Answers to Exercises

A bird does not sing because he has an answer, he sings because he has a song.

—Chinese Proverb

1: abstemious, abstentious, adventitious, annelidous, arsenious, arterious, facetious, sacrilegious.

2: When a software house has a popular product, they tend to come up with new versions. A user can update an old version to a new one, and the update usually comes as a compressed file on a floppy disk. Over time the updates get bigger and, at a certain point, an update may not fit on a single floppy. This is why good compression is important in the case of software updates. The time it takes to compress and decompress the update is unimportant since these operations are typically done just once. Recently, software makers have taken to providing updates over the Internet, but even in such cases it is important to have small files because of the download times involved.

1.1: 6,8,0,1,3,1,4,1,3,1,4,1,3,1,4,1,3,1,2,2,2,2,6,1,1. The first two are the bitmap resolution (6×8). If each number occupies a byte on the output stream, then its size is 25 bytes, compared to a bitmap size of only 6×8 bits $= 6$ bytes. The method does not work for small images.

1.2: Each of the first four rows yields the eight runs 1,1,1,2,1,1,1,eol. Rows 6 and 8 yield the four runs 0,7,1,eol each. Rows 5 and 7 yield the two runs 8,eol each. The total number of runs (including the eol's) is thus 44.

When compressing by columns, columns 1, 3, and 6 yield the five runs 5,1,1,1,eol each. Columns 2, 4, 5, and 7 yield the six runs 0,5,1,1,1,eol each. Column 8 gives 4,4,eol, so the total number of runs is 42. This image is thus "balanced" with respect to rows and columns.

1.3: As "11 22 90 00 00 33 44". The 00 following the 90 indicates no run, and the following 00 is interpreted as a regular character.

1.4: Table Ans.1 summarizes the results. In (a), the first string is encoded with $k = 1$. In (b) it is encoded with $k = 2$. Columns (c) and (d) are the encodings of the second string with $k = 1$ and $k = 2$, respectively. The averages of the four columns are 3.4375, 3.25, 3.56 and 3.6875; very similar! The move-ahead-k method used with small values of k does not favor strings satisfying the concentration property.

a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0
b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1
c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2
d	bcadmnop	3	d	cbadmnop	3	d	bcadmnop	3	d	cbadmnop	3
d	bcdamnop	2	d	cdbamnop	1	m	bcdamnop	4	m	cdbamnop	4
c	bdcamnop	2	c	dcbamnop	1	n	bcdmanop	5	n	cdmbanop	5
b	bcdamnop	0	b	cdbamnop	2	o	bcdmnaop	6	o	cdmnbaop	6
a	bcdamnop	3	a	bcdamnop	3	p	bcdmnoap	7	p	cdmnobap	7
m	bcadmnop	4	m	bacdmnop	4	a	bcdmnopa	7	a	cdmnopba	7
n	bcamdnop	5	n	bamcdnop	5	b	bcdmnoap	0	b	cdmnoapb	7
o	bcamndop	6	o	bamncdop	6	c	bcdmnoap	1	c	cdmnobap	0
p	bcamnodp	7	p	bamnocdp	7	d	cbdmnoap	2	d	cdmnobap	1
p	bcamnopd	6	p	bamnopcd	5	m	cdbmnoap	3	m	dcmnobap	2
o	bcamnpod	6	o	bampnocd	5	n	cdmbnoap	4	n	mdcnobap	3
n	bcamnopd	4	n	bamopncd	5	o	cdmnboap	5	o	mndcobap	4
m	bcanmopd	4	m	bamnopcd	2	p	cdmnobap	7	p	mnodcbap	7
	bcamnopd			mbanopcd			cdmnobpa			mnodcpba	
	(a)			(b)			(c)			(d)	

Table Ans.1: Encoding with Move-Ahead-k.

1.5: Table Ans.2 summarizes the decoding steps. Notice how similar it is to Table 1.16, indicating that move-to-front is a symmetric data compression method.

Code input	A (before adding)	A (after adding)	Word
0the	()	(the)	the
1boy	(the)	(the, boy)	boy
2on	(boy, the)	(boy, the, on)	on
3my	(on, boy, the)	(on, boy, the, my)	my
4right	(my, on, boy, the)	(my, on, boy, the, right)	right
5is	(right, my, on, boy, the)	(right, my, on, boy, the, is)	is
5	(is, right, my, on, boy, the)	(is, right, my, on, boy, the)	the
2	(the, is, right, my, on, boy)	(the, is, right, my, on, boy)	right
5	(right, the, is, my, on, boy)	(right, the, is, my, on, boy)	boy
	(boy, right, the, is, my, on)		

Table Ans.2: Decoding Multiple-Letter Words.

2.1: It is $1 + \lfloor \log_2 i \rfloor$ as can be seen by simple experimenting.

2.2: We assume an alphabet with two symbols a_1 and a_2, with probabilities P_1 and P_2, respectively. Since $P_1 + P_2 = 1$, the entropy of the alphabet is $-P_1 \log_2 P_1 - (1 - P_1) \log_2(1 - P_1)$. Table Ans.3 shows the entropies for certain values of the probabilities. When $P_1 = P_2$, at least 1 bit is required to encode each symbol, reflecting the fact that the entropy is at its maximum, the redundancy is zero, and the data cannot be compressed. However, when the probabilities are very different, the minimum number of bits required per symbol drops significantly. We may not be able to develop a compression method using 0.08 bits per symbol but we know that when $P_1 = 99\%$, this is the theoretical minimum.

P_1	P_2	Entropy
99	1	0.08
90	10	0.47
80	20	0.72
70	30	0.88
60	40	0.97
50	50	1.00

Table Ans.3: Probabilities and Entropies of Two Symbols.

An essential tool of this theory [information] is a quantity for measuring the amount of information conveyed by a message. Suppose a message is encoded into some long number. To quantify the information content of this message, Shannon proposed to count the number of its digits. According to this criterion, 3.14159, for example, conveys twice as much information as 3.14, and six times as much as 3. Struck by the similarity between this recipe and the famous equation on Boltzman's tomb (entropy is the number of digits of probability), Shannon called his formula the "information entropy."

Hans Christian von Baeyer, *Maxwell's Demon*, 1998

2.3: It is easy to see that the unary code satisfies the prefix property, so it definitely can be used as a variable-size code. Since its length L satisfies $L = n$ we get $2^{-L} = 2^{-n}$, so it makes sense to use it in cases were the input data consists of integers n with probabilities $P(n) \approx 2^{-n}$. If the data lends itself to the use of the unary code, the entire Huffman algorithm can be skipped, and the codes of all the symbols can easily and quickly be constructed before compression or decompression starts.

2.4: This is straightforward. Table Ans.4 shows the code. There are only three different codewords since "start" and "stop" are so close, but there are many codes since "start" is large.

n	$a = 10 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	10	$0\underbrace{x...x}_{10}$	$2^{10} = 1K$	0–1023
1	12	$10\underbrace{xx...x}_{12}$	$2^{12} = 4K$	1024–5119
2	14	$11\underbrace{xx...xx}_{14}$	$2^{14} = 16K$	5120–21503
		Total	21504	

Table Ans.4: The General Unary Code (10,2,14).

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.20	1	0			:101
3.	0.15	1	0			:100
4.	0.15	0	1			:01
5.	0.10	0	0	1		:001
6.	0.10	0	0	0	0	:0001
7.	0.05	0	0	0	0	:0000

Table Ans.5: Shannon-Fano Example.

2.5: Subsequent splits can be done in different ways, but Table Ans.5 shows one way of assigning Shannon-Fano codes to the 7 symbols.
The average size in this case is $0.25 \times 2 + 0.20 \times 3 + 0.15 \times 3 + 0.15 \times 2 + 0.10 \times 3 + 0.10 \times 4 + 0.05 \times 4 = 2.75$ bits/symbols.

2.6: Figure Ans.6a,b,c shows the three trees. The codes sizes for the trees are

$$(5 + 5 + 5 + 5\cdot 2 + 3\cdot 3 + 3\cdot 5 + 3\cdot 5 + 12)/30 = 76/30,$$
$$(5 + 5 + 4 + 4\cdot 2 + 4\cdot 3 + 3\cdot 5 + 3\cdot 5 + 12)/30 = 76/30,$$
$$(6 + 6 + 5 + 4\cdot 2 + 3\cdot 3 + 3\cdot 5 + 3\cdot 5 + 12)/30 = 76/30.$$

2.7: The second row of Table Ans.7 (due to Guy Blelloch) shows a symbol whose Huffman code is three bits long, but for which $\lceil -\log_2 0.3\rceil = \lceil 1.737\rceil = 2$.

P_i	Code	$-\log_2 P_i$	$\lceil -\log_2 P_i\rceil$
.01	000	6.644	7
*.30	001	1.737	2
.34	01	1.556	2
.35	1	1.515	2

Table Ans.7: A Huffman Code Example.

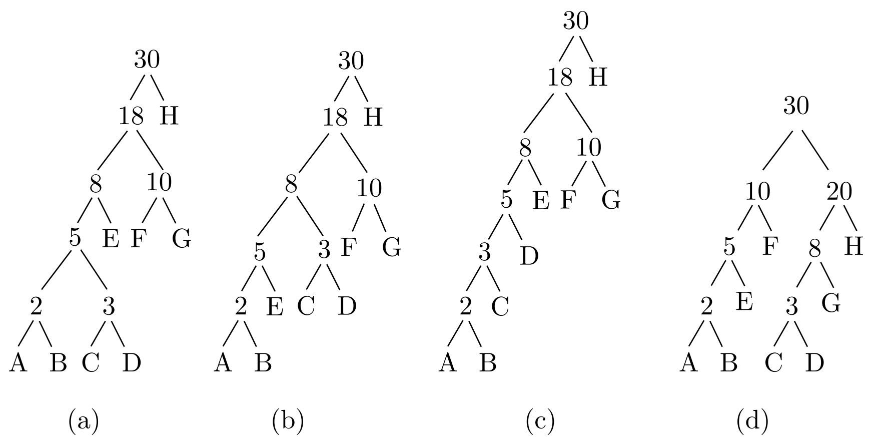


Figure Ans.6: Three Huffman Trees for Eight Symbols.

2.8: Figure Ans.8 shows Huffman codes for 5, 6, 7, and 8 symbols with equal probabilities. In the case where n is a power of 2, the codes are simply the fixed-sized ones. In other cases the codes are very close to fixed-size. This shows that symbols with equal probabilities do not benefit from variable-size codes. (This is another way of saying that random text cannot be compressed.) Table Ans.9 shows the codes, their average sizes and variances.

2.9: The binary value of 127 is 01111111 and that of 128 is 10000000. Half the pixels in each bitplane will therefore be 0 and the other half, 1. In the worst case, each bitplane will be a checkerboard, i.e., will have many runs of size one. In such a case, each run requires a 1-bit code, leading to one codebit per pixel per bitplane, or eight codebits per pixel for the entire image, resulting in no compression at all. In comparison, a Huffman code for such an image requires just two codes (since there are just two pixel values) and they can be one bit each. This leads to one codebit per pixel, or a compression factor of eight.

2.10: Figure Ans.10 shows how the loop continues until the heap shrinks to just one node that is the single pointer 2. This indicates that the total frequency (which happens to be 100 in our example) is stored in `A[2]`. All other frequencies have been replaced by pointers. Figure Ans.11a shows the heaps generated during the loop.

2.11: The code lengths for the seven symbols are 2, 2, 3, 3, 4, 3, and 4 bits. This can also be verified from the Huffman code-tree of Figure Ans.11b. A set of codes derived from this tree is shown in the following table:

Count:	25	20	13	17	9	11	5
Code:	01	11	101	000	0011	100	0010
Length:	2	2	3	3	4	3	4

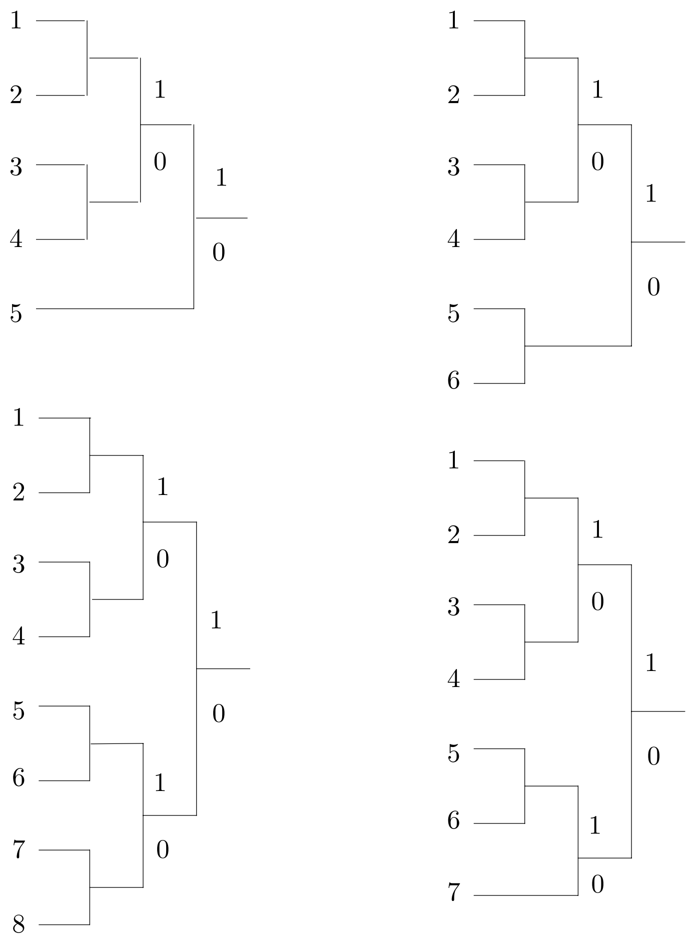


Figure Ans.8: Huffman Codes for Equal Probabilities.

n	p	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	Avg. size	Var.
5	0.200	111	110	101	100	0				2.6	0.64
6	0.167	111	110	101	100	01	00			2.672	0.2227
7	0.143	111	110	101	100	011	010	00		2.86	0.1226
8	0.125	111	110	101	100	011	010	001	000	3	0

Table Ans.9: Huffman Codes for 5–8 Symbols.

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{7}&\mathit{11}&\mathit{6}&\mathit{8}&\mathit{9}]&24&14&25&20&\mathit{6}&17&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{11}&\mathit{9}&\mathit{8}&\mathit{6}]&&24&14&25&20&\mathit{6}&17&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{11}&\mathit{9}&\mathit{8}&\mathit{6}]&17{+}14&24&14&25&20&\mathit{6}&17&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{5}&\mathit{9}&\mathit{8}&\mathit{6}]&31&24&\mathit{5}&25&20&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{9}&\mathit{6}&\mathit{8}&\mathit{5}]&31&24&\mathit{5}&25&20&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{6}&\mathit{8}&\mathit{5}]&&31&24&\mathit{5}&25&20&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{6}&\mathit{8}&\mathit{5}]&20{+}24&31&24&\mathit{5}&25&20&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{4}&\mathit{8}&\mathit{5}]&44&31&\mathit{4}&\mathit{5}&25&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{8}&\mathit{5}&\mathit{4}]&44&31&\mathit{4}&\mathit{5}&25&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{5}&\mathit{4}]&&44&31&\mathit{4}&\mathit{5}&25&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{5}&\mathit{4}]&25{+}31&44&31&\mathit{4}&\mathit{5}&25&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{3}&\mathit{4}]&56&44&\mathit{3}&\mathit{4}&\mathit{5}&\mathit{3}&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

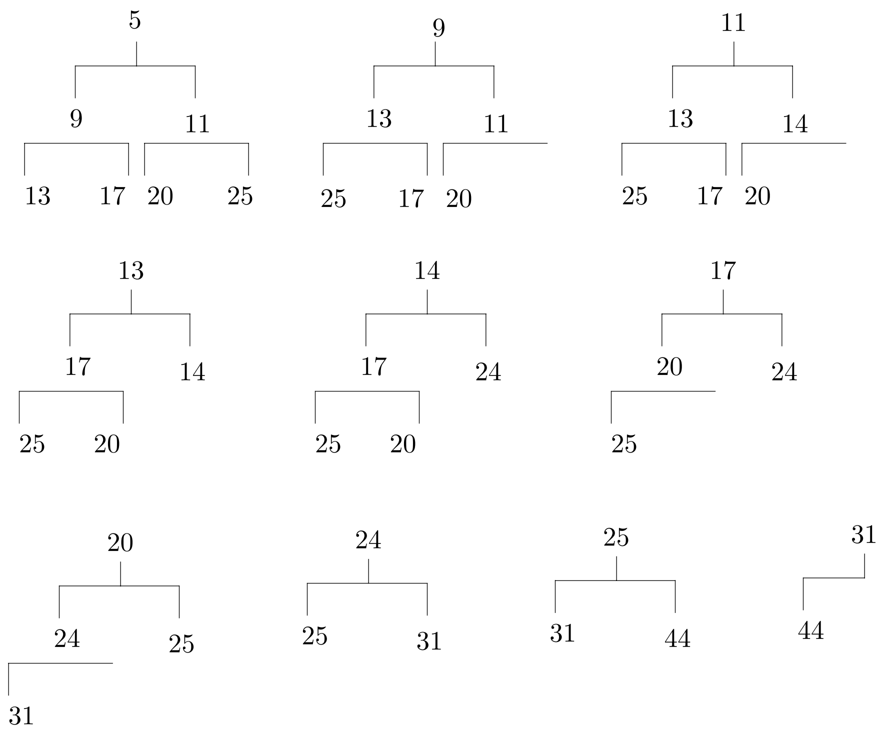
$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{4}&\mathit{3}]&56&44&\mathit{3}&\mathit{4}&\mathit{5}&\mathit{3}&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{3}]&&56&44&\mathit{3}&\mathit{4}&\mathit{5}&\mathit{3}&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

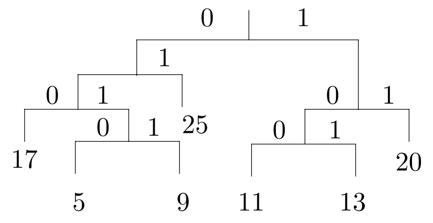
$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{3}]&56{+}44&56&44&\mathit{3}&\mathit{4}&\mathit{5}&\mathit{3}&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

$$\begin{array}{cccccccccccccc}\underline{1}&\underline{2}&\underline{3}&\underline{4}&\underline{5}&\underline{6}&\underline{7}&\underline{8}&\underline{9}&\underline{10}&\underline{11}&\underline{12}&\underline{13}&\underline{14}\\ [\mathit{2}]&100&\mathit{2}&\mathit{2}&\mathit{3}&\mathit{4}&\mathit{5}&\mathit{3}&\mathit{4}&\mathit{6}&\mathit{5}&\mathit{7}&\mathit{6}&\mathit{7}\end{array}$$

Figure Ans.10: Sifting the Heap.



(a)



(b)

Figure Ans.11: (a) Heaps. (b) Huffman Code-Tree.

2.12: Figure Ans.12 shows the initial tree and how it is updated in the 11 steps (a) through (k). Notice how the *esc* symbol gets assigned different codes all the time, and how the different symbols move about in the tree and change their codes. Code 10, e.g., is the code of symbol "i" in steps (f) and (i), but is the code of "s" in steps (e) and (j). The code of a blank space is 011 in step (h), but 00 in step (k).

The final output is: "`s0i00r100␣1010000d011101000`". A total of $5\times 8+22=62$ bits. The compression ratio is thus $62/88\approx 0.7$.

2.13: Each scan line starts with a white pel, so when the decoder inputs the next code it knows whether it is for a run of white or black pels. This is why the codes of Table 2.40 have to satisfy the prefix property in each column but not between the columns.

2.14: Figure Ans.13 shows the modes and the actual code generated from the two lines.

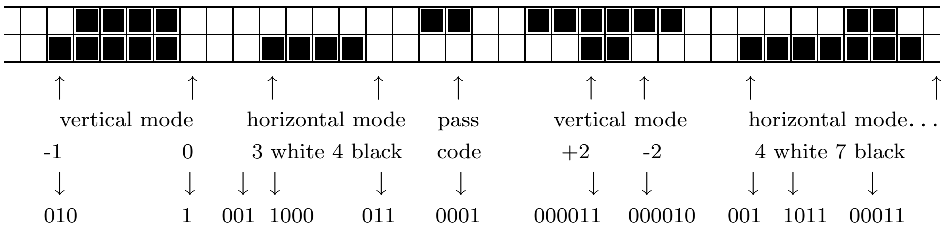


Figure Ans.13: Two-Dimensional Coding Example.

2.15: Table Ans.14 shows the steps of encoding the string $a_2a_2a_2a_2$. Because of the high probability of a_2 the low and high variables start at very different values and approach each other slowly.

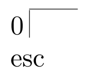
$$\begin{array}{lr}
a_2 & 0.0+(1.0-0.0)\times 0.023162{=}0.023162\\
 & 0.0+(1.0-0.0)\times 0.998162{=}0.998162\\
a_2 & 0.023162+.975\times 0.023162{=}0.04574495\\
 & 0.023162+.975\times 0.998162{=}0.99636995\\
a_2 & 0.04574495+0.950625\times 0.023162{=}0.06776322625\\
 & 0.04574495+0.950625\times 0.998162{=}0.99462270125\\
a_2 & 0.06776322625+0.926859375\times 0.023162{=}0.08923124309375\\
 & 0.06776322625+0.926859375\times 0.998162{=}0.99291913371875
\end{array}$$

Table Ans.14: Encoding the String $a_2a_2a_2a_2$.

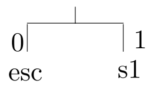
2.16: The encoding steps are simple (see first example on page 109). We start with the interval $[0,1)$. The first symbol a_2 reduces the interval to $[0.4,0.9)$. The second one, to $[0.6,0.85)$, the third one to $[0.7,0.825)$ and the eof symbol, to $[0.8125,0.8250)$. The approximate binary values of the last interval are 0.1101000000 and 0.1101001100, so we select the 7-bit number 1101000 as our code.

The probability of the string "$a_2a_2a_2$eof" is $(0.5)^3\times 0.1=0.0125$, but $-\log_2 0.0125\approx 6.322$, which is why the practical minimum code size is 7 bits.

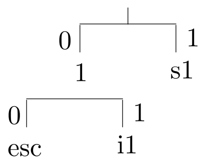
Initial tree



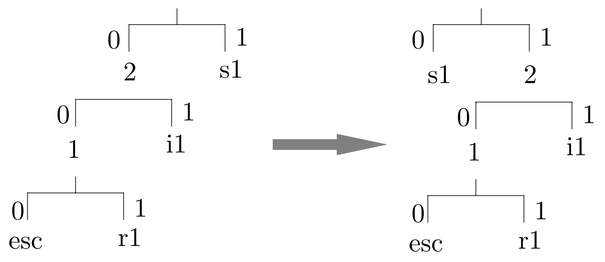
(a). Input: s. Output: ‘s’.
$esc\, s_1$



(b). Input: i. Output: 0‘i’.
$esc\, i_1\; 1\; s_1$



(c). Input: r. Output: 00‘r’.
$esc\, r_1\, 1\, i_1\, 2\, s_1 \rightarrow$
$esc\, r_1\, 1\, i_1\, s_1\, 2$



(d). Input: ␣. Output: 100‘␣’.
$esc_{␣1}\, 1\, r_1\, 2\, i_1\, s_1\, 3 \rightarrow$
$esc_{␣1}\, 1\, r_1\, s_1\, i_1\, 2\, 2$

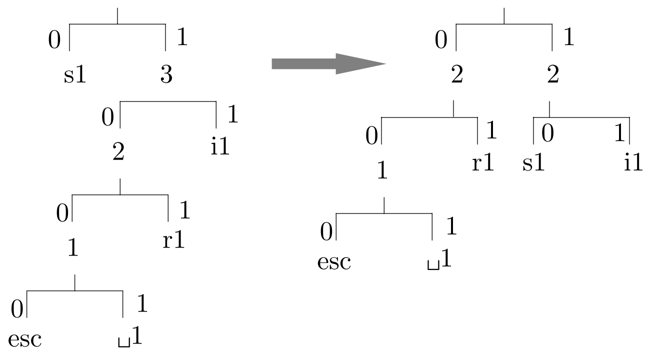
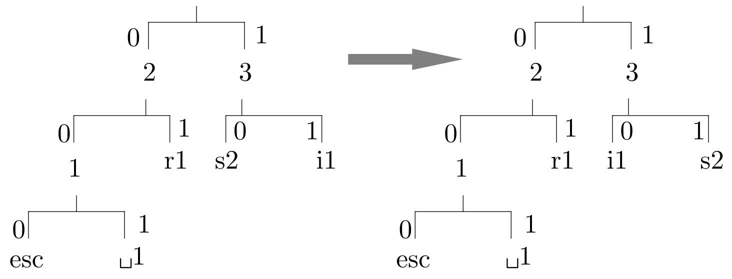
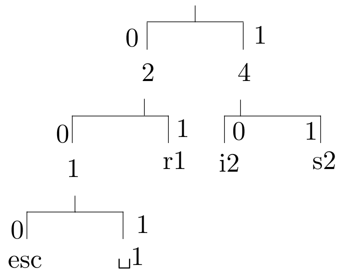


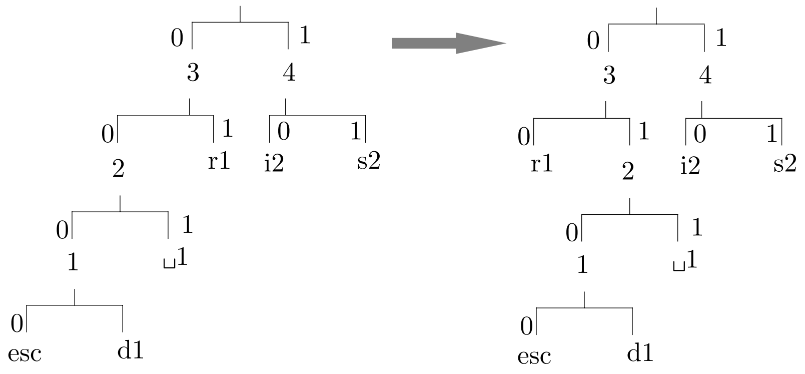
Figure Ans.12: Exercise 2.12. Part I.



(e). Input: s. Output: 10.
$esc\,{}_{\sqcup 1}\,1\,r_1\,s_2\,i_1\,2\,3 \rightarrow$
$esc\,{}_{\sqcup 1}\,1\,r_1\,i_1\,s_2\,2\,3$

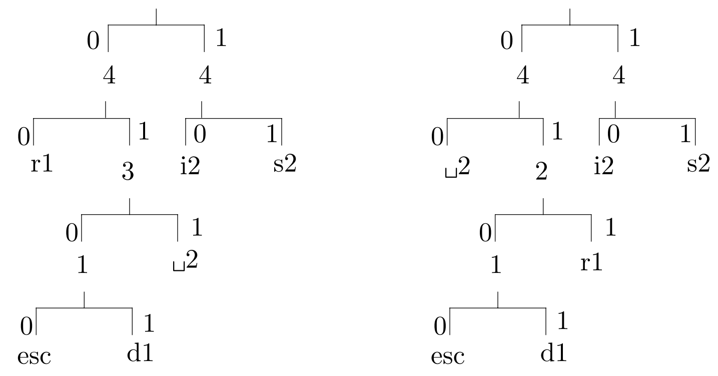


(f). Input: i. Output: 10.
$esc\,{}_{\sqcup 1}\,1\,r_1\,i_2\,s_2\,2\,4$

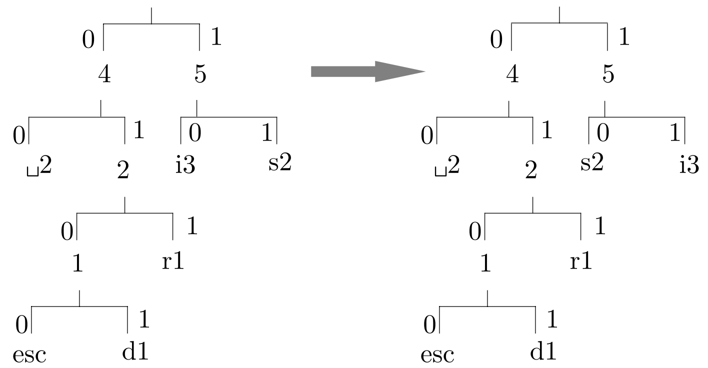


(g). Input: d. Output: 000‘d’.
$esc\,d_1\,1\,{}_{\sqcup 1}\,2\,r_1\,i_2\,s_2\,3\,4 \rightarrow$
$esc\,d_1\,1\,{}_{\sqcup 1}\,r_1\,2\,i_2\,s_2\,3\,4$

Figure Ans.12: Exercise 2.12. Part II.

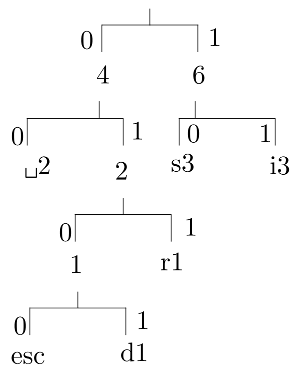


(h). Input: ␣. Output: 011.
$esc\, d_1\, 1\, {}_{␣2}\, r_1\, 3\, i_2\, s_2\, 4\, 4 \rightarrow$
$esc\, d_1\, 1\, r_1\, {}_{␣2}\, 2\, i_2\, s_2\, 4\, 4$

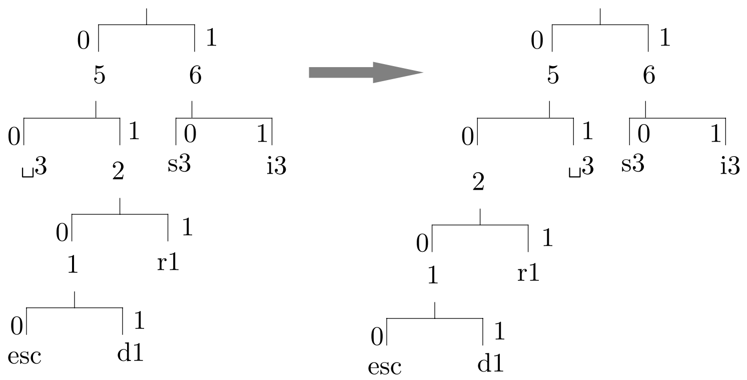


(i). Input: i. Output: 10.
$esc\, d_1\, 1\, r_1\, {}_{␣2}\, 2\, i_3\, s_2\, 4\, 5 \rightarrow$
$esc\, d_1\, 1\, r_1\, {}_{␣2}\, 2\, s_2\, i_3\, 4\, 5$

Figure Ans.12: Exercise 2.12. Part III.



(j). Input: s. Output: 10.
$esc\,d_1\,1\,r_1\,{␣}_2\,2\,s_3\,i_3\,4\,6$



(k). Input: ␣. Output: 00.
$esc\,d_1\,1\,r_1\,{␣}_3\,2\,s_3\,i_3\,5\,6 \rightarrow$
$esc\,d_1\,1\,r_1\,2\,{␣}_3\,s_3\,i_3\,5\,6$

Figure Ans.12: Exercise 2.12. Part IV.

2.17: The results are shown in Tables Ans.15 and Ans.16. When all symbols are LPS, the output C always points at the bottom $A(1-Qe)$ of the upper (LPS) subinterval. When the symbols are MPS, the output always points at the bottom of the lower (MPS) subinterval, i.e., 0.

2.18: The results are shown in Tables Ans.17 and Ans.18 (compare with the answer to exercise 2.17).

2.19: The four decoding steps are as follows:

Step 1: $C = 0.981$, $A = 1$, the dividing line is $A(1-Qe) = 1(1-0.1) = 0.9$, so the LPS and MPS subintervals are $[0, 0.9)$ and $[0.9, 1)$. Since C points to the upper subinterval, an LPS is decoded. The new C is $0.981-1(1-0.1) = 0.081$ and the new A is $1\times 0.1 = 0.1$.
Step 2: $C = 0.081$, $A = 0.1$, the dividing line is $A(1-Qe) = 0.1(1-0.1) = 0.09$, so the LPS and MPS subintervals are $[0, 0.09)$ and $[0.09, 0.1)$, and an MPS is decoded. C is unchanged and the new A is $0.1(1-0.1) = 0.09$.
Step 3: $C = 0.081$, $A = 0.09$, the dividing line is $A(1-Qe) = 0.09(1-0.1) = 0.0081$, so the LPS and MPS subintervals are $[0, 0.0081)$ and $[0.0081, 0.09)$, and an LPS is decoded. The new C is $0.081 - 0.09(1-0.1) = 0$ and the new A is $0.09\times 0.1 = 0.009$.
Step 4: $C = 0$, $A = 0.009$, the dividing line is $A(1-Qe) = 0.009(1-0.1) = 0.00081$, so the LPS and MPS subintervals are $[0, 0.00081)$ and $[0.00081, 0.009)$, and an MPS is decoded. C is unchanged and the new A is $0.009(1-0.1) = 0.00081$.

2.20: In practice, an encoder may encode texts other than English, such as a foreign language or the source code of a computer program. Acronyms, such as QED and abbreviations, such as qwerty, are also good examples. Even in English there are some examples of a `q` not followed by a `u`, such as in this sentence. (The author has noticed that science-fiction writers tend to use non-English sounding words, such as Qaal, to name characters in their works.)

2.21: An object file generated by a compiler or an assembler normally has several distinct parts including the machine instructions, symbol table, relocation bits, and constants. Such parts may have different bit distributions.

2.22: The alphabet has to be extended, in such a case, to include one more symbol. If the original alphabet consisted of all the possible 256 8-bit bytes, it should be extended to 9-bit symbols, and should include 257 values.

2.23: Table Ans.19 shows the groups generated in both cases and makes it clear why these particular probabilities were assigned.

2.24: The first three cases don't change. They still code a symbol with 1, 1.32, and 6.57 bits, which is less than the 8 bits required for a 256-symbol alphabet without compression. Case 4 is different since the `d` is now encoded with a probability of 1/256, producing 8 instead of 4.8 bits. The total number of bits required to encode the `d` in case 4 is now $1 + 1.32 + 1.93 + 8 = 12.25$.

Symbol	C	A
Initially	0	1
s1 (LPS)	$0+1(1-0.5)=0.5$	$1\times0.5=0.5$
s2 (LPS)	$0.5+0.5(1-0.5)=0.75$	$0.5\times0.5=0.25$
s3 (LPS)	$0.75+0.25(1-0.5)=0.875$	$0.25\times0.5=0.125$
s4 (LPS)	$0.875+0.125(1-0.5)=0.9375$	$0.125\times0.5=0.0625$

Table Ans.15: Encoding Four Symbols With $Qe=0.5$.

Symbol	C	A
Initially	0	1
s1 (MPS)	0	$1\times(1-0.1)=0.9$
s2 (MPS)	0	$0.9\times(1-0.1)=0.81$
s3 (MPS)	0	$0.81\times(1-0.1)=0.729$
s4 (MPS)	0	$0.729\times(1-0.1)=0.6561$

Table Ans.16: Encoding Four Symbols With $Qe=0.1$.

Symbol	C	A	Renor. A	Renor. C
Initially	0	1		
s1 (LPS)	$0+1-0.5=0.5$	0.5	1	1
s2 (LPS)	$1+1-0.5=1.5$	0.5	1	3
s3 (LPS)	$3+1-0.5=3.5$	0.5	1	7
s4 (LPS)	$7+1-0.5=6.5$	0.5	1	13

Table Ans.17: Renormalization Added to Table Ans.15.

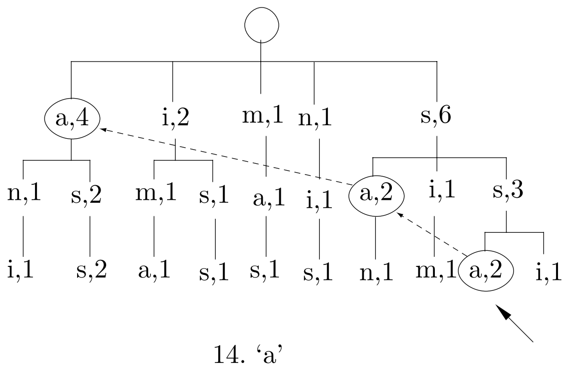
Symbol	C	A	Renor. A	Renor. C
Initially	0	1		
s1 (MPS)	0	$1-0.1=0.9$		
s2 (MPS)	0	$0.9-0.1=0.8$		
s3 (MPS)	0	$0.8-0.1=0.7$	1.4	0
s4 (MPS)	0	$1.4-0.1=1.3$		

Table Ans.18: Renormalization Added to Table Ans.16.

Context	f	p
abc→x	10	10/11
Esc	1	1/11

Context		f	p
abc→	a_1	1	1/20
→	a_2	1	1/20
→	a_3	1	1/20
→	a_4	1	1/20
→	a_5	1	1/20
→	a_6	1	1/20
→	a_7	1	1/20
→	a_8	1	1/20
→	a_9	1	1/20
→	a_{10}	1	1/20
Esc		10	10/20
Total		20	

Table Ans.19: Stable vs. Variable Data.



14. ‘a’

Figure Ans.20: Final Trie of “assanissimassa”.

2.25: The final trie is shown in Figure Ans.20.

2.26: For the first string the single bit has a suffix of 00, so the probability of leaf 00 is $P_e(1,0) = 1/2$. This is equal to the probability of string 0 without any suffix. For the second string each of the two zero bits has suffix 00, so the probability of leaf 00 is $P_e(2,0) = 3/8 = 0.375$. This is greater than the probability 0.25 of string 00 without any suffix. Similarly, the probabilities of the remaining three strings are $P_e(3,0) = 5/8 \approx 0.625$, $P_e(4,0) = 35/128 \approx 0.273$, and $P_e(5,0) = 63/256 \approx 0.246$. As the strings get longer, their probabilities get smaller but they are greater than the probabilities without the suffix. Having a suffix of 00 thus increases the probability of having strings of zeros following it.

2.27: The four trees are shown in Figure Ans.21a–d. The weighted probability that the next bit will be a zero given that three zeros have just been generated is 0.5. The weighted probability to have two consecutive zeros given the suffix 000 is 0.375, higher than the 0.25 without the suffix.

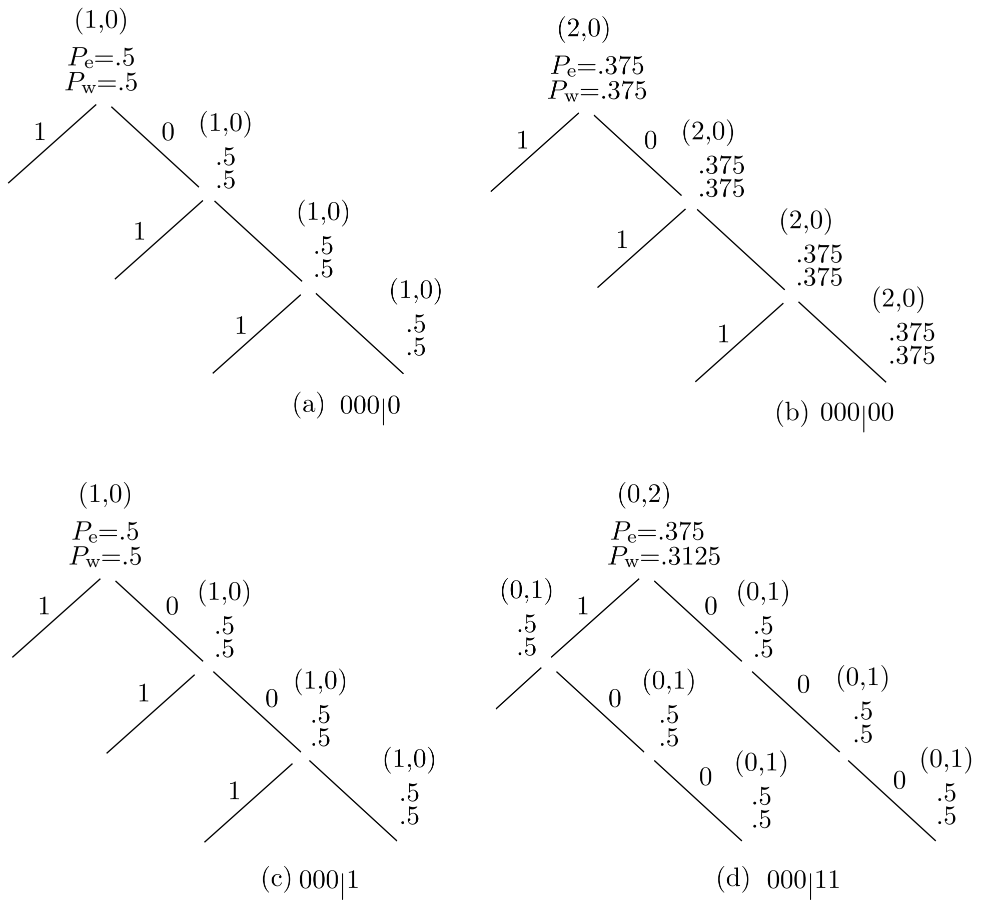


Figure Ans.21: Context Trees For 000|0, 000|00, 000|1, and 000|11.

Dictionary		Token	Dictionary		Token
15	"␣t"	(4, "t")	21	"␣si"	(19,"i")
16	"e"	(0, "e")	22	"c"	(0, "c")
17	"as"	(8, "s")	23	"k"	(0, "k")
18	"es"	(16,"s")	24	"␣se"	(19,"e")
19	"␣s"	(4, "s")	25	"al"	(8, "l")
20	"ea"	(4, "a")	26	"s(eof)"	(1, "(eof)")

Table Ans.22: Next 12 Encoding Steps in the LZ78 Example.

3.1: This is straightforward. The remaining steps are shown in Table Ans.22

3.2: Table Ans.23 shows the last three steps.
The final compressed stream consists of 1 control word followed by 11 items (9 literals and 2 copy items)

p_src	3 chars	Hash index	P	Output	Binary output
11	“`h␣t`”	7	any→7	`h`	01101000
12	“`␣th`”	5	5→5	4,7	0000|0011|00000111
16	“`ws`”			`ws`	01110111|01110011

Table Ans.23: Last Steps of Encoding “`that thatch thaws`”.

0000010010000000|01110100|01101000|01100001|01110100|00100000|0000|0011
|00000101|01100011|01101000|0000|0011|00000111|01110111|01110011.

3.3: Table Ans.24 summarizes the steps. The output emitted by the encoder is

97 (`a`), 108 (`l`), 102 (`f`), 32 (`␣`), 101 (`e`), 97 (`a`), 116 (`t`), 115 (`s`), 32 (`␣`), 256 (`al`), 102 (`f`), 265 (`alf`), 97 (`a`),

and the following new entries are added to the dictionary

(256: `al`), (257: `lf`), (258: `f␣`), (259: `␣e`), (260: `ea`), (261: `at`), (262: `ts`), (263: `s␣`), (264: `␣a`), (265: `alf`), (266: `fa`), (267: `alfa`).

I	in dict?	new entry	output	I	in dict?	new entry	output
`a`	Y			`s␣`	N	263-`s␣`	115 (`s`)
`al`	N	256-`al`	97 (`a`)	`␣`	Y		
`l`	Y			`␣a`	N	264-`␣a`	32 (`␣`)
`lf`	N	257-`lf`	108 (`l`)	`a`	Y		
`f`	Y			`al`	Y		
`f␣`	N	258-`f␣`	102 (`f`)	`alf`	N	265-`alf`	256 (`al`)
`␣`	Y			`f`	Y		
`␣e`	N	259-`␣e`	32 (`w`)	`fa`	N	266-`fa`	102 (`f`)
`e`	Y			`a`	Y		
`ea`	N	260-`ea`	101 (`e`)	`al`	Y		
`a`	Y			`alf`	Y		
`at`	N	261-`at`	97 (`a`)	`alfa`	N	267-`alfa`	265 (`alf`)
`t`	Y			`a`	Y		
`ts`	N	262-`ts`	116 (`t`)	`a`,eof	N		97 (`a`)
`s`	Y						

Table Ans.24: LZW Encoding of “`alf eats alfalfa`”.

3.4: The encoder inputs the first `a` into I, searches and finds `a` in the dictionary. It inputs the next `a` but finds that Ix, which is now `aa`, is not in the dictionary. The encoder thus adds string `aa` to the dictionary as entry 256 and outputs the token 97 (`a`). Variable I is initialized to the second `a`. The third `a` is input, so Ix is the string `aa`, which is now in the dictionary. I becomes this string, and the fourth `a` is input. Ix is now `aaa` which is not in the dictionary. The encoder thus adds string `aaa` to the dictionary as entry 257 and outputs 256 (aa). I is initialized to the fourth a. Continuing this process is straightforward.

The result is that strings aa, aaa, aaaa,... are added to the dictionary as entries 256, 257, 258,..., and the output is

97 (a), 256 (aa), 257 (aaa), 258 (aaaa),...

The output consists of pointers pointing to longer and longer strings of as. The first k pointers thus point at strings whose total length is $1+2+\cdots+k=(k+k^2)/2$.

Assuming an input stream that consists of one million as, we can find the size of the compressed output stream by solving the quadratic equation $(k+k^2)/2=1000000$ for the unknown k. The solution is $k\approx 1414$. The original, 8-million bit input is thus compressed into 1414 pointers, each at least 9-bit (and in practice, probably 16-bit) long. The compression factor is thus either $8\text{M}/(1414\times 9)\approx 628.6$ or $8\text{M}/(1414\times 16)\approx 353.6$.

This is an impressive result but such input streams are rare (notice that this particular input can best be compressed by generating an output stream containing just "1000000 a", and without using LZW).

3.5: We simply follow the decoding steps described in the text. The results are:

1. Input 97. This is in the dictionary so set I="a" and output "a". String "ax" needs to be saved in the dictionary but x is still unknown..
2. Input 108. This is in the dictionary so set J="l" and output "l". Save "al" in entry 256. Set I="l".
3. Input 102. This is in the dictionary so set J="f" and output "f". Save "lf" in entry 257. Set I="f".
4. Input 32. This is in the dictionary so set J="␣" and output "␣". Save "f␣" in entry 258. Set I="␣".
5. Input 101. This is in the dictionary so set J="e" and output "e". Save "␣e" in entry 259. Set I="e".
6. Input 97. This is in the dictionary so set J="a" and output "a". Save "ea" in entry 260. Set I="a".
7. Input 116. This is in the dictionary so set J="t" and output "t". Save "at" in entry 261. Set I="t".
8. Input 115. This is in the dictionary so set J="s" and output "s". Save "ts" in entry 262. Set I="t".
9. Input 32. This is in the dictionary so set J="␣" and output "␣". Save "s␣" in entry 263. Set I="␣".
10. Input 256. This is in the dictionary so set J="al" and output "al". Save "␣a" in entry 264. Set I="al".
11. Input 102. This is in the dictionary so set J="f" and output "f". Save "alf" in entry 265. Set I="f".
12. Input 265. This has just been saved in the dictionary so set J="alf" and output "alf". Save "fa" in dictionary entry 266. Set I="alf".
13. Input 97. This is in the dictionary so set J="a" and output "a". Save "alfa" in entry 267 (even though it will never be used). Set I="a".
14. Read eof. Stop.

3.6: We assume that the dictionary is initialized to just the two entries (1: `a`) and (2: `b`). The encoder outputs

1 (`a`), 2 (`b`), 3 (`ab`), 5(`aba`), 4(`ba`), 7 (`bab`), 6 (`abab`), 9 (`ababa`), 8 (`baba`),...

and adds the new entries (3: `ab`), (4: `ba`), (5: `aba`), (6: `abab`), (7: `bab`), (8: `baba`), (9: `ababa`), (10: `ababab`), (11: `babab`),... to the dictionary. This regular behavior can be analyzed and the kth output pointer and dictionary entry predicted, but the effort is probably not worth it.

3.7: This is straightforward (Table Ans.25) but not very efficient since only one two-symbol dictionary phrase is used.

Step	Input	Output	S	Add to dict.	S'
	`swiss␣miss`				
1	`s`	115	`s`	—	`s`
2	`w`	119	`w`	256-`sw`	`w`
3	`i`	105	`i`	257-`wi`	`i`
4	`s`	115	`s`	258-`is`	`s`
5	`s`	115	`s`	259-`ss`	`s`
6	`-`	32	`␣`	260-`s␣`	`␣`
7	`m`	109	`m`	261-`␣m`	`m`
8	`is`	258	`is`	262-`mis`	`is`
9	`s`	115	`s`	263-`iss`	`s`

Table Ans.25: LZMW Compression of "`swiss miss`".

3.8: Table Ans.26 shows all the steps. In spite of the short input, the result is quite good (13 codes to compress 18-symbols) because the input contains concentrations of `a`s and `b`s.

3.9: 1. The encoder starts by shifting the first two symbols `xy` to the search buffer, outputting them as literals and initializing all locations of the index table to the null pointer.
2. The current symbol is `a` (the first `a`) and the context is `xy`. It is hashed to, say, 5, but location 5 of the index table contains a null pointer, so `P` is null. Location 5 is set to point to the first `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.
3. The current symbol is the second `a` and the context is `ya`. It is hashed to, say, 1, but location 1 of the index table contains a null pointer, so `P` is null. Location 1 is set to point to the second `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.
4. The current symbol is the third `a` and the context is `aa`. It is hashed to, say, 2, but location 2 of the index table contains a null pointer, so `P` is null. Location 2 is set to point to the third `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

Step	Input	Output	S	Add to dict.	S'
	yabbadabbadabbadoo				
1	y	121	y	—	y
2	a	97	a	256-ya	a
3	b	98	b	257-ab	b
4	b	98	b	258-bb	b
5	a	97	a	259-ba	a
6	d	100	a	260-ad	a
7	ab	257	ab	261-dab	ab
8	ba	259	ba	262-abba	ba
9	dab	261	dab	263-badab	dab
10	ba	259	ba	264-dabba	ba
11	d	100	d	265-bad	d
12	o	111	o	266-do	o
13	o	111	o	267-o	o

Table Ans.26: LZMW Compression of "yabbadabbadabbadoo".

5. The current symbol is the fourth a and the context is aa. We know from step 4 that it is hashed to 2, and location 2 of the index table points to the third a. Location 2 is set to point to the fourth a, and the encoder tries to match the string starting with the third a to the string starting with the fourth a. Assuming that the look-ahead buffer is full of as, the match length L will be the size of that buffer. The encoded value of L will be written to the compressed stream, and the data in the buffer shifted L positions to the left.

6. If the original input stream is long, more a's will be shifted into the look-ahead buffer, and this step will also result in a match of length L. If only n as remain in the input stream, they will be matched, and the encoded value of n output.

The compressed stream will consist of the three literals x, y, and a, followed by (perhaps several values of) L, and possibly ending with a smaller value.

3.10: T percent of the compressed stream is made up of literals, some appearing consecutively (and thus getting the flag "1" for two literals, half a bit per literal) and others with a match length following them (and thus getting the flag "01", one bit for the literal). We assume that two thirds of the literals appear consecutively and one third are followed by match lengths. The total number of flag bits created for literals is thus

$$\frac{2}{3}T\times 0.5+\frac{1}{3}T\times 1.$$

A similar argument for the match lengths yields

$$\frac{2}{3}(1-T)\times 2+\frac{1}{3}(1-T)\times 1$$

for the total number of the flag bits. We now write the equation

$$\frac{2}{3}T\times 0.5+\frac{1}{3}T\times 1+\frac{2}{3}(1-T)\times 2+\frac{1}{3}(1-T)\times 1=1,$$

which is solved to yield $T=2/3$. This means that if two thirds of the items in the compressed stream are literals, there would be 1 flag bit per item on the average. More literals would result in fewer flag bits.

4.1: This is shown by multiplying the largest four n-bit number, $\underbrace{11\ldots1}_{n}$ by 4, which is easily done by shifting it 2 positions to the left. The result is the $n+2$-bit number $\underbrace{11\ldots1}_{n}00$.

4.2: Each new codebook entry $C_i^{(k)}$ is calculated, in step 4 of iteration k, as the average of the block images comprising partition $P_i^{(k-1)}$. In our example the image blocks (points) are concentrated in four separate regions, so the partitions calculated for iteration $k=1$ are the same as those for $k=0$. Another iteration, for $k=2$, will therefore compute the same partitions in its step 1 yielding, in step 3, an average distortion $D^{(2)}$ that equals $D^{(1)}$. Step 3 will therefore indicate convergence.

4.3: Monitor the compression ratio and delete the dictionary and start afresh each time compression performance drops below a certain threshold.

4.4: The two previously seen neighbors of P=8 are A=1 and B=11. P is thus in the central region, where all codes start with a zero, and L=1, H=11. The computations are straightforward:

$$k=\lfloor\log_2(11-1+1)\rfloor=3,\qquad a=2^{3+1}-11=5,\qquad b=2(11-2^3)=6.$$

Table Ans.27 lists the five 3-bit codes and six 4-bit codes for the central region. The code for 8 is thus 0|111.

The two previously seen neighbors of P=7 are A=2 and B=5. P is thus in the right outer region, where all codes start with 11, and L=2, H=7. We are looking for the code of $7-5=2$. Choosing $m=1$ yields, from Table 4.127, the code 11|01.

The two previously seen neighbors of P=0 are A=3 and B=5. P is thus in the left outer region, where all codes start with 10, and L=3, H=5. We are looking for the code of $3-0=3$. Choosing $m=1$ yields, from Table 4.127, the code 10|100.

4.5: The decoder knows this pixel since it knows the value of average $\mu[i-1,j]=0.5(I[2i-2,2j]+I[2i-1,2j+1])$ and since it has already decoded pixel $I[2i-2,2j]$

Pixel P	Region code	Pixel code
1	0	0000
2	0	0010
3	0	0100
4	0	011
5	0	100
6	0	101
7	0	110
8	0	111
9	0	0001
10	0	0011
11	0	0101

Table Ans.27: The Codes for a Central Region.

4.6: the four vectors are

$$\begin{aligned}\mathbf{a} &= (90,95,100,80,90,85),\\ \mathbf{b}^{(1)} &= (100,90,95,102,80,90),\\ \mathbf{b}^{(2)} &= (101,128,108,100,90,95),\\ \mathbf{b}^{(3)} &= (128,108,110,90,95,100),\end{aligned}$$

and the code of Figure Ans.28 produces the solutions $w_1 = 0.1051$, $w_2 = 0.3974$, and $w_3 = 0.3690$. Their total is 0.8715, compared with the original solutions, which added up to 0.9061. The point is that the numbers involved in the equations (the elements of the four vectors) are not independent (for example, pixel 80 appears in $\mathbf{a}$ and in $\mathbf{b}^{(1)}$) except for the last element (85 or 91) of $\mathbf{a}$ and the first element 101 of $\mathbf{b}^{(2)}$, which are independent. Changing these two elements affects the solutions, which is why the solutions do not always add up to unity. However, compressing nine pixels produces solutions whose total is closer to one than in the case of six pixels. Compressing an entire image, with many thousands of pixels, produces solutions whose sum is very close to 1.

```
a={90.,95,100,80,90,85};
b1={100,90,95,100,80,90};
b2={100,128,108,100,90,95};
b3={128,108,110,90,95,100};
Solve[{b1.(a-w1 b1-w2 b2-w3 b3)==0,
b2.(a-w1 b1-w2 b2-w3 b3)==0,
b3.(a-w1 b1-w2 b2-w3 b3)==0},{w1,w2,w3}]
```

Figure Ans.28: Solving For Three Weights.

4.7: Figure Ans.29a,b,c shows the results, with all H_i values shown in small type. Most H_i values are zero because the pixels of the original image are so highly correlated. The H_i values along the edges are very different because of the simple edge rule used. The result is that the H_i values are highly decorrelated and have low entropy. Thus, they are candidates for entropy coding.

1	.	3	.	5	.	7	.
.	0	.	0	.	0	.	-5
17	.	19	.	21	.	23	.
.	0	.	0	.	0	.	-13
33	.	35	.	37	.	39	.
.	0	.	0	.	0	.	-21
49	.	51	.	53	.	55	.
.	-33	.	-34	.	-35	.	-64

(a)

1	.	7	.	5	.	5	.
.	.	.	.	.	.	.	.
15	.	19	.	11	.	23	.
.	.	.	.	.	.	.	.
33	.	0	.	37	.	0	.
.	.	.	.	.	.	.	.
-33	.	51	.	-35	.	55	.
.	.	.	.	.	.	.	.

(b)

1	.	.	.	5	.	.	.
.	.	.	.	.	.	.	.
.	.	0	.	.	.	-5	.
.	.	.	.	.	.	.	.
33	.	.	.	37	.	.	.
.	.	.	.	.	.	.	.
.	.	-33	.	.	.	-55	.
.	.	.	.	.	.	.	.

(c)

Figure Ans.29: (a) Bands L_2 and H_2. (b) Bands L_3 and H_3. (c) Bands L_4 and H_4.

4.8: They are shown in Figure Ans.30

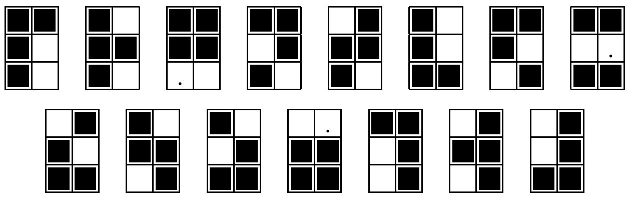


Figure Ans.30: The 15 6-Tuples With Two White Pixels.

4.9: The two halves of L_0 are distinct, so L_1 consists of the two elements

$$L_1 = (0101010101010101, 1010101010101010),$$

and the first indicator vector is $I_1 = (0, 0)$. The two elements of L_1 are distinct, so L_2 has the four elements

$$L_2 = (01010101, 01010101, 10101010, 10101010),$$

and the second indicator vector is $I_2 = (0, 1, 0, 2)$. Two elements of L_2 are distinct, so L_3 has the four elements $L_3 = (0101, 0101, 1010, 1010)$, and the third indicator vector is $I_3 = (0, 1, 0, 2)$. Again two elements of L_3 are distinct, so L_4 has the four elements $L_4 = (01, 01, 10, 10)$, and the fourth indicator vector is $I_4 = (0, 1, 0, 2)$. Only two elements of L_4 are distinct, so L_5 has the four elements $L_5 = (0, 1, 1, 0)$.

The output thus consists of $k = 5$, the value 2 (indicating that I_2 is the first nonzero vector) I_2, I_3, and I_4 (encoded), followed by $L_5 = (0, 1, 1, 0)$.

4.10: This particular numbering makes it easy to convert between the number of a subsquare and its image coordinates. (We assume that the origin is located at the bottom-left corner of the image and that image coordinates vary from 0 to 1.) As an example, translating the digits of the number 1032 to binary results in (01)(00)(11)(10). The first bits of these groups constitute the x coordinate of the subsquare, and the second bits constitute the y coordinate. Thus, the image coordinates of subsquare 1032 are $x = .0011_2 = 3/16$ and $y = .1010_2 = 5/8$, as can be directly verified from Figure 4.170c.

4.11: Figure Ans.31a,b shows the six states and all 21 edges. We use the notation $i(q,t)j$ for the edge with quadrant number q and transformation t from state i to state j. This GFA is more complex than pervious ones since the original image is less self-similar.

4.12: All three transformations shrink an image to half its original size. In addition, w_2 and w_3 place two copies of the shrunken image at relative displacements of $(0, 1/2)$ and $(1/2, 0)$, as shown in Figure Ans.32. The result is the familiar Sierpiński gasket but in a different orientation.

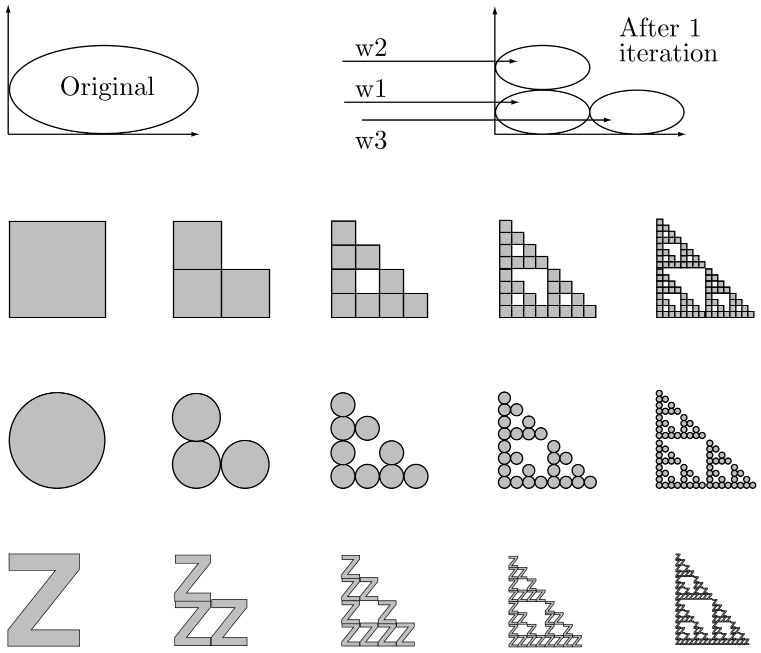
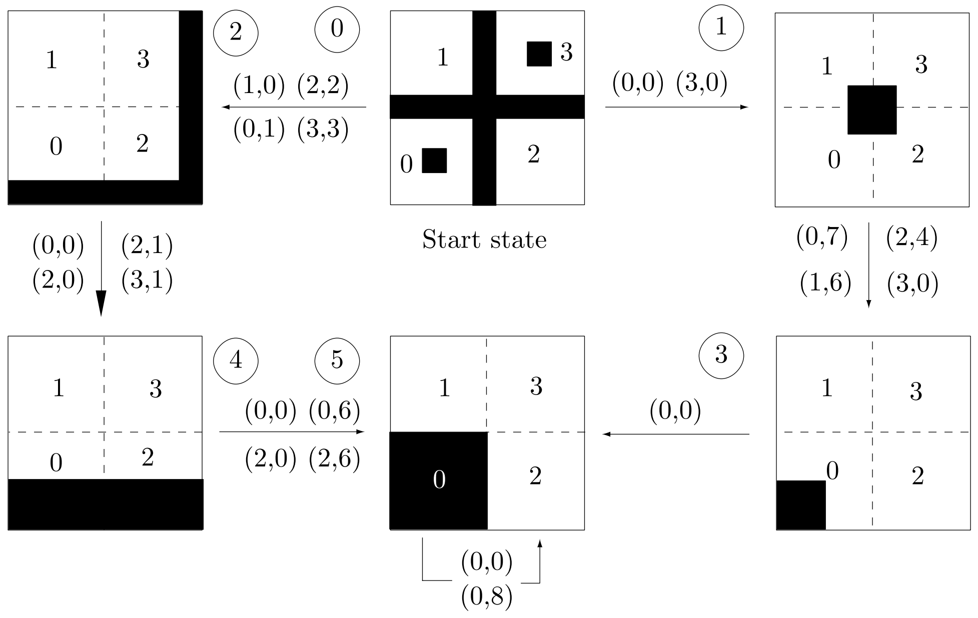


Figure Ans.32: Another Sierpiński Gasket.

5.1: Figure Ans.33 shows $f(t)$ and three shifted copies of the wavelet, for $a = 1$ and $b = 2$, 4, and 6. The inner product $W(a, b)$ is plotted below each copy of the wavelet. It is easy to see how the inner products are affected by the increasing frequency.



(a)

$$\begin{tikzcd}
2 \arrow[d, "(0,0)\ (2,0)"', "(2,1)\ (3,1)"] & 0 \arrow[l, "(1,0)\ (2,2)"', "(0,1)\ (3,3)"] \arrow[r, "(0,0)\ (3,0)"] & 1 \arrow[d, "(0,7)\ (1,6)"', "(2,4)\ (3,0)"] \\
4 & 5 \arrow[l, "(0,0)\ (0,6)"', "(2,0)\ (2,6)"] \arrow[r, "(0,0)"] \arrow[loop below, "(0,0)\ (0,8)"] & 3
\end{tikzcd}$$

(b)

```
0(0,0)1   0(3,0)1   0(0,1)2   0(1,0)2   0(2,2)2   0(3,3)2   1(0,7)3
1(1,6)3   1(2,4)3   1(3,0)3   2(0,0)4   2(2,0)4   2(2,1)4   2(3,1)4
3(0,0)5   4(0,0)5   4(0,6)5   4(2,0)5   4(2,6)5   5(0,0)5   5(0,8)5
```

Figure Ans.31: A GFA for Exercise 4.11.

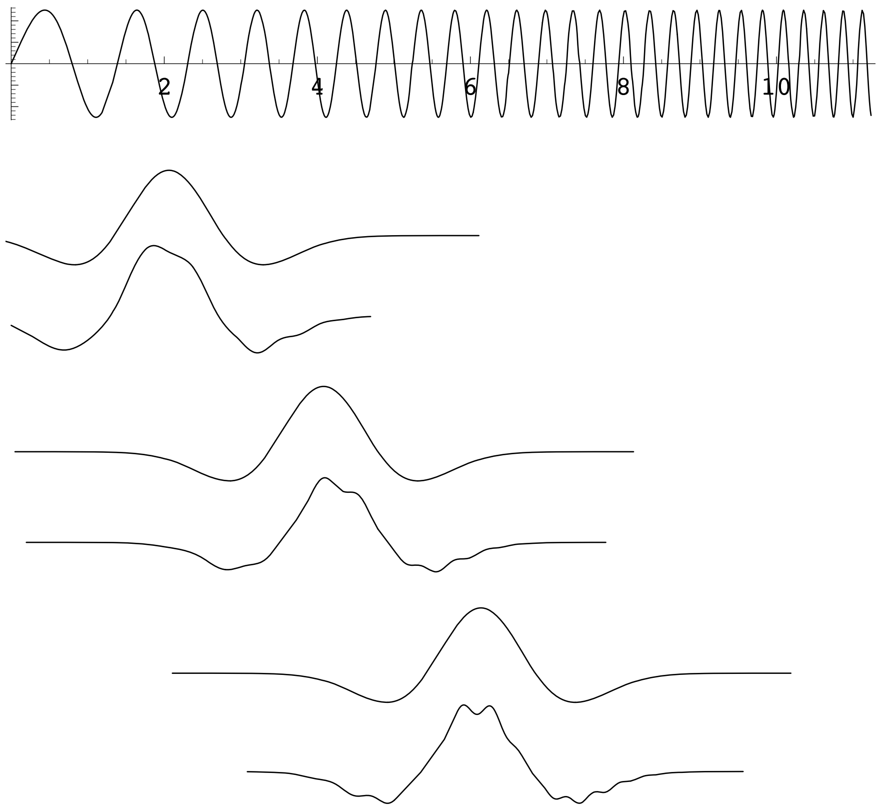


Figure Ans.33: An Inner Product for $a = 1$ and $b = 2, 4, 6$.

a	$b = 2$	3	4	5	6
1	0.032512	0.000299	1.10923×10^{-6}	2.73032×10^{-9}	8.33866×10^{-11}
2	0.510418	0.212575	0.0481292	0.00626348	0.00048097
3	0.743313	0.629473	0.380634	0.173591	0.064264

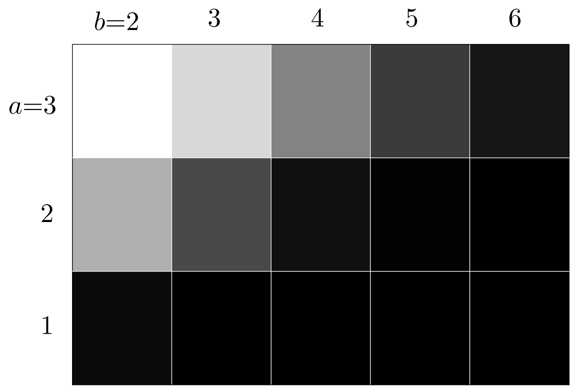


Figure Ans.34: Fifteen Values And a Density Plot of $W(a, b)$.

The table of Figure Ans.34 lists 15 values of $W(a,b)$, for $a = 1$, 2, and 3 and for $b = 2$ through 6. The density plot of the figure, where the bright parts correspond to large values, shows those values graphically. For each value of a, the CWT yields values that drop with b, reflecting the fact that the frequency of $f(t)$ increases with t. The five values of $W(1,b)$ are small and very similar, while the five values of $W(3,b)$ are larger and differ more. This shows how scaling the wavelet up makes the CWT more sensitive to frequency changes in $f(t)$.

5.2: Figure Ans.35a shows a simple, 8×8 image with one diagonal line above the main diagonal. Figure Ans.35b,c shows the first two steps in its pyramid decomposition. It is obvious that the transform coefficients in the bottom-right subband (HH) indicate a diagonal artifact located above the main diagonal. It is also easy to see that subband LL is a low-resolution version of the original image.

$$
\begin{array}{cccccccc}
12&16&12&12&12&12&12&12\\
12&12&16&12&12&12&12&12\\
12&12&12&16&12&12&12&12\\
12&12&12&12&16&12&12&12\\
12&12&12&12&12&16&12&12\\
12&12&12&12&12&12&16&12\\
12&12&12&12&12&12&12&16\\
12&12&12&12&12&12&12&12
\end{array}
\qquad
\begin{array}{cccc|cccc}
14&12&12&12&\underline{4}&0&0&0\\
12&14&12&12&0&4&0&0\\
12&14&12&12&0&\underline{4}&0&0\\
12&12&14&12&0&0&4&0\\
12&12&14&12&0&0&\underline{4}&0\\
12&12&12&14&0&0&0&4\\
12&12&12&14&0&0&0&\underline{4}\\
12&12&12&12&0&0&0&0
\end{array}
\qquad
\begin{array}{cccc|cccc}
13&13&12&12&\underline{2}&2&0&0\\
12&13&13&12&0&\underline{2}&2&0\\
12&12&13&13&0&0&\underline{2}&2\\
12&12&12&13&0&0&0&\underline{2}\\
\hline
2&\underline{2}&0&0&\underline{4}&\underline{4}&0&0\\
0&2&\underline{2}&0&0&\underline{4}&\underline{4}&0\\
0&0&2&\underline{2}&0&0&\underline{4}&\underline{4}\\
0&0&0&2&0&0&0&\underline{4}
\end{array}
$$

(a) (b) (c)

Figure Ans.35: The Subband Decomposition of a Diagonal Line.

5.3: The Matlab code of Figure Ans.36 calculates W as the product of the three matrices A_1, A_2, and A_3 and computes the 8×8 matrix of transform coefficients. Notice that the top-left value 131.375 is the average of all the 64 image pixels.

5.4: Figure Ans.37 lists the Matlab code of the inverse wavelet transform function `iwt1(wc,coarse,filter)` and a test.

5.5: Figure Ans.38 shows the result of blurring the "lena" image. Parts (a) and (b) show the logarithmic multiresolution tree and the subband structure, respectively. Part (c) shows the results of the quantization. The transform coefficients of subbands 5–7 have been divided by two, and all the coefficients of subbands 8–13 have been cleared. We can say that the blurred image of part (d) has been reconstructed from the coefficients of subbands 1–4 (1/64th of the total number of transform coefficients) and half of the coefficients of subbands 5-7 (half of 3/64, or 3/128). On average, the image has been reconstructed from $5/128 \approx 0.039$ or 3.9% of the transform coefficients. Notice that the Daubechies D8 filter was used in the calculations. Readers are encouraged to use this code and experiment with the performance of other filters.

```
clear
a1=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
 0 0 0 0 1/2 1/2 0 0; 0 0 0 0 0 0 1/2 1/2;
 1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
 0 0 0 0 1/2 -1/2 0 0; 0 0 0 0 0 0 1/2 -1/2];
% a1*[255; 224; 192; 159; 127; 95; 63; 32];
a2=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
 1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
 0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
a3=[1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0;
 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0;
 0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
w=a3*a2*a1;
dim=8; fid=fopen('8x8','r');
img=fread(fid,[dim,dim])'; fclose(fid);
w*img*w' % Result of the transform
```

131.375	4.250	−7.875	−0.125	−0.25	−15.5	0	−0.25
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75

Figure Ans.36: Code and Results For the Calculation of Matrix W and Transform $W\cdot I\cdot W^T$.

5.6: We sum Equation (5.13) over all the values of l to get

$$\sum_{l=0}^{2^{j-1}-1} s_{j-1,l}=\sum_{l=0}^{2^{j-1}-1}(s_{j,2l}+d_{j-1,l}/2)=\frac{1}{2}\sum_{l=0}^{2^{j-1}-1}(s_{j,2l}+s_{j,2l+1})=\frac{1}{2}\sum_{l=0}^{2^{j}-1}s_{j,l}. \quad \text{(Ans.1)}$$

Therefore, the average of set s_{j-1} equals

$$\frac{1}{2^{j-1}}\sum_{l=0}^{2^{j-1}-1}s_{j-1,l}=\frac{1}{2^{j-1}}\frac{1}{2}\sum_{l=0}^{2^{j}-1}s_{j,l}=\frac{1}{2^{j}}\sum_{l=0}^{2^{j}-1}s_{j,l}$$

the average of set s_j.

5.7: The Matlab code of Figure Ans.39 does that and produces the transformed integer vector $y=(111,-1,84,0,120,25,84,3)$. The inverse transform generates vector z that is identical to the original data x. Notice how the detail coefficients are much smaller than the weighted averages. Notice also that Matlab arrays are indexed from 1, whereas

```
function dat=iwt1(wc,coarse,filter)
% Inverse Discrete Wavelet Transform
dat=wc(1:2^coarse);
n=length(wc); j=log2(n);
for i=coarse:j-1
 dat=ILoPass(dat,filter)+ ...
  IHiPass(wc((2^(i)+1):(2^(i+1))),filter);
end

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% returns a vector of length 2*n with zeros
% placed between consecutive values
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

Figure Ans.37: Code For the One-Dimensional Inverse Discrete Wavelet Transform.

A simple test of `iwt1` is

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)
```

the discussion in the text assumes arrays indexed from 0. This causes the difference in index values in Figure Ans.39.

5.8: In the sorting pass of the third iteration the encoder transmits the number $l = 3$ (the number of coefficients $c_{i,j}$ in our example that satisfy $2^{12} \le |c_{i,j}| < 2^{13}$), followed by the three pairs of coordinates $(3,3)$, $(4,2)$, and $(4,1)$ and by the signs of the three coefficients. In the refinement step it transmits the six bits *cdefgh*. These are the 13th most significant bits of the coefficients transmitted in all the previous iterations.

The information received so far enables the decoder to further improve the 16 approximate coefficients. The first nine become

$$c_{2,3} = s1ac0\ldots0,\ c_{3,4} = s1bd0\ldots0,\ c_{3,2} = s01e00\ldots0,$$
$$c_{4,4} = s01f00\ldots0,\ c_{1,2} = s01g00\ldots0,\ c_{3,1} = s01h00\ldots0,$$
$$c_{3,3} = s0010\ldots0,\ c_{4,2} = s0010\ldots0,\ c_{4,1} = s0010\ldots0,$$

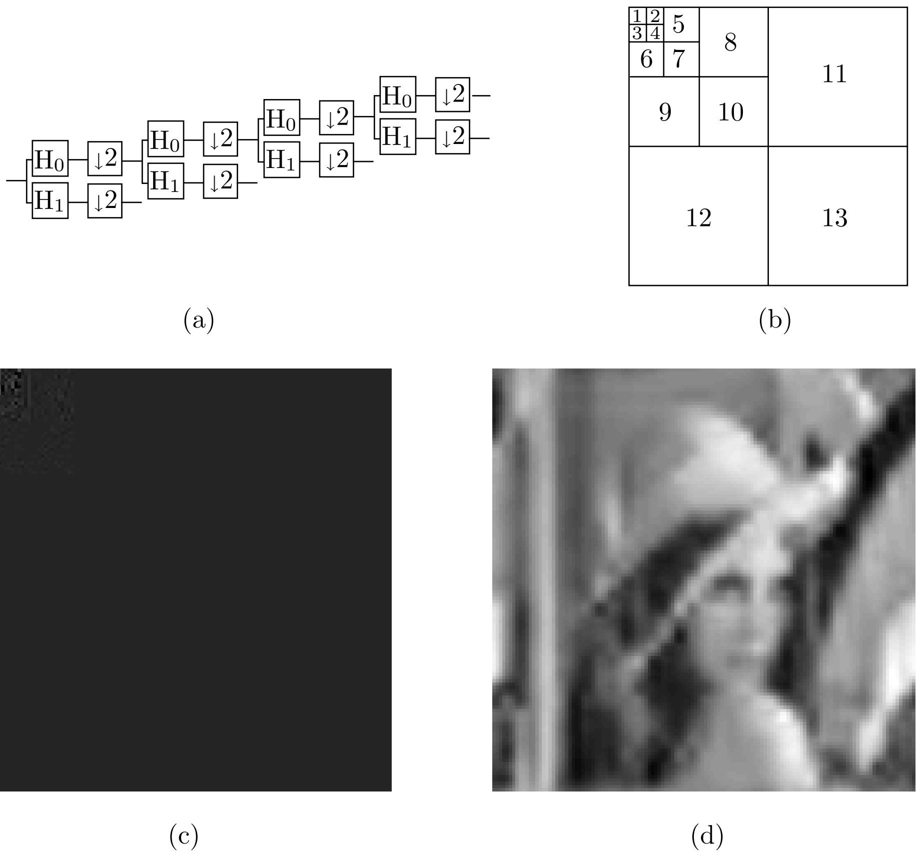


Figure Ans.38: Blurring As A Result of Coarse Quantization.

```
clear, colormap(gray);
filename='lena128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
filt=[0.23037,0.71484,0.63088,-0.02798, ...
 -0.18703,0.03084,0.03288,-0.01059];
fwim=fwt2(img,3,filt);
figure(1), imagesc(fwim), axis square
 fwim(1:16,17:32)=fwim(1:16,17:32)/2;
 fwim(1:16,33:128)=0;
 fwim(17:32,1:32)=fwim(17:32,1:32)/2;
 fwim(17:32,33:128)=0;
 fwim(33:128,:)=0;
figure(2), colormap(gray), imagesc(fwim)
rec=iwt2(fwim,3,filt);
figure(3), colormap(gray), imagesc(rec)
```

Code For Figure Ans.38.

```
clear;
N=8; k=N/2;
x=[112,97,85,99,114,120,77,80];
% Forward IWT into y
for i=0:k-2,
 y(2*i+2)=x(2*i+2)-floor((x(2*i+1)+x(2*i+3))/2);
end;
y(N)=x(N)-x(N-1);
y(1)=x(1)+floor(y(2)/2);
for i=1:k-1,
 y(2*i+1)=x(2*i+1)+floor((y(2*i)+y(2*i+2))/4);
end;
% Inverse IWT into z
z(1)=y(1)-floor(y(2)/2);
for i=1:k-1,
 z(2*i+1)=y(2*i+1)-floor((y(2*i)+y(2*i+2))/4);
end;
for i=0:k-2,
 z(2*i+2)=y(2*i+2)+floor((z(2*i+1)+x(2*i+3))/2);
end;
z(N)=y(N)+z(N-1);
```

Figure Ans.39: Matlab Code For Forward and Inverse IWT.

and the remaining seven are not changed.

6.1: The vertical height of the picture on the author's 27 in. television set is 16 in., which translates to a viewing distance of $7.12\times16 = 114$ in. or about 9.5 feet. It is easy to see that individual scan lines are visible at any distance shorter than about 6 feet.

6.2: The golden ratio $\phi \approx 1.618$ has traditionally been considered the aspect ratio that is most pleasing to the eye. This suggests that 1.77 is the better aspect ratio.

6.3: Since $(4,4)$ is at the center of the "+", the value of s is halved, to 2. The next step searches the four blocks labeled 4, centered on $(4,4)$. Assuming that the best match is at $(6,4)$, the two blocks labeled 5 are searched. Assuming that $(6,4)$ is the best match, s is halved to 1, and the eight blocks labeled 6 are searched. The diagram shows that the best match is finally found at location $(7,4)$.

7.1: An average book may have 60 characters per line, 45 lines per page, and 400 pages. This comes to $60\times45\times400 = 1,080,000$ characters, requiring one byte of storage each.

7.2: Assuming that a noise level P_1 translates to x decibels

$$20\log\left(\frac{P_1}{P_2}\right) = x \text{ dB SPL},$$

results in the relation

$$20\log\left(\frac{\sqrt[3]{2}P_1}{P_2}\right) = 20\left[\log_{10}\sqrt[3]{2} + \log\left(\frac{P_1}{P_2}\right)\right] = 20(0.1 + x/20) = x + 2.$$

Thus, increasing the sound level by a factor of $\sqrt[3]{2}$ increases the decibel level by 2 dB SPL.

7.3: Table 7.37 shows that the scale factor is 111 and the select information is 2. The third rule in Table 7.38 shows that a scfsi of 2 means that only one scale factor was coded, occupying just six bits in the compressed output. The decoder assigns these six bits as the values of all three scale factors.

8.1: Figure Ans.40 shows the rotations of S and the sorted matrix. The last column, L of Ans.40b happens to be identical to S, so S=L="`sssssssssh`". Since A=(`s`,`h`), a move-to-front compression of L yields $C = (1, 0, 0, 0, 0, 0, 0, 0, 0, 1)$. Since C contains just the two values 0 and 1, they can serve as their own Huffman codes, so the final result is 1000000001, 1 bit per character!

`sssssssssh`	`hsssssssss`
`sssssssshs`	`shssssssss`
`ssssssshss`	`sshsssssss`
`sssssshsss`	`ssshssssss`
`ssssshssss`	`sssshsssss`
`sssshsssss`	`ssssshssss`
`ssshssssss`	`sssssshsss`
`sshsssssss`	`ssssssshss`
`shssssssss`	`sssssssshs`
`hsssssssss`	`sssssssssh`
(a)	(b)

Figure Ans.40: Permutations of "`sssssssssh`".

8.2: Table Ans.41 shows the sorted contexts. Equation (Ans.2) shows the context ranking at each step.

$$\begin{array}{l}
\begin{array}{ccccc} 0\ , & & 0 \rightarrow 2\ , & & 1 \rightarrow 3 \rightarrow 0\ , \\ u & & u\quad\ b & & l\quad b\quad u \end{array} \\
\begin{array}{cc} 0 \rightarrow 2 \rightarrow 3 \rightarrow 4\ , & 2 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 0\ , \\ u\quad l\quad a\quad b & l\quad a\quad d\quad b\quad u \end{array} \\
\begin{array}{c} 3 \rightarrow 5 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 0\ . \\ i\quad a\quad l\quad b\quad d\quad u \end{array}
\end{array} \tag{Ans.2}$$

The final output is "u 2 b 3 l 4 a 5 d 6 i 6." Notice that each of the distinct input symbols appears once in this output in raw format.

0	λ	u
1	u	x

(a)

0	λ	u
1	ub	x
2	u	b

(b)

0	λ	u
1	ub	l
2	ubl	x
3	u	b

(c)

0	λ	u
1	ubla	x
2	ub	l
3	ubl	a
4	u	b

(d)

0	λ	u
1	ubla	d
2	ub	l
3	ublad	x
4	ubl	a
5	u	b

(e)

0	λ	u
1	ubla	d
2	ub	l
3	ublad	i
4	ubladi	x
5	ubl	a
6	u	b

(f)

Table Ans.41: Constructing the Sorted Lists For `ubladiu`.

8.3: The input stream consists of:
1. A run of three zero groups, coded as 10|1 since 3 is in second position in class 2.
2. The nonzero group 0100, coded as 111100.
3. Another run of three zero groups, again coded as 10|1.
4. The nonzero group 1000, coded as 01100.
5. A run of four zero groups, coded as 010|00 since 4 is in first position in class 3.
6. 0010, coded as 111110.
7. A run of two zero groups, coded as 10|0.

The output is thus the 31-bit string 1011111001010110001000111110100.

8.4: The input stream consists of:
1. A run of three zero groups, coded as R_2R_1 or 101|11.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as 101|11.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $R_4 = 1001$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $R_2 = 101$.

The output is thus the 32-bit string 10111001001011101000100100010101.

8.5: The input stream consists of:
1. A run of three zero groups, coded as F_3 or 1001.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as 1001.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $F_3F_1 = 1001|11$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $F_2 = 101$.

The output is thus the 32-bit string 10010010010010100010011100010101.

8.6: Figure Ans.42 shows how state A feeds into the new state D' which, in turn, feeds into states E and F. Notice how states B and C haven't changed. Since the new state D' is identical to D, it is possible to feed A into either D or D' (cloning can be done in

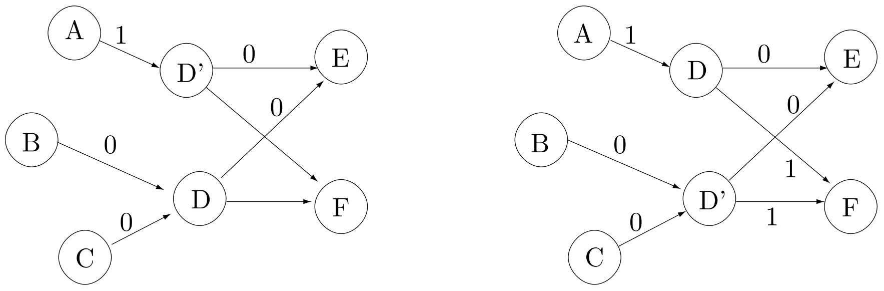


Figure Ans.42: New State D' Cloned.

two different but identical ways). The original counts of state D should now be divided between D and D' in proportion to the counts of the transitions $A \to D$ and $B, C \to D$.

8.7: In each of these cases, the encoder removes one edge from the boundary and inserts two new edges. There is a net gain of one edge.

8.8: They create triangles $(18, 2, 3)$ and $(18, 3, 4)$, and reduce the boundary to the sequence of vertices

$$(4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18).$$

A problem can be found for almost every solution.
Unknown