EXCERPTED FROM

STEPHEN WOLFRAM
A NEW KIND OF SCIENCE

SECTION 10.5

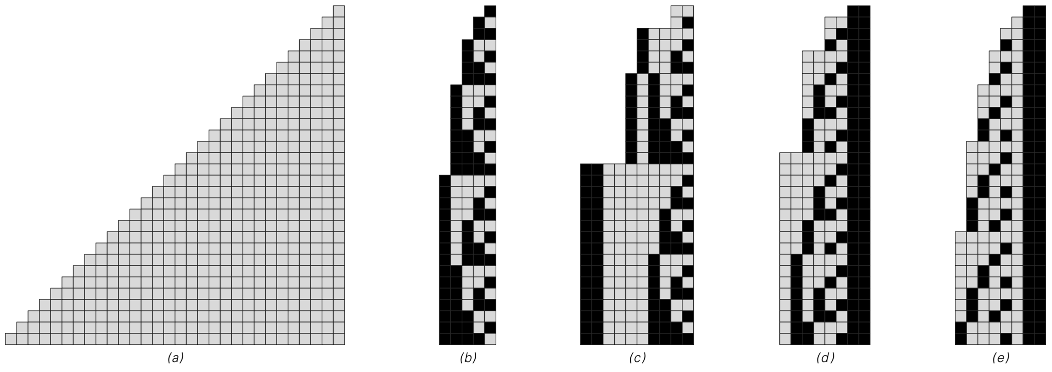
Data Compression

Data Compression

One usually thinks of perception and analysis as being done mainly in order to provide material for direct human consumption. But in most modern computer and communications systems there are processes equivalent to perception and analysis that happen all the time when data is compressed for more efficient storage or transmission.

One simple example of such a process is run-length encoding—a method widely used in practice to compress data that involves long sequences of identical elements, such as bitmap images of pages of text with large areas of white.

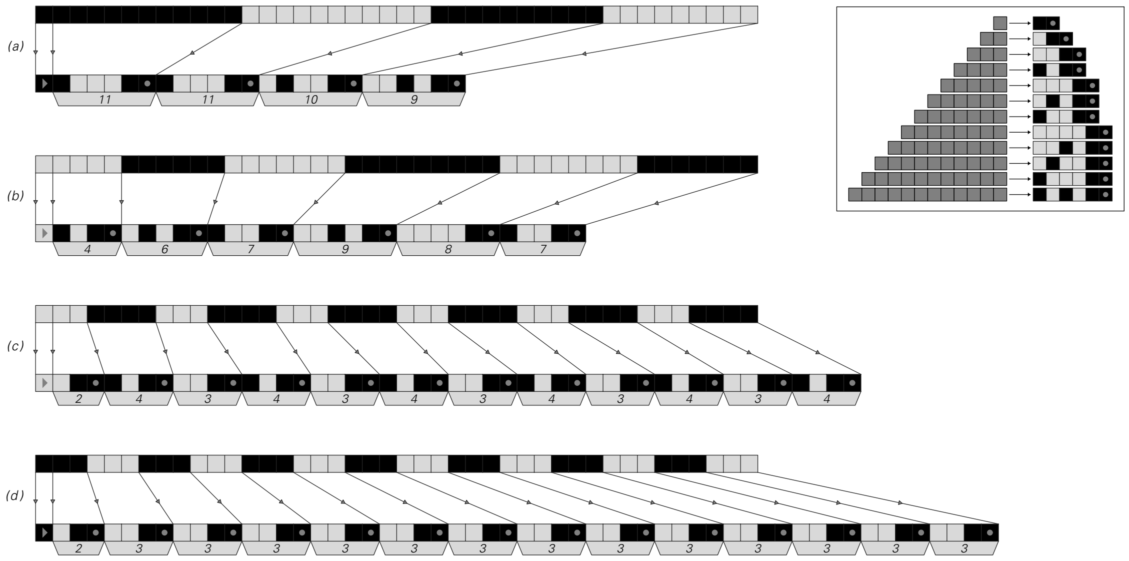
The basic idea of run-length encoding is to break data into runs of identical elements, and then to specify the data just by giving the lengths of these runs. This means, for example, that instead of having to list explicitly all the cells in a run of, say, 53 identical cells, one instead just gives the number "53". And the point is that even if the "53" is itself represented in terms of black and white cells, this representation can be much shorter than 53 cells.



Various representations of numbers from 1 to 30. (a) is unary, in which any given number is represented by a sequence of cells whose length is equal to that number. (b) is ordinary binary or base 2 representation. (c), (d) and (e) are set up to be self-delimiting, so that the end of a number can be recognized purely by looking at the cells within it. (c) is like (b), except that it has a specification of the number of digits at the front. (d) is essentially binary-coded-ternary, with the end of the number indicated by a pair of black cells. (e) uses a non-integer base derived from the Fibonacci sequence, with the property that a pair of black cells can appear only at the end of each number.

Indeed, any digit sequence can be thought of as providing a short representation for a number. But for run-length encoding it turns out that ordinary base 2 digit sequences do not quite work. For if the numbers corresponding to the lengths of successive runs are given one after another then there is no way to tell where the digits of one number end and the next begin.

Several approaches can be used, however, to avoid this problem. One, illustrated in picture (c) at the bottom of the facing page, is to insert at the beginning of each number a specification of how many digits the number contains. Another approach, illustrated in picture (d), is in effect to have two cells representing each digit, and then to indicate the end of the number by a pair of black cells. A variant on this approach, illustrated in picture (e), uses a non-integer base in which pairs of black cells can occur only at the end of a number.

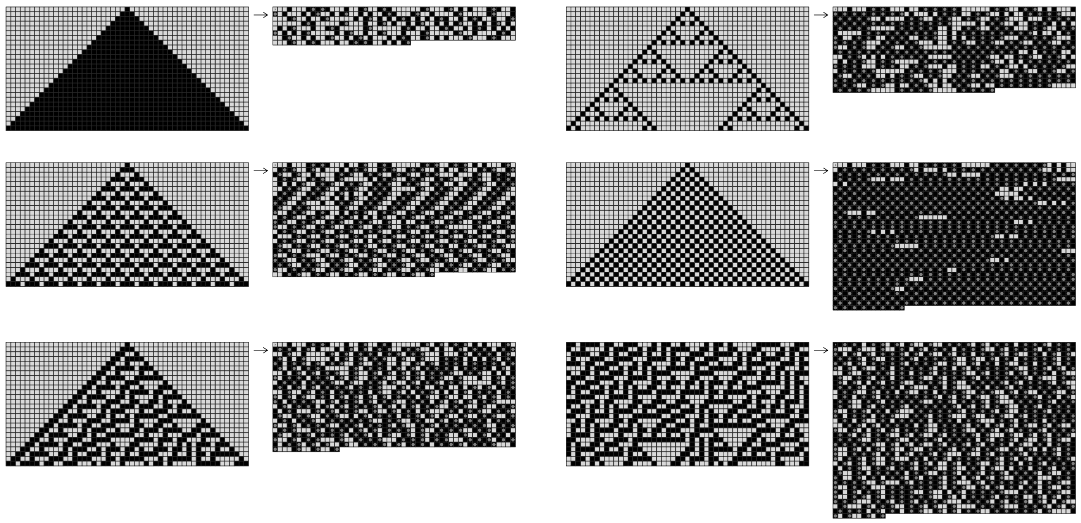


Examples of run-length encoding. In each case the input data is shown on top, and the output is shown below. The arrows between input and output show how the data is broken into runs of identical elements. Each run is then specified by a number, represented as a sequence ending with two black cells, as indicated in the inset picture, and in picture (e) on the facing page. For the first two sets of input data there are enough long runs present that compression is achieved. But for the other two sets no compression is achieved. Note that the first cell in the output is used to specify whether the first run is black or white. In this picture and those that follow, the output consists purely of black and white cells; the gray annotations are included purely as aids to interpretation.

For small numbers, all these approaches yield representations that are at least somewhat longer than the explicit sequences shown in picture (a). But for larger numbers, the representations quickly become much shorter. And this means that they can potentially be used to achieve compression in run-length encoding.

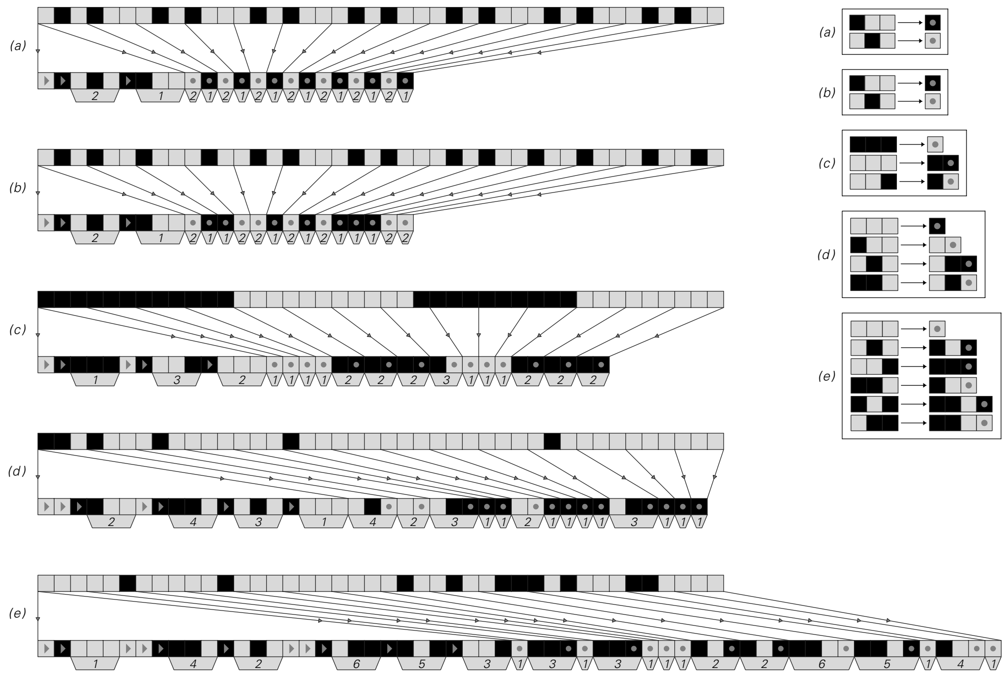
The pictures at the bottom of the previous page show what happens when one applies run-length encoding using representation (e) from page 560 to various sequences of data. In the first two cases, there are sufficiently many long runs that compression is achieved. But in the last two cases, there are too many short runs, and the output from run-length encoding is actually longer than the input.

The pictures below show the results of applying run-length encoding to typical patterns produced by cellular automata. When the patterns contain enough regions of uniform color, compression is achieved. But when the patterns are more intricate—even in a simple repetitive way—little or no compression is achieved.


Examples of applying run-length encoding to patterns produced by cellular automata. Successive rows in each original image are placed end to end so as to give a one-dimensional sequence, then run-length encoded, and then chopped into rows again. Compression is typically achieved whenever most of the image consists of regions of uniform color.

Run-length encoding is based on the idea of breaking data up into runs of identical elements of varying lengths. Another common approach to data compression is based on forming blocks of fixed length, and then representing whatever distinct blocks occur by specific codewords.

The pictures below show a few examples of how this works. In each case the input is taken to be broken into blocks of length 3. In the first two cases, there are then only two distinct blocks that occur, so each of these can be represented by a codeword consisting of a single cell, with the result that substantial compression is achieved.

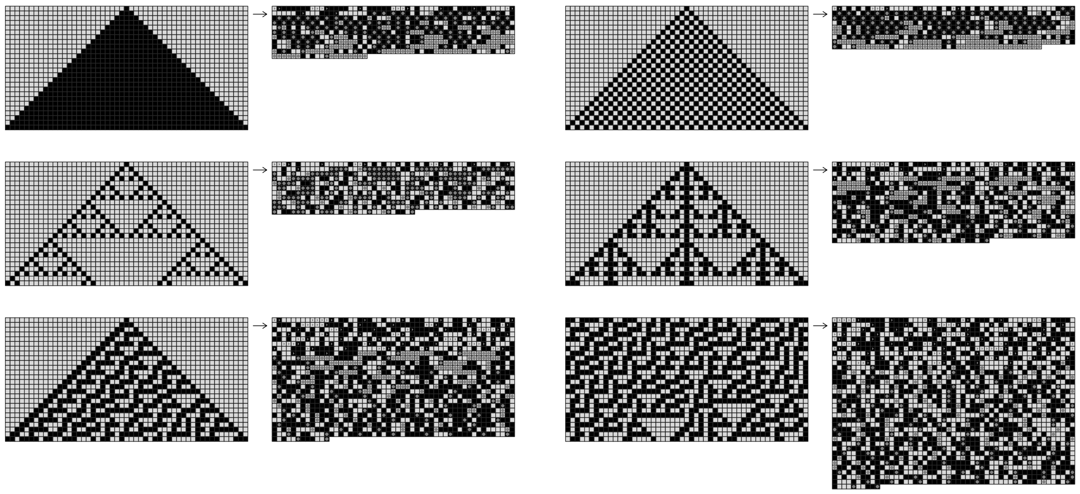


Examples of Huffman coding based on blocks of length 3. In cases (a) and (b), only two possible blocks occur, and these are assigned codewords consisting of a single black cell and a single white cell. In case (c), 3 possible blocks occur; the most common is assigned a codeword consisting of a single white cell, while the others are assigned codewords consisting of two cells. In case (d) 4 out of the 8 possible blocks occur, while in case (e) 6 occur. In all cases, the output begins with a preamble specifying which block is to be represented by which codeword. The blocks appear explicitly in this preamble, and are indicated by numbered tabs. The codewords are represented implicitly by the arrangement of cells shown with arrows. The preamble is followed by the actual codewords representing the data. The codewords are self-delimiting, so that they can be given one after another, with no separator in between.

When a larger number of distinct blocks occur, longer codewords must inevitably be used. But compression can still be achieved if the codewords for common blocks are sufficiently much shorter than the blocks themselves.

One simple strategy for assigning codewords is to number all distinct blocks in order of decreasing frequency, and then just to use the resulting numbers—given, say, in one of the representations discussed above—as the codewords. But if one takes into account the actual frequencies of different blocks, as well as their ranking, then it turns out that there are better ways to assign codewords.

The pictures below show examples based on a method known as Huffman coding. In each case the first part of the output specifies which blocks are to be represented by which codewords, and then the remainder of the output gives the actual succession of codewords that correspond to the blocks appearing in the data. And as the pictures below illustrate, whenever there are fairly few distinct blocks that occur with high frequency, substantial compression is achieved.

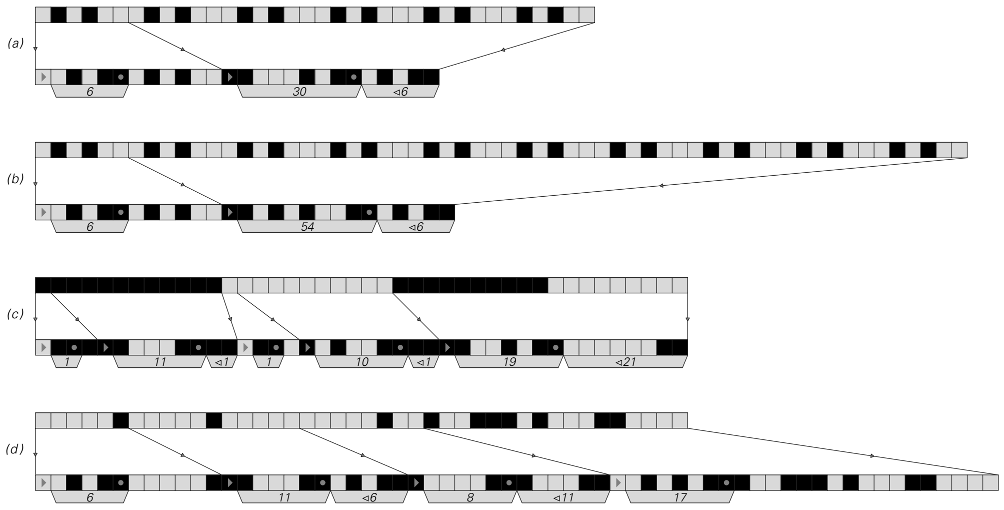


Huffman encoding with blocks of length 6 applied to patterns produced by cellular automata. The maximum possible compression is by a factor of 6; the maximum achieved here is roughly a factor of 3. The difference between the size of the results for the last two examples is mostly a consequence of the presence of large areas of white in the first of them.

But ultimately there is a limit to the degree of compression that can be obtained with this method. For even in the very best case any block of cells in the input can never be compressed to less than one cell in the output.

So how can one achieve greater compression? One approach—which turns out to be similar to what is used in practice in most current high-performance general-purpose compression systems—is to set up an encoding in which any particular sequence of elements above some length is given explicitly only once, and all subsequent occurrences of the same sequence are specified by pointers back to the first one.

The pictures below show what happens when this approach is applied to a few short sequences. In each case, the output consists of two kinds of objects, one giving sequences that are occurring for the first time, and the other giving pointers to sequences that have occurred before. Both kinds of objects start with a single cell that specifies their type. This is

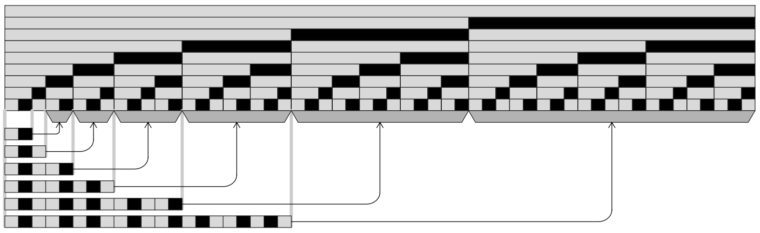


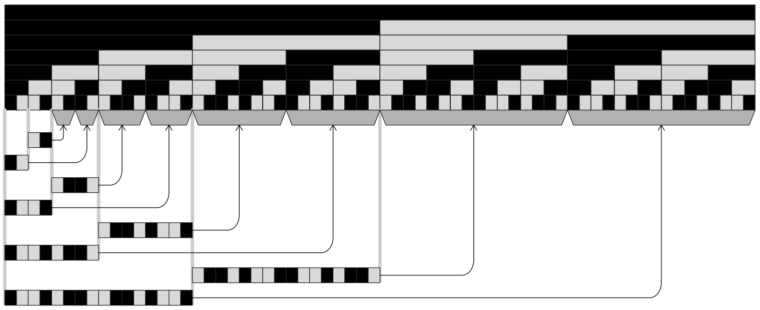
Examples of pointer-based encoding, in which sequences that have occurred once in the data are subsequently specified just by pointers. Each section of output starts with an element which indicates whether what follows is a new sequence, or a pointer to a previous one. After this comes a specification of the length of sequence represented by this section of output, with the number given in the form used for run-length encoding above. Then comes either a literal sequence, or a number giving the offset to where the required sequence last occurred in the data. In the examples shown, pointers are used only for sequences of length at least 6. Pointer-based encoding is similar to the Lempel-Ziv algorithm widely used in practical high-performance general-purpose compression systems.

followed by a specification of the length of the sequence that the object describes. In the first kind of object, the actual sequence is then given, while in the second kind of object what is given is a specification of how far back in the data the required sequence can be found.

With data that is purely repetitive this method achieves quite dramatic compression. For having once specified the basic sequence to be repeated, all that then needs to be given is a pointer to this sequence, together with a representation of the total length of the data.

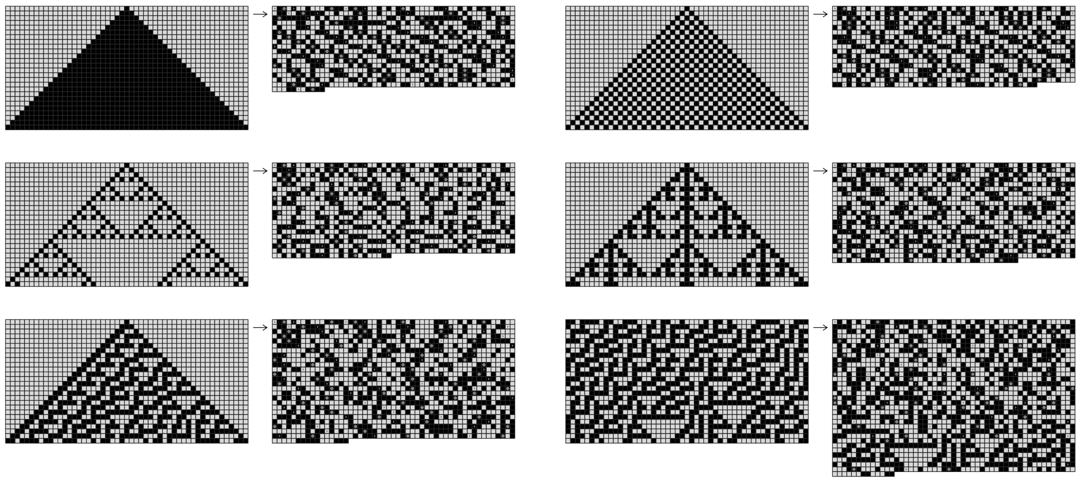
Purely nested data can also be compressed nearly as much. For as the pictures below illustrate, each whole level of nesting can be viewed just as adding a fixed number of repeated sequences.





Examples of the pattern of repeats found in purely nested data. As indicated in these pictures, any such data must correspond to the output of a neighbor-independent substitution system (see page 83). In pointer-based encoding, the number of pointers required to represent the data increases essentially like the number of steps in the evolution of the substitution system. Taking into account the length of the representation for each pointer, the compressed form of a nested sequence of length *n* will typically grow in length like *Log[n]*^{2}. (This can be compared with *Log[n]* growth for a purely repetitive sequence.) Note that actual algorithms for pointer-based encoding will typically find a slightly less regular pattern of repeats than is shown in the pictures here.

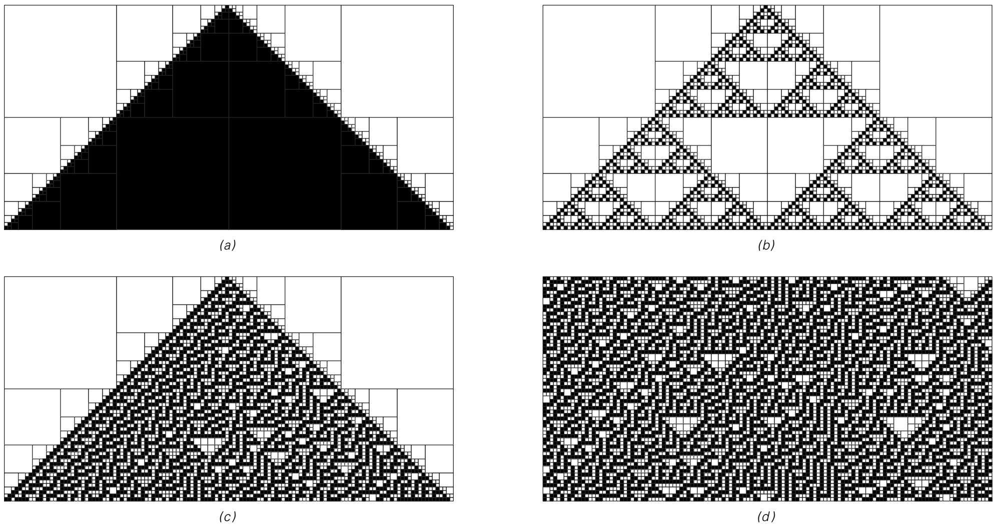
So what about two-dimensional patterns? The pictures below show what happens if one takes various patterns, arranges their rows one after another in a long line, and then applies pointer-based encoding to the resulting sequences. When there are obvious regularities in the original pattern, some compression is normally achieved—but in most cases the amount is not spectacular.

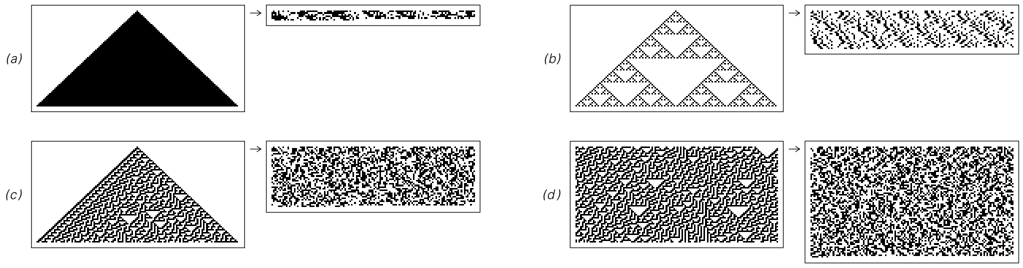

Examples of one-dimensional pointer-based encoding applied to patterns produced by cellular automata. Successive rows in each image are placed end to end so as to get a sequence to which the encoding can be applied. Pointers are used only for repeats that are of length at least 4. In the last example, large regions contain no such repeats, and therefore appear in the output just as they do in the input.

So how can one do better? The basic answer is that one needs to take account of the two-dimensional nature of the patterns. Most compression schemes used in practice have in the past primarily been set up just to handle one-dimensional sequences. But it is not difficult to set up schemes that operate directly on two-dimensional data.

The picture on the next page shows one approach based on the idea of breaking images up into collections of nested pieces, each with a uniform color. In some respects this scheme is a two-dimensional analog of run-length encoding, and when there are large regions of uniform color it yields significant compression.

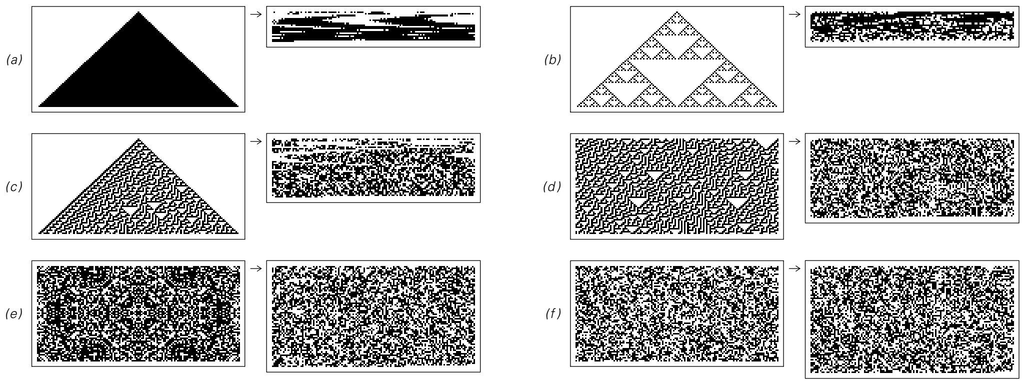
It is also easy to extend block-based encoding to two dimensions: all one need do is to assign codewords to two-dimensional rather than





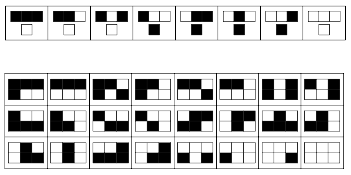
Examples of encoding by two-dimensional recursive subdivision. The idea is to use a generalization of a two-dimensional substitution system, in which at each step a square either remains the same or is subdivided into four small squares. The encoding specifies which choice is made at each step for each square. The method is analogous to the quadtree representation sometimes used in computer graphics. The substantial compression seen even in case (c) is a consequence of the large areas of uniform white that are present.

one-dimensional blocks. And as the pictures at the top of the facing page demonstrate, this procedure can lead to substantial compression. Particularly notable is what happens in case (d). For even though this pattern is produced by a simple one-dimensional cellular automaton rule, and even though one can see by eye that it contains at least some small-scale regularities, none of the schemes we have discussed up till now have succeeded in compressing it at all.



Examples of two-dimensional block-based encoding. Each image is broken into 3 × 2 blocks, and codewords are then assigned to these blocks using the Huffman scheme. Note the presence of compression even in case (d). This is a consequence of the fact that the cellular automaton rule allows only certain blocks to appear in the pattern, as illustrated in the picture below. (e) is generated by a two-dimensional cellular automaton; (f) is the sequence that appears on the center column of rule 30.

The picture below demonstrates why two-dimensional block encoding does, however, manage to compress it. The point is that the two-dimensional blocks that one forms always contain cells whose colors are connected by the cellular automaton rule—and this greatly reduces the number of different arrangements of colors that can occur.



Cellular automaton rule 30, and the 3 × 2 blocks which appear in large patterns generated by it. There are a total of $2^6 = 64$ possible 3 × 2 blocks of black and white cells; the fact that only 24 of them appear in patterns generated by rule 30 is what makes it possible for two-dimensional block-based encoding to compress such patterns.

In cases (e) and (f), however, there is no simple rule for going from one row to the next, and two-dimensional block encoding—like all the other encoding schemes we have discussed so far—does not yield any substantial compression.

Like block encoding, pointer-based encoding can also be extended to two dimensions. The basic idea is just to scan two-dimensional data looking for repeats not of one-dimensional sequences, but instead of two-dimensional regions. And although such a procedure does not in the past appear to have been used in practice, it is quite straightforward to implement. The pictures on the facing page show some examples of the results one gets. And in many cases it turns out that the overall level of compression obtained is considerably greater than with any of the other schemes discussed in this section. But what is perhaps still more striking is that the patterns of repeated regions seem to capture almost every regularity that we readily notice by eye—as well as some that we do not. In pictures (c) and (d), for example, fairly subtle repetition on the left-hand side is captured, as is fourfold symmetry in picture (e).

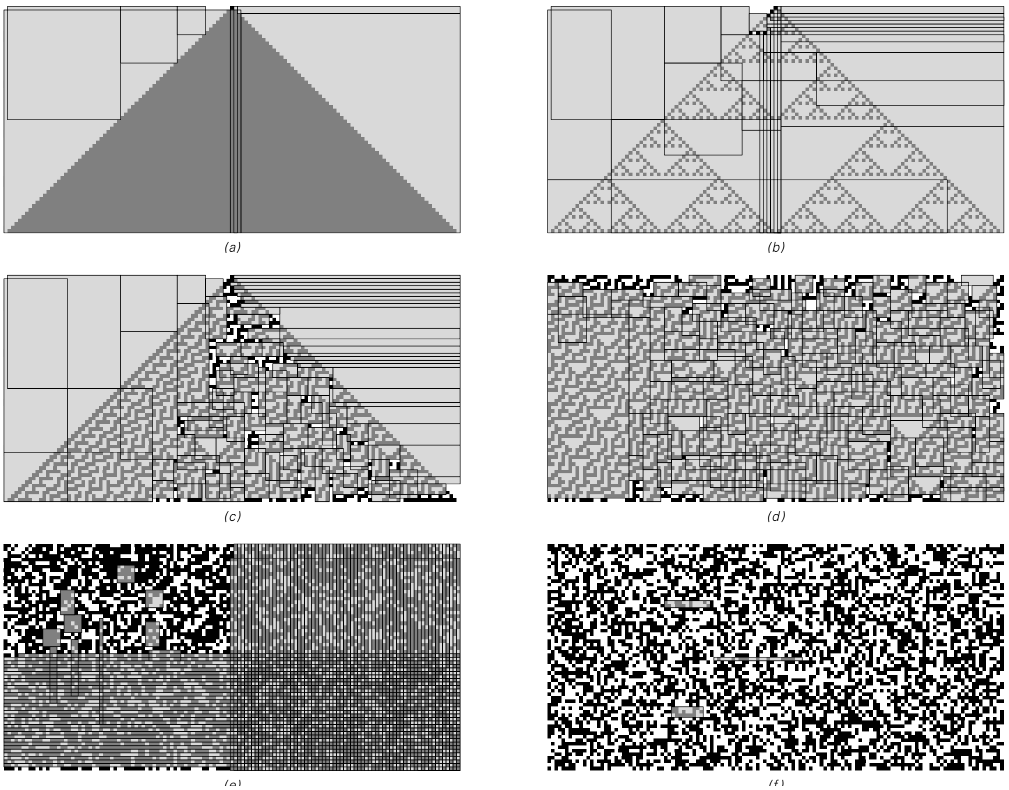
One might have thought that to capture all these kinds of regularities would require a whole collection of complicated procedures. But what the pictures on the facing page demonstrate is that in fact just a single rather straightforward procedure is quite sufficient. And indeed the amount of compression achieved by this procedure in different cases seems to agree rather well with our intuitive impression of how much regularity is present.

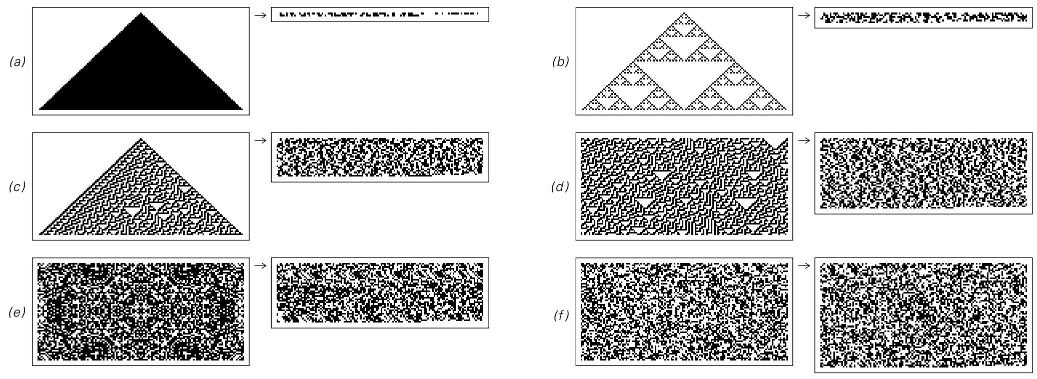
All of the methods of data compression that we have discussed in this section can be thought of as corresponding to fairly simple programs. But each method involves a program with a rather different structure, and so one might think that it would inevitably be sensitive to rather different kinds of regularities.

But what we have seen in this section is that in fact different methods of data compression have remarkably similar characteristics. Essentially every method, for example, will successfully compress large regions of uniform color. And most methods manage to compress behavior that is repetitive, and at least to some extent behavior that is nested—exactly the two kinds of simple behavior that we have noted many times in this book.

For more complicated behavior, however, none of the methods seem capable of substantial compression. It is not that no compression is ever in principle possible. Indeed, as it happens, every single one of the pictures on the facing page can for example be generated from very short cellular automaton programs.

But the point is that except when the overall behavior shows repetition or nesting none of the standard methods of data compression





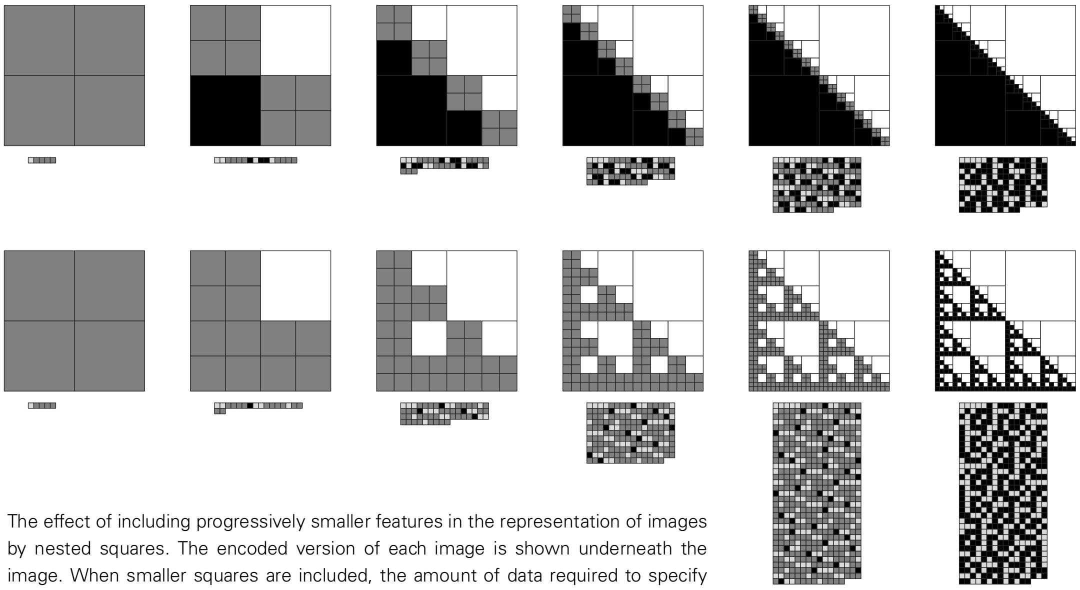
Examples of two-dimensional pointer-based encoding. The gray rectangles in the upper pictures indicate repeated regions that are encoded using pointers. In the particular scheme used here, each of these regions is required to contain at least 25 cells that have not already been encoded using pointers. The images are scanned sequentially and at every point the maximal rectangle extending to the right and down is found that is a repeat of a rectangle previously encountered, and contains the largest number of cells not already encoded using pointers. In many cases this maximal rectangle overlaps those found at subsequent points.

as we have discussed them in this section come even close to finding such short descriptions. And as a result, at least with respect to any of these methods all we can reasonably say is that the behavior we see seems for practical purposes random.

Irreversible Data Compression

All the methods of data compression that we discussed in the previous section are set up to be reversible, in the sense that from the encoded version of any piece of data it is always possible to recover every detail of the original. And if one is dealing with data that corresponds to text or programs such reversibility is typically essential. But with images or sounds it is typically no longer so necessary: for in such cases all that in the end usually matters is that one be able to recover something that looks or sounds right. And by being able to drop details that have little or no perceptible effect one can often achieve much higher levels of compression.

In the case of images a simple approach is just to ignore features that are smaller than some minimum size. The pictures below show


The effect of including progressively smaller features in the representation of images by nested squares. The encoded version of each image is shown underneath the image. When smaller squares are included, the amount of data required to specify the image increases rapidly.