Chapter 6

Saving Space

As computers get ever faster, we ask ever more of them: a higher-resolution film streamed in real time, a faster download, or the same experience on a mobile device over a slow connection as we have at home or in the office over a fast one. When we talk of efficiency, we are concerned with the time taken to do a task, the space required to store data, and knock-on effects such as how often we have to charge our device's battery. And so we cannot simply say "things are getting faster all the time: we need not worry about efficiency."

An important tool for reducing the space information takes up (and so, increasing the speed with which it can be moved around) is *compression*. The idea is to process the information in such as way that it becomes smaller, but also so that it may be *decompressed* – that is to say, the process must be reversible.

Imagine we want to send a coffee order. Instead of writing "Four espressos, two double espressos, a cappuccino, and two lattes", we might write "4E2DC2L". This relies, of course, on the person to whom we are sending the order knowing how to decompress it. The instructions for decompressing might be longer than the message itself, but if we are sending similar messages each day, we need only share the instructions once. We have reduced the message from 67 characters to 7, making it almost ten times smaller.

This sort of compression happens routinely, and it is really just a matter of choosing a better representation for storing a particular kind of information. It tends to be more successful the more uniform the data is. Can we come up with a compression method which works for any data? If not, what about one which works well for a whole class of data, such as text in the English language, or photographs, or video?

First, we should address the question of whether or not this kind of universal compression is even possible. Imagine that our message is just one character long, and our alphabet (our set of possible characters) is the familiar `A`,`B`,`C`... `Z`. There are then exactly 26 different possible messages, each consisting of a single character. Assuming each message is equally likely, there is no way to reduce the length of messages, and so compress them. In fact, this is not entirely true: we can make a tiny improvement – we could send the empty message for, say, `A`, and then one out of twenty-six messages would be smaller. What about a message of length two? Again, if all messages are equally likely, we can do no better: if we were to encode some of the two-letter sequences using just one letter, we would have to use two-letter sequences to indicate the one-letter ones – we would have gained nothing. The same argument applies for sequences of length three or four or five or indeed of any length.

However, all is not lost. Most information has patterns in it, or elements which are more or less common. For example, most of the words in this book can be found in an English dictionary. When there are patterns, we can reserve our shorter codes for the most common sequences, reducing the overall length of the message. It is not immediately apparent how to go about this, so we shall proceed by example. Consider the following text:

> Whether it was embarrassment or impatience, the
> judge rocked backwards and forwards on his seat.
> The man behind him, whom he had been talking
> with earlier, leant forward again, either to give him
> a few general words of encouragement or some
> specific piece of advice. Below them in the hall the
> people talked to each other quietly but animatedly.
> The two factions had earlier seemed to hold views
> strongly opposed to each other but now they began to
> intermingle, a few individuals pointed up at K., others
> pointed at the judge. The air in the room was fuggy
> and extremely oppressive, those who were standing
> furthest away could hardly even be seen through it.
> It must have been especially troublesome for those
> visitors who were in the gallery, as they were forced
> to quietly ask the participants in the assembly what
> exactly was happening, albeit with timid glances at

the judge. The replies they received were just as quiet, and given behind the protection of a raised hand.

We shall take as our dictionary the 100 most commonly-used English words of three or more letters:

00	the	25	there	50	two	75	part
01	and	26	use	51	more	76	over
02	you	27	each	52	write	77	new
03	that	28	which	53	see	78	sound
04	was	29	she	54	number	79	take
05	for	30	how	55	way	80	only
06	are	31	their	56	could	81	little
07	with	32	will	57	people	82	work
08	his	33	other	58	than	83	know
09	they	34	about	59	first	84	place
10	this	35	out	60	water	85	year
11	have	36	many	61	been	86	live
12	from	37	then	62	call	87	back
13	one	38	them	63	who	88	give
14	had	39	these	64	its	89	most
15	word	40	some	65	now	90	very
16	but	41	her	66	find	91	after
17	not	42	would	67	long	92	thing
18	what	43	make	68	down	93	our
19	all	44	like	69	day	94	just
20	were	45	him	70	did	95	name
21	when	46	into	71	get	96	good
22	your	47	time	72	come	97	sentence
23	can	48	has	73	made	98	man
24	said	49	look	74	may	99	think

These words will be compressed by representing them as the two-character sequences 00, 01, 02, . . . , 97, 98, 99. We don't bother with the one and two letter words, common though they are, because they are already as short or shorter than our codes. We assume our text does not contain digits, so that any digit sequence may be interpreted as a code. Any word, text, or punctuation not in the word list will be rendered literally. If we substitute these codes into our text, we find a somewhat underwhelming level of compression:

> Whether it 04 embarrassment or impatience, 00 judge rocked backwards 01 forwards on 08 seat. The 98 behind 45, whom he 14 61 talking 07 earlier, leant forward again, either to 88 45 a few general 15s of encouragement or 40 specific piece of advice. Below 38 in 00 hall 00 people talked to 27 33 quietly 16 animatedly. The 50 factions 14 earlier seemed to views strongly opposed to 27 33 16 65 09 began to intermingle, a few individuals pointed up to K., 33s pointed at 00 judge. The air in 00 room 04 fuggy 01 extremely oppressive, those 63 20 standing furthest away could hardly ever be 53n through it. It must 11 61 especially troublesome 05 those visitors 63 20 in 00 gallery, as 09 20 forced to quietly ask 00 participants in 00 assembly 18 exactly 04 happening, albeit 07 timid glances at 00 judge. The replies 09 received 20 94 as quiet, 01 given behind 00 protection of a raised hand.

The original text had 975 characters; the new one has 891. One more small change can be made – where there is a sequence of codes, we can squash them together if they have only spaces between them in the source:

> Whether it 04 embarrassment or impatience, 00 judge rocked backwards 01 forwards on 08 seat. The 98 behind 45, whom he 1461 talking 07 earlier, leant forward again, either to 8845 a few general 15s of encouragement or 40 specific piece of advice. Below 38 in 00 hall 00 people talked to 2733 quietly 16 animatedly. The 50 factions 14 earlier seemed to views strongly opposed to 2733166509 began to intermingle, a few individuals pointed up to K., 33s pointed at 00 judge. The air in 00 room 04 fuggy 01 extremely oppressive, those 6320 standing furthest away could hardly ever be 53n through it. It must 11 61 especially troublesome 05 those visitors 6320 in 00 gallery, as 0920 forced to quietly ask 00 participants in 00 assembly 18 exactly 04 happening, albeit 07 timid glances at 00 judge. The replies 09 received 2094 as quiet, 01 given behind 00 protection of a raised hand.

We are down to 880 characters, a reduction of about 10% compared with the original. The top 100 words in English are known to cover about half of the printed words, in general. We have not quite achieved that in this example.

Let us try counting the number of each character in our text to see if we can take advantage of the fact that some letters are more common than others (our current method makes no use of the fact that, for example, spaces are very common):

167	*space*	30	l	10	,
120	e	24	w	8	.
71	t	19	p	5	k
62	a	19	m	4	j
55	i	19	g	4	T
51	h	19	c	3	q
49	o	18	u	2	x
45	r	15	y	1	W
42	n	13	f	1	K
41	s	13	b	1	I
33	d	10	v	1	B

The space character is by far the most common (we say it has the highest frequency). The frequencies of the lower case letters are roughly what we might expect from recalling the value of Scrabble tiles, the punctuation characters are infrequent, and the capital letters very infrequent.

We have talked about what a bit is, how 8 bits make a byte, and how one byte is sufficient to store a character (at least in English). Our original message is 975 bytes, or $975 \times 8 = 7800$ bits. We could encode each of the 33 characters we have found in our text using a different pattern of 6 bits, since 33 is less than 64, which is the number of 6-bit combinations 000000,000001,...,111110,111111. (The number of 5-bit combinations is 32, which is not quite enough.) This would reduce our space to $975 \times 6 = 5850$ bits. However, we would have wasted much of the possible set of codes and taken no advantage of our knowledge of how frequently each character occurs. What we should like is a code which uses shorter bit patterns for more common characters, and longer bit patterns for less common ones. Let us write out the beginnings of such a code:

space	0
e	1

t	00
a	01
i	10
h	11
o	000
⋮	⋮

There is a problem, though. It is very easy to encode a word; for example, "heat" encodes as 1110100 (that is, 11 for "h", 1 for "e", 01 for "a", and 00 for "t"). However, we can decode it in many different ways. The sequence 1110100 might equally be taken to mean "ee*space*e*space*" or "hii*space*". Our code is ambiguous. What we require is a code with the so-called *prefix property* – that is, arranged such that no code in the table is a prefix of another. For example, we cannot have both 001 and 0010 as codes, since 001 appears at the beginning of 0010. This property allows for unambiguous decoding. Consider the following alternative code:

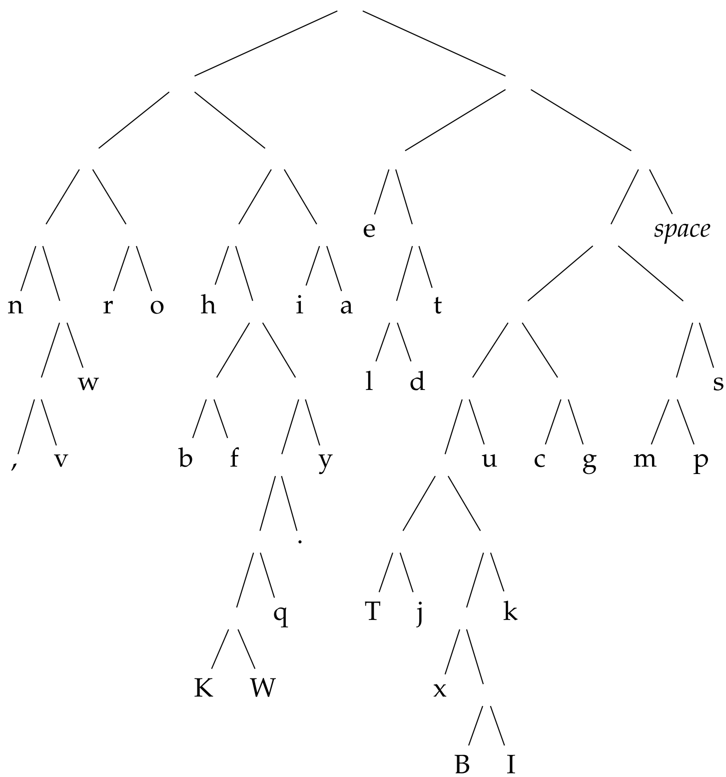
space	00
e	010
t	011
a	100
i	101
h	110
o	111
⋮	⋮

This code is unambiguous – no code is a prefix of another. The word "heat" encodes as 110010100011 and may be decoded unambiguously. We can have the computer automatically create an appropriate code for our text, taking into account the frequencies. Then, by sending the code table along with the text, we ensure it may be unambiguously decoded. Here is the full table of unambiguous codes for the frequencies derived from our text:

space	111	l	10100	,	000100
e	100	w	00011	.	0101101
t	1011	p	110101	k	11000011
a	0111	m	110100	j	11000001
i	0110	g	110011	T	11000000

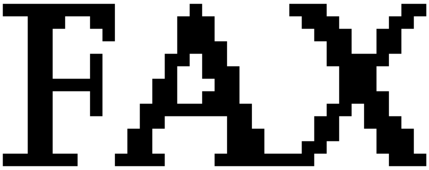
h	0100	c	110010	q	01011001
o	0011	u	110001	x	110000100
r	0010	y	010111	W	010110001
n	0000	f	010101	K	010110000
s	11011	b	010100	I	1100001011
d	10101	v	000101	B	1100001010

The information in this table can, alternatively, be viewed as a diagram:

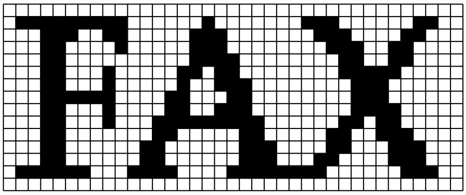


In order to find the code for a letter, we start at the top, adding 0 each time we go left and 1 each time we go right. For example, we can see that the code for the letter "g" is Right Right Left Left Right Right or 110011. You can see that all the letters are at the bottom edge of the diagram, a visual reinforcement of the prefix property. The compressed message length for our example text is 4171 bits, or 522 bytes, about half of the original message length. Sending the tree requires another 197 bits, or 25 bytes. (We do not discuss the method here.) Of course, the longer the message, the less it matters, since the message will be so big by comparison. These codes are called *Huffman codes*, named after David A. Huffman, who invented them whilst a PhD student at MIT in the 1950s.

A common use for this sort of encoding is in the sending of faxes. A fax consists of a high-resolution black and white image. In this case, we are not compressing characters, but the black and white image of those characters itself. Take the following fragment:



This image is 37 pixels wide and 15 tall. Here it is with a grid superimposed to make it easier to count pixels:



We cannot compress the whole thing with Huffman encoding, since we do not know the frequencies at the outset – a fax is sent incrementally. One machine scans the document whilst the machine at the other end of the phone line prints the result as it pulls paper from its roll. It had to be this way because, when fax machines were in their infancy, computer memory was very expensive, so receiving and storing the whole image in one go and only then printing it out was not practical.

The solution the fax system uses is as follows. Instead of sending individual pixels, we send, a line at a time, a list of *runs*. Each run is a length of white pixels or a length of black pixels. For example, a line of width 39 might contain 12 pixels of white, then 4 of black, then 2 of white, then 18 of black, and then 3 of white. We look up the code for each run and send the codes in order. To avoid the problem of having to gather frequency data for the whole page, a pre-prepared master code table is used, upon which everyone agrees. The table has been built by gathering frequencies from thousands of text documents in several languages and typefaces, and then collating the frequencies of the various black and white runs.

Here is the table of codes for black and white runs of lengths 0 to 63. (We need length 0 because a line is always assumed to begin white, and a zero-length white run is required if the line actually begins black.)

Run	White	Black	Run	White	Black
0	00110101	0000110111	32	00011011	000001101010
1	0000111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	1011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	0000101	00001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	00000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011

28	0011000	000011001100	60	00110010	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

Notice that the prefix property applies only to alternating black and white codes. There is never a black code followed by a black code or a white code followed by a white code. The shortest codes are reserved for the most common runs – the black ones of length two and three. We can write out the codes for the first two lines of our image by counting the pixels manually:

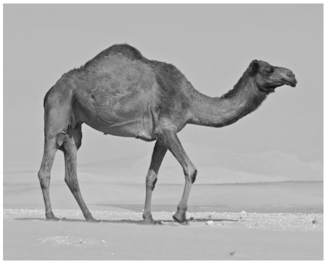
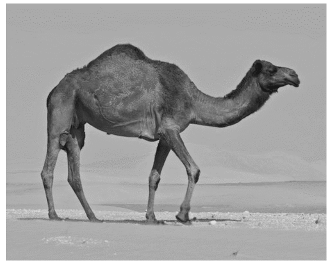
Run length	Colour	Bit pattern	Pattern length
37	white	00010110	8
1	white	0000111	7
9	black	000100	6
6	white	1110	4
1	black	010	3
7	white	1111	4
3	black	10	2
6	white	1110	4
2	white	0111	4

So we transmit the bit pattern 00010110 0000111 000100 1110 010 1111 10 1110 0111. The number of bits required to transmit the image has dropped from $37 \times 2 = 74$ to $8 + 7 + 6 + 4 + 3 + 4 + 2 + 4 + 2 + 4 = 46$. Due to the preponderance of white space in written text (blank lines, spaces between words, and page margins), faxes can often be compressed to less than twenty per cent of their original size. Modern fax systems which take advantage of the fact that successive lines are often similar can reduce this to five per cent.

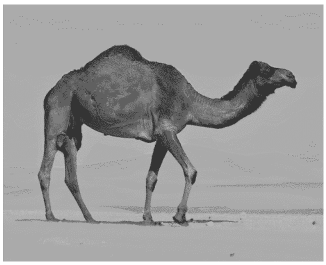
Of course, we often want more than just black and white. (Even black and white television was not really just black and white – there were shades of grey.) How can we compress grey and colour photographic images? The reversible (lossless) compression we have used so far tends not to work well, so we look at methods which do not retain all the information in an image. This is known as lossy compression. One option is simply to use fewer colours. Figure A on page 76 shows a picture reduced from the original to 64, then 8, then 2 greys. We see a marked decrease in size, but the quality reduces rapidly. On the printed page, we can certainly see that 8 and 2 greys are too few, but 64 seems alright. On a computer screen, you would see that even 64 is a noticeable decrease in quality.

If we can't reduce the number of greys with a satisfactory result, what about the resolution? Let us try discarding one out of every two pixels in each row of the original, and one out of every two pixels in each column. Then we will go further and discard three from every four, and finally seven from every eight. The result is Figure B. In these examples, we removed some information and then scaled up the image again when printing it on the page. Again, the first reduction is not too bad – at least at the printed size of this book. The 3/4 is a little obvious, and the 7/8 is dreadful. Algorithms have been devised which can take the images which have had data discarded like those above and, when scaling them back to normal size, attempt to smooth the image. This will reduce the "blocky" look, but it can lead to indistinctness. Figure C shows the same images as Figure B, displayed using a modern smoothing method.

Finally, Figure D shows the images compressed using an algorithm especially intended for photographic use, the JPEG (Joint Photographic Experts Group) algorithm, first conceived in the 1980s. At "75% quality", the image is down to nineteen per cent of its original size and almost indistinguishable from the original.

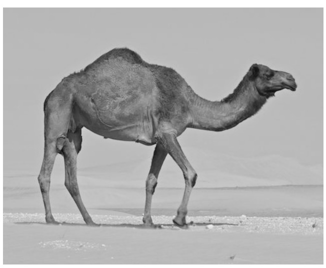
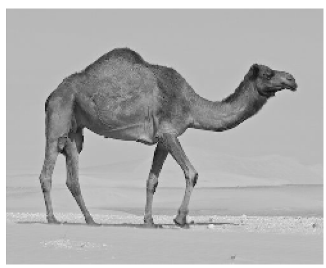



original – 100% | 64 greys – 40%

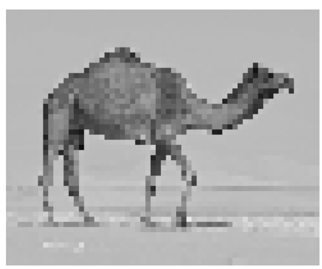



8 greys – 14% | 2 greys – 5%

Figure A

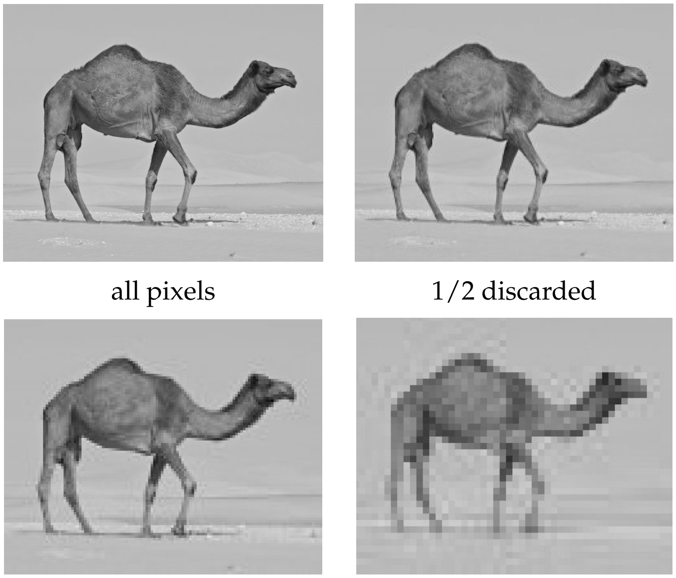



all pixels | 1/2 discarded

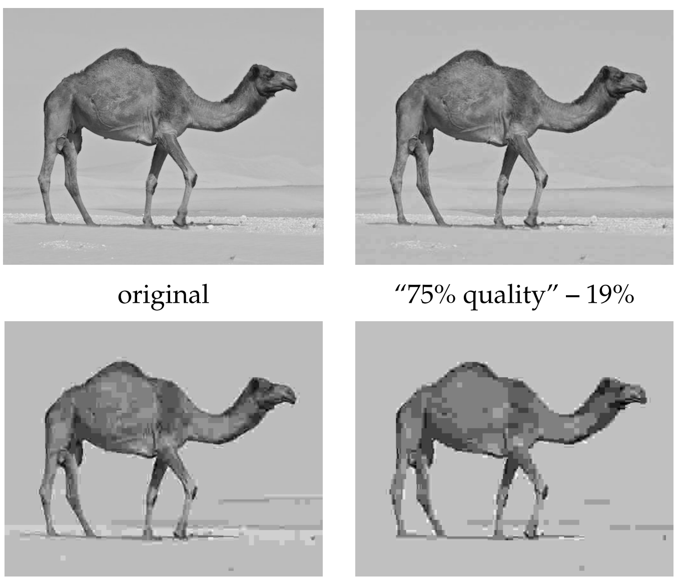
3/4 discarded | 7/8 discarded

Figure B



all pixels | 1/2 discarded | 3/4 discarded | 7/8 discarded

Figure C



original | “75% quality” – 19% | “50% quality” – 11% | “25% quality” – 9%

Figure D

Problems

Solutions on page 154.

1. Count the frequencies of the characters in this piece of text and assign them to the Huffman codes, filling in the following table. Then encode the text up to "more lightly.".

> 'I have a theory which I suspect is rather immoral,' Smiley went on, more lightly. 'Each of us has only a quantum of compassion. That if we lavish our concern on every stray cat, we never get to the centre of things.'

Letter	Frequency	Code	Letter	Frequency	Code
		111			110100
		100			110011
		1011			110010
		0111			110001
		0110			010111
		0100			010101
		0011			01010000
		0010			01010001
		0000			01010010
		11011			01010011
		10101			01011000
		10100			01011001
		00011			01011010
		110101			01011011

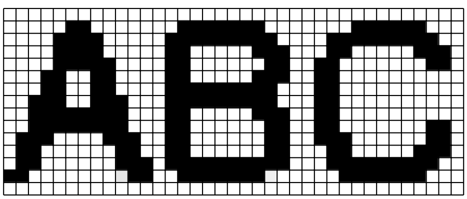
2. Consider the following frequency table and text. Decode it.

Letter	Frequency	Code	Letter	Frequency	Code
space	20	111	s	2	00011
e	12	100	d	2	110101
t	9	1011	T	1	110100
h	7	0111	n	1	110011
o	7	0110	w	1	110010
m	6	0100	p	1	110001
r	5	0011	b	1	010111

a	4	0010	l	1	010101
f	4	0000	v	1	01010000
c	4	11011	y	1	01010001
u	4	10101	.	1	01010010
i	3	10100			

```
1101000111100001110011100100011100111010001100100
10011001101100011111110010011110100110110111111100
10001110011101000010110101100111101011100011110111
00000011101101100110111010010101011101101111111000
11011101010000000001110000011000111110110111100010
01110110110111010111100010101101000101000010011101
01111001010111111101101111001111011101000100100111
10110111100010100011110110110111101110101001101 01
0010
```

3. Encode the following fax image. There is no need to use zero-length white runs at the beginning of lines starting with a black pixel.



4. Decode the following fax image to the same 37x15 grid. There are no zero-length white runs at the beginning of lines starting with a black pixel.

```
00010110000001110001111110001111000000111000000001001
01100001001000000010001111111001010001011001001111
11100100000011111111011011110111111011111111011000
01111111001001111110111101111111100100000011100010 0
10000111011110111000100011100010010001110111101110
00100011111110010011111101111011111111001000001111
11110110111111011110111111111011000011111111011011
11011101001111111101100001111111101101110111100 11
10001111101100001110000100100000000100100000010001
110000111000111111001011100010101100010110
```