 or 2 predecessors. The predecessors of a given state can be found from

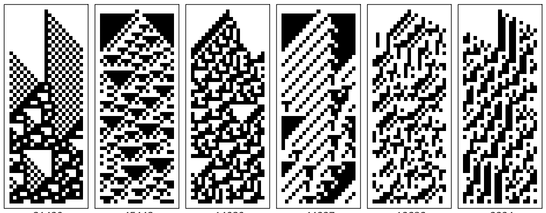
```
Cases[Map[Fold[Prepend[#1, If[#2 == 1 ⊻
    Take[#1, 2] == {0, 0}, 0, 1]] &, #, Reverse[list]] &, {{0,
  0}, {0, 1}, {1, 0}, {1, 1}}], {a_, b_, c___, a_, b_} → {b, c, a}]
```



▪ **Directional sampling.** One can consider sampling cells not in a vertical column but on lines at any angle. In a rule 30 system of infinite size, it turns out that at 45 ° clockwise from vertical all possible sequences can occur on any two adjacent lines, probably making cryptanalysis more difficult in this case. (Note that directional sampling is always equivalent to looking at a vertical column in the evolution of a cellular automaton whose basic rule has been composed with an appropriate shift rule.)

▪ **Alternative rules.** Among elementary rules, rule 45 is the only plausible alternative to rule 30. It usually yields longer repetition periods (see page 260), but shows slightly slower responses to changes in the key. (Changes expand about 1.24 cells per step in rule 30, and about 1.17 in rule 45.) Rule 45 shares with rule 30 the property of one-sided additivity. With the occasional exception of the additive rule 60, elementary rules not equivalent to 30 or 45 tend to exhibit vastly shorter repetition periods. (The completely non-additive rule with largest typical repetition period is rule 110.) (See page 951.)

If one considers rules that depend on 4 rather than 3 cells, then the results turn out to be surprisingly similar: out of all 65536 possible such rules the ones with longest periods essentially always seem to be variants of rules 45, 30 or 60. In a region of size 15, for example, the longest period is 20460, and this is achieved by rule 13251, which is just rule 45 applied to the first three cells in the neighborhood. (Rule 45 itself has period 6820 in this case.) After a few rules with long periods, the periods obtained drop off rapidly. (In general the number of rules with a given period seems to decrease roughly exponentially with period.) For size 15, the 33 rules with the longest periods are all additive with respect to one position. The pictures below show the first rules that are not additive with respect to any position.


31420 (1635) 45443 (1620) 14030 (1560) 44227 (1545) 12686 (1380) 2924 (1320)

Among the 4,294,967,296 $r = 2$ rules which depend on 5 cells, there are again just a few that give long periods, but now only a small fraction of these seem directly related to rules 45 and 30, and perhaps half are not additive with respect to any position. The pictures below show the rules with longest periods for size 15; these same rules also yield the longest periods for many other sizes. The first two are additive with respect to one position, but do not appear to be directly related to rules 45 or 30; the last two are not additive with respect to any position. Formulas for the rules are respectively:

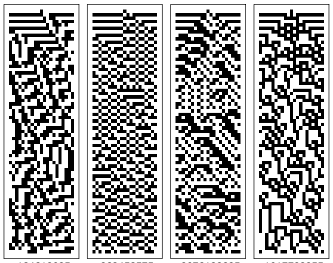
$p \veebar (\neg q \lor r \lor s \land \neg t)$

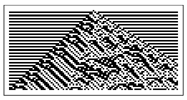
$r \veebar (\neg p \lor q \lor s \land \neg t)$

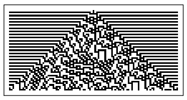
$u = \neg p \land \neg q \lor q \land t; \neg r \land u \lor q \land \neg s \land (p \lor \neg r) \lor r \land s \land \neg u$

$s \land (q \land \neg r \lor p \land \neg q \land t) \lor \neg (s \lor (p \lor q) \land (r \veebar (q \lor t)))$

Note that for size 15 the maximum possible period is 32730 (see page 950).


184612095 (31455) 263458575 (29865) 2076199695 (25395) 1017723955 (23370)


2076199695


1017723955

■ **Nonlinear feedback shift registers.** Linear feedback shift registers of the kind discussed on page 974 can be generalized to allow any function *f* (note the slight analogy with cyclic tag systems):

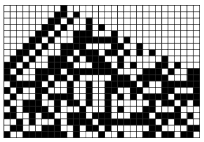
```
NLFSRStep[f_, taps_, list_] :=
  Append[Rest[list], f[list[[taps]]]]
```

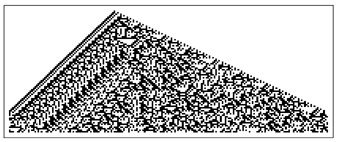
With the choice *f = IntegerDigits[s, 2, 8][[8 - # . {4, 2, 1}]] &* and *taps = {1, 2, 3}* this is essentially a rule *s* elementary cellular automaton. With a list of length *n*, *Nest[NLFSRStep[f, taps, #] &, list, n]* gives one step in the evolution of the cellular automaton in a register of width *n*, with a certain kind of spiral boundary condition. The case analogous to rule 30 yields some of the longest repetition periods—usually remarkably close to the absolute maximum of $2^n - 1$ (for $n = 21$ the result is 1999864, 95% of the maximum).

Nonlinear feedback shift registers were apparently studied in the context of military cryptography in the 1950s, but very little about them has made its way into the open literature (see page 878). An empirical investigation of repetition periods in such systems was made by Solomon Golomb in 1959. The main conclusion drawn from extensive data was that nothing like the linear theory applies. One set of computations concerned functions

```
f[{w_, x_, y_, z_}] := Mod[w + y + z + x y + x z + y z, 2]
```

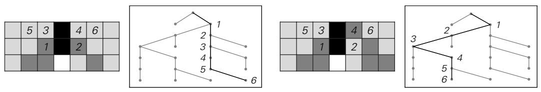
(apparently chosen to have balance between 0's and 1's that would minimize correlations). Tap positions *{1, 2, 3, 4}* were among those studied, but nothing like the pictures below were apparently ever explicitly generated—and nearly three decades passed before I noticed the remarkable behavior of the rule 30 cellular automaton.



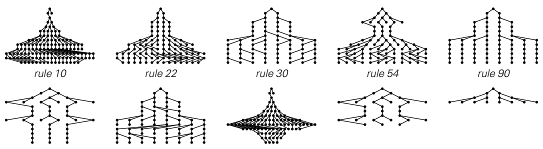


Sequences of states in any shift register must correspond to paths through a network of the kind shown on page 941. And as noted by Nicolaas de Bruijn in 1946 there are $2^{2^{n-1}-n}$ such paths with length 2^n, and thus this number of functions f out of the 2^{2^n} possible must yield sequences of maximal length. (For k colors, the number of paths is $k!^{k^{n-1}}/k^n$.)

■ **Backtracking.** If one wants to find out which of the 2^n possible initial conditions of width n evolve to yield a specific column of colors in a system like an elementary cellular automaton one can usually do somewhat better than just testing all possibilities. The picture below illustrates a typical approach, applied to 3 steps of rule 30. The idea is successively to look at each numbered cell, and to make a tree of possibilities representing what happens if one tries to fill in each possible color for each cell. A branch in the resulting tree continues only if it corresponds to a configuration of cell colors whose evolution is consistent with the specified column of colors.

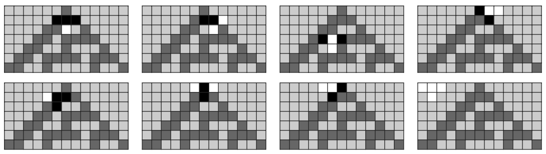


The picture below shows trees obtained for the column ■□■ in various elementary cellular automata. In cases like rules 250 and 254 no initial condition gives the specified column, so all branches eventually die out. In class 2 examples like rule 10 many intermediate configurations are possible. Rules like 90 and to some extent 30 that allow sideways evolution yield comparatively simple trees.



If one wants to find just a single initial condition that works then one can set up a recursive algorithm that in effect does a depth-first traversal of the tree. No doubt in many cases the number of nodes that have to be visited eventually increases like 2^t, but many branches usually die off quickly, greatly reducing the typical effort required in practice.

■ **Deducing cellular automaton rules.** Given a complete cellular automaton pattern it is easy to deduce the rule which produced it just by identifying examples of places where each element in the rule was used, as in the picture at the top of the next column. Given an incomplete pattern, deducing the rule in effect requires solving Boolean equations.



■ **Linear congruential generators.** Cryptanalysis of linear congruential generators is fairly straightforward. Given only an output list *NestList[Mod[a #, m] &, x, n]* parameters *{a, m}* that generate the list can be found for sufficiently large *n* from

```
With[{α = Apply[#2 . Rest[list]/#1 &, Apply[
      ExtendedGCD, Drop[list, -1]]]}, ({Mod[α, #], #} &)[
   Fold[GCD[#1, If[#1 == 0, #2, Mod[#2, #1]]] &, 0,
     ListCorrelate[{α, -1}, list]]]]
```

With slightly more effort both x and *{a, m}* can be found just from *First[IntegerDigits[list, 2, p]]*.

■ **Digit sequence encryption.** One can consider using as encrypting sequences the digit sequences of numbers obtained from standard mathematical functions. As discussed on page 139 such digit sequences often seem locally very random. But in many cases one can immediately tell how a sequence was made just by globally applying appropriate mathematical functions. Thus, for example, given the digit sequence of $\sqrt{s}$ one can retrieve the key s just by squaring the number obtained from early digits in the sequence. Whenever a number x is known to satisfy *Sum[a[i] f[i][x], {i, n}] == 0* with fixed *f[i]* one can take the early digits of x and use *LatticeReduce* to find integer solutions for the *a[i]*. With *f[i_] =* $\#^i$ *&* this method allows algebraic numbers to be recognized. If no linear equation is satisfied by any combination of known functions of x, however, the method fails, and it seems quite likely that in such cases secure encrypting sequences can be generated, albeit less efficiently than with systems like cellular automata.

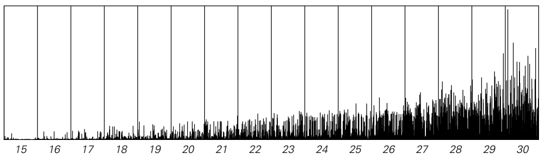
■ **Problem-based cryptography.** Particularly following the work of Whitfield Diffie and Martin Hellman in 1976 it became popular to consider cryptography systems based on mathematical problems that are easy to state but have been found difficult to solve. It was at first hoped that the problems could be NP-complete ones, which are universal in the sense that their solution can be used to provide a solution to any problem in the class NP (see page 1086). To date, however, no system has been devised whose cryptanalysis is known to be NP-complete. Indeed, essentially the only problem on which cryptography systems have so far successfully been based is factoring of integers (see below). And while this problem has resisted a fair number of attempts at solution, it is not known to be NP-complete (and indeed its ability to be solved in polynomial time on a formal quantum computer may suggest that it is not).

My cellular automaton cryptography system follows the principle of being based on a problem that is easy to state. And indeed the general problem of finding initial conditions for a cellular automaton is NP-complete (see page 767). But the problem is not known to be NP-complete for the specific case of, say, rule 30. Significantly less work has been done on the problem of finding initial conditions for rule 30 than on the problem of factoring integers. But the greater simplicity of rule 30 might make one already have almost as much confidence in the difficulty of solving this problem as of factoring integers.

■ **Factoring integers.** The difficulty of factoring is presumably related to the irregularity of the pattern of divisors shown on page 909. One approach to factoring a number *n* is just to try dividing it by each of the numbers up to $\sqrt{n}$. A sequence of much faster methods have however been developed over the past few decades, one simple example that works for most *n* being the so-called rho method of John Pollard (compare the quadratic residue sequences discussed below):

```
Module[{f = Mod[#^2 + 1, n] &, a = 2, b = 5, c},
  While[(c = GCD[n, a - b]) == 1, {a, b} = {f[a], f[f[b]]}]; c]
```

Most existing methods depend on facts in number theory that are fairly easy to state, though implementing them for maximum efficiency tends to lead to complex programs. Typical running times for *FactorInteger[n]* in *Mathematica* 4 are shown below for the first 1000 numbers with each of 15 through 30 digits. Different current methods asymptotically require slightly different numbers of steps—but all typically at least *Exp[Sqrt[Log[n]]]*. Nevertheless, to test whether a number is prime (*PrimeQ*) it is known that only a few more than *Log[n]* steps suffice.

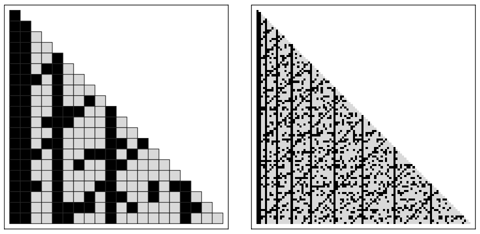


■ **RSA cryptography.** Widely used in practice, the idea is to encode messages using a public key specified by a number *n*, but to make it so that to decode the messages requires a private key based on the factors of *n*. An element *m* in a message is encoded as *c = PowerMod[m, d, n]*. It can then be decoded as *PowerMod[c, e, n]*, where *e = PowerMod[d, -1, EulerPhi[n]]*. But to find *EulerPhi[n]* (see page 1093) is equivalent in difficulty to finding the factors of *n*.

■ **Quadratic residue sequences.** As an outgrowth of ideas related to RSA cryptography it was shown in 1982 by Lenore Blum, Manuel Blum and Michael Shub that the sequence

```
Mod[NestList[Mod[#^2, m] &, x0, n], 2]
```

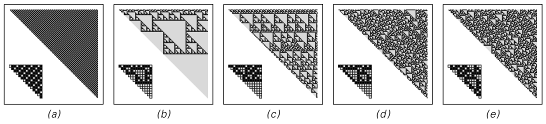
discussed on page 975 has the property that if $m = p q$ with *p* and *q* primes (congruent to 3 modulo 4) then any systematic regularities detected in the sequence can eventually be used to discover factors of *m*. What is behind this is that each of the numbers in the basic sequence here must be a so-called quadratic residue of the form *Mod[v^2, m]*, and given any such quadratic residue *x* the expression *GCD[x + Mod[x^2, m], m]* turns out always to be a factor of *m*—and at least sometimes a non-trivial one. So if one could reconstruct sufficiently many complete numbers *x* from the sequence of *Mod[x, 2]* values then this would provide a way to factor *m* (compare the Pollard rho method above). But in practice it is difficult to do this, because without knowing the factors of *m* one cannot even readily tell whether a given *x* is a quadratic residue modulo *m*. The pictures below show as black squares all the quadratic residues for each successive *m* going down the page (the ordinary squares 1, 4, 9, 16, … show up as vertical black stripes). If *m* is a prime *p*, then the simple tests *JacobiSymbol[x, p] == 1* (see page 1081) or *Mod[$x^{(p-1)/2}$, p] == 1* determine whether *x* is a quadratic residue. But with $m = p q$, one has to factor *m* and find *p* and *q* in order to carry out similar tests. The condition *Mod[p, 4] == Mod[q, 4] == 3* ensures that only one of the solutions *+v* and *-v* to *x == Mod[v^2, m]* is ever a quadratic residue, with the result that the iterated mapping *x → Mod[x^2, m]* always has a unique inverse. But unlike in a cellular automaton even given a complete *x* (the analog of a complete cellular automaton state) it is difficult to invert the mapping and solve for the *x* on the previous step.



Traditional Mathematics and Mathematical Formulas

■ **Practical empirical mathematics.** In looking for formulas to describe behavior seen in this book I have in practice typically taken associated sequences of numbers and then tested whether obvious regularities are revealed by combinations of such operations as: computing successive differences (see note below), computing running totals, looking for repeated blocks, picking out running maxima, picking out numbers with particular modular residues, and looking at positions of particular values, and at the forms of the digit sequences of these positions.

■ **Difference tables and polynomials.** A common mathematical approach to analyzing sequences is to form a difference table by repeatedly evaluating *d[list_] := Drop[list, 1] - Drop[list, -1]*. If the elements of *list* correspond to values of a polynomial of degree *n* at successive integers, then *Nest[d, list, n + 1]* will contain only zeros. If the differences are computed modulo *k* then the difference table corresponds essentially to the evolution of an additive cellular automaton (see page 597). The pictures below show the results with *k = 2* (rule 60) for (a) *Fibonacci[n]*, (b) Thue-Morse sequence, (c) Fibonacci substitution system, (d) *(Prime[n] - 1)/2*, (e) digits of π. (See also page 956.)



■ **Page 607 · Implementation.** The color of a cell at position *{x, y}* in the pattern shown is given by *Extract[{{1, 0, 1}, {0, 1, 0}}, Mod[{y, x}, {2, 3}] + 1]*.

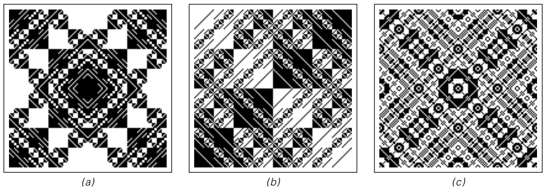
■ **Page 608 · Nested patterns and numbers.** See page 931.

■ **Page 609 · Implementation.** Given the rules for a substitution system in the form used on page 931 a finite automaton (as on page 957) which yields the color of each cell from the digit sequences of its position is

```
Map[Flatten[MapIndexed[#2 - 1 → Position[rules, #1 → _][[
    1, 1]] &, Last[#], {-1}]] &, rules]
```

This works in any number of dimensions so long as each replacement yields a block of the same cuboidal form.

■ **Arbitrary digit operations.** If the operation on digit sequences that determines whether a square will be black can be performed by a finite automaton (see page 957) then the pattern generated must always be either repetitive or nested. The pictures below show examples with more general operations. Picture (a) in effect shows which words in a simple context-free language of parenthesis matching (see page 939) are syntactically correct. Scanning the digit sequences from the left, one starts with 0 open parentheses, then adds 1 whenever corresponding digits in the *x* and *y* coordinates differ, and subtracts 1 whenever they are the same. A square is black if no negative number ever appears. Picture (b) has a black square wherever digits at more than half the possible positions differ between the *x* and *y* coordinates. Picture (c) has a black square wherever the maximum run of either identical or different digits has a length which is an odd number. All the patterns shown have the kind of intricate substructure typical of nesting. But none of the patterns are purely nested.



■ **Page 610 · Generating functions.** A convenient algebraic way to describe a sequence of numbers *a[n]* is to give a generating function *Sum[a[n] x^n, {n, 0, ∞}]*. *1/(1 - x)* thus corresponds to the constant sequence and *1/(1 - x - x^2)* to the Fibonacci sequence (see page 890). A 2D array can be described by *Sum[a[t, n] x^n y^t, {n, -∞, ∞}, {t, -∞, ∞}]*. The array for rule 60 is then *1/(1 - (1 + x) y)*, for rule 90 *1/(1 - (1/x + x) y)*, for rule 150 *1/(1 - (1/x + 1 + x) y)* and for second-order reversible rule 150 (see page 439) *1/(1 - (1/x + 1 + x) y - y^2)*. Any rational function is the generating function for some additive cellular automaton.

■ **Page 611 · Pascal's triangle.** See notes on page 870.

■ **Nesting in bitwise functions.** See page 871.

■ **Trinomial coefficients.** The coefficient of x^n in the expansion of *(1 + x + x^2)^t* is

```
Sum[Binomial[n + t - 1 - 3 k, n - 3 k]
    Binomial[t, k] (-1)^k, {k, 0, t}]
```

which can be evaluated as

```
Binomial[2 t, n] Hypergeometric2F1[-n, n - 2 t, 1/2 - t, 1/4]
```

or finally *GegenbauerC[n, -t, -1/2]*. This result follows directly from the generating function formula

```
(1 - 2 x z + x^2)^-m == Sum[GegenbauerC[n, m, z] x^n, {n, 0, ∞}]
```

■ **Gegenbauer functions.** Introduced by Leopold Gegenbauer in 1893 *GegenbauerC[n, m, z]* is a polynomial in *z* with integer coefficients for all integer *n* and *m*. It is a special case of *Hypergeometric2F1* and *JacobiP* and satisfies a second-order ordinary differential equation in *z*. The *GegenbauerC[n, d/2 - 1, z]* form a set of orthogonal functions on a *d*-dimensional sphere. The *GegenbauerC[n, 1/2, z]* obtained for *d = 3* are *LegendreP[n, z]*.

■ **Standard mathematical functions.** There are an infinite number of possible functions with integer or continuous arguments. But in practice there is a definite set of standard named mathematical functions that are considered reasonable to include as primitives in formulas, and that are implemented as built-in functions in *Mathematica*. The so-called elementary functions (logarithms, exponentials, trigonometric and hyperbolic functions, and their inverses) were mostly introduced before about 1700. In the 1700s and 1800s another several hundred so-called special functions were introduced. Most arose first as solutions to specific differential equations, typically in physics and astronomy; some arose as products, sums of series or inverses of other functions. In the mid-1800s it became clear that despite their different origins most of these functions could be viewed as special cases of *Hypergeometric2F1[a, b, c, z]*, and that the functions covered the solutions to all linear differential equations of a certain type. (*Zeta* and *PolyLog* are parametric derivatives of *Hypergeometric2F1*; elliptic modular functions are inverses.) Rather few new special functions have been introduced over the past century. The main reason has been that the obvious generalizations seem to yield classes of functions whose properties cannot be worked out with much completeness. So, for example, if there are more parameters it becomes difficult to find continuous definitions that work for all complex values of these parameters. (Typically one needs to generalize formulas that are initially set up with integer numbers of terms; examples include taking *Power[x, y]* to be *Exp[Log[x] y]* and *x!* to be *Gamma[x + 1]*.) And if one modifies the usual hypergeometric equation *y″[x] == f[y[x], y′[x]]* by making *f* nonlinear then solutions typically become hard to find, and vary greatly in character with the form of *f*. (For rational *f* Paul Painlevé in the 1890s identified just 6 additional types of functions that are needed, but even now series expansions are not known for all of them.) Generalizations of special functions can in principle be used to represent the results of many kinds of computations. Thus, for example, generalized elliptic theta functions represent solutions to arbitrary polynomial equations, while multivariate hypergeometric functions represent arbitrary conformal mappings. In *Mathematica*, however, functions like *Root* provide more convenient ways to access such results.

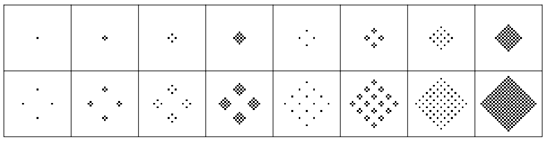
A variety of standard mathematical functions with integer arguments were introduced in the late 1800s and early 1900s in connection with number theory. A few functions that involve manipulation of digits have also become standard since the use of computers became widespread.

■ **1D sequences.** Generating functions that are rational always lead to sequences which after reduction modulo 2 are purely repetitive. Algebraic generating functions can also lead to nested sequences. (Note that to get only integer sequences such generating functions have to be specially chosen.) *Sqrt[1 - 4 x]/2* yields a sequence with 1's at positions 2^m, as essentially obtained from the substitution system *{2 → {2, 1}, 1 → {1, 0}, 0 → {0, 0}}*. *Sqrt[(1 - 3 x)/(1 + x)]/2* yields sequence (a) on page 84. *(1 + Sqrt[(1 - 3 x)/(1 + x)])/(2 (1 + x))* (see page 890) yields the Thue-Morse sequence. (This particular generating function satisfies the equation $(1 + x)^3 f^2 - (1 + x)^2 f + x == 0$.) $(1 - 9 x)^{1/3}$ yields almost the Cantor set sequence from page 83. *EllipticTheta[3, π, x]/2* gives a sequence with 1's at positions m^2.

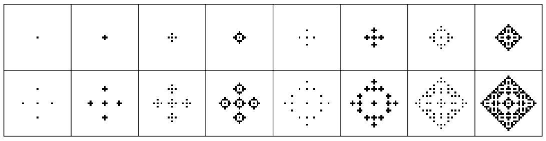
For any sequence with an algebraic generating function and thus for any nested sequence the nth element can always be expressed in terms of hypergeometric functions. For the Thue-Morse sequence the result is

$$1/2\,(-1)^n + (-3)^n \sqrt{\pi}\ \mathit{Hypergeometric2F1}[3/2, -n, 3/2 - n, -1/3]/(4\, n!\ \mathit{Gamma}[3/2 - n])$$

■ **Multidimensional additive rules.** The 2D analog of rule 90 yields the patterns shown below. The colors of cells are given essentially by *Mod[Multinomial[t, x, y], 2]*. In *d* dimensions *(2 d)^DigitCount[t, 2, 1]* cells are black at step *t*. The fractal dimension of the *(d+1)*-dimensional structure formed from all black cells is *Log[2, 1 + 2 d]*.

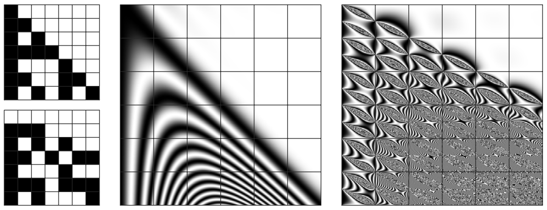


The 2D analog of rule 150 yields the patterns below; the fractal dimension of the structure in this case is *Log[2, (1 + Sqrt[1 + 4/d]) d]*.

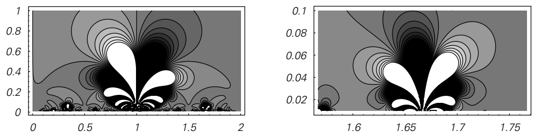


■ **Continuous generalizations.** Functions such as *Binomial[t, n]* and *GegenbauerC[n, -t, -1/2]* can immediately be evaluated for continuous *t* and *n*. The pictures on the right below show $\mathit{Sin}[1/2\, \pi\, a[t, n]]^2$ for these functions (equivalent to *Mod[a[t, n], 2]* for integer *a[t, n]*). The discrete results on the left can be obtained by sampling only where integer grid lines cross. Note that without further conditions the continuous forms cannot be considered unique extensions of the discrete ones. The presence of poles in quantities such as

GegenbauerC[1/2, -t, -1/2] leads to essential singularities in the rightmost picture below. (Compare page 922.)



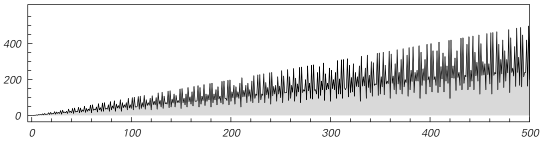
■ **Nested continuous functions.** Most standard continuous mathematical functions never show any kind of nested behavior. Elliptic theta and elliptic modular functions are exceptions. Each of these functions has definite finite values only in a limited region of the complex plane, and on the boundary of this region they exhibit singularities at every single rational point. The picture below shows *Im[ModularLambda[x + ⅈ y]]*. Like other elliptic modular functions, *ModularLambda* satisfies *f[z] == f[(a + b z)/(c + d z)]* with *a*, *b*, *c*, *d* integers such that *a c - b d == 1*. The function can be obtained as the solution to a second-order nonlinear ordinary differential equation. Nested behavior is also found for example in *EllipticTheta[3, 0, z]*, which is given essentially by *Sum[*z^{n^2}*, {n, ∞}]*.



■ **Page 613 · GCD array.** (See also page 950.) There are various deviations from perfect randomness. The density of white squares is asymptotically $6/\pi^2 \simeq 0.61$. (The probability for *s* randomly chosen integers to be relatively prime is *1/Zeta[s]*.) No 2×2 or larger block of white squares can ever occur. An arrangement of black squares with any list of relative offsets will always eventually occur. (This follows from the Chinese Remainder Theorem.) The first 2×2 block of black squares occurs at *{14, 20}*, the first 3×3 block at *{1274, 1308}* and the first 4×4 block at *{7247643, 10199370}*. The densities of such blocks are respectively about 0.002, 2×10^{-6} and 10^{-14}. In general the density for an arrangement of white squares with offsets *v* is given in *s* dimensions by (no simple closed formula seems to exist except for the 1×1 case)

```
Product[With[{p = Prime[n]},
    1 - Length[Union[Mod[v, p]]]/p^s], {n, ∞}]
```

White squares correspond to lattice points that are directly visible from the origin at the top left of the picture, so that lines to them do not pass through any other integer points. On row *n* the number of white squares encountered before reaching the leading diagonal is *EulerPhi[n]*. This function is shown below. Its computation is known in general to be equivalent in difficulty to factoring *n* (see page 1090). *GCD* can be computed using Euclid's algorithm as discussed on page 915.



■ **Power cellular automata.** Multiplication by *m* in base *k* corresponds to a local cellular automaton operation on digit sequences when every prime that divides *m* also divides *k*. The first non-trivial cases for which this is so are $k = 6$, $m = 2^i 3^j$ and $k = 10$, $m = 2^i 5^j$. When *m* itself divides *k*, the cellular automaton rule is *{_, b_, c_} → m Mod[b, k/m] + Quotient[c, k/m]*; in other cases the rule can be obtained by composition. A similar result holds for rational *m*, obtained for example by allowing *i* and *j* above to be negative. In all cases the cellular automaton rule, like the original operation on numbers, is invertible. The inverse rule, corresponding to multiplication by *1/m*, can be obtained by applying the rule for multiplication by the integer k^q/m, then shifting right by *q* positions. (See page 903.)

The condition for locality in negative bases (see page 902) is more stringent. The first non-trivial example is $k = -6$, $m = 8$, corresponding to a rule that depends on four neighboring cells.

Non-trivial examples of multiplication by *m* in base *k* all appear to be class 3 systems (see page 250), with small changes in initial conditions growing at a roughly fixed rate.

■ **Page 615 · Computing powers.** The method of repeated squaring (also known as the binary power method, Russian peasant method and Pingala's method) computes the quantity m^t by performing about *Log[t]* multiplications and building up the sequence

```
FoldList[#1^2 m^#2 &, 1, IntegerDigits[t, 2]]
```

(related to the Horner form for the base 2 representation of *t*). Given two numbers *x* and *y* their product can be computed in base *k* by (*FromDigits* does the carries)

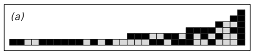
```
FromDigits[ListConvolve[IntegerDigits[x, k],
    IntegerDigits[y, k], {1, -1}, 0], k]
```

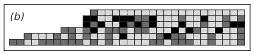
For numbers with *n* digits direct evaluation of the convolution would take about n^2 steps. But FFT-related methods reduce this to about *n Log[n]* steps (see also page 1142). And this implies that to find a particular digit of m^t in base *k* will take altogether about *t Log[t]*^{2} steps.

One might think that a more efficient approach would be to start with the trivial length t digit sequence for c^t in base c, then to find a particular base k digit just by converting to base k. However, the straightforward method for converting a t-digit number x to base k takes about t divisions, though this can be reduced to around Log[t] by using a recursive method such as

```
FixedPoint[Flatten[Map[If[# < k, #, With[
        {e = Ceiling[Log[k, #]/2]}, {Quotient[#, k^e], With[
          {s = Mod[#, k^e]}, If[s == 0, Table[0, {e}], {Table[0,
             {e - Floor[Log[k, s]] - 1}], s}]]}]] &, #]] &, {x}]
```

The pictures below show stages in the computation of 3^{20} (a) by a power tree in base 2 and (b) by conversion from base 3. Both approaches seem to require about the same number of underlying steps. Note that even though one may only want to find a single digit in m^t, I know of no way to do this without essentially computing all the other digits in m^t as well.

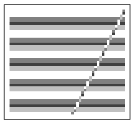


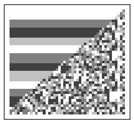


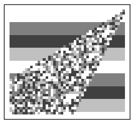
▪ **Complex powers.** The pictures below show successive powers of complex numbers z with digits extracted according to

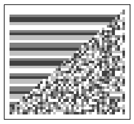
```
(2 d[Re[#], w] + d[Im[#], w] &)[z^t]
d[x_, w_] := If[x < 0, 1 - d[-x, w], IntegerDigits[x, 2, w]]
```

Non-trivial cases of complex number multiplication never correspond to local cellular automaton operations. (Compare page 933.)


$z = 1 + i$


$z = 2 + i$


$z = 3 + i$


$z = 1 + 2i$

▪ **Additive cellular automata.** As discussed on page 951 a step in the evolution of an additive cellular automaton can be thought of as multiplication by a polynomial modulo k. After t steps, therefore, the configuration of such a system is given by PolynomialMod[poly^t, k]. This quantity can be computed using power tree methods (see below), though as discussed on page 609, even more efficient methods are also available. (A similar formalism can be set up for any of the cellular automata with generalized additivity discussed on page 952; see also page 886.)

▪ **The more general case.** One can think of a single step in the evolution of any system as taking a rule r and state s, and producing a new state h[r, s]. Usually the representations that are used for r and s will be quite different, and the function h will have no special properties. But for both multiplication rules and additive cellular automata it turns out that rules and states can be represented in the same way, and the evolution functions h have the property of being associative, so that h[a, h[b, c]] == h[h[a, b], c]. This means that in effect one can always choose to evolve the rule rather than a state. A consequence is that for example 4 steps of evolution can be computed not only as h[r, h[r, h[r, h[r, s]]]] but also as h[h[h[r, r], h[r, r]], s] or u = h[r, r]; h[h[u, u], s]—which requires only 3 applications of h. And in general if h is associative the result Nest[h[r, #] &, s, t] of t steps of evolution can be rewritten for example using the repeated squaring method as

```
h[Fold[If[#2 == 0, h[#1, #1], h[r, h[#1, #1]]] &,
    r, Rest[IntegerDigits[t, 2]]], s]
```

which requires only about Log[t] rather than t applications of h.

As a very simple example, consider a system which starts with the integer 1, then at each step just adds 1. One can compute the result of 9 steps of evolution as $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$, but a better scheme is to use partial results and compute successively $1 + 1; 2 + 2; 1 + 4; 5 + 5$—which is what the repeated squaring method above does when h = Plus, $r = s = 1$. This same basic scheme can be used with any associative function h—Max, GCD, And, Dot, Join or whatever—so long as suitable forms for r and s are used.

For the multiplication rules discussed in the main text both states and rules can immediately be represented by integers, with h = Times, and $r = m$ giving the multiplier. For additive cellular automata, states and rules can be represented as polynomials (see page 951), with h[a_, b_] := PolynomialMod[a b, k] and for example $r = 1 + x$ for elementary rule 60. The correspondence between multiplication rules and additive cellular automata can be seen even more directly if one represents all states by integers and computes h in terms of base k digits. In both cases it then turns out that h can be obtained from (see note above)

```
h[a_, b_] := FromDigits[g[ListConvolve[
        IntegerDigits[a, k], IntegerDigits[b, k], {1, -1}, 0]], k]
```

where for multiplication rules g = Identity and for additive cellular automata g = Mod[#, k] &. For multiplication rules, there are normally carries (handled by FromDigits), but for power cellular automata, these have only limited range, so that g = Mod[#, k^σ] & can be used.

For any associative function h the repeated squaring method allows the result of t steps of evolution to be computed with only about Log[t] applications of h. But to be able to do this some of the arguments given to h inevitably need to be larger.

So whether a speedup is in the end achieved will depend on how fast *h* can be evaluated for arguments of various sizes. Typically the issue is whether *h[a, b]* for large *a* and *b* can be found with much less effort than it would take to evaluate *h[r, b]* about *a* times. If *h = Times*, then as discussed in the note above, the most obvious procedure for evaluating *h[a, b]* would involve about *m n* operations, where *m* and *n* are the numbers of digits in *a* and *b*. But when $m \simeq n$ FFT-related methods allow this to be reduced to about *n Log[n]* operations. And in fact whenever *h* is commutative (*Orderless*) it turns out that such methods can be used, and substantial speedups obtained. But whether anything like this will work in other cases is not clear.

(See also page 886.)

■ **Evaluation chains.** The idea of building up computations like *1 + 1 + 1 + …* from partial results has existed since Egyptian times. Since the late 1800s there have been efforts to find schemes that require the absolute minimum number of steps. The method based on *IntegerDigits* in the previous two notes can be improved (notably by power tree methods), but apparently about *Log[t]* steps are always needed. (Finding the optimal addition chain for given *t* may be NP-complete.)

One can also consider building up lists of non-identical elements, say by successively using *Join*. In general a length *n* list can require about *n* steps. But if the list contains a nested sequence, say generated using a substitution system, then about *Log[n]* steps should be sufficient. (Compare page 566.)

■ **Boolean formulas.** A Boolean function of *n* variables can always be specified by an explicit table giving values for all 2^n possible inputs. (Any cellular automaton rule with an *n*-cell neighborhood corresponds to such a function; digit sequences in rule numbers correspond to explicit tables of values.) Like ordinary algebraic functions, Boolean functions can also be represented by a variety of kinds of formulas. Those on pages 616 and 618 use so-called disjunctive normal form (DNF) *And[…] ∨ And[…] ∨ …*, which is common in practice in programmable logic arrays (PLAs). (The addition and multiplication operators in the main text should be interpreted as *Or* and *And* respectively.) In general any given function will allow many DNF representations; minimal ones can be found as described below. Writing a Boolean function in DNF is the rough analog of applying *Expand* to a polynomial. Conjunctive normal form (CNF) *Or[…] ∧ Or[…] ∧ …* is the rough analog of applying *Factor*. DNF and CNF both involve Boolean formulas of depth 2. As in the note on multilevel formulas below, one can also in effect introduce intermediate variables to get recursive formulas of larger depth, somewhat analogous to results from *Collect*. (Unbalanced depths in different parts of a formula lead to latencies in a circuit, reducing practical utility.)

■ **DNF minimization.** From a table of values for a Boolean function one can immediately get a DNF representation just by listing cases where the value is 1. For one step in rule 30, for example, this yields *{{1, 0, 0}, {0, 1, 1}, {0, 1, 0}, {0, 0, 1}}*, as shown on page 616. One can think of this as specifying corners that should be colored on an *n*-dimensional Boolean hypercube. To reduce the representation, one must introduce “don’t care” elements _; in this example the final minimal form consists of the list of 3 so-called implicants *{{1, 0, 0}, {0, 1, _}, {0, _, 1}}*. In general, an implicant with *m* _’s can be thought of as corresponding to an *m*-dimensional hyperplane on the Boolean hypercube. The problem of minimization is then to find the minimal set of hyperplanes that will cover the corners for a particular Boolean function. The first step is to work out so-called prime implicants corresponding to hyperplanes that cannot be contained in higher-dimensional ones. Given an original DNF list *s*, this can be done using *PI[s, n]*:

```
PI[s_, n_] := Union[Flatten[
      FixedPointList[f[Last[#], n] &, {{}, s}][[All, 1]], 1]]
g[a_, b_] := With[{i = Position[Transpose[{a, b}], {0, 1}]},
    If[Length[i] == 1 && Delete[a, i] === Delete[b, i],
      {ReplacePart[a, _, i]}, {}]]
f[s_, n_] := With[
   {w = Flatten[Apply[Outer[g, #1, #2, 1] &, Partition[Table[
        Select[s, Count[#, 1] == i &], {i, 0, n}], 2, 1], {1}],
     3]}, {Complement[s, w, SameTest → MatchQ], w}]
```

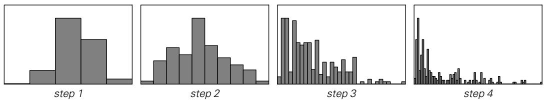
The minimal DNF then consists of a collection of these prime implicants. Sometimes it is all of them, but increasingly often when $n \geq 3$ it is only some. (For example, in *{{0, 0, _}, {0, _, 1}, {_, 0, 0}}* the first prime implicant is covered by the others, and can therefore be dropped.) Given the original list *s* and the complete prime implicant list *p* the so-called Quine-McCluskey procedure can be used to find a minimal list of prime implicants, and thus a minimal DNF:

```
QM[s_, p_] := First[Sort[Map[p[[#]] &,
      h[{}, Range[Length[s]], Outer[MatchQ, s, p, 1]]]]]
h[i_, r_, t_] := Flatten[Map[h[Join[i, r[[#]]], Drop[r, #],
      Delete[Drop[t, {}, #], Position[t[[All, #]], {True}]]] &,
    First[Sort[Map[Position[#, True] &, t]]]], 1]
h[i_, _, {}] := {i}
```

The number of steps required in this procedure can increase exponentially with the length of *p*. Other procedures work slightly more efficiently, but in general the problem of finding the minimal DNF for a Boolean function of *n* variables is NP-complete (see page 768) and is thus expected to grow in difficulty faster than any polynomial in n. In practice, however, cases up to about $n = 12$ are nevertheless currently handled quite routinely.

■ **Formula sizes.** There are a total of 2^{2^n} possible Boolean functions of n variables. The maximum number of terms needed to represent any of these functions in DNF is 2^{n-1}. The actual numbers of functions which require 0, 1, 2, … terms is for $n = 2$: $\{1, 9, 6\}$; for $n = 3$: $\{1, 27, 130, 88, 10\}$, and for $n = 4$: $\{1, 81, 1804, 13472, 28904, 17032, 3704, 512, 26\}$. The maximal length turns out always to be realized for the simple parity function *Xor*, as well as its negation. The reason for this is essentially that these functions are the ones that make the coloring of the Boolean hypercube maximally fragmented. (Other functions with maximal length are never additive, at least for $n \le 4$.)

■ **Cellular automaton formulas.** See page 869. The maximum length DNF for elementary rules after 1 step is 4, and this is achieved by rules 105, 107, 109, 121, 150, 151, 158, 182, 214 and 233. These rules have behavior of quite varying complexity. Rules 150 and 105 are additive, and correspond to *Xor* and its negation. After t steps the maximum conceivable DNF would be of length 2^{2t}. In practice, after 2 steps, the maximum length is 9, achieved by rules 107, 121 and 182; after 3 steps, it is 33 achieved by rule 182; after 4 steps, 78 achieved by rule 129; after 5 steps 256 achieved by rules 105 and 150. The distributions of lengths for all elementary rules are shown below.



Note that the length of a minimal DNF representation cannot be considered a reliable measure of the complexity of a function, since among other things, just exchanging the role of black and white can substantially change this length (as in the case of rule 126 versus rule 129).

■ **Primitive functions.** There are several possible choices of primitive functions that can be combined to represent any Boolean function. In DNF *And*, *Or* and *Not* are used. *Nand = Not[And[##]] &* alone is also sufficient, as shown on page 619 and further discussed on page 807. (It is indicated by $\bar{\wedge}$ in the main text.) The functions *And*, *Xor* and *Not* are equivalent to *Times*, *Plus* and *1 - # &* for variables modulo 2, and in this case algebraic functions like *PolynomialReduce* can be used for minimization. (See also page 1102.)

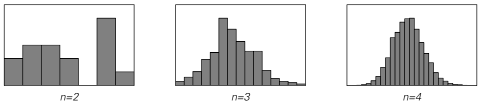
■ **Multilevel formulas.** DNF formulas always have depth 2. By allowing larger depths one can potentially find smaller formulas for functions. A major result from the 1980s is that it requires a formula with depth at least *Log[n]/(c + Log[Log[n]])* to make it possible to represent an *Xor* of n variables using a polynomial number of *And*, *Or* and *Not* operations. If one chooses an n-variable Boolean function at random out of the 2^{2^n} possibilities, it is typical that regardless of depth a formula involving at least $2^n/n$ operations will be needed to represent it. A formula of polynomial size and logarithmic depth exists only when a function is the computational complexity class NC discussed on page 1149.

Little is known about systematic minimization of Boolean formulas with depths above 2. Nevertheless, some programs for circuit design such as SIS do include a few heuristics. And this for example allows SIS to generate higher depth formulas somewhat smaller than the minimal DNF for the first three steps of rule 30 evolution.

$b_1 = a_2 + a_3; \bar{a}_1 b_1 + a_1 \bar{b}_1$
$b_1 = \bar{a}_2 a_3 + a_2 \bar{a}_3; b_2 = a_4 + a_5; a_1 b_1 + a_1 \bar{b}_2 + \bar{a}_1 \bar{b}_1 b_2$
$b_1 = a_6 + a_7; b_2 = a_4 + a_5; b_3 = \bar{a}_5 b_1 + a_4 \bar{b}_1; b_4 = \bar{b}_1 + b_2; \bar{a}_1 \bar{a}_3 b_3 + \bar{a}_1 a_2 \bar{b}_2 + a_1 \bar{a}_2 \bar{a}_3 \bar{b}_3 + \bar{a}_1 a_2 a_4 \bar{b}_3 + a_1 a_2 \bar{a}_4 b_2 + \bar{a}_1 \bar{a}_2 a_3 b_4 + a_1 \bar{a}_2 a_3 \bar{b}_4 + a_1 a_2 a_3 b_3 b_4$

■ **Page 619 · NAND expressions.** If one allows a depth of at most $2n$ any n-input Boolean function can be obtained just by combining 2-input *Nand* functions. (See page 807.) (Note that unless one introduces an explicit copy operation—or adds variables as in the previous note—there is no way to use the same intermediate result multiple times without recomputing it.)

The pictures below show the distributions of numbers of *Nand* operations needed for all 2^{2^n} n-input Boolean functions. For $n = 2$, the largest number of such operations is 6, achieved by *Nor*; for $n = 3$, it is 14, achieved by *Xor* (rule 150); for $n = 4$, it is 27, achieved by rule 5737, which is *Not[Xor[##]] &* except when all inputs are *True*. The average number of operations needed when $n = 2, 3, 4$ is about $\{2.875, 6.09, 12.23\}$.

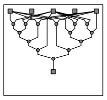


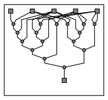
The maximum depths for the expressions of minimal size are respectively 4, 6 and 7, always achieved among others for the function taking the most *Nand* operations. The total numbers of functions involving successive depths are: $n = 2$: $\{2, 3, 5, 6\}$, $n = 3$: $\{3, 6, 22, 99, 72, 54\}$, $n = 4$: $\{4, 10, 64, 923, 9663, 54622, 250\}$, corresponding to averages $\{2.9, 4.5, 5.8\}$.

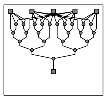
The following generates explicit lists of *n*-input Boolean functions requiring successively larger numbers of *Nand* operations:

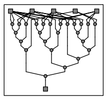
```
Map[FromDigits[#, 2] &, NestWhile[Append[#,
    Complement[Flatten[Table[Outer[1 - Times[##] &,
        #[[i]], #[[-i]], 1], {i, Length[#]}], 2], Flatten[#, 1]]] &,
  {1 - Transpose[IntegerDigits[Range[2^n] - 1, 2, n]]},
  Length[Flatten[#, 1]] < 2^2^n &], {2}]
```

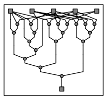
The results for 2-step cellular automaton evolution in the main text were found by a recursive procedure. First, expressions containing progressively more *Nand* operations were enumerated, and those for functions that had not been seen before were kept. It then turned out that this made it possible to get to expressions at least half as large as any needed, so that it could be assumed that remaining expressions could be decomposed as *f[##] ⊼ g[##] &*, where *f* had already been found. The pictures below show some more results obtained in this way.


rule 150


rule 110


rule 126


rule 45


rule 54

■ **Cellular automaton formulas.** For 1 step, the elementary cellular automaton rules are exactly the 256 $n = 3$ Boolean functions. For 2 steps, they represent a small subset of the 2^{32} $n = 5$ functions. They require an average of about 11.6 *Nand* operations, and a maximum of 27 (achieved by rules 107 and 121).

For rule 254 the result after t steps (which is always asymmetric, even though the rule is symmetric) is

```
Nest[{{#, #[[2]] + 1}, #[[2]] + 1} &, {{1, 1}, {2, 2}}, t - 2]
```

If explicit copy operations were allowed, then the number of *Nand* operations after t steps could not increase faster than t^2 for any rule. But without copy (fanout) operations no corresponding result is immediately clear.

■ **Binary decision diagrams.** One can specify a Boolean function of n variables by giving a finite automaton (and thus a network) in which paths exist only for those lists of values for which the function yields *True*. The resulting so-called binary decision diagram (BDD) can be minimized using the methods of page 957. Out of all possible Boolean functions the number that require BDDs of sizes 1, 2, … is for $n = 2$: *{1, 0, 6, 9}* and for $n = 3$: *{1, 0, 0, 27, 36, 132, 60}*; the absolute maximum grows roughly like 2^n. For cellular automata with simple behavior, the minimal BDD typically grows linearly on successive steps. For rule 254, for example, it is $8t + 2$, while for rule 90 it is $4t + 2$. For cellular automata with more complex behavior, it typically grows roughly exponentially. Thus for rule 30 it is *{7, 14, 29, 60, 129}* and for rule 110 *{7, 15, 27, 52, 88}*. The size of the minimal BDD can depend on the order in which variables are specified; thus for example, just reflecting rule 30 to give rule 86 yields *{6, 11, 20, 36, 63}*.

In practical system design BDDs have become fairly popular in the past ten years, and by maintaining minimality when logical combinations of functions are formed, cases with millions of nodes have been studied. (Some practical systems are found to yield fairly small BDDs, while others are not.)

■ **History.** Logic has been used as an abstraction of arguments in ordinary language since antiquity. Its serious mathematical formulation began with the work of George Boole in the mid-1800s. (See page 1151.) Concepts of Boolean algebra were applied to electronic switching circuits by Claude Shannon in 1937, and became a standard part of electronic design methodology by the 1950s. DNF had been introduced as part of the development of mathematical logic in the early 1900s, but became particularly popular in the 1970s with the advent of programmable logic arrays (PLAs) used in application-specific integrated circuits (ASICs). Diagrammatic and mechanical methods for minimizing simple logic expressions have existed since at least medieval times. More systematic methods for minimizing complex expressions began to be developed in the early 1950s, but until well into the 1980s a diagrammatic method known as a Karnaugh map was the most commonly used in practice. In the late 1970s there began to be computer programs for large-scale Boolean minimization—the best known being *Espresso*. Only in the 1990s, however, did exact minimization of complex DNF expressions become common. Minimization of Boolean expressions with depth larger than 2 has been considered off and on since the late 1950s, and became popular in the 1990s in connection with the BDDs discussed above. Various forms of Boolean minimization have routinely been used in chip and circuit design since the late 1980s, though often physical and geometrical constraints are now more important than pure logical ones. In addition, theoretical studies of minimal Boolean circuits became increasingly popular starting in the 1980s, as discussed on page 1148.

■ **Reversible logic.** In an ordinary Boolean function with n inputs there is no unique way to tell from its output which of the 2^n possible sets of inputs was given. But as noted in the 1970s, it is possible to set up systems that evaluate Boolean functions, yet operate reversibly. The basic idea is to have m outputs as well as m inputs—with every one of the 2^m possible sets of inputs mapping to a unique set of outputs. Normally one specifies the first n inputs, taking the others to be fixed, and then looks say at the first output, ignoring all others. One can represent the inside of such a system much like a sorting network from page 1142—but with s-input s-output gates instead of pair comparisons. If each such gate is itself reversible, then overall reversibility is guaranteed. With gates that in effect implement $\{p, q\} \to \{p \barwedge q\}$ and $\{p\} \to \{p, p\}$ (with other inputs constant, and other outputs ignored) one can set up a direct translation of Boolean functions given in the form shown on page 619. Of the 24 possible reversible $s = 2$ gates, none can yield anything other than additive Boolean functions (as formed from *Xor* and *Not*). But of the 40,320 ($8!$) reversible $s = 3$ gates (in 52 distinct classes) it turns out that 38,976 (in 23 classes) can be used to reproduce any possible Boolean function. A simple example of such a universal gate is $\{p\_, q\_, 1\} \to \{q, p, 1\}$—and not allowing permutations of gate inputs (or in effect wire crossings) a simple example is $\{p\_, q\_, q\_\} \to \{q, 1 - p, 1 - p\}$. (Compare pages 1147 and 1173.)

■ **Continuous systems.** The systems I discuss in the main text of this section are mostly discrete. But from experience with traditional mathematics one might have the impression that it would at some basic level be easier to get formulas for continuous systems. I believe, however, that this is not the case, and that the reason for the impression is just that it is usually so much more difficult even to represent the states of continuous systems that one normally tends to work only with ones that have comparatively simple overall behavior—and are therefore more readily described by formulas. (See also pages 167 and 729.)

As an example of what can happen in continuous systems consider iterated mappings $x \to a\,x\,(1 - x)$ from page 920. Each successive step in such a mapping can in principle be represented by an algebraic formula. But the table below gives for example the actual algebraic formulas obtained in the case $a = 4$ after applying *FullSimplify*—and shows that these increase quite rapidly in complexity.

x
$4\,(1 - x)\,x$
$16\,(1 - 2\,x)^2\,(1 - x)\,x$
$64\,(1 - 2\,x)^2\,(1 - x)\,x\,(8\,(x - 1)\,x + 1)^2$
$256\,(1 - 2\,x)^2\,(1 - x)\,x\,(8\,(x - 1)\,x + 1)^2\,(32\,(x - 1)\,x\,(1 - 2\,x)^2 + 1)^2$
$1024\,(1 - 2\,x)^2\,(1 - x)\,x\,(8\,(x - 1)\,x + 1)^2\,(32\,(x - 1)\,x\,(1 - 2\,x)^2 + 1)^2\,(128\,(1 - 2\,x)^2\,(x - 1)\,x\,(8\,(x - 1)\,x + 1)^2 + 1)^2$

In the specific case $a = 4$, however, it turns out that by allowing more sophisticated mathematical functions one can get a complete formula: the result after any number of steps t can be written in any of the forms

$$Sin[2^t\,ArcSin[\sqrt{x}\,]]^2$$
$$(1 - Cos[2^t\,ArcCos[1 - 2\,x]])/2$$
$$(1 - ChebyshevT[2^t,\,1 - 2\,x])/2$$

where these follow from functional relations such as

$$Sin[2\,x]^2 == 4\,Sin[x]^2\,(1 - Sin[x]^2)$$
$$ChebyshevT[m\,n,\,x] == ChebyshevT[m,\,ChebyshevT[n,\,x]]$$

For $a = 2$ it also turns out that there is a complete formula:

$$(1 - (1 - 2\,x)^{2^t})/2$$

And the same is true for $a = -2$:

$$1/2 - Cos[1/3\,(\pi - (-2)^t\,(\pi - 3\,ArcCos[1/2 - x]))]$$

In all these examples t enters essentially only in a^t. And if one assumes that this is a general feature then one can formally derive for any a the result

$$1/2\,(1 - g[a^t\,InverseFunction[g][1 - 2\,x]])$$

where g is a function that satisfies the functional equation

$$g[a\,x] == 1 + 1/2\,a\,(g[x]^2 - 1)$$

When $a = 4$, $g[x]$ is $Cosh[Sqrt[2\,x]]$. When $a = 2$ it is $Exp[x]$ and when $a = -2$ it is $2\,Cos[1/3\,(\pi - \sqrt{3}\,x)]$. But in general for arbitrary a there is no standard mathematical function that seems to satisfy the functional equation. (It has long been known that only elliptic functions such as *JacobiSN* satisfy polynomial addition formulas—but there is no immediate analog of this for replication formulas.) Given the functional equation one can find a power series for $g[x]$ for any a. The series has an accumulation of poles on the circle $Abs[a]^2 == 1$; the coefficient of x^m turns out to have denominator

$$2\,\hat{}\,(m - DigitCount[m,\,2,\,1])\,Apply[Times,\,Table[Cyclotomic[s,\,a]\,\hat{}\,Floor[(m - 1)/s],\,\{s,\,m - 1\}]]$$

For other iterated maps general formulas also seem rare. But for example $x \to a\,x + b$ and $x \to 1/(a + b\,x)$ both give results just involving powers, while $x \to Sqrt[a\,x + b]$ sometimes yields trigonometric functions, as on page 915. In addition, from a known replication formula for an elliptic or other function one can often construct an iterated map whose behavior can be expressed in terms of that function. (See also page 919.)

Human Thinking

■ **The brain.** There are a total of about 100 billion neurons in a human brain (see page 1075), each with an average of a few thousand synapses connecting it to other cells. On a small scale the arrangement of neurons seems quite haphazard. But on a larger scale the brain seems to be organized into areas with very definite functions. This organization is sometimes revealed by explicitly following nerve fibers. More often it has been deduced by looking at what happens if parts of the brain are disabled or stimulated. In recent times it has also begun to be possible to image local electrical and metabolic activity while the brain is in normal operation. From all these methods it is known that each kind of sensory input is first processed in its own specific area of the brain. Inputs from different senses are integrated in an area that effectively maintains a map of the body; a similar area initiates output to muscles. Certain higher mental functions are known to be localized in definite areas of the brain, though within these areas there is often variability between individuals. Areas are currently known for specific aspects of language, memory (see below) and various cognitive tasks. There is some evidence that thinking about seemingly rather similar things can lead to significantly different patterns of activity.

Most of the action of the brain seems to be associated with local electrical connections between neurons. Some collective electrical activity is however revealed by EEG. In addition, levels of chemicals such as hormones, drugs and neurotransmitters can have significant global effects on the brain.

▪ **History.** Ever since antiquity immense amounts have been written about human thinking. Until recent centuries most of it was in the tradition of philosophy, and indeed one of the major themes of philosophy throughout its history has been the elucidation of principles of human thinking. However, almost all the relevant ideas generated have remained forever controversial, and almost none have become concrete enough to be applied in science or technology. An exception is logic, which was introduced in earnest by Aristotle in the 4th century BC as a way to model certain patterns of human reasoning. Logic developed somewhat in medieval times, and in the late 1600s Gottfried Leibniz tried to use it as the foundation for a universal language to capture all systematic thinking. Beginning with the work of George Boole in the mid-1800s most of logic began to become more closely integrated with mathematics and even less convincingly relevant as a model for general human thinking.

The notion of applying scientific methods to the study of human thinking developed largely with the rise of the field of psychology in the mid-1800s. Two somewhat different approaches were taken. The first concentrated on doing fairly controlled experiments on humans or animals and looking at responses to specific stimuli. The second concentrated on trying to formulate fairly general theories based on observations of overall human behavior, initially in adults and later especially in children. Both approaches achieved some success, but by the 1930s many of their positions had become quite extreme, and the identification of phenomena to contradict every simple conclusion reached led increasingly to the view that human thinking would allow no simple explanations.

The idea that it might be possible to construct machines or other inanimate objects that could emulate human thinking existed already in antiquity, and became increasingly popular starting in the 1600s. It began to appear widely in fiction in the 1800s, and has remained a standard fixture in portrayals of the future ever since.

In the early 1900s it became clear that the brain consists of neurons which operate electrically, and by the 1940s analogies between brains and electrical machines were widely discussed, particularly in the context of the cybernetics movement. In 1943 Warren McCulloch and Walter Pitts formulated a simple idealized model of networks of neurons and tried to analyze it using methods of mathematical logic. In 1949 Donald Hebb then argued that simple underlying neural mechanisms could explain observed psychological phenomena such as learning. Computer simulations of neural networks were done starting in the mid-1950s, but the networks were too small to have any chance to exhibit behavior that could reasonably be identified with thinking. (Ironically enough, as mentioned on page 879, the phenomenon central to this book of complex behavior with simple underlying rules was in effect seen in some of these experiments, but it was considered a distraction and ignored.) And in the 1960s, particularly after Frank Rosenblatt's introduction of perceptrons, neural networks were increasingly used only as systems for specific visual and other tasks (see page 1076).

The idea that computers could be made to exhibit human-like thinking was discussed by Alan Turing in 1950 using many of the same arguments that one would give today. Turing made the prediction that by 2000 a computer would exist that could pass the so-called Turing test and be able to imitate a human in a conversation. (René Descartes had discussed a similar test for machines in 1637, but concluded that it would never be passed.) When electronic computers were first becoming widespread in the 1950s they were often popularly referred to as "electronic brains". And when early efforts to make computers perform tasks such as playing games were fairly successful, the expectation developed that general human-like thinking was not far away. In the 1960s, with extensive support from the U.S. government, great effort was put into the field of artificial intelligence. Many programs were written to perform specific tasks. Sometimes the programs were set up to follow general models of the high-level processes of thinking. But by the 1970s it was becoming clear that in almost all cases where programs were successful (notable examples being chess, algebra and autonomous control), they typically worked by following definite algorithms not closely related to general human thinking.

Occasional work on neural networks had continued through the 1960s and 1970s, with a few definite results being obtained using methods from physics. Then in the early 1980s, particularly following work by John Hopfield, computer simulations of neural networks became widespread. Early applications, particularly by Terrence Sejnowski and Geoffrey Hinton, demonstrated that simple neural networks could be made to learn tasks of at least some sophistication. But by the mid-1990s it was becoming clear that—probably in large part as a consequence of reliance on methods from traditional mathematics—typical neural network models were mostly being successful only in situations where what was needed was a fairly straightforward extension of standard continuous probabilistic models of data.

▪ **The future.** To achieve human-like thinking with computers will no doubt require advances in both basic science and technology. I strongly suspect that a key element is to be able to store a collection of experiences comparable to those of a human. Indeed, to succeed even with specific tasks such as speech recognition or language translation seems to require human-like amounts of background knowledge. Present-day computers are beginning to have storage capacities that are probably comparable to those of the brain. From looking at the brain one might guess that parallel or other non-standard hardware might be required to achieve efficient human-like thinking. But I rather suspect that—much as in the analogy between birds and airplanes—it will in the end be possible to set up algorithms that achieve the same basic functions but work satisfactorily even on standard sequential-processing computers.

▪ **Sleep.** A common feature of higher organisms is the existence of distinct behavioral states of sleep and wakefulness. There are various theories that sleep is somehow fundamental to the process of thinking. But my guess is that its most important function is quite mundane: just as muscles build up lactic acid waste products, so also I suspect synapses in the brain build up waste products, and these can only safely be cleared out when the brain is not in normal use.

▪ **Page 621 · Pointer encoding.** The pointer encoding compression method discussed on page 571 implements a very simple form of memory based on literal repetitions, and already leads to fairly good compression of many kinds of data.

▪ **Page 622 · Hashing.** Given data in the form of sequences of numbers between 0 and $k-1$, a very simple hashing scheme is just to compute *FromDigits[Take[list, n], k]*. But for data corresponding, say, to English words this scheme yields a very nonuniform distribution of hash codes, since, for example, there are many words beginning with "ba", but none beginning with "bb". The slightly modified but still very simple scheme *Mod[FromDigits[list, k], m]*, where m is usually chosen to be a prime, is what is most often used in practice. For a fair fraction of values of m, the hash codes obtained from this scheme change whenever any element of *list* is changed. If $m = k^s - 1$ then it turns out that interchanging a pair of adjacent length s blocks in *list* never affects the result. Out of the many hundreds of times that I have used hashing in practice, I recall only a couple of cases where schemes like the one just described were not adequate, and in these cases the data always turned out to have quite dramatic regularities.

In typical applications hash codes give locations in computer memory, from which actual data is found either by following a chain of pointers, or by probing successive locations until an empty one is reached. In the internals of *Mathematica* the most common way that hashing is used is for recognizing data and finding unique stored versions of it. There are several subtleties associated with setting up hash codes that appropriately handle approximate real numbers and *Mathematica* patterns.

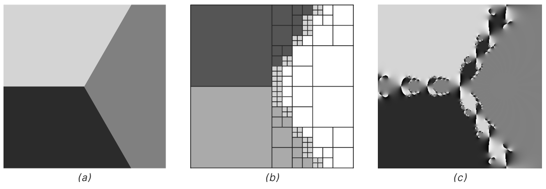
Hashing is a sufficiently simple idea that it has been invented independently many times since at least the 1950s. The main alternative to hashing is to store data with successive elements corresponding to successive levels in a tree. In the past decade, hashing has become widely used not only for searching but also for authentication. The basic idea in this case is to take a document and to compute from it a small hash code that changes when almost any change is made in the document, and for which it is a difficult problem of cryptanalysis to work out what changes in the document will lead to no change in the hash code. Schemes for such hash codes can fairly easily be constructed using rule 30 and other cellular automata.

▪ **Page 623 · Similar words.** The soundex system for hashing names according to sound was first used on 1880 U.S. census data, and is still today widely used by telephone information services. The system works essentially by dropping vowels and assigning consonants to six possible groups. More sophisticated systems along the same lines can be set up using finite automata.

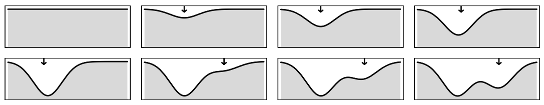
Natural language query systems usually work by stripping words to their linguistic roots (e.g. "stripping" → "strip") before looking them up. Spell-checking systems typically find suggested corrections by doing a succession of lookups after applying transformations based on common errors.

Even given two specific words it can be difficult to find out whether they should be considered similar. Fairly efficient algorithms are known for cases such as genetic sequences where small numbers of insertions, deletions and substitutions are expected. But if more complicated transformations are allowed—say corresponding to rules in a multiway system—the problem rapidly becomes intractable (see page 765).

▪ **Numerical data.** In situations where pieces of data can be thought of as points in space similarity can often be defined in terms of spatial distance. And this means that around every point corresponding to a piece of data in memory there is a region of points that can be considered more similar to that point than to any other. Picture (a) shows a so-called Voronoi diagram (see page 1038) obtained in this way in two dimensions. Particularly in higher dimensions, it becomes rather difficult in practice to determine for certain which existing point is closest to some new point. But to do it approximately is considerably easier. One approach, illustrated in picture (b), is to use a d-dimensional tree. Another approach, illustrated in picture (c), is to set up a continuous function with minima at the existing points, and then to search for the closest minimum. In most cases, this search will be done using some iterative scheme such as Newton's method; the result is that the boundaries between regions typically take on an intricate nested form. (The case shown corresponds to iteration of the map $z \to z - (z^3 - 1)/(3 z^2)$ corresponding to Newton's method for finding the complex roots of $z^3 == 1$.)



The pictures below show how one can build up a kind of memory landscape by successively adding points. In a first approximation, the regions considered similar to a particular minimum are delimited by sharp watersheds corresponding to local maxima in the landscape. But if an iterative scheme for minimization is used, these watersheds are typically no longer sharp, but take on a local nested structure, much as in picture (c) above.



In numbers earlier digits are traditionally considered more important than later ones, and this allows numbers to be arranged in a simple one-dimensional sequence. But in strings where each element is considered equally important, no such layout is possible. A vague approximation, perhaps useful for some applications, is nevertheless to use a space-filling curve (see page 893).

▪ **Error-correcting codes.** In many information transmission and storage applications one needs to be able to recover data even if some errors are introduced into it. The standard way to do this is to set up an error-correcting code in which blocks of m original data elements are represented by a codeword of length n that in effect includes some redundant elements. Then—somewhat in analogy to retrieving closest memories—one can take a sequence of length n that one receives and find the codeword that differs from it in the fewest elements. If the codewords are chosen so that every pair differs by at least r elements (or equivalently, have so-called Hamming distance at least r), then this means that errors in up to *Floor[(r - 1)/2]* elements can be corrected, and finding suitable codewords is like finding packings of spheres in n-dimensional space. It is common in practice to use so-called linear codes which can be analyzed using algebraic methods, and for which the spheres are arranged in a repetitive array. The Hamming codes with $n = 2^s - 1$, $m = n - s$, $r = 3$ are an example, invented by Marcel Golay in 1949 and Richard Hamming in 1950. Defining

```
PM[s_] := IntegerDigits[Range[2^s - 1], 2, s]
```

blocks of data of length m can be encoded with

```
Join[data, Mod[data . Select[PM[s], Count[#, 1] > 1 &], 2]]
```

while blocks of length n (and at most one error) can be decoded with

```
Drop[(If[# == 0, data, MapAt[1 - # &, data, #]] &)[
    FromDigits[Mod[data . PM[s], 2], 2]], -s]
```

A number of families of linear codes are known, together with a few nonlinear ones. But in all cases they tend to be based on rather special mathematical structures which do not seem likely to occur in any system like the brain.

▪ **Matrix memories.** Many times since the 1950s it has been noted that methods from linear algebra suggest ways to construct associative memories in which data can potentially be retrieved on the basis of some form of similarity. Typically one starts from some list of vectors to be stored, then forms a matrix such as $m = PseudoInverse[list]$. Given a new piece of data corresponding to a vector v, its decomposition in terms of stored vectors can be found by computing $v . m$. And by applying various forms of thresholding one can often pick out at least approximately the stored vector closest to the piece of data given. But such schemes tend to be inefficient in practice, as well as presumably being unrealistic as actual models of the brain.

■ **Neural network models.** The basic rule used in essentially all neural network models is extremely simple. Each neuron is assumed to have a value between -1 and 1 corresponding roughly to a firing rate. Then given a list *s[i]* of the values of one set of neurons, one finds the values of another set using *s[i + 1] = u[w . s[i]]*, where in early models *u = Sign* was usually chosen, and now *u = Tanh* is more common, and *w* is a rectangular matrix which gives weights—normally assumed to be continuous numbers, often between -1 and +1—for the synaptic connections between the neurons in each set. In the simplest case, studied especially in the context of perceptrons in the 1960s, one has only two sets of neurons: an input layer and an output layer. But with suitable weights one can reproduce many functions. For example, with three inputs and one output, *w = {{-1, +1, -1}}* yields essentially the rule for the rule 178 elementary cellular automaton. But out of the 2^{2^n} possible Boolean functions of *n* inputs, only 14 (out of 16) can be obtained for *n = 2*, 104 (out of 256) for *n = 3*, 1882 for *n = 4*, and 94304 for *n = 5*. (The VC dimension is *n + 1* for such systems.) The key idea that became popular in the early 1980s was to consider neural networks with an additional layer of “hidden units”. By introducing enough hidden units it is then possible—just as in the formulas discussed on page 616—to reproduce essentially any function. Suitable weights (which are typically far from unique) are in practice usually found by gradient descent methods based either on minimization of deviations from desired outputs given particular inputs (supervised learning) or on maximization of some discrimination or other criterion (unsupervised learning).

Particularly in early investigations of neural networks, it was common to consider systems more like very simple cellular automata, in which the *s[i]* corresponded not to states of successive layers of neurons, but rather to states of the same set of neurons at successive times. For most choices of weights, such a system exhibits typical class 3 behavior and never settles down to give an obvious definite output. But in special circumstances probably not of great biological relevance it can yield class 2 behavior. An example studied by John Hopfield in 1981 is a symmetric matrix *w* with neuron values being updated sequentially in a random order rather than in parallel.

■ **Memory.** Since the early 1900s it has been suspected that long-term memory is somehow encoded in the strengths of synaptic connections between nerve cells. It is known that at least in specific cases such strengths can remain unchanged for at least hours or more, but can immediately change if connected nerve cells have various patterns of simultaneous excitation. The changes that occur appear to be associated changes in ionic channels in cell membranes and sometimes with the addition of new synapses between cells.

Observations suggest that in humans there are several different types of memory, with somewhat different characteristics. (Examples include memory for facts and for motor skills.) Usually there is a short-term or so-called working component, lasting perhaps 30 seconds, and typically holding perhaps seven items, and a long-term component that can apparently last a lifetime. Specific parts of the brain (such as the hippocampus) appear necessary for the long-term component to form. In at least some cases there is evidence for specialized areas that handle particular types of memories. When new data is first presented, many parts of the brain are often active in processing it. But once the data has somehow been learned, only parts directly associated with handling it usually appear to be active.

Memories often seem at some level to be built up incrementally, as reflected in smooth learning curves for motor skills. It is not clear whether this is due to actual incremental changes in nerve cells or just to the filling in of progressively more cases that differ in detail.

Experiments on human learning suggest that a particular memory typically involves an association between components from several sensory systems, as well as emotional state.

When several incomplete examples of data are presented, there appears to be some commonality in the character of generalizations that we make. One mathematically convenient but probably unrealistic model studied in recent years in the context of computational learning theory involves building up minimal Boolean formulas consistent with the examples seen.

■ **Child development.** As children get older their thinking becomes progressively more sophisticated, advancing through a series of fairly definite stages that appear to be associated with an increasing ability to handle generalization and abstraction. It is not clear whether this development is primarily associated with physiological changes or with the accumulation of more experiences (or, in effect, with the addition of more layers of software). Nor is it clear how it relates to the fact that the number of items that can be stored in short-term memory seems steadily to increase.

■ **Computer interfaces.** The earliest computer interfaces were essentially just numerical. By the 1960s text-based interfaces were common, and in the decade following the introduction of the Macintosh in 1984 graphical interfaces based on menus and dialogs came to largely dominate consumer software. Such interfaces work well if what one wants is basically to take a single object and apply operations to it. And they can be extended somewhat by using visual block diagrams or flowcharts. But whenever there is neither just a single active data element nor an obvious sequence of independent execution steps—as for many of the programs in this book—my experience has always been that the only viable choice of interface is a computer language like *Mathematica*, based essentially on one-dimensional sequences of word-like constructs. The rule diagrams in this book represent a possible new method for specifying some simpler programs, but it remains to be seen whether such diagrams can readily both be created incrementally by humans and interpreted by computer.

▪ **Page 627 · Structure of *Mathematica*.** Beneath all the sophisticated capabilities of *Mathematica* lies a remarkably simple basic structure. The key idea is to represent data of any kind by a symbolic expression of the general form *head[arg*$_1$*, arg*$_2$*, …]*. (*a + b*^{2} is thus *Plus[a, Power[b, 2]]*, *{a, b, c}* is *List[a, b, c]* and *a = b + 1* is *Set[a, Plus[b, 1]]*.) The basic action of *Mathematica* is then to transform such expressions according to whatever rules it knows. Most often these rules are specified in terms of *Mathematica* patterns—expressions in which _ can stand for any expression.

▪ **Context-free languages.** The set of valid expressions in a context-free language can be defined recursively by rules such as *"e" → "e + e"* and *"e" → "(e)"* that specify how one expression can be built up from sequences of literal objects or "tokens" and other expressions. (As discussed on page 939, the fact that the left-hand side contains nothing more than *"e"* is what makes the language context free.) To interpret or parse an expression in a context-free language one has to go backwards and find out which rules could be used to generate that expression. (For the built-in syntax of *Mathematica* this is achieved using *ToExpression*.)

It is convenient to think of expressions in a language as having forms such as *s["(", "(", ")", ")"]* with *Attributes[s] = Flat*. Then the rules for the language consisting of balanced runs of parentheses (see page 939) can be written as

```
{s[e] → s[e, e], s[e] → s["(", e, " )"], s[e] → s["(", ")"]}
```

Different expressions in the language can be obtained by applying different sequences of these rules, say using (this gives so-called leftmost derivations)

```
Fold[#1 /. rules[[#2]] &, s[e], list]
```

Given an expression, one can then use the following to find a list of rules that will generate it—if this exists:

```
Parse[rules_, expr_] := Catch[Block[{t = { }}, NestWhile[
        ReplaceList[#, MapIndexed[ReverseRule, rules]] &,
        {{expr, { }}}, # /. {s[e], u_} :→ Throw[u]; # =!= { } &];]]
ReverseRule[a_ → b_, {i_}] := {___, {s[x___, b, y___], {u___}},
      ___} :→ {s[x, a, y], {i, u}} /; FreeQ[s[x], s[a]]
```

In general, there will in principle be more than one such list, and to pick the appropriate list in a practical situation one normally takes the rules of the language to apply with a certain precedence—which is how, for example, *x + y z* comes to be interpreted in *Mathematica* as *Plus[x, Times[y, z]]* rather than *Times[Plus[x, y], z]*. (Note that in practice the output from a parser for a context-free language is usually represented as a tree—as in *Mathematica FullForm*—with each node corresponding to one rule application.)

Given only the rules for a context-free language, it is often very difficult to find out the properties of the language (compare page 944). Indeed, determining even whether two sets of rules ultimately yield the same set of expressions is in general undecidable (see page 1138).

▪ **Languages.** There are about 140 human languages and 15 full-fledged computer languages currently in use by a million people or more. Human languages typically have perhaps 50,000 reasonably common words; computer languages usually have a few hundred at most (*Mathematica*, however, has at least nominally somewhat over 1000). In expressing general human issues, different human languages tend to be largely equivalent—though they often differ when it comes to matters of special cultural or environmental interest to their users. Computer languages are also mostly equivalent in their handling of general programming issues—and indeed among widespread languages the only substantial exception is *Mathematica*, which supports symbolic, functional and pattern-based as well as procedural programming. Human languages have mostly evolved quite haphazardly over the course of many centuries, becoming sometimes simpler, sometimes more complicated. Computer languages are almost always specifically designed once and for all, usually by a single person. New human languages have sometimes been developed—a notable example being Esperanto in the 1890s—but for reasons largely of political history none have in practice become widely used.

Human languages always seem to have fairly definite rules for what is grammatically correct. And in a first approximation these rules can usually be thought of as specifying that every sentence must be constructed from various independent nested phrases, much as in a context-free grammar (see above). But in any given language there are always many exceptions, and in the end it has proved essentially impossible to identify specific detailed features—beyond for example the existence of nouns and verbs—that are convincingly universal across more than just languages with clear historical connections (such as the Indo-European ones). (One obvious general deviation from the context-free model is that in practice subordinate clauses can never be nested too deep if a sentence is expected to be understood.)

All the computer languages that are in widespread use today are based quite explicitly on context-free grammars. And even though the original motivation for this was typically ease of specification or implementation, I strongly suspect that it has also been critical in making it readily possible for people to learn such languages. For in my observation, exceptions to the context-free model are often what confuse users of computer languages the most—even when those users have never been exposed to computer languages before. And indeed the same seems to be true for traditional mathematical notation, where occasional deviations from the context-free model in fields like logic seem to make material particularly hard to read. (A notable feature that I was surprised to discover in designing *Mathematica* 3 is that users of mathematical notation seem to have a remarkably universal view of the precedence of different mathematical operators.)

The idea of describing languages by grammars dates back to antiquity (see page 875). And starting in the 1800s extensive studies were made of the comparative grammars of different languages. But the notion that grammars could be thought of like programs for generating languages did not emerge with clarity until the work of Noam Chomsky beginning in 1956. And following this, there were for a while many efforts to formulate precise models for human languages, and to relate these to properties of the brain. But by the 1980s it became clear—notably through the failure of attempts to automate natural language understanding and translation—that language cannot in most cases (with the possible exception of grammar-checking software) meaningfully be isolated from other aspects of human thinking.

Computer languages emerged in the early 1950s as higher-level alternatives to programming directly in machine code. FORTRAN was developed in 1954 with a syntax intended as a simple idealization of mathematical notation. And in 1958, as part of the ALGOL project, John Backus used the idea of production systems from mathematical logic (see page 1150) to set up a recursive specification equivalent to a context-free grammar. A few deviations from this approach were tried—notably in LISP and APL—but by the 1970s, following the development of automated compiler generators such as yacc, so-called Backus-Naur context-free specifications for computer languages had become quite standard. (A practical enhancement to this was the introduction of two-dimensional grammar in *Mathematica* 3 in 1996.)

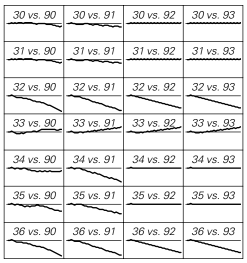
▪ **Page 631 · Computer language fluency.** It is common that when one knows a human language sufficiently well, one feels that one can readily "think in that language". In my experience the same is eventually true with computer languages. In particular, after many years of using *Mathematica*, I have now got to the point where I can effectively think directly in *Mathematica,* so that I can start entering a *Mathematica* program even though I may be a long way from being able to explain in English what I want to do.

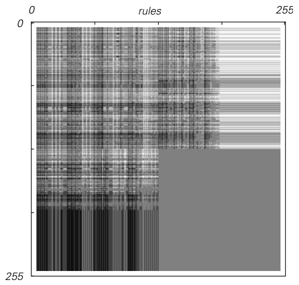
▪ **Brainteasers.** In many puzzles and IQ tests the setup is to give a few elements in some sequence of numbers, strings or pictures, then to ask what the next element would be. The correct answer is normally assumed to be the one that in a sense allows the simplest description of all the data. But despite attempts to remove cultural and other biases such questions in practice seem almost always to rely on being able to retrieve from memory various specific forms and transformations. And I strongly suspect that if one were, for example, to construct similar questions using outputs from many of the simple programs I discuss in this book then unless one had studied almost exactly the cases of such programs used one would never manage to work out the answers.

▪ **Human generation of randomness.** If asked to type a random sequence of 0's and 1's, most people will at first produce a sequence with too many alternations between 0 and 1. But with modest learning time my experience is that one can generate sequences with quite good randomness.

▪ **Game theory.** Remarkably simple models are often believed to capture features of what might seem like sophisticated decision making by humans, animals and human organizations. A particular case on which many studies have been done is the so-called iterated Prisoner's Dilemma, in which two players make a sequence of choices *a* and *b* to "cooperate" (*1*) or "defect" (*2*), each trying to maximize their score *m[[a, b]]* with *m = {{1, -1}, {2, 0}}*. At a single step, standard static game theory from the 1940s implies that a player should always defect, but in the 1960s a folk theorem emerged that if a whole sequence of steps is considered then a possible strategy for perfectly rational players is always to cooperate—in apparent agreement with some observations on human and animal behavior. In 1979 Robert Axelrod tried setting up computer programs as players and found that in tournaments between them the winner was often a simple "tit-for-tat" program that cooperates on the first step, then on subsequent steps just does whatever its opponent did on the previous step. The same winner was also often obtained by natural selection—a fact widely taken to explain cooperation phenomena in evolutionary biology and the social sciences. In the late 1980s similar studies were done on processes such as auctions (cf page 1015), and in the late 1990s on games such as Rock, Paper, Scissors (RoShamBo) (with $m = \{\{0, -1, 1\}, \{1, 0, -1\}, \{-1, 1, 0\}\}$). (A simpler game—certainly played since antiquity—is Penny Matching or Evens and Odds, with $m = \{\{1, -1\}, \{-1, 1\}\}$.) But even though they seemed to capture or better actual human behavior, the programs considered in all these cases typically just used standard statistical or Markov model methods, or matching of specific sequences—making them far too weak to make predictions about the kinds of complex behavior shown in this book. (Note that a program can always win the games above if it can in effect successfully predict each move its opponent will make. In a game between two arbitrary programs it can be undecidable which will win more often over the course of an infinite number of moves.)

■ **Games between programs.** One can set up a game between two programs generating single bits of output by for example taking the input at each step to be the concatenation of the historical sequences of outputs from the two programs. The pictures below show what happens if the programs operate by applying elementary cellular automaton rules t times to $2t+1$ inputs. The plots on the left show cumulative scores in the Evens and Odds game; the array on the right indicates for each of the 256 possible rules the average number of wins it gets against each of the 256 rules. At some level considerable complexity is evident. But the rules that win most often typically seem to do so in rather simple ways.





Higher Forms of Perception and Analysis

■ **Biological perception.**Animals can process data not only from visual or auditory sources (as discussed on pages 577 and 585), but also from mechanical, thermal, chemical and other sources. Usually special receptors for each type of data convert it into electrical impulses in nerve cells. Mechanical and thermal data are often mapped onto an array of nerve cells in the brain, from which features are extracted similar to those in visual perception. Taste involves data from solids and liquids; smell data from gases. The human tongue has millions of taste buds scattered on its surface, each with many tens of nerve cells. Rather little is currently known about how taste data is processed, and it is not even clear whether the traditional notion that there are just four or so primary tastes is correct. The human nose has several tens of millions of receptors, apparently broken into a few hundred distinct types. Each of these types probably has proteins that form pockets with definite shapes, making it respond to molecules whose shapes fit into these pockets. People typically distinguish a few thousand odors, presumably by comparing responses of different receptor types. (Foods usually contain tens of distinct odors; manufactured scents hundreds.) There is evidence that at the first level of processing in the brain all receptors of a given type excite nerve cells that lie in the same spatial region. But just how different regions are laid out is not clear, and may well differ between individuals. Polymers whose lengths differ by more than one or two repeating units often seem to smell different, and it is conceivable that elaborate general features of shapes of molecules can be perceived. But more likely there is no way to build up sophisticated taste or smell data—and no analog of any properties such as repetition or nesting.

■ **Page 634 · Evolving to predict.** If one thinks that biological evolution is infinitely powerful one might imagine that by emulating it one would always be able to find ways to predict any sequence of data. But in practice methods based, for example, on genetic programming seem to do at best only about as well as all sorts of other methods discussed in this chapter. And typically what limits them seems to be much the same as I argue in Chapter 8 limits actual biological evolution: that incremental changes are difficult to make except when the behavior is fairly simple. (See also page 985.)

It is common for animals to move in apparently random ways when they are trying to avoid predators. Yet I suspect that the randomness they use is often generated by quite simple rules (see page 1011)—so that in principle it could be predictable. So it is then notable that biological evolution has apparently never made predators able to catch their prey by predicting anything that looks to us particularly random; instead strategies tend to be based on tricks that do not require predicting more than at most repetition.

■ **Page 635 · Familiar features.** What makes features familiar to us is that they are common in our typical environment and are readily recognized by our built-in human powers of perception. In the distant past humans were presumably exposed only to features generated by ordinary natural processes. But ever since the dawn of civilization humans have increasingly been exposed to things that were explicitly constructed through engineering, architecture, art, mathematics and other human activities. And indeed as human knowledge and culture have progressed, humans have ended up being exposed to new kinds of features. For example, while repetition has been much emphasized for several millennia, it is only in the past couple of decades that precise nesting has had much emphasis. So this may make one wonder what features will be emphasized in the future. The vast majority of forms created by humans in the past—say in art or architecture—have had basic features that are either directly copied from systems in nature, or are in effect built up by using extremely simple kinds of rules. On the basis of the discoveries in this book I thus tend to suspect that almost any feature that might end up becoming emphasized in the future will already be present—and probably even be fairly common—in the behavior of the kinds of simple programs that I have discussed in this book. (When future technology is routinely able to interact with individual atoms there will presumably quickly be a new class of quantum and other features that become familiar.)

■ **Relativism and postmodernism.** See pages 1131 and 1196.