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: (1) ask a question, (2) absolutely necessary, (3) advance warning, (4) boiling hot, (5) climb up, (6) close scrutiny, (7) exactly the same, (8) free gift, (9) hot water heater, (10) my personal opinion, (11) newborn baby, (12) postponed until later, (13) unexpected surprise, (14) unsolved mysteries.

1.2: An obvious way is to use them to code the five most common strings in the text. Since irreversible text compression is a special-purpose method, the user may know what strings are common in any particular text to be compressed. The user may specify five such strings to the encoder, and they should also be written at the start of the output stream, for the decoder's use.

1.3: 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 numbers 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.

1.4: RLE of images is based on the idea that adjacent pixels tend to be identical. The last pixel of a row, however, has no reason to be identical to the first pixel of the next row.

1.5: 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.

1.6: This results in five groups as follows:

$$\begin{aligned}
W_1 \text{ to } W_2 &: 00000, 11111,\\
W_3 \text{ to } W_{10} &: 00001, 00011, 00111, 01111, 11110, 11100, 11000, 10000,\\
W_{11} \text{ to } W_{22} &: 00010, 00100, 01000, 00110, 01100, 01110,\\
&\quad 11101, 11011, 10111, 11001, 10011, 10001,\\
W_{23} \text{ to } W_{30} &: 01011, 10110, 01101, 11010, 10100, 01001, 10010, 00101,\\
W_{31} \text{ to } W_{32} &: 01010, 10101.
\end{aligned}$$

1.7: The seven codes are

$$0000, 1111, 0001, 1110, 0000, 0011, 1111.$$

Forming a string with six runs. Applying the rule of complementing yields the sequence

$$0000, 1111, 1110, 1110, 0000, 0011, 0000,$$

with *seven* runs. The rule of complementing does not always reduce the number of runs.

1.8: 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.

1.9: The six characters "`123ABC`" have ASCII codes 31, 32, 33, 41, 42 and 43. Translating these hexadecimal numbers to binary produces "`00110001 00110010 00110011 01000001 01000010 01000011`".
The next step is to divide this string of 48 bits into 6-bit blocks. They are 001100=12, 010011=19, 001000=8, 110011=51, 010000=16, 010100=20, 001001=9,

000011=3. The character at position 12 in the BinHex table is "`-`" (position numbering starts at zero). The one at position 19 is "`6`". The final result is the string "`-6)c38*$`".

1.10: Exercise 2.1 shows that the binary code of the integer i is $1 + \lfloor \log_2 i \rfloor$ bits long. We add $\lfloor \log_2 i \rfloor$ zeros, bringing the total size to $1 + 2\lfloor \log_2 i \rfloor$ bits.

1.11: 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.

1.12: 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.

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

2.2: Two is the smallest integer that can serve as the basis for a number system.

2.3: Replacing 10 by 3 we get $x = k \log_2 3 \approx 1.58k$. A trit is thus worth about 1.58 bits.

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.

2.4: 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

2.5: 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.

2.6: The triplet $(n, 1, n)$ defines the standard n-bit binary codes, as can be verified by direct construction. The number of such codes is easily seen to be

$$\frac{2^{n+1} - 2^n}{2^1 - 1} = 2^n.$$

The triplet $(0, 0, \infty)$ defines the codes 0, 10, 110, 1110,... which are the unary codes but assigned to the integers 0, 1, 2,... instead of 1, 2, 3,... .

2.7: The number is $(2^{30} - 2^1)/(2^1 - 1) \approx$ A billion.

2.8: 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).

2.9: Each part of C_4 is the standard binary code of some integer, so it starts with a 1. A part that starts with a 0 thus signals to the decoder that this is the last bit of the code.

2.10: We use the property that the Fibonacci representation of an integer does not have any adjacent 1's. If R is a positive integer, we construct its Fibonacci representation and append a 1-bit to the result. The Fibonacci representation of the integer 5 is 001, so the Fibonacci-prefix code of 5 is 0011. Similarly, the Fibonacci representation of 33 is 1010101, so its Fibonacci-prefix code is 10101011. It is obvious that each of these codes ends with two adjacent 1's, so they can be decoded uniquely. However, the property of not having adjacent 1's restricts the number of binary patterns available for such codes, so they are longer than the other codes shown here.

2.11: Subsequent splits can be done in different ways, but Table Ans.5 shows one way of assigning Shannon-Fano codes to the 7 symbols.

	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.

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.

2.12: This is immediate $-2(0.25 \times \log_2 0.25) - 4(0.125 \times \log_2 0.125) = 2.5$.

2.13: If method C does not expand any of the n-bit strings, then the result of applying it to all the n-bit input strings is a set of 2^n binary strings, none of which is longer than n bits and some of which are shorter. This is impossible because the total number of n-bit strings is 2^n, so the total number of shorter strings must be less than that.

2.14: The method does not exploit the redundancy in the input. Even in our simple example the two consecutive `SS` of `SWISS` (and also the two `SS` of `MISS`) result in $5 + 1 = 6$ bits. If the dictionary size is 256, then a pair of identical input bytes results in $256 + 1 = 257$ bits; a considerable expansion. On the other hand, two consecutive 1-bits are generated only if two consecutive input characters are identical to two consecutive dictionary characters. It seems reasonable to assume that the number of consecutive zeros between two 1-bits is, on average, half the dictionary size. If half the dictionary size is greater than eight (the size of a character), expansion occurs.

Later that evening, we were all sitting around the table talking when someone said something and Murray Gell-Mann remarked, "Oh, that's a pleonasm." Everyone went, "What?" "It's a sentence with a triple redundancy," Gell-Mann stated. Gell-Mann is well known among his associates for his pedantic knowledge of language and facts. Feynman and I sneaked into my library where we looked it up in the dictionary. Gell-Mann was right. Feynman hit his fist on the table, and exclaimed, "DAMN IT! He's always GODDAMNED right, always!" "Let's see if we can catch him tonight," I replied.

— Al Seckel

2.15: 1: The initial 10 indicates that $v_1 = 1$. The following 0 indicates that $v_2 = 0$. The fourth bit is 1, indicating that there are more nonzero vector elements to be decoded. Bits 5–6 are 10, indicating that $v_3 = 1$. Bits 7–8 are 11, indicating that $v_4 = -1$. Bit 9 is zero, indicating that the remaining elements are all zeros.

2: The shortest code, for all-zero vectors, is 000, just three bits. As an example of the longest code let's assume an 8-component vector where all the elements are 1. The resulting code is 101011010110101010, an 18-bit number, longer than the fixed-size code of 13 bits.

This method results in very short codes for certain vectors and very long codes for others. Thus, this method should be used in cases where we have to compress vectors that tend to have trailing zeros.

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

$$\begin{aligned}
&(5+5+5+5{\cdot}2+3{\cdot}3+3{\cdot}5+3{\cdot}5+12)/30 = 76/30,\\
&(5+5+4+4{\cdot}2+4{\cdot}3+3{\cdot}5+3{\cdot}5+12)/30 = 76/30,\\
&(6+6+5+4{\cdot}2+3{\cdot}3+3{\cdot}5+3{\cdot}5+12)/30 = 76/30.
\end{aligned}$$

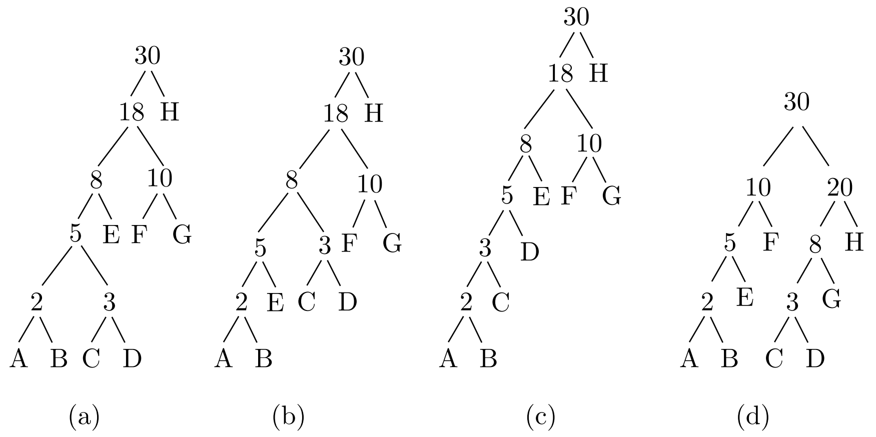


Figure Ans.6: Three Huffman Trees For Eight Symbols.

2.17: After adding symbols A, B, C, D, E, F, and G to the tree we were left with the three symbols ABEF (with probability 10/30), CDG (with probability 8/30), and H (with probability 12/30). The two symbols with lowest probabilities were ABEF and CDG, so they had to be merged. Instead, symbols CDG and H were merged, creating a non-Huffman tree.

2.18: 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.

2.19: Imagine a large alphabet where all the symbols have (about) the same probability. Since the alphabet is large, that probability will be small, resulting in long codes. Imagine the other extreme case, where certain symbols have high probabilities (and, therefore, short codes). Since the probabilities have to add up to 1, the rest of the symbols will have low probabilities (and, therefore, long codes). We thus see that the size of a code depends on the probability, but is indirectly affected by the size of the alphabet.

2.20: Answer not provided.

2.21: 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.

2.22: The number of groups increases exponentially from 2^s to $2^{s+n} = 2^s \times 2^n$.

2.23: 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.

2.24: The two trees are shown in Figure 2.24c,d. The average code size for the binary Huffman tree is

$$1\times.49 + 2\times.25 + 5\times.02 + 5\times.03 + 5\times.04 + 5\times.04 + 3\times.12 = 2\,\text{bits/symbol},$$

and that of the ternary tree is

$$1\times.26 + 3\times.02 + 3\times.03 + 3\times.04 + 2\times.04 + 2\times.12 + 1\times.49 = 1.34\,\text{trits/symbol}.$$

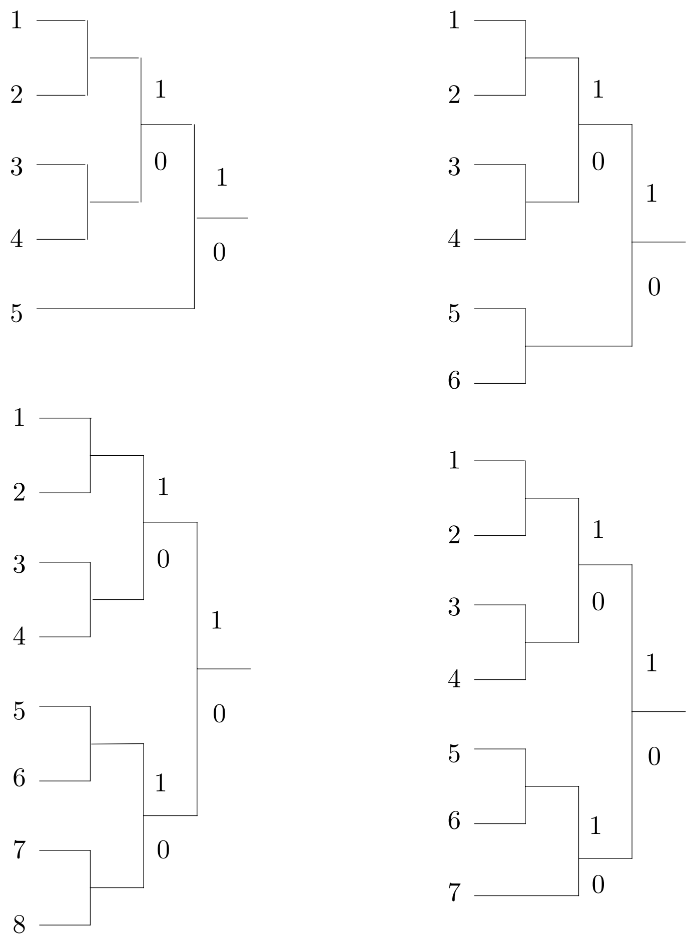


Figure Ans.8: Huffman Codes for Equal Probabilities.

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.

2.25: 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.

2.26: 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

2.27: A symbol with high frequency of occurrence should be assigned a shorter code. Therefore it has to appear high in the tree. The requirement that at each level the frequencies be sorted from left to right is artificial. In principle it is not necessary but it simplifies the process of updating the tree.

2.28: 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 $62/88\approx 0.7$.

2.29: A simple calculation shows that the average size of a token in Table 2.32 is about 9 bits. In stage 2, each 8-bit byte will be replaced, on the average, by a 9-bit token, resulting in an expansion factor of $9/8=1.125$ or 12.5%.

2.30: The decompressor will interpret the input data as "111110 0110 11000 0...", which is the string "XRP...".

2.31: A typical fax machine scans lines that are about 8.2 inches wide ($\approx$ 208 mm). A blank scan line thus produces 1,664 consecutive white pels.

2.32: These codes are needed for cases such as example 4, where the run length is 64, 128 or any length for which a make-up code has been assigned.

2.33: There may be fax machines (now or in the future) built for wider paper, so the Group 3 code was designed to accommodate them.

2.34: 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 2.38 have to satisfy the prefix property in each column but not between the columns.

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

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

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

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

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

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

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

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

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

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

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

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

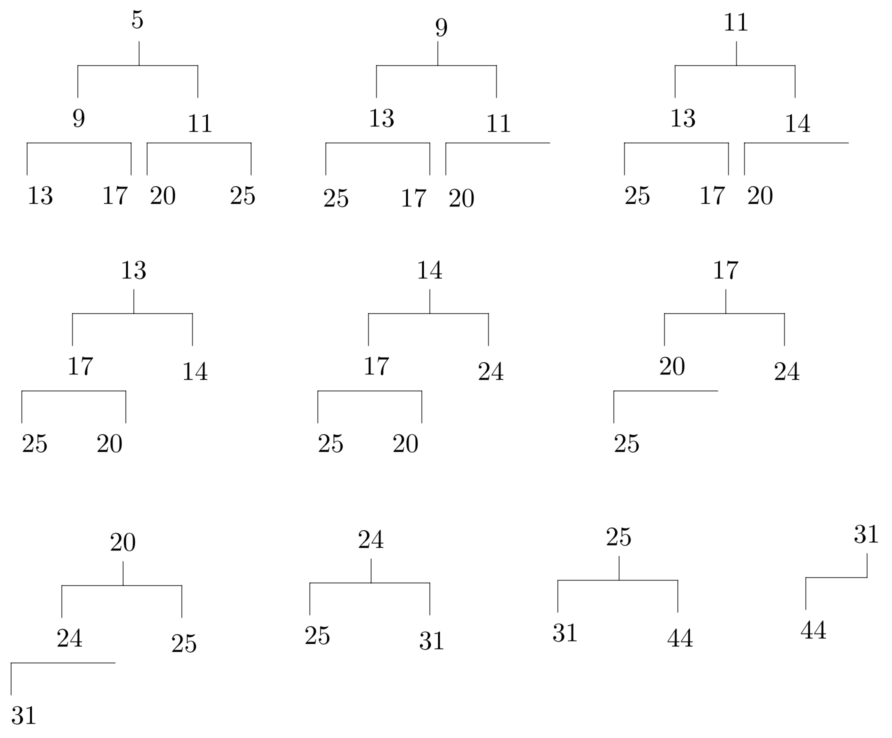
$$\begin{array}{cccccccccccccc}
\underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & \underline{6} & \underline{7} & \underline{8} & \underline{9} & \underline{10} & \underline{11} & \underline{12} & \underline{13} & \underline{14} \\
[\mathit{4} & \mathit{3}] & 56 & 44 & \mathit{3}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

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

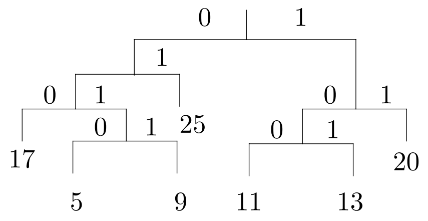
$$\begin{array}{cccccccccccccc}
\underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & \underline{6} & \underline{7} & \underline{8} & \underline{9} & \underline{10} & \underline{11} & \underline{12} & \underline{13} & \underline{14} \\
[\mathit{3}] & 56{+}44 & 56 & 44 & \mathit{3}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

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

Figure Ans.10: Sifting the Heap.



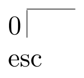
(a)



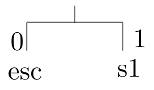
(b)

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

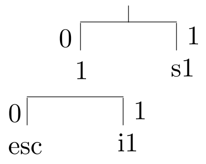
Initial tree



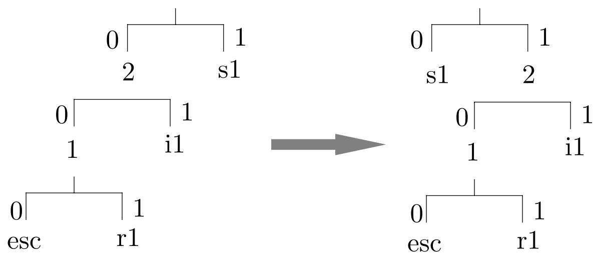
(a). Input: s. Output: ‘s’.
$esc\, s_1$



(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‘␣’.
$esc\, ␣_1\, 1\, r_1\, 2\, i_1\, s_1\, 3 \rightarrow$
$esc\, ␣_1\, 1\, r_1\, s_1\, i_1\, 2\, 2$

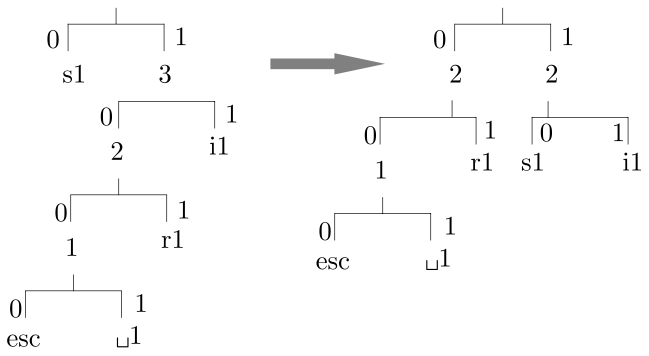
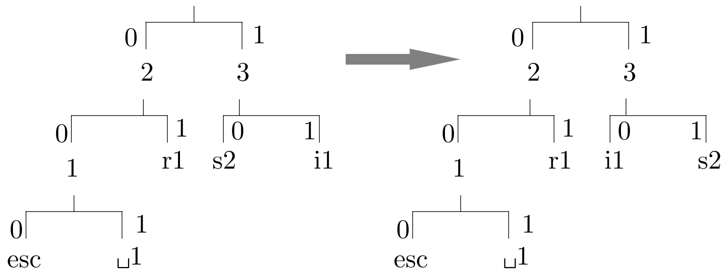
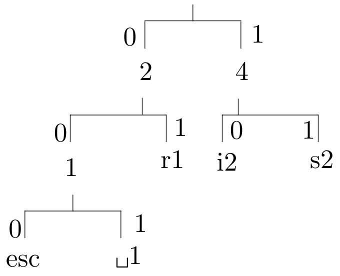


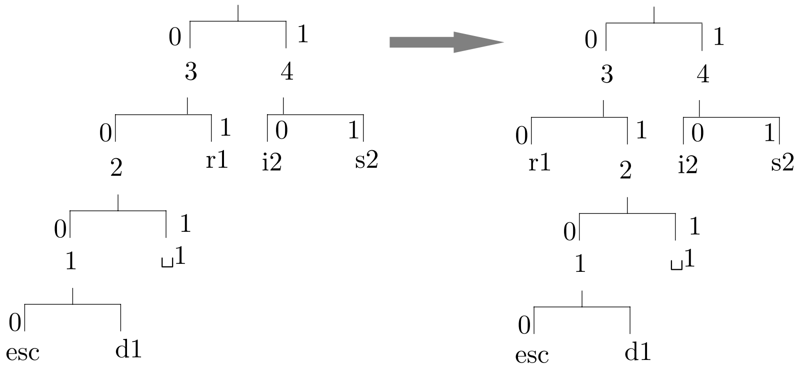
Figure Ans.12: Exercise 2.28. Part I.



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

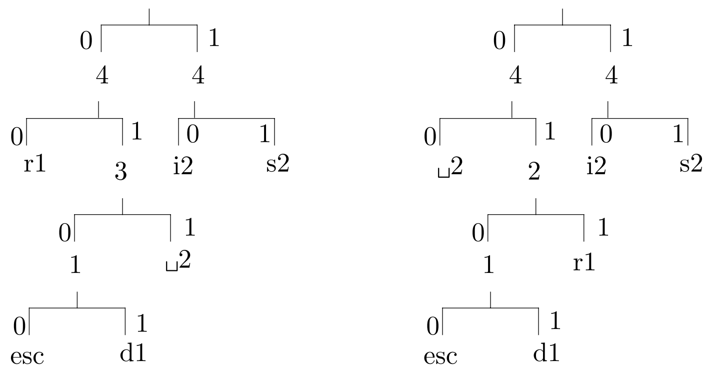


(f). Input: i. Output: 10.
$esc_{\sqcup 1}\,1\,r_1\,i_2\,s_2\,2\,4$

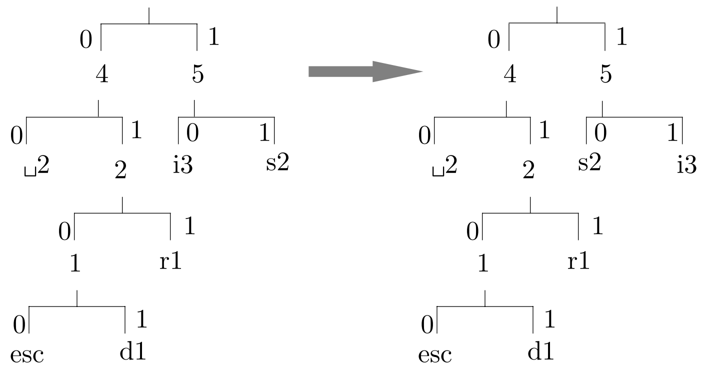


(g). Input: d. Output: 000‘d’.
$esc\,d_1\,1_{\sqcup 1}\,2\,r_1\,i_2\,s_2\,3\,4 \rightarrow$
$esc\,d_1\,1_{\sqcup 1}\,r_1\,2\,i_2\,s_2\,3\,4$

Figure Ans.12: Exercise 2.28. Part II.

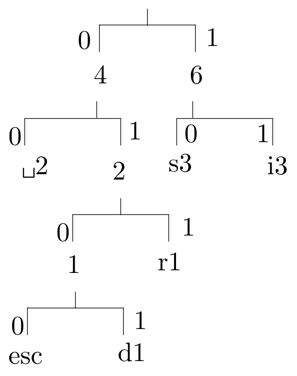


(h). Input: ␣. Output: 011.
$esc\, d_1\, 1\, {}_{\sqcup 2}\, r_1\, 3\, i_2\, s_2\, 4\, 4 \rightarrow$
$esc\, d_1\, 1\, r_1\, {}_{\sqcup 2}\, 2\, i_2\, s_2\, 4\, 4$

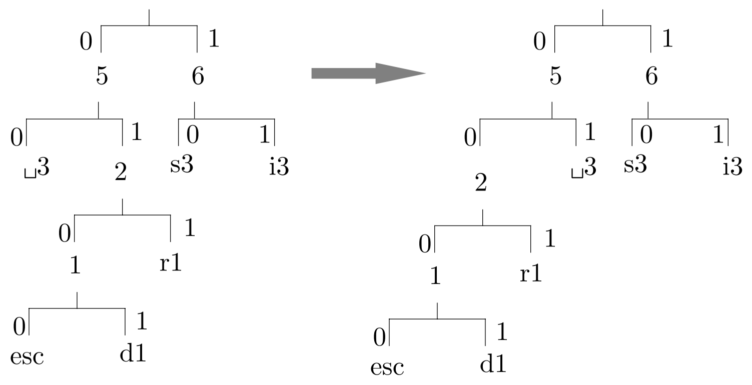


(i). Input: i. Output: 10.
$esc\, d_1\, 1\, r_1\, {}_{\sqcup 2}\, 2\, i_3\, s_2\, 4\, 5 \rightarrow$
$esc\, d_1\, 1\, r_1\, {}_{\sqcup 2}\, 2\, s_2\, i_3\, 4\, 5$

Figure Ans.12: Exercise 2.28. 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 2.28. Part IV.

2.35: The code of a run length of one white pel is 000111, and that of one black pel is 010. Two consecutive pels of different colors are thus coded into 9 bits. Since the uncoded data requires just two bits (01 or 10), the compression ratio is 9/2=4.5 (the compressed stream is 4.5 times longer than the uncompressed one; a large expansion).

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

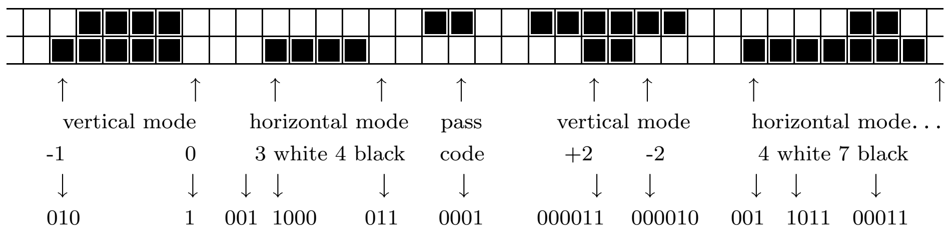


Figure Ans.13: Two-Dimensional Coding Example.

2.37: 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.

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	$0.06776322625 + 0.926859375 \times 0.023162{=}0.08923124309375$
	$0.06776322625 + 0.926859375 \times 0.998162{=}0.99291913371875$

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

2.38: An argument similar to the one in the previous exercise shows that there are two ways of writing this number. It can be written either as 0.1000... or 0.0111... .

2.39: In practice, the eof symbol has to be included in the original table of frequencies and probabilities. This symbol is the last to be encoded and the decoder stops when it detects an eof.

2.40: The encoding steps are simple (see first example on page 101). 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 since $-\log_2 0.125 \approx 6.322$ it follows that the practical minimum code size is 7 bits.

2.41: Perhaps the simplest way is to calculate a set of Huffman codes for the symbols, using their probabilities. This converts each symbol to a binary string, so the input stream can be encoded by the QM-coder. After the compressed stream is decoded by the QM-decoder, an extra step is needed, to convert the resulting binary strings back to the original symbols.

2.42: 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.

2.43: If the current input bit is an LPS, A is shrunk to Qe, which is always 0.5 or less, so A always has to be renormalized in such a case.

2.44: The results are shown in Tables Ans.17 and Ans.18 (compare with the answer to exercise 2.42).

2.45: 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 $0.009(1-0.1) = 0.00081$.

2.46: In practice, an encoder may encode texts other than English, such as a foreign language or the source code of a computer program. 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.)

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

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.

2.47: $256^2 = 65,536$, a manageable number, but $256^3 = 16,777,216$, perhaps too big for a practical implementation, unless a sophisticated data structure is used, or unless the encoder gets rid of older data from time to time.

2.48: A color or gray-scale image with 4-bit pixels. Each symbol is a pixel, and there are 16 different ones.

2.49: 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.

2.50: 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.

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

Context	f	p
`abc`$\to$`x`	10	10/11
Esc	1	1/11

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

Table Ans.19: Stable vs. Variable Data.

2.52: The `d` is added to the order-0 contexts with frequency 1. The escape frequency should be incremented from 5 to 6, bringing the total frequencies from 19 up to 21. The probability assigned to the new `d` is therefore 1/21, and that assigned to the escape is 6/21. All other probabilities are reduced from $x/19$ to $x/21$.

2.53: The new `d` would require switching from order-2 to order-0, sending two escapes that take 1 and 1.32 bits. The `d` is now found in order-0 with probability 1/21, so it is encoded in 4.39 bits. The total number of bits required to encode the second `d` is thus $1 + 1.32 + 4.39 = 6.71$, still greater than 5.

2.54: 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 $1 + 1.32 + 1.93 + 8 = 12.25$.

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

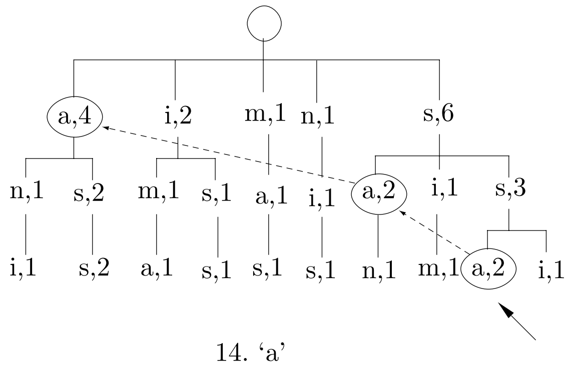


Figure Ans.20: Final Trie of "`assanissimassa`".

2.56: This is, of course

$$1 - P_e(b_{t+1} = 1|b_1^t) = 1 - \frac{b + 1/2}{a + b + 1} = \frac{a + 1/2}{a + b + 1}.$$

2.57: 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.

2.58: This is straightforward and is shown in Figure 2.77b.

2.59: 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.

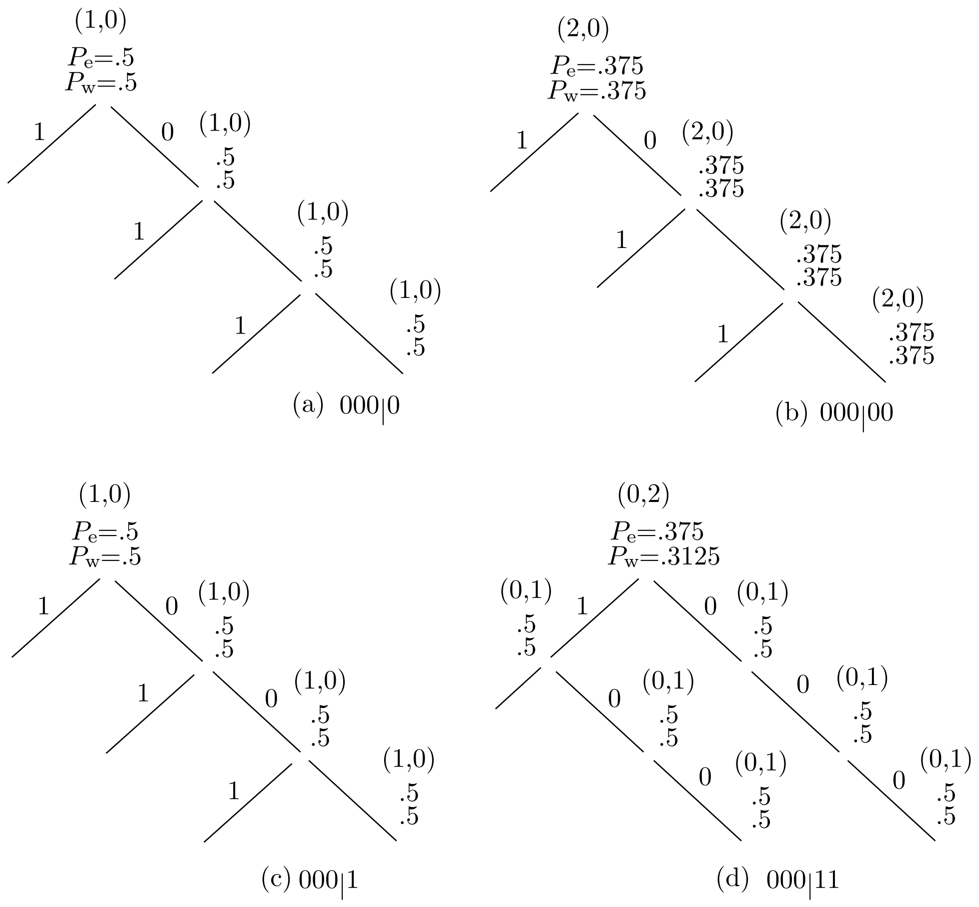


Figure Ans.21: Context Trees For 000|0, 000|00, 000|1, and 000|11.

3.1: The size of the output stream is $N[48 - 28P] = N[48 - 25.2] = 22.8N$. The size of the input stream is, as before, $40N$. The compression factor is thus $40/22.8 \approx 1.75$.

3.2: The decoder doesn't know but it does not need to know. The decoder simply reads tokens and uses each offset to find a string of text without having to know whether the string was a first or a last match.

3.3: The next step matches the space and encodes the string "␣e".

```
    sir␣sid|␣eastman␣easily␣  ⇒ (4,1,"e")
  sir␣sid␣e|astman␣easily␣te  ⇒ (0,0,"a")
```

and the next one matches nothing and encodes the "a".

3.4: The first two characters `CA` at positions 17–18 are a repeat of the `CA` at positions 9–10, so they will be encoded as a string of length 2 at offset $18 - 10 = 8$.

The next two characters AC at positions 19–20 are a repeat of the string at positions 8–9, so they will be encoded as a string of length 2 at offset $20 - 9 = 11$.

3.5: The decoder interprets the first 1 of the end marker as the start of a token. The second 1 is interpreted as the prefix of a 7-bit offset. The next 7 bits are 0 and they identify the end-marker as such, since a "normal" offset cannot be zero.

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

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.

3.7: Table Ans.23 shows the last three steps.

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".

The final compressed stream consists of 1 control word followed by 11 items (9 literals and 2 copy items)
0000010010000000|01110100|01101000|01100001|01110100|00100000|0000|0011
|00000101|01100011|01101000|0000|0011|00000111|01110111|01110011.

3.8: Imagine a compression utility for a personal computer that maintains all the files (or groups of files) on the hard disk in compressed form, to save space. Such a utility should be transparent to the user; it should automatically decompress a file every time it is opened and automatically compress it when it is being closed. In order to be transparent, such a utility should be fast; with compression ratio being only a secondary feature.

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

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

3.9: 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`).

3.10: 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 `a`s. 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 `a`s, 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 8M/(1414 × 9) ≈ 628.6 or 8M/(1414 × 16) ≈ 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).

3.11: 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.

3.12: 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.

3.13: The answer to exercise 3.10 shows the relation between the size of the compressed file and the size of the largest dictionary string for the "worst case" situation (input that creates the longest strings). For a 1Mbyte input stream, there will be 1,414 strings in the dictionary, the largest of which is 1,414 symbols long.

3.14: 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".

3.15: 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 as and bs.

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".

3.16: 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.
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 `a`s, 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 `a`s 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.

3.17: 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.

3.18: The first three ones indicate six literals. The following 01 indicates a literal (`b`) followed by a match length (of 3). The 10 is the code of match length 3, and the last 1 indicates two more literals (`x` and `y`).

4.1: No. The definition of redundancy (Section 2.1) tells us that an image where each color appears with the same frequency has no redundancy (statistically), yet it is not necessarily random and may even be interesting and/or useful.

4.2: Figure Ans.27 shows two 32×32 matrices. The first one, a, with random (and therefore decorrelated) values and the second one, b, is its inverse (and therefore with correlated values). Their covariance matrices are also shown and it is obvious that matrix cov(a) is close to diagonal, whereas matrix cov(b) is far from diagonal. The Matlab code for this figure is also listed.

4.3: The results are shown in Table Ans.28 together with the Matlab code used to calculate it.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	01100	**10000**	11000	1**1000**	10100
0000**1**	00001	0100**1**	01101	1000**1**	11001	1100**1**	10101
000**10**	00011	010**10**	01111	100**10**	11011	110**10**	10111
0001**1**	00010	0101**1**	01110	1001**1**	11010	1101**1**	10110
00**100**	00110	01**100**	01010	10**100**	11110	11**100**	10010
0010**1**	00111	0110**1**	01011	1010**1**	11111	1110**1**	10011
001**10**	00101	011**10**	01001	101**10**	11101	111**10**	10001
0011**1**	00100	0111**1**	01000	1011**1**	11100	1111**1**	10000

Table Ans.28: First 32 Binary and Gray Codes.

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b); dec2bin(b)
```

Code For Table Ans.28.

4.4: One feature is the regular way in which each of the five code bits alternates periodically between 0 and 1. It is easy to write a program that will set all five bits to 0, will flip the rightmost bit after two codes have been calculated, and will flip any of the other four code bits in the middle of the period of its immediate neighbor on the right.

Another feature is the fact that the second half of the table is a mirror image of the first half, but with the most significant bit set to one. After the first half

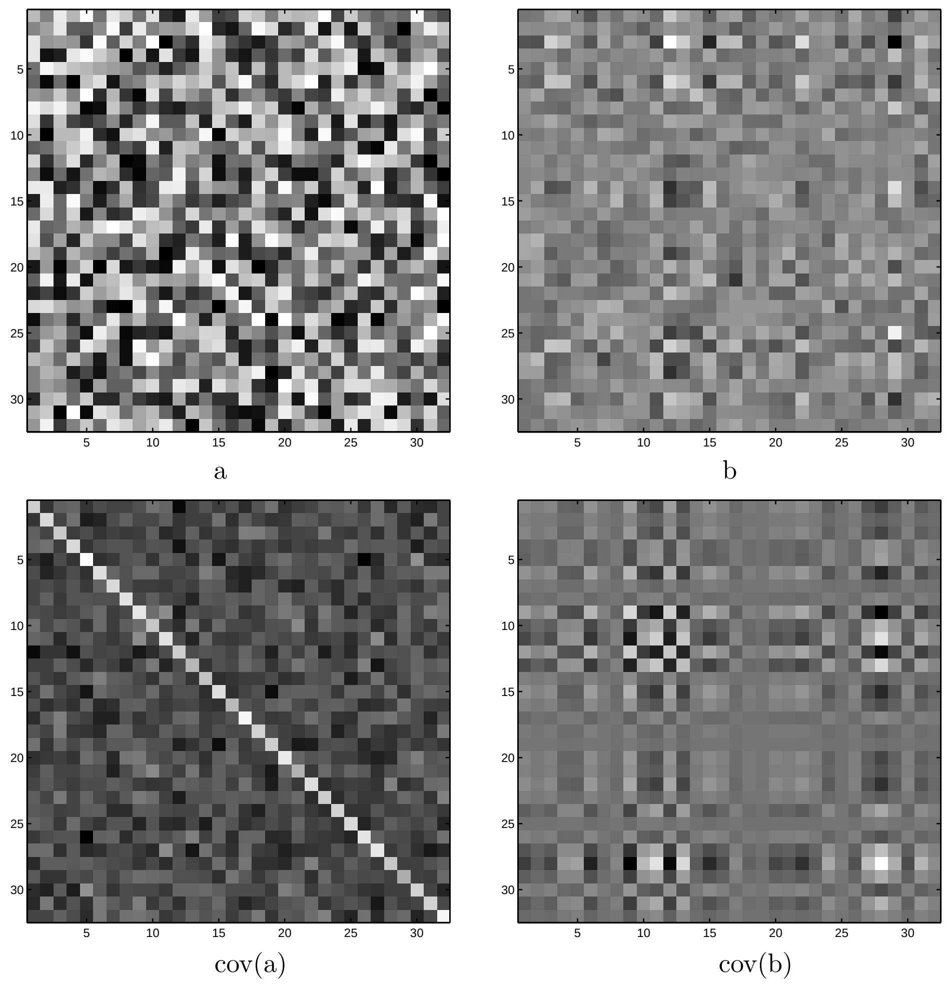


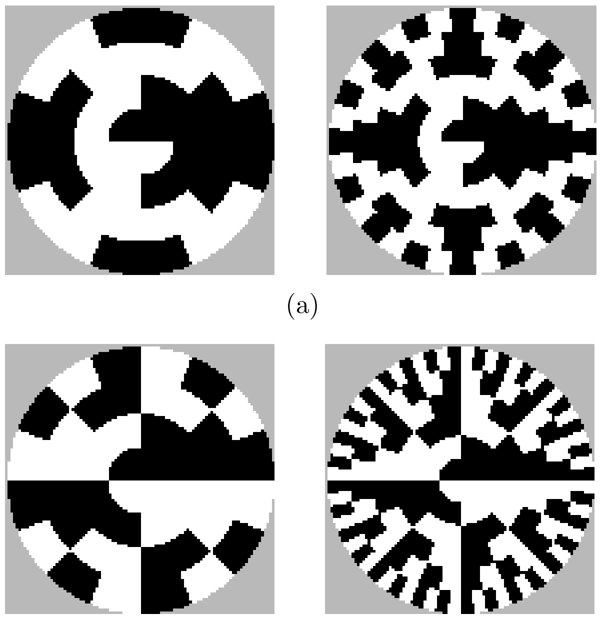
Figure Ans.27: Covariance Matrices of Correlated and Decorrelated Values

```
a=rand(32); b=inv(a);
figure(1), imagesc(a), colormap(gray); axis square
figure(2), imagesc(b), colormap(gray); axis square
figure(3), imagesc(cov(a)), colormap(gray); axis square
figure(4), imagesc(cov(b)), colormap(gray); axis square
```

Code for Figure Ans.27.

of the table has been computed, using any method, this symmetry can be used to quickly calculate the second half.

4.5: Figure Ans.29 is an *angular code wheel* representation of the 4 and 6-bit RGC codes (part a) and the 4 and 6-bit binary codes (part b). The individual bitplanes are shown as rings, with the most significant bits as the innermost ring. It is easy to see that the maximum angular frequency of the RGC is half that of the binary code.



(a)

(b)

Figure Ans.29: Angular Code Wheels of RGC and Binary Codes.

4.6: No. If pixel values are in the range $[0, 255]$, a difference $(P_i - Q_i)$ can be at most 255. The worst case is where all the differences are 255. It is easy to see that such a case yields an RMSE of 255.

4.7: The code of Figure Ans.30 yields the coordinates of the rotated points

$$(7.071, 0),\ (9.19, 0.7071),\ (17.9, 0.78),\ (33.9, 1.41),\ (43.13, -2.12),$$

(notice how all the y coordinates are small numbers) and shows that the cross-correlation drops from 1729.72 before the rotation to -23.0846 after it. A significant reduction!

4.8: The eight values in the top row are close (the distances between them are either 2 or 3). Each of the other rows is obtained as a right-circular shift of the preceding row.

```
p={{5,5},{6, 7},{12.1,13.2},{23,25},{32,29}};
rot={{0.7071,-0.7071},{0.7071,0.7071}};
Sum[p[[i,1]]p[[i,2]], {i,5}]
q=p.rot
Sum[q[[i,1]]q[[i,2]], {i,5}]
```

Figure Ans.30: Code For Rotating Five Points.

4.9: It is obvious that such a block can be represented as a linear combination of the patterns in the leftmost column of Figure 4.21. The actual calculation yields the eight weights 4, 0.72, 0, 0.85, 0, 1.27, 0, and 3.62 for the patterns of this column. The other 56 weights are zero or very close to zero.

4.10: The *Mathematica* code below produces the eight coefficients

$$140, -71, 0, -7, 0, -2, 0, 0.$$

After clearing the last two nonzero weights (-7 and -2) and applying the one-dimensional IDCT, Equation(4.8),

```
DCT[i_]:={(1/2)Cr[i]Sum[Pixl[[x+1]]Cos[(2x+1)i Pi/16], {x,0,7,1}]};
 IDCT[x_]:={(1/2)Sum[Cr[i]G[[i+1]]Cos[(2x+1)i Pi/16], {i,0,7,1}]};
```

to the sequence $140, -71, 0, 0, 0, 0, 0, 0$, we get $15, 20, 30, 43, 56, 69, 79$, and 84. These are not identical to the original values, but the maximum difference is only 4.

4.11: Figure Ans.31 shows the results and the Matlab code. Notice that the same code can also be used to calculate and display the DCT basis images.

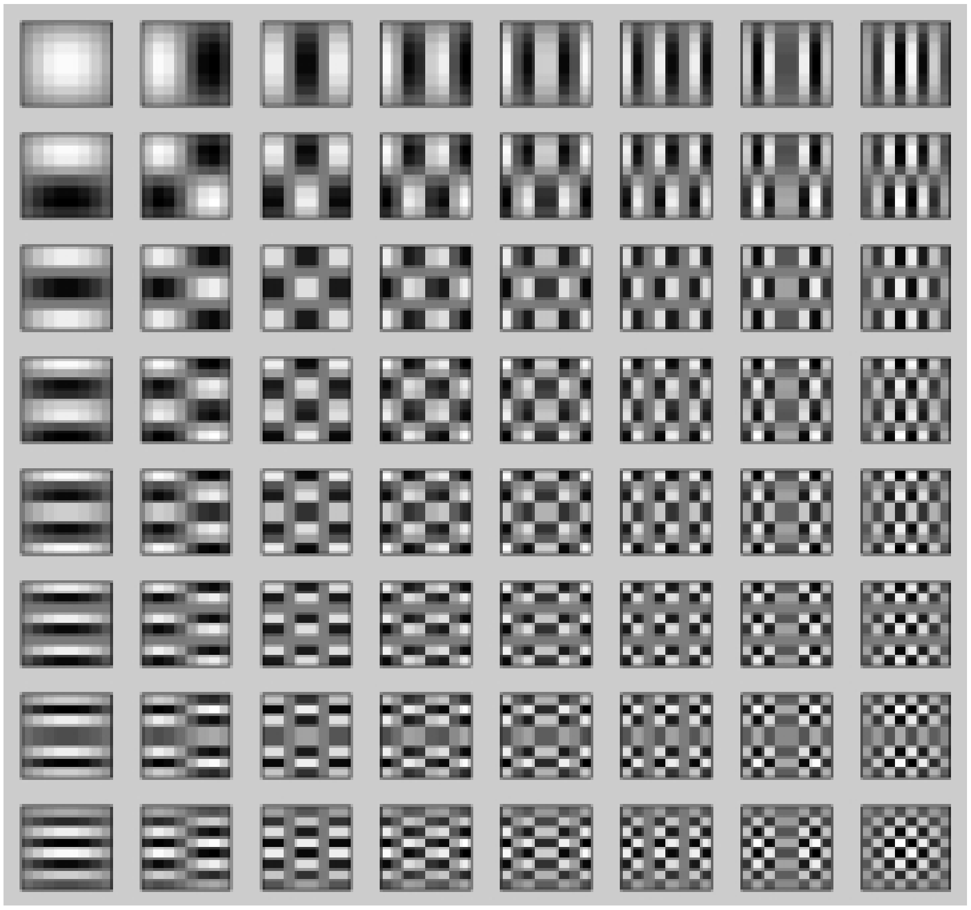
4.12: Figure Ans.32 shows the 64 basis images and the Matlab code to calculate and display them. Each basis image is an 8×8 matrix.

4.13: $\mathbf{A}_4$ is the 4×4 matrix

$$\mathbf{A}_4 = \begin{pmatrix} h_0(0/4) & h_0(1/4) & h_0(2/4) & h_0(3/4) \\ h_1(0/4) & h_1(1/4) & h_1(2/4) & h_1(3/4) \\ h_2(0/4) & h_2(1/4) & h_2(2/4) & h_2(3/4) \\ h_3(0/4) & h_3(1/4) & h_3(2/4) & h_3(3/4) \end{pmatrix} = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}.$$

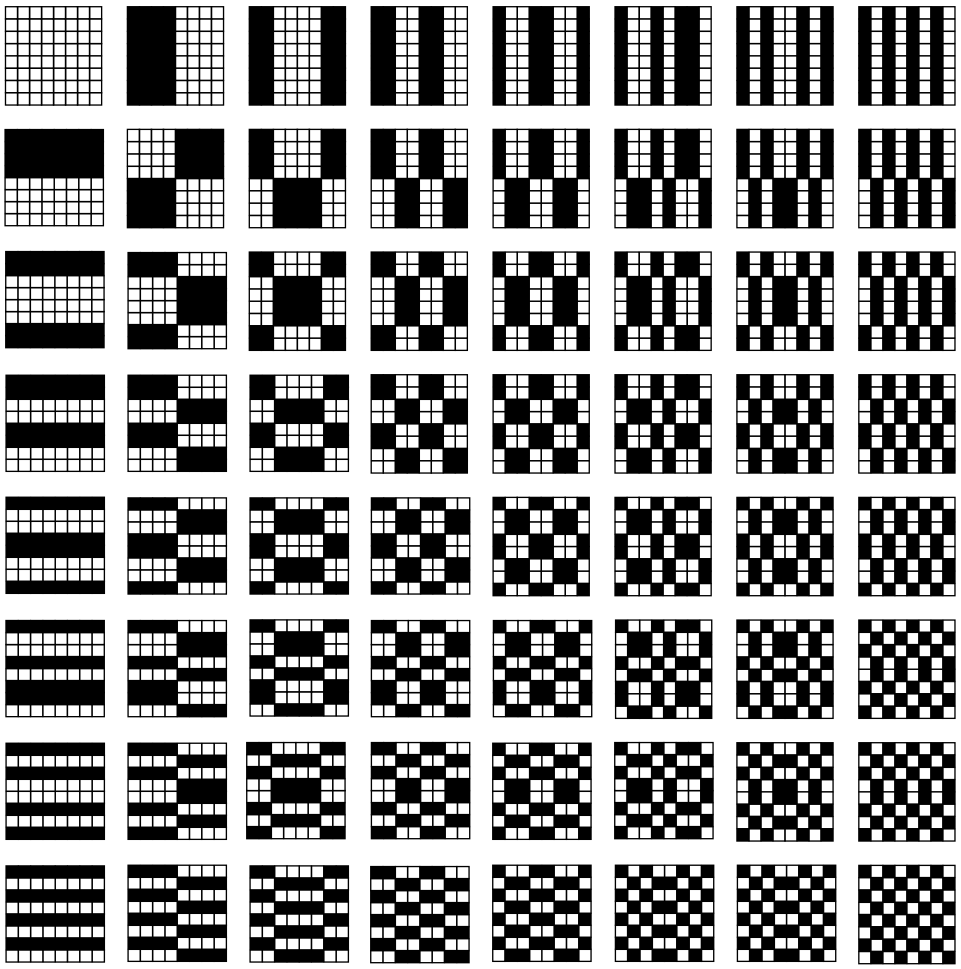
Similarly, $\mathbf{A}_8$ is the matrix

$$\mathbf{A}_8 = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{pmatrix}$$



```
N=8;
m=[1:N]'*ones(1,N); n=m';
% can also use cos instead of sin
%A=sqrt(2/N)*cos(pi*(2*(n-1)+1).*(m-1)/(2*N));
A=sqrt(2/N)*sin(pi*(2*(n-1)+1).*(m-1)/(2*N));
A(1,:)=sqrt(1/N);
C=A';
for row=1:N
  for col=1:N
    B=C(:,row)*C(:,col).'; %tensor product
    subplot(N,N,(row-1)*N+col)
    imagesc(B)
    drawnow
  end
end
```

Figure Ans.31: The 64 Basis Images of the Two-Dimensional DST.



```
M=3; N=2^M; H=[1 1; 1 -1]/sqrt(2);
for m=1:(M-1) % recursion
  H=[H H; H -H]/sqrt(2);
end
A=H';
map=[1 5 7 3 4 8 6 2]; % 1:N
for n=1:N, B(:,n)=A(:,map(n)); end;
A=B;
sc=1/(max(abs(A(:))).^2); % scale factor
for row=1:N
  for col=1:N
    BI=A(:,row)*A(:,col).'; % tensor product
    subplot(N,N,(row-1)*N+col)
    oe=round(BI*sc); % results in -1, +1
    imagesc(oe), colormap([1 1 1; .5 .5 .5; 0 0 0])
    drawnow
  end
end
```

Figure Ans.32: The 8×8 WHT Basis Images and Matlab Code.

4.14: The average of vector $\mathbf{w}^{(i)}$ is zero, so Equation (4.17) yields

$$\left(\mathbf{W}\cdot\mathbf{W}^T\right)_{jj} = \sum_{i=1}^{k} w_j^{(i)} w_j^{(i)} = \sum_{i=1}^{k}\left(w_j^{(i)} - 0\right)^2 = \sum_{i=1}^{k}\left(c_i^{(j)} - 0\right)^2 = k\,\text{Variance}(\mathbf{c}^{(j)}).$$

4.15: The arguments of the cosine functions used by the DCT are of the form $(2x+1)i\pi/16$, where i and x are integers in the range $[0,7]$. Such an argument can be written in the form $n\pi/16$, where n is an integer in the range $[0, 15\times 7]$. Since the cosine function is periodic, it satisfies $\cos(32\pi/16) = \cos(0\pi/16)$, $\cos(33\pi/16) = \cos(\pi/16)$, and so on. As a result, only the 32 values $\cos(n\pi/16)$ for $n = 0, 1, 2, \ldots, 31$ are needed. The author is indebted to V. Saravanan for pointing out this feature of the DCT.

4.16: When the following *Mathematica*™ code is applied to Table 4.53b it creates a data unit with 64 pixels, all having the value 140, which is the average value of the pixels in the original data unit 4.50.

```
Cr[i_]:=If[i==0, 1/Sqrt[2], 1];
IDCT[x_,y_]:={(1/4)Sum[Cr[i]Cr[j]G[[i+1,j+1]]Quant[[i+1,j+1]]*
 Cos[(2x+1)i Pi/16]Cos[(2y+1)j Pi/16], {i,0,7,1}, {j,0,7,1}]};
```

4.17: Selecting $R = 1$ has produced the quantization coefficients of Table Ans.33a and the quantized data unit of Table Ans.33b. This table has 18 nonzero coefficients which, when used to reconstruct the original data unit, produce Table Ans.34, only a small improvement over Table 4.51.

4.18: The zigzag sequence is $1118, 2, 0, -2, \underbrace{0, \ldots, 0}_{13}, -1, 0, \ldots$ (there are only four nonzero coefficients).

4.19: Perhaps the simplest way is to manually figure out the zigzag path and to record it in an array `zz` of structures, where each structure contains a pair of coordinates for the path as shown, e.g., in Figure Ans.35.

If the two components of a structure are `zz.r` and `zz.c`, then the zig-zag traversal can be done by a loop of the form :

```
for (i=0; i<64; i++){
row:=zz[i].r; col:=zz[i].c
...data_unit[row][col]...}
```

4.20: It is located in row 3 column 5, so it is encoded as 1110|101.

4.21: Thirteen consecutive zeros precede this coefficient, so Z = 13. The coefficient itself is found in Table 4.55 in row 1, column 0, so R = 1 and C = 0. Assuming that the Huffman code in position (R, Z) = (1, 13) of Table 4.58 is 1110101, the final code emitted for 1 is 1110101|0.

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15

(a)

1118.	3	2	-1	1	0	0	0
-1	0	1	1	1	0	0	0
-3	-2	0	-2	0	0	0	0
-1	-2	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	-1	1	0	0	0
0	0	0	1	0	0	0	0

(b)

Table Ans.33: (a): The Quantization table $1 + (i + j) \times 1$. (b): Quantized Coefficients Produced by (a).

139	139	138	139	139	138	139	140
140	140	140	139	139	139	139	140
142	141	140	140	140	139	139	140
142	141	140	140	140	140	139	139
142	141	140	140	140	140	140	139
140	140	140	140	139	139	139	141
140	140	140	140	139	139	140	140
139	140	141	140	139	138	139	140

Table Ans.34: Restored data unit of Table 4.50.

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

Figure Ans.35: Coordinates for the Zigzag Path.

4.22: 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 $\underbrace{11\ldots1}_{n}00$.

4.23: Simple growth rules make for a more natural progressive growth of the image. They make it easier for a person watching the image develop on the screen to decide if and when to stop the decoding process and accept or discard the image.

4.24: The only specification that depends on the particular bits assigned to the two colors is Equation (4.19). All the other parts of JBIG are independent of the bit assignment.

4.25: For the 16-bit template of Figure 4.89a the relative coordinates are

$$A_1 = (3, -1), \quad A_2 = (-3, -1), \quad A_3 = (2, -2), \quad A_4 = (-2, -2).$$

For the 13-bit template of Figure 4.89b the relative coordinates of A_1 are $(3, -1)$. For the 10-bit templates of Figure 4.89c,d the relative coordinates of A_1 are $(2, -1)$.

4.26: It produces better compression in cases where the text runs vertically.

4.27: Going back to step 1 we have the same points participate in the partition for each codebook entry (this happens because our points are concentrated in four distinct regions, but in general a partition $P_i^{(k)}$ may consist of different image blocks in each iteration k). The distortions calculated in step 2 are summarized in Table Ans.37. The average distortion $D_i^{(1)}$ is

$$D^{(1)} = (277{+}277{+}277{+}277{+}50{+}50{+}200{+}117{+}37{+}117{+}162{+}117)/12 = 163.17,$$

much smaller than the original 603.33. If step 3 indicates no convergence, more iterations should follow (Exercise 4.28), reducing the average distortion and improving the values of the four codebook entries.

4.28: 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.

4.29: It is $4^{-8} \approx 0.000015$ the area of the entire space.

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

4.31: Here are steps 4 and 5. Step 4: Point $(2, 0)$ is popped out of the GPP. The pixel value at this position is 7. The best match for this point is with the dictionary entry containing 7. The encoder outputs the pointer 7. The match does not have any concave corners, so we push the point on the right of the matched block, $(2, 1)$, and the point below it, $(3, 0)$, into the GPP. The GPP now contains points $(2, 1)$, $(3, 0)$, $(0, 2)$, and $(1, 1)$. The dictionary is updated by appending to it (at location 18) the block $\boxed{\begin{matrix}4\\7\end{matrix}}$.

Step 5: Point $(1, 1)$ is popped out of the GPP. The pixel value at this position is 5. The best match for this point is with the dictionary entry containing 5. The encoder outputs the pointer 5. The match does not have any concave corners, so

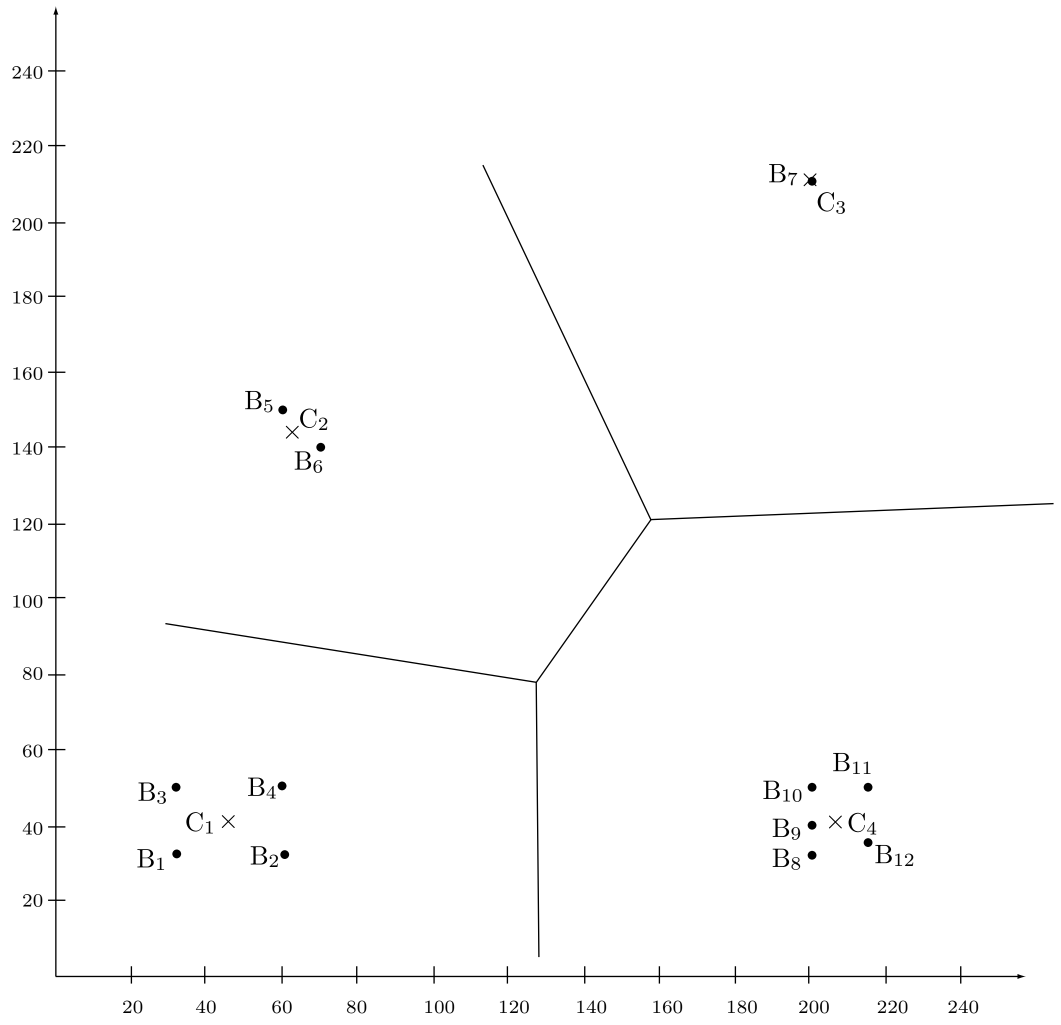


Figure Ans.36: Twelve Points and Four Codebook Entries $C_i^{(1)}$.

$$\begin{array}{rrr}
\text{I:} & (46-32)^2+(41-32)^2=277, & (46-60)^2+(41-32)^2=277,\\
 & (46-32)^2+(41-50)^2=277, & (46-60)^2+(41-50)^2=277,\\
\text{II:} & (65-60)^2+(145-150)^2=\ 50, & (65-70)^2+(145-140)^2=\ 50,\\
\text{III:} & (210-200)^2+(200-210)^2=200, & \\
\text{IV:} & (206-200)^2+(41-32)^2=117, & (206-200)^2+(41-40)^2=\ 37,\\
 & (206-200)^2+(41-50)^2=117, & (206-215)^2+(41-50)^2=162,\\
 & (206-215)^2+(41-35)^2=117. &
\end{array}$$

Table Ans.37: Twelve Distortions For $k = 1$.

we push the point to the right of the matched block, $(1,2)$, and the point below it, $(2,1)$, into the GPP. The GPP contains points $(1,2)$, $(2,1)$, $(3,0)$, and $(0,2)$. The dictionary is updated by appending to it (at locations 19, 20) the two blocks $\boxed{\begin{smallmatrix}2\\5\end{smallmatrix}}$ and $\boxed{4|5}$.

4.32: It may simply be too long. When compressing text, each symbol is normally 1-byte long (2 bytes in Unicode). However, images with 24-bit pixels are very common, and a 16-pixel block in such an image is 48-bytes long.

4.33: If the encoder uses a $(2,1,k)$ general unary code, then the value of k should be included in the header.

4.34: The mean and standard deviation are $\bar{p}=115$ and $\sigma=77.93$, respectively. The counts become $n^+=n^-=8$, and Equations (4.27) are solved to yield $p^+=193$ and $p^-=37$. The original block is compressed to the 16 bits

$$\begin{pmatrix}1&0&1&1\\1&0&0&1\\1&1&0&1\\0&0&0&0\end{pmatrix},$$

and the two 8-bit values 37 and 193.

4.35: Table Ans.38 summarizes the results. Notice how a 1-pixel with a context of 00 is assigned high probability after being seen 3 times.

#	Pixel	Context	Counts	Probability	New counts
5	0	10=2	1,1	1/2	2,1
6	1	00=0	1,3	3/4	1,4
7	0	11=3	1,1	1/2	2,1
8	1	10=2	2,1	1/3	2,2

Table Ans.38: Counts and Probabilities for Next four Pixels.

4.36: Such a thing is possible for the encoder but not for the decoder. A compression method using "future" pixels in the context is useless because its output would be impossible to decompress.

4.37: A 2nd order Markov model. In such a model the value of the current data item depends on just two of its past neighbors, not necessarily the two immediate ones.

4.38: 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.39 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 4.112, 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 4.112, the code 10|100.

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.39: The Codes for a Central Region.

4.39: Because the decoder has to resolve ties in the same way as the encoder.

4.40: Because this will result in a weighted sum whose value is in the same range as the values of the pixels. If pixel values are, e.g., in the range $[0, 15]$ and the weights add up to 2, a prediction may result in values of up to 30.

4.41: Each of the three weights 0.0039, -0.0351, and 0.3164 is used twice. The sum of the weights is thus 0.5704 and the result of dividing each weight by this sum is 0.0068, -0.0615, and 0.5547. It is easy to verify that the sum of the renormalized weights $2(0.0068 - 0.0615 + 0.5547)$ equals 1.

4.42: One such example is an archive of static images. NASA has a large archive of images taken by various satellites. They should be kept highly compressed, but they never change so each image has to be compressed only once. A slow encoder is therefore acceptable but a fast decoder is certainly handy. Another example is an art collection. Many museums have digitized their collections of paintings, and those are also static.

4.43: 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 $I[2i-2,2j]$

4.44: When the decoder inputs the 5, it knows that the difference between p (the pixel being decoded) and the reference pixel starts at position 6 (counting from left). Since bit 6 of the reference pixel is 0, that of p must be 1.

4.45: Yes, but compression would suffer. One way to apply this method is to separate each byte into two 4-bit pixels and encode each pixel separately. This approach is bad since the prefix and suffix of a 4-bit pixel may often consist of more than 4 bits. Another approach is to ignore the fact that a byte contains two pixels, and use the method as originally described. This may still compress the image, but is not very efficient, as the following example illustrates.

Example: The two bytes 1100|1101 and 1110|1111 represent four pixels, each differing from its immediate neighbor by its least significant bit. The four pixels thus have similar colors (or grayscales). Comparing consecutive pixels results in prefixes of 3 or 2, but comparing the 2 bytes produces the prefix 2.

4.46: Because this produces a value X in the same range as A, B, and C. If the weights were, for instance, 1, 100, and 1, X would have much bigger values than any of the three pixels.

4.47: 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.40 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.

4.48: Figure Ans.41a,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.

```
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.40: Solving For Three Weights.

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.41: (a) Bands L_2 and H_2. (b) Bands L_3 and H_3. (c) Bands L_4 and H_4.

4.49: There are 16 values. The value 0 appears nine times, and each of the other seven values appears once. The entropy is thus

$$-\sum p_i \log_2 p_i = -\frac{9}{16}\log_2\left(\frac{9}{16}\right) - 7\frac{1}{16}\log_2\left(\frac{1}{16}\right) \approx 2.2169.$$

Not very small, since seven of the 16 values have the same probability. In practice, values of an H_i difference band tend to be small, are both positive and negative, and are concentrated around zero, so their entropy is small.

4.50: Because the decoder needs to know how the encoder estimated X for each H_i difference value. If the encoder uses one of three methods for prediction, it has to precede each difference value in the compressed stream with a code that tells the decoder which method was used. Such a code can have variable size (for example, 0, 10, 11) but even adding just one or two bits to each prediction reduces compression performance significantly, since each H_i value needs to be predicted, and the number of these values is close to the size of the image.

4.51: The binary tree is shown in Figure Ans.42. From this tree, it is easy to see that the progressive image file is 3 6|5 7|7 7 10 5.

4.52: They are shown in Figure Ans.43

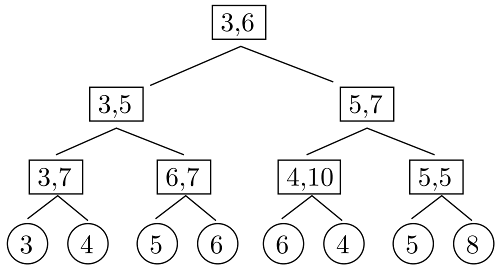


Figure Ans.42: A Binary Tree For An 8-Pixel Image.

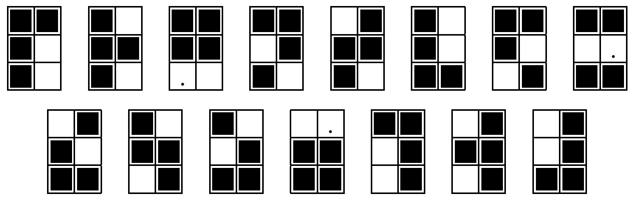


Figure Ans.43: The 15 6-Tuples With Two White Pixels.

4.53: No. An image with little or no correlation between the pixels will not compress with quadrisection, even though the size of the last matrix is always small. Even without knowing the details of quadrisection we can confidently state that such an image will produce a sequence of matrices M_j with few or no identical rows. In the extreme case, where the rows of any M_j are all distinct, each M_j will have four times the number of rows of its predecessor. This will create indicator vectors I_j that get longer and longer, thereby increasing the size of the compressed stream and reducing the overall compression performance.

4.54: This is just the concatenation of the 12 distinct rows of M_4

$$M_5^T = (0000|0001|1111|0011|1010|1101|1000|0111|1110|0101|1011|0010).$$

4.55: M_4 has four columns, so it can have at most 16 distinct rows, implying that M_5 can have at most $4\times 16 = 64$ elements.

4.56: The decoder has to read the entire compressed stream, save it in memory, and start the decoding with L_5. Grouping the eight elements of L_5 yields the four distinct elements 01, 11, 00, and 10 of L_4, so I_4 can now be used to reconstruct L_4. The four zeros of I_4 correspond to the four distinct elements of L_4, and the remaining 10 elements of L_4 can be constructed from them. Once L_4 has been constructed, its 14 elements are grouped to form the seven distinct elements of L_3. These elements are 0111, 0010, 1100, 0110, 1111, 0101, and 1010, and they correspond to the seven zeros of I_3. Once L_3 has been constructed, its eight elements are grouped to form the four distinct elements of L_2. Those four elements are the entire L_2 since I_2 is all zero. Reconstructing L_1 and L_0 is now trivial.

4.57: 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 $L_5 = (0, 1, 1, 0)$.

4.58: Using a Hilbert curve produces the 21 runs 5, 1, 2, 1, 2, 7, 3, 1, 2, 1, 5, 1, 2, 2, 11, 7, 2, 1, 1, 1, 6. RLE produces the 27 runs 0, 1, 7, eol, 2, 1, 5, eol, 5, 1, 2, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 4, 1, 3, eol, 3, 1, 4, eol.

4.59: The string 2011.

4.60: 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 4.152c.

4.61: This is shown in Figure Ans.44.

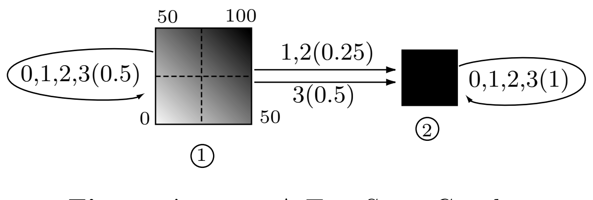


Figure Ans.44: A Two-State Graph.

4.62: The function is

$$f(x,y)=\begin{cases} x+y, & \text{if } x+y\le 1,\\ 0, & \text{if } x+y>1.\end{cases}$$

4.63: The graph has five states, so each transition matrix is of size 5×5. Direct computation from the graph yields

$$W_0=\begin{pmatrix}0&1&0&0&0\\0&0.5&0&0&0\\0&0&0&0&1\\0&-0.5&0&0&1.5\\0&-0.25&0&0&1\end{pmatrix},\quad W_3=\begin{pmatrix}0&0&0&0&0\\0&0&0&0&1\\0&0&0&0&0\\0&0&0&0&0\\0&-1&0&0&1.5\end{pmatrix},$$

$$W_1=W_2=\begin{pmatrix}0&0&1&0&0\\0&0.25&0&0&0.5\\0&0&0&1.5&0\\0&0&-0.5&1.5&0\\0&-0.375&0&0&1.25\end{pmatrix}.$$

The final distribution is the five-component vector

$$F=(0.25,0.5,0.375,0.4125,0.75)^T.$$

4.64: One way to specify the center is to construct string $033\ldots3$. This yields

$$\begin{aligned}\psi_i(03\ldots3)&=(W_0\cdot W_3\cdots W_3\cdot F)_i\\ &=\begin{pmatrix}0.5&0\\0&1\end{pmatrix}\begin{pmatrix}0.5&0.5\\0&1\end{pmatrix}\cdots\begin{pmatrix}0.5&0.5\\0&1\end{pmatrix}\begin{pmatrix}0.5\\1\end{pmatrix}_i\\ &=\begin{pmatrix}0.5&0\\0&1\end{pmatrix}\begin{pmatrix}0&1\\0&1\end{pmatrix}\begin{pmatrix}0.5\\1\end{pmatrix}_i=\begin{pmatrix}0.5\\1\end{pmatrix}_i.\end{aligned}$$

4.65: Figure Ans.45 shows Matlab code for such a matrix.

```
dim=256;
for i=1:dim
 for j=1:dim
   m(i,j)=(i+j-2)/(2*dim-2);
 end
end
m
```

Figure Ans.45: Matlab Code For A Matrix $m_{i,j}=(i+j)/2$.

4.66: A direct examination of the graph yields the ψ_i values

$$\begin{aligned}
\psi_i(0) &= (W_0\cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)^T_i,\\
\psi_i(01) &= (W_0\cdot W_1\cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)^T_i,\\
\psi_i(1) &= (W_1\cdot F)_i = (0.375, 0.5, 0.61875, 0.43125, 0.75)^T_i,\\
\psi_i(00) &= (W_0\cdot W_0\cdot F)_i = (0.25, 0.125, 0.625, 0.8125, 0.5625)^T_i,\\
\psi_i(03) &= (W_0\cdot W_3\cdot F)_i = (0.75, 0.375, 0.625, 0.5625, 0.4375)^T_i,\\
\psi_i(3) &= (W_3\cdot F)_i = (0, 0.75, 0, 0, 0.625)^T_i,
\end{aligned}$$

and the f values

$$\begin{aligned}
&f(0) = I\cdot\psi(0) = 0.5, \quad f(01) = I\cdot\psi(01) = 0.5, \quad f(1) = I\cdot\psi(1) = 0.375,\\
&f(00) = I\cdot\psi(00) = 0.25, \quad f(03) = I\cdot\psi(03) = 0.75, \quad f(3) = I\cdot\psi(3) = 0.
\end{aligned}$$

4.67: Figure Ans.46a,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.

4.68: The transformation can be written $(x,y) \rightarrow (x, -x+y)$, so $(1,0) \rightarrow (1,-1)$, $(3,0) \rightarrow (3,-3)$, $(1,1) \rightarrow (1,0)$ and $(3,1) \rightarrow (3,-2)$. The original rectangle is thus transformed into a parallelogram.

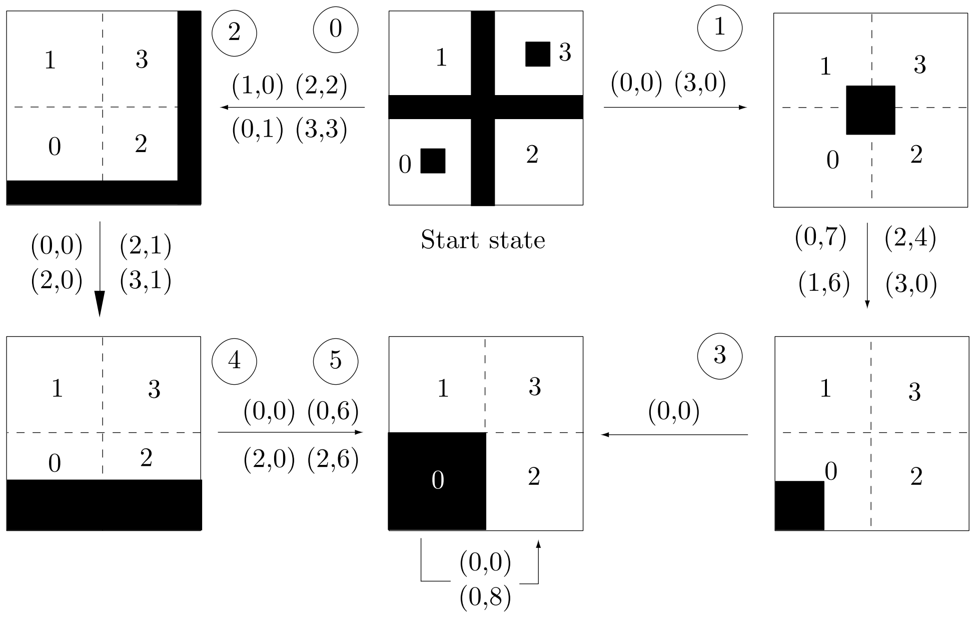
4.69: The two sets of transformations produce the same Sierpiński triangle but at different sizes and orientations.

4.70: 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.47. The result is the familiar Sierpiński gasket but in a different orientation.

4.71: There are $32\times 32 = 1,024$ ranges and $(256-15)\times(256-15) = 58,081$ domains. The total number of steps is thus $1,024 \times 58,081\times 8 = 475,799,552$, still a large number. PIFS is thus computationally intensive.

4.72: Suppose that the image has G levels of gray. A good measure of data loss is the difference between the value of an average decompressed pixel and its correct value, expressed in number of gray levels. For large values of G (hundreds of gray levels) an average difference of $\log_2 G$ gray levels (or fewer) is considered satisfactory.

5.1: A written page. A person can place marks on a page and read them later as text, mathematical expressions, and drawings. This is a two-dimensional representation of the information on the page. The page can later be scanned by, e.g., a fax machine, and its contents transmitted as a one-dimensional stream of bits that constitute a different representation of the same information.



(a)

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

(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.46: A GFA for Exercise 4.67.

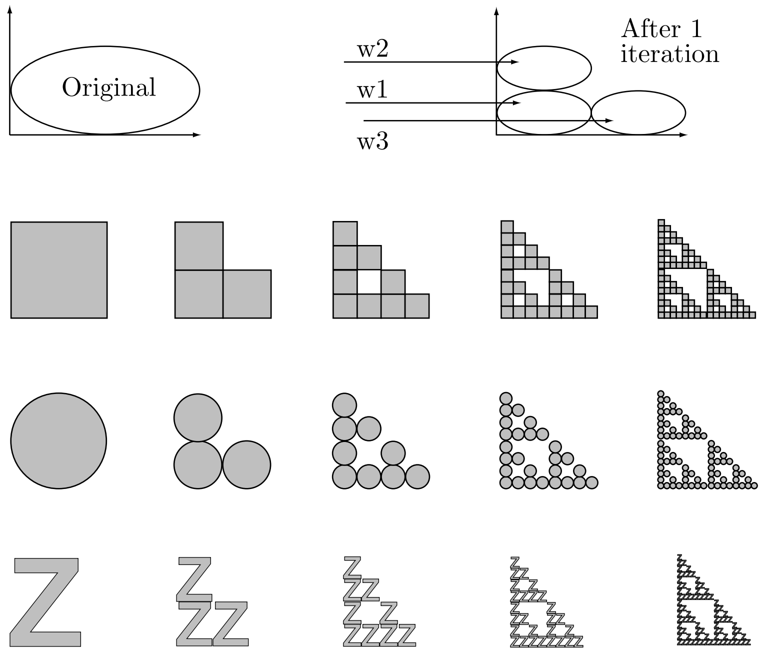


Figure Ans.47: Another Sierpiński Gasket.

5.2: Figure Ans.48 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.

The table of Figure Ans.49 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 $f(t)$.

5.3: They are shown in Figure 5.11c.

5.4: Figure Ans.50a shows a simple, 8×8 image with one diagonal line above the main diagonal. Figure Ans.50b,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.

5.5: The average can easily be calculated. It turns out to be 131.375, which is exactly 1/8 of 1051. The reason the top-left transform coefficient is eight times the

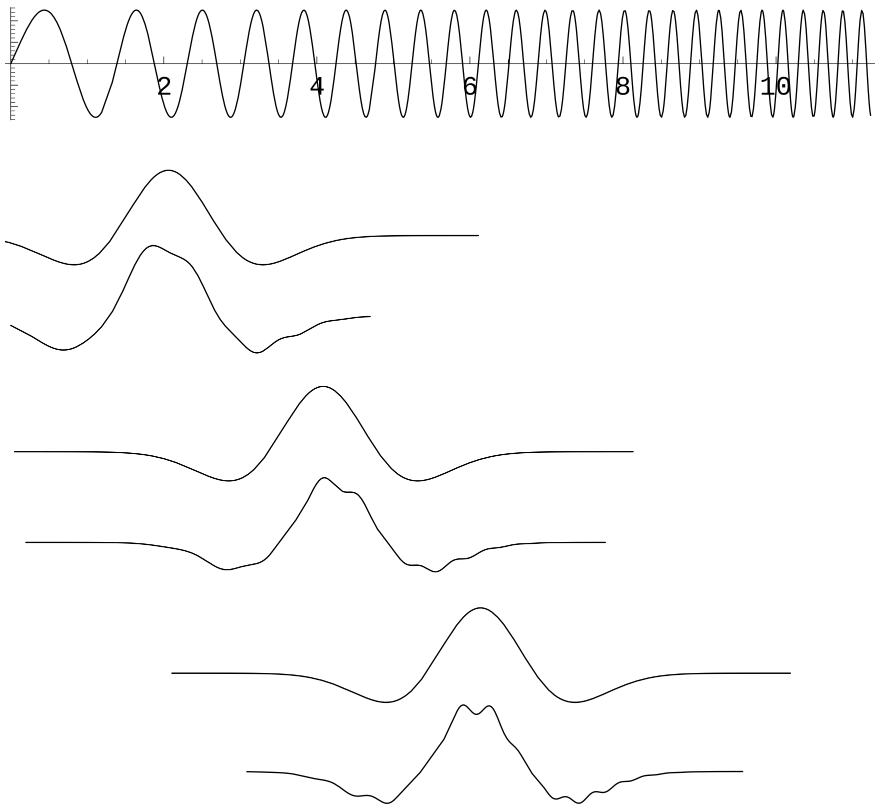


Figure Ans.48: An Inner Product for $a = 1$ and $b = 2, 4, 6$.

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

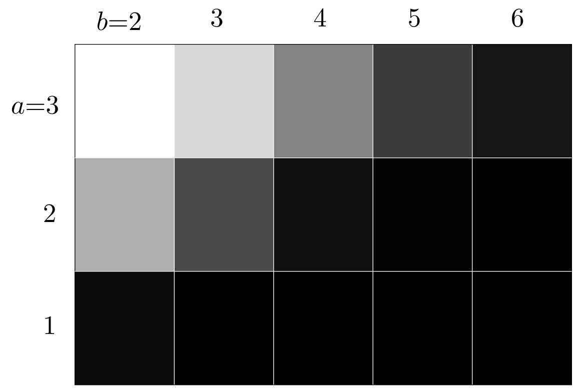


Figure Ans.49: Fifteen Values And a Density Plot of $W(a, b)$.

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

(a) (b) (c)

Figure Ans.50: The Subband Decomposition of a Diagonal Line.

average is that the Matlab code that did the calculations uses $\sqrt{2}$ instead of 2 (see function `individ(n)` in Figure 5.22).

5.6: Figure Ans.51a–c shows the results of reconstruction from 3277, 1639, and 820 coefficients, respectively. Despite the heavy loss of wavelet coefficients, only a very small loss of image quality is noticeable. The number of wavelet coefficients is, of course, the same as the image resolution $128\times128 = 16,384$. Using 820 out of 16,384 coefficients corresponds to discarding 95% of the smallest of the transform coefficients (notice, however, that some of the coefficients were originally zero, so the actual loss may amount to less than 95%).

5.7: The Matlab code of Figure Ans.52 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.

5.8: A simple example of such input is the vector of alternating values $x = (\ldots,1,-1,1,-1,1,\ldots)$.

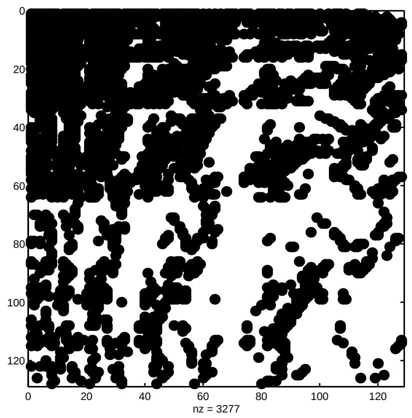
5.9: For eight-tap filters, rules 1 and 2 imply

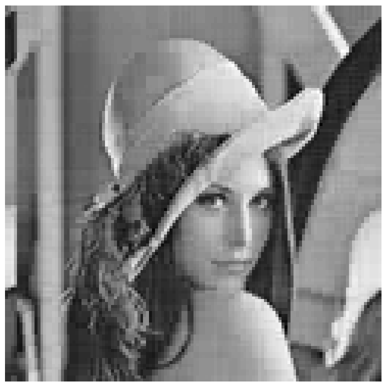
$$
\begin{gathered}
h_0^2(0)+h_0^2(1)+h_0^2(2)+h_0^2(3)+h_0^2(4)+h_0^2(5)+h_0^2(6)+h_0^2(7)=1,\\
h_0(0)h_0(2)+h_0(1)h_0(3)+h_0(2)h_0(4)+h_0(3)h_0(5)+h_0(4)h_0(6)+h_0(5)h_0(7)=0,\\
h_0(0)h_0(4)+h_0(1)h_0(5)+h_0(2)h_0(6)+h_0(3)h_0(7)=0,\\
h_0(0)h_0(6)+h_0(1)h_0(7)=0,
\end{gathered}
$$

and rules 3–5 yield

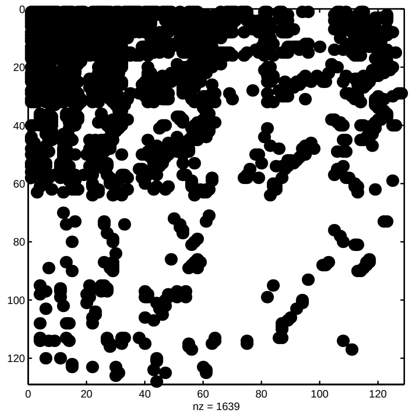
$$
\begin{aligned}
f_0&=\bigl(h_0(7),h_0(6),h_0(5),h_0(4),h_0(3),h_0(2),h_0(1),h_0(0)\bigr),\\
h_1&=\bigl(-h_0(7),h_0(6),-h_0(5),h_0(4),-h_0(3),h_0(2),-h_0(1),h_0(0)\bigr),\\
f_1&=\bigl(h_0(0),-h_0(1),h_0(2),-h_0(3),h_0(4),-h_0(5),h_0(6),-h_0(7)\bigr).
\end{aligned}
$$

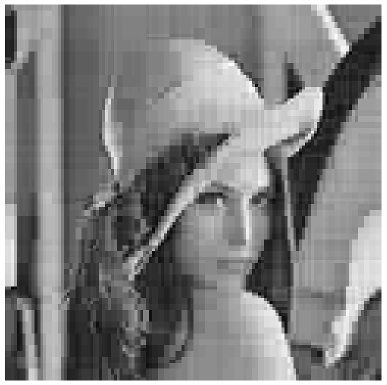
The eight coefficients are listed in Table 5.35 (this is the Daubechies D8 filter).



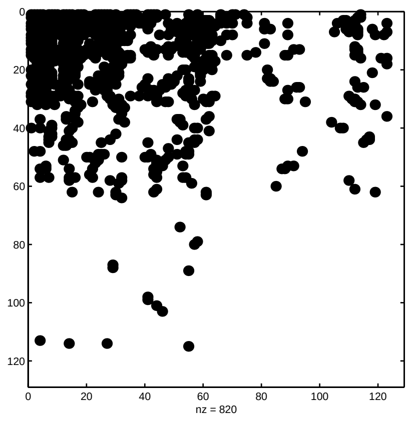


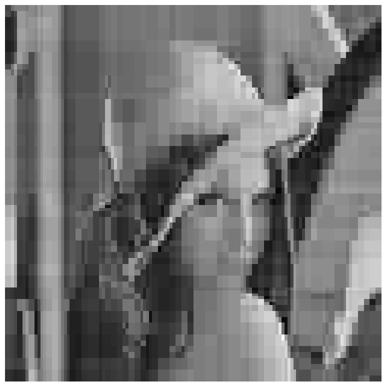
(a)





(b)





(c)

Figure Ans.51: Three Lossy Reconstructions of the 128×128 Lena Image.

```
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.52: Code and Results For the Calculation of Matrix W and Transform $W{\cdot}I{\cdot}W^T$.

5.10: Figure Ans.53 lists the Matlab code of the inverse wavelet transform function `iwt1(wc,coarse,filter)` and a test.

5.11: Figure Ans.54 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.

5.12: This is written `a-=b/2;  b+=a;`.

```
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.53: 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)
```

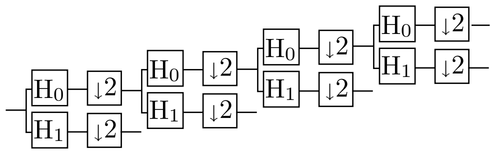
5.13: We sum Equation (5.13) over all the values of l to get

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

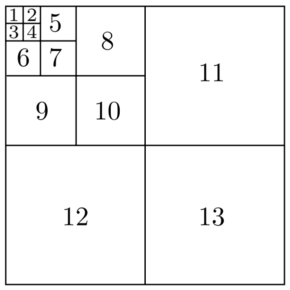
Therefore, the average of set s_{j-1} equals

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

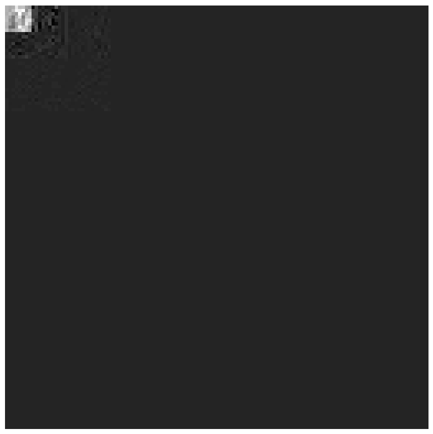
the average of set s_j.



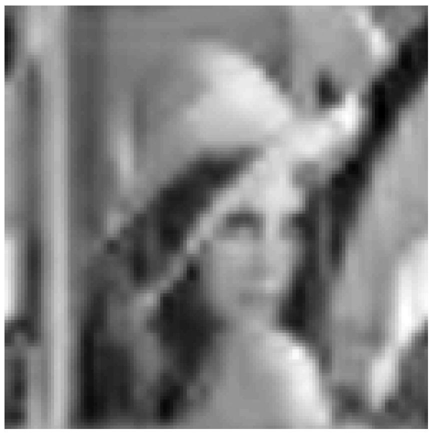
(a)



(b)



(c)



(d)

Figure Ans.54: 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.54.

5.14: The code of Figure Ans.55 produces the expression

$$0.0117\mathbf{P}_1 - 0.0977\mathbf{P}_2 + 0.5859\mathbf{P}_3 + 0.5859\mathbf{P}_4 - 0.0977\mathbf{P}_5 + 0.0117\mathbf{P}_6.$$

```
Clear[p,a,b,c,d,e,f];
p[t_]:=a t^5+b t^4+c t^3+d t^2+e t+f;
Solve[{p[0]==p1, p[1/5.]==p2, p[2/5.]==p3,
p[3/5.]==p4, p[4/5.]==p5, p[1]==p6}, {a,b,c,d,e,f}];
sol=ExpandAll[Simplify[%]];
Simplify[p[0.5] /.sol]
```

Figure Ans.55: Code for a Degree-5 Interpolating Polynomial.

5.15: The Matlab code of Figure Ans.56 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 the discussion in the text assumes arrays indexed from 0. This causes the difference in index values in Figure Ans.56.

```
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.56: Matlab Code For Forward and Inverse IWT.

5.16: Images g_0 through g_5 will have dimensions

$$(3\cdot2^5+1\times 4\cdot2^5+1) = 97\times129, \quad 49\times65, \quad 25\times33, \quad 13\times17, \text{ and } 7\times9.$$

5.17: 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

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

and the remaining seven are not changed.

5.18: The simple equation $10\times2^{20}\times8 = (500x)\times(500x)\times8$ is solved to yield $x^2 = 40$ square inches. If the card is square, it is approximately 6.32 inches on a side. Such a card has 10 rolled impressions (about 1.5×1.5 each), two plain impressions of the thumbs (about 0.875×1.875 each), and simultaneous impressions of both hands (about 3.125×1.875 each). All the dimensions are in inches.

5.19: The bit of 10 is encoded, as usual, in pass 2. The bit of 1 is encoded in pass 1 since this coefficient is still insignificant but has significant neighbors. This bit is 1, so coefficient 1 becomes significant (a fact that is not used later). Also, this bit is the first 1 of this coefficient, so the sign bit of the coefficient is encoded following this bit. The bits of coefficients 3 and -7 are encoded in pass 2 since these coefficients are significant.

6.1: It is easy to calculate that $525\cdot4/3 = 700$ pixels.

6.2: 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.

6.3: There aren't many, but here are three examples: (1) Surveillance camera, (2) an old, silent movie being restored and converted from film to video, and (3) a video presentation taken underwater.

6.4: 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.

6.5: Imagine a camera panning from left to right. New objects will enter the field of view from the right all the time. A block on the right side of the frame may thus contain objects that did not exist in the previous frame.

6.6: 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 $(7,4)$.

6.7: The picture consists of 18×18 macroblocks, and each macroblock constitutes six 8×8 blocks of samples. The total number of samples is, thus, $18\times 18\times 6\times 64 = 124,416$.

6.8: The size category of zero is 0, so code 100 is emitted, followed by zero bits. The size category of 4 is 3, so code 110 is first emitted, followed by the three least-significant bits of 4, which are 100.

6.9: The zigzag sequence is

$$118, 2, 0, -2, \underbrace{0, \ldots, 0}_{13}, -1, 0, \ldots.$$

The run-level pairs are $(0,2)$, $(1,-2)$, and $(13,-1)$, so the final codes are (notice the sign bits following the run-level codes)

$$0100\,0|000110\,1|00100000\,1|10,$$

(without the vertical bars).

6.10: There are no nonzero coefficients, no run-level codes, just the 2-bit EOB code. However, in nonintra coding, such a block is encoded in a special way.

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

7.2: The period of a wave is its speed divided by its frequency. For sound we get

$$\frac{34380\,\text{cm/s}}{22000\,\text{Hz}} = 1.562\text{ cm}, \qquad \frac{34380}{20} = 1719\text{ cm}.$$

7.3: The (base-10) logarithm of x is a number y such that $10^y = x$. The number 2 is the logarithm of 100 since $10^2 = 100$. Similarly, 0.3 is the logarithm of 2 since $10^{0.3} = 2$. Also, The base-b logarithm of x is a number y such that $b^y = x$ (for any real $b > 1$).

7.4: Each doubling of the sound intensity increases the dB level by 3. Therefore, the difference of 9 dB $(3+3+3)$ between A and B corresponds to three doublings of the sound intensity. Thus, source B is $2\cdot 2\cdot 2 = 8$ times louder than source A.

7.5: Each 0 would result in silence and each sample of 1, in the same tone. The result would be a nonuniform buzz. Such sounds were common on early personal computers.

7.6: The experiment should be repeated with several persons, preferably of different ages. The person should be placed in a sound insulated chamber and a pure tone of frequency f should be played. The amplitude of the tone should be gradually increased from zero until the person can just barely hear it. If this happens at a decibel value d, point (d, f) should be plotted. This should be repeated for many frequencies until a graph similar to Figure 7.4a is obtained.

7.7: Imagine that the sound being compressed contains one second of a pure tone (just one frequency). This second will be digitized to 44,100 consecutive samples per channel. The samples indicate amplitudes, so they don't have to be the same. However, after filtering, only one subband (although in practice perhaps two subbands) will have nonzero signals. All the other subbands correspond to different frequencies, so they will have signals that are either zero or very close to zero.

7.8: 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.

7.9: The decoder has to decode $44{,}100/384 \approx 114.84$ frames per second. Thus, each frame has to be decoded in approximately 8.7 ms. In order to output 114.84 frames in 64,000 bits, each frame must have $B_f = 557$ bits available to encode it. The number of slots per frame is thus $557/32 \approx 17.41$. Thus, the last (18th) slot is not full and has to padded.

7.10: Table 7.31 shows that the scale factor is 111 and the select information is 2. The third rule in Table 7.32 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.

7.11: Typical layer II parameters are (1) a sampling rate of 48000 samples/s, (2) a bitrate of 64000 bits/s, and (3) 1152 quantized signals per frame. The decoder has to decode 48000/1152 = 41.66 frames per second. Thus, each frame has to be decoded in 24 ms. In order to output 41.66 frames in 64000 bits, each frame must have $B_f = 1536$ bits available to encode it.

7.12: A program to play `.mp3` files is an MPEG layer III *decoder*, not an encoder. Decoding is much simpler since it does not use a psychoacoustic model, nor does it have to anticipate preechoes and maintain the bit reservoir.

8.1: Because the original string S can be reconstructed from L but not from F.

8.2: A direct application of equation (8.1) eight more times produces:

S[10-1-2]=L[T^2[I]]=L[T[T^1[I]]]=L[T[7]]=L[6]=i;
S[10-1-3]=L[T^3[I]]=L[T[T^2[I]]]=L[T[6]]=L[2]=m;
S[10-1-4]=L[T^4[I]]=L[T[T^3[I]]]=L[T[2]]=L[3]=␣;
S[10-1-5]=L[T^5[I]]=L[T[T^4[I]]]=L[T[3]]=L[0]=s;
S[10-1-6]=L[T^6[I]]=L[T[T^5[I]]]=L[T[0]]=L[4]=s;
S[10-1-7]=L[T^7[I]]=L[T[T^6[I]]]=L[T[4]]=L[5]=i;
S[10-1-8]=L[T^8[I]]=L[T[T^7[I]]]=L[T[5]]=L[1]=w;
S[10-1-9]=L[T^9[I]]=L[T[T^8[I]]]=L[T[1]]=L[9]=s;

The original string "`swiss miss`" is indeed reproduced in S from right to left.

8.3: Figure Ans.57 shows the rotations of S and the sorted matrix. The last column, L of Ans.57b 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
ssssssssh s	shssssssss
sssssssshss	sshsssssss
ssssssshsss	ssshssssss
sssssshssss	sssshsssss
ssssshsssss	ssssshssss
sssshssssss	sssssshsss
ssshsssssss	ssssssshss
sshssssssss	sssssssshs
hsssssssss	sssssssssh
(a)	(b)

Figure Ans.57: Permutations of "`sssssssssh`".

8.4: The encoder starts at T[0], which contains 5. The first element of L is thus the last symbol of permutation 5. This permutation starts at position 5 of S, so its last element is in position 4. The encoder thus has to go through symbols S[T[i-1]] for $i = 0, \ldots, n-1$, where the notation $i-1$ should be interpreted cyclically (i.e., $0-1$ should be $n-1$). As each symbol S[T[i-1]] is found, it is compressed using move-to-front. The value of I is the position where T contains 0. In our example, T[8]=0, so I=8.

8.5: The first element of a triplet is the distance between two dictionary entries, the one best matching the content and the one best matching the context. In this case there is no content match, no distance, so any number could serve as the first element, 0 being the best (smallest) choice.

8.6: Because the three lines are sorted in ascending order. The bottom two lines of Table 8.13c are not in sorted order. This is why the "`zz...z`" part of string S must be preceded and followed by complementary bits.

8.7: The encoder places S between two entries of the sorted associative list and writes the (encoded) index of the entry above or below S on the compressed stream. The fewer the number of entries, the smaller this index, and the better the compression.

8.8: Context 5 is compared to the three remaining contexts 6, 7, and 8, and it is most similar to context 6 (they share a suffix of "b"). Context 6 is compared to 7 and 8 and, since they don't share any suffix, context 7, the shorter of the two, is selected. The remaining context 8 is, of course, the last one in the ranking. The final context ranking is

$$1 \to 3 \to 4 \to 0 \to 5 \to 6 \to 7 \to 8.$$

8.9: Equation (8.3) shows that the third "a" is assigned rank 1 and the "b" and "a" following it are assigned ranks 2 and 3, respectively.

8.10: Table Ans.58 shows the sorted contexts. Equation (Ans.2) shows the context ranking at each step.

$$\begin{array}{l}
\underset{u}{0}\,,\qquad \underset{u}{0}\to\underset{b}{2}\,,\qquad \underset{l}{1}\to\underset{b}{3}\to\underset{u}{0}\,,\\
\underset{u}{0}\to\underset{l}{2}\to\underset{a}{3}\to\underset{b}{4}\,,\qquad \underset{l}{2}\to\underset{a}{4}\to\underset{d}{1}\to\underset{b}{5}\to\underset{u}{0}\,,\\
\underset{i}{3}\to\underset{a}{5}\to\underset{l}{2}\to\underset{b}{6}\to\underset{d}{5}\to\underset{u}{0}\,.
\end{array}\tag{Ans.2}$$

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.58: Constructing the Sorted Lists For `ubladiu`.

8.11: All n_1 bits of string L_1 need be written on the output stream. This already shows that there is going to be no compression. String L_2 consists of n_1/k 1's, so all of it has to be written on the output stream. String L_3 similarly consists of n_1/k^2 1's, and so on. The size of the output stream is thus

$$n_1 + \frac{n_1}{k} + \frac{n_1}{k^2} + \frac{n_1}{k^3} + \cdots + \frac{n_1}{k^m} = n_1 \frac{k^{m+1}-1}{k^m(k-1)},$$

for some value of m. The limit of this expression, when $m \to \infty$, is $n_1 k/(k-1)$. For $k = 2$ this equals $2n_1$. For larger values of k this limit is always between n_1 and $2n_1$.

For the curious reader, here is how the sum above is calculated. Given the series

$$S = \sum_{i=0}^{m} \frac{1}{k^i} = 1 + \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^{m-1}} + \frac{1}{k^m},$$

we multiply both sides by $1/k$

$$\frac{S}{k} = \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^m} + \frac{1}{k^{m+1}} = S + \frac{1}{k^{m+1}} - 1,$$

and subtract

$$\frac{S}{k}(k-1) = \frac{k^{m+1}-1}{k^{m+1}} \to S = \frac{k^{m+1}-1}{k^m(k-1)}.$$

8.12: 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.

8.13: 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.

8.14: 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.

8.15: Yes, if they are located in different quadrants or subquadrants. Pixels 123 and 301, for example, are adjacent in Figure 8.27 but have different prefixes.

8.16: No, since all prefixes have the same probability of occurrence. In our example the prefixes are four bits long and all 16 possible prefixes have the same probability since a pixel may be located anywhere in the image. A Huffman code calculated for 16 equally-probable symbols has an average size of four bits per symbol, so nothing would be gained. The same is true for suffixes.

8.17: This is possible, but it places severe limitations on the size of the string. In order to rearrange a one-dimensional string into a four-dimensional cube, the string size should be 2^{4n}. If the string size happens to be $2^{4n}+1$, it has to be extended to $2^{4(n+1)}$, which doubles its size. It is possible to rearrange the string into a rectangular box, not just a cube, but then its size will have to be of the form $2^{n1}2^{n2}2^{n3}2^{n4}$ where the four ni's are integers.

8.18: The LZW method, which starts with the entire alphabet stored at the beginning of its dictionary. However, an adaptive version of LZW can be designed to compress words instead of individual characters.

8.19: Relative values (or *offsets*). Each (x, y) pair may specify the position of a character relative to its predecessor. This results in smaller numbers for the coordinates, and smaller numbers are easier to compress.

8.20: There may be such letters in other, "exotic" alphabets, but a more common example is a rectangular box enclosing text. The four rules comprising such a box should be considered a mark, but the text characters inside the box should be identified as separate marks.

8.21: Because this guarantees that the two probabilities will add up to 1.

8.22: Figure Ans.59 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 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 $B, C \to D$.

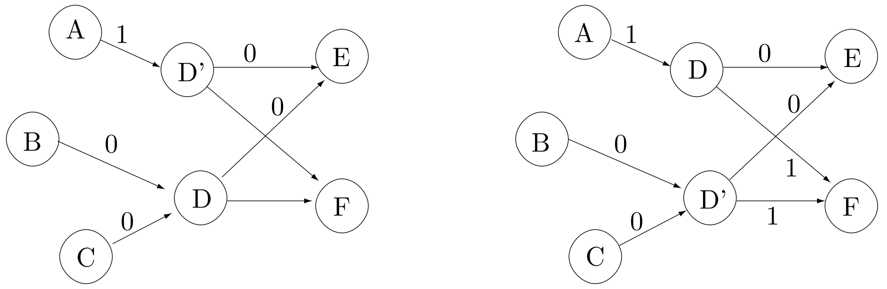


Figure Ans.59: New State D' Cloned.

8.23: Figure Ans.60 shows the new state 6. Its 1-output is identical to that of state 1, and its 0-output is a copy of the 0-output of state 3.

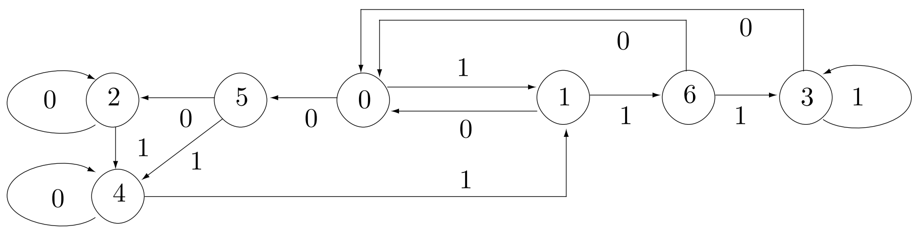


Figure Ans.60: State 6 Added.

8.24: A precise answer requires many experiments with various data files. A little thinking, though, shows that the larger k, the better the initial model that is created when the old one is discarded. Larger values of k thus minimize the loss of compression. However, very large values may produce an initial model that is already large and cannot grow much. The best value for k is therefore one that produces an initial model large enough to provide information about recent correlations in the data, but small enough so it has room to grow before it too has to be discarded.

8.25: The number of marked points can be written $8(1+2+3+5+8+13) = 256$ and the numbers in parentheses are the Fibonacci numbers.

8.26: It is very small. A segment pointing in direction D_i can be preceded by another segment pointing in the same direction only if the original curve is straight or very close to straight for more than 26 coordinate units (half the width of grid S_{13}).

8.27: A point has two coordinates. If each coordinate occupies 8 bits, then the use of Fibonacci numbers reduces the 16-bit coordinates to an 8-bit number, a compression ratio of 0.5. The use of Huffman codes can typically reduce this 8-bit number to (on average) a 4-bit code, and the use of the Markov model can perhaps cut this by another bit. The result is an estimated compression ratio of $3/16 = 0.1875$. If each coordinate is a 16-bit number, then this ratio improves to $3/32 = .09375$.

8.28: The resulting, shorter grammar is shown in Figure Ans.61. It is one rule and one symbol shorter.

Input	Grammar
`S` → `abcdbcabcdbc`	`S` → `CC`
	`A` → `bc`
	`C` → `aAdA`

Figure Ans.61: Improving the Grammar of Figure 8.42.

8.29: Generating rule `C` has made rule `B` underused (i.e., used just once).

8.30: Rule `S` consists of two copies of rule `A`. The first time rule `A` is encountered, its contents `aBdB` are sent. This involves sending rule `B` twice. The first time rule `B` is sent, its contents `bc` are sent (and the decoder does not know that the string `bc` it is receiving is the contents of a rule). The second time rule `B` is sent, the pair $(1, 2)$ is sent (offset 1, count 2). The decoder identifies the pair and uses it to set up the rule `1` → `bc`. Sending the first copy of rule `A` therefore amounts to sending `abcd`$(1, 2)$. The second copy of rule `A` is sent as the pair $(0, 4)$ since `A` starts at offset 0 in `S` and its length is 4. The decoder identifies this pair and uses it to set up the rule `2` → $\mathtt{a}\boxed{\mathtt{1}}\mathtt{d}\boxed{\mathtt{1}}$. The final result is therefore `abcd`$(1, 2)(0, 4)$.

8.31: 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.

8.32: 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.1: It is a combination of `"4x` and `"9`, or `"49`.

B.1: The values of X are listed below. Each has a probability of 1/8.

$$\begin{array}{cccc} X(HHH)=3, & X(HHT)=2, & X(HTH)=2, & X(HTT)=1, \\ X(THH)=2, & X(THT)=1, & X(TTH)=1, & X(TTT)=0. \end{array}$$

B.2: The definition of expectation implies that $E(X)=v$. The expected value of a constant random variable is the constant value.

B.3: Denote $m=E(X)$. From $\text{Var}(X)=E[(X-m)^2]$ we get

$$\begin{aligned} \text{Var}(X) &= E(X^2-2Xm+m^2) \\ &= E(X^2)-E(-2mX)+E(m^2) \\ &= E(X^2)-2mE(X)+m^2 \\ &= E(X^2)-2m^2+m^2 \\ &= E(X^2)-m^2. \end{aligned}$$

B.4: The probability of rolling a double-six with two dice is 1/36. The complement probability is 35/36. The probability of rolling a double-six in the first throw, or the second throw,..., or the 24th throw is

$$1-\underbrace{(35/36)(35/36)\cdots(35/36)}_{24}\approx 1-0.5086=0.4914.$$

For game B, the probability of rolling a six is 1/6, its complement is 5/6, so the probability of rolling a six in four tries is

$$1-(5/6)(5/6)(5/6)(5/6)\approx 1-0.482=0.518;$$

slightly higher than the winning probability of game A.

B.5: This problem is easy to solve intuitively. Once B has withdrawn, one of the remaining two companies will win the contract. All we have to do to find their new chances is to scale their old chances such that they add up to 1. Since the old chances add up to 3/5, they have to be scaled by 5/3 to bring their new sum to 1. The new chances are therefore $(2/5)(5/3)=2/3$ and $(1/5)(5/3)=1/3$.

Next, we use conditional probabilities to solve the same problem. We are looking for the conditional probabilities $P(A|\bar{B})$ and $P(C|\bar{B})$. We know that $P(B)$ was 2/5, so $P(\bar{B})=1-P(B)=3/5$. The quantity $P(A\cdot\bar{B})$ is the probability that A will win *and* B will not win. It equals $P(A)$, since if A wins, B cannot win. Equation (B.2) therefore yields the conditional probabilities

$$\begin{aligned} P(A|\bar{B}) &= \frac{P(A\cdot\bar{B})}{P(\bar{B})}=\frac{(2/5)}{(3/5)}=\frac{2}{3}, \\ P(C|\bar{B}) &= \frac{P(C\cdot\bar{B})}{P(\bar{B})}=\frac{(1/5)}{(3/5)}=\frac{1}{3}. \end{aligned}$$

B.6: Applying Bayes' theorem, the results are 0.24, 0.24, and 0.36, respectively. The knowledge that the student got an A made is less likely that he selected mathematics, more likely that he selected biology, and did not much affect the probabilities that he selected physics or chemistry.

B.7: Using appropriate mathematical software it is easy to obtain this integral separately for negative and nonnegative values of x.

$$\int L(V,x)\,dx = \begin{cases} \frac{-1}{V\exp\left(\sqrt{\frac{2}{V}}x\right)}, & x \geq 0, \\ \frac{1}{\sqrt{2V}}\exp\left(\sqrt{\frac{2}{V}}x\right), & x < 0. \end{cases}$$

C.1: A straight line segment from a to b is an example of a one-dimensional curve that passes through every point in the interval a, b.

C.2: The key is to realize that P_0 is a single point, and P_1 is constructed by connecting nine copies of P_0 with straight segments. Similarly, P_2 consists of nine copies of P_1, in different orientations, connected by segments (the dashed segments in Figure Ans.62).

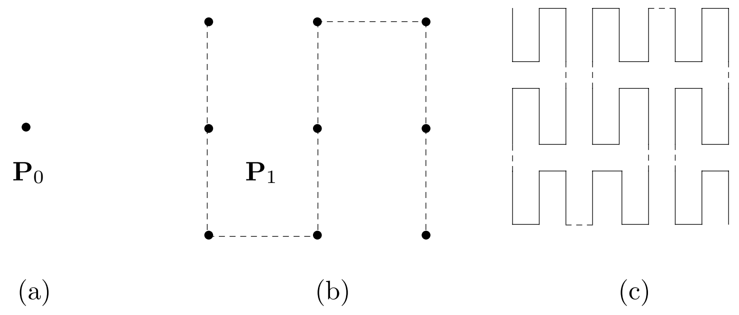


Figure Ans.62: The First Three Iterations of the Peano Curve.

C.3: Written in binary, the coordinates are $(1101, 0110)$. We iterate four times, each time taking 1 bit from the x coordinate and 1 bit from the y coordinate to form an (x, y) pair. The pairs are 10, 11, 01, 10. The first one yields [from Table C.12(1)] 01. The second pair yields [also from Table C.12(1)] 10. The third pair [from Table C.12(1)] 11, and the last pair [from Table C.12(4)] 01. The result is thus $01|10|11|01 = 109$.

C.4: Table C.2 shows that this traversal is based on the sequence 2114.

C.5: This is straightforward

$$\begin{aligned}(00,01,11,10) &\to (000,001,011,010)(100,101,111,110)\\ &\to (000,001,011,010)(110,111,101,100)\\ &\to (000,001,011,010,110,111,101,100).\end{aligned}$$

D.1: It is $j\cdot m+(i+1)$.

D.2: It makes sense to store the degree n of the polynomial in the first array location.

D.3: Yes. In a ternary tree, the three children of node a are stored in locations $2a$, $2a+1$, and $2a+2$ and the parent of a can be found at array location $\lfloor a/3\rfloor$.

D.4: The pre-order, in-order, and level-order traversals are, respectively

$$\begin{gathered}A,((B,(D,E)),(C,(F,(G,H)))),\\ ((D,B,E),A,(null,C,(G,F,H))),\\ (A,(B,C),(D,E,F),(G,H)).\end{gathered}$$

D.5: Each of the 8 characters of a name can be one of the 26 letters or the ten digits, so the total number of names is $36^8 = 2,821,109,907,456$; close to 3 trillion.

E.1: A direct check shows that when only a single bit is changed in any of the codewords of code$_2$, the result is not any of the other codewords.

E.2: A direct check shows that code$_4$ has a Hamming distance of 4, which is more than enough to detect all 2-bit errors.

E.3: b_2 is the parity of b_3, b_6, b_7, b_{10}, and b_{11}. b_4 is the parity of b_5, b_6, and b_7. b_8 is the parity of b_9, b_{10}, and b_{11}.

E.4: Table Ans.63 summarizes the definitions of the five parity bits required for this case.

Parity	Data bits															
bits	3	5	6	7	9	10	11	12	13	14	15	17	18	19	20	21
1	x	x		x	x		x		x		x	x		x		x
2	x		x	x		x	x			x	x		x	x		
4		x	x	x				x	x	x	x				x	x
8					x	x	x	x	x	x	x					
16												x	x	x	x	x

Table Ans.63: Hamming Code for $m = 16$.

E.5: It is a variable-size code (Chapter 2). It's easy to see that it satisfies the prefix property.

F.1: Yes. It alternates between states 1, 2, 3, 4, and 1.

H.1: All the shades of gray.

H.2: A yellow surface absorbs blue and reflects green and red.

H.3: Yes! These are three colors that produce white when mixed. Examples are red, green, and blue; cyan, magenta, and yellow.

H.4: For point 1, it is easy to see that $R = 0$ and $G = 1$. Since it is on the $U = 0$ plane, where $Y = B$, it is easy to calculate that $B = 0.663$. For point 2 we have $R = 0$ and $B = 1$. Since it is on the $Y = 0.3$ plane, we get $G = 0.317$. Similarly, point 3 has $R = 0$ and the two equations $Y = 0.3$ and $U = 0$ are solved to yield $G = 0.453$ and $B = 0.3$.

H.5: Because white isn't a pure color; it is a mixture of all colors.

H.6: Recall that the sum of a dyad is white. Since illuminant white is in the middle of the line connecting c and d, it is obtained by adding equal amounts of them $(0.5c + 0.5d)$. This is why they are complementary.

H.7: Saturation refers to the amount of white in a color. Point f corresponds to full saturation, whereas illuminant white corresponds to no saturation. The saturation of the color of point e is, therefore, the ratio of the distances fw/ew.

H.8: If we continue the line from w to g, it intercepts the pure spectral curve at the bottom, an area that does not correspond to any wavelength. We therefore continue the line in the opposite direction until it intercepts the pure spectral curve at h and we say that the dominant wavelength of point g is 497_c (where c stands for "complement").

H.9: Direct calculations using matrix D_{44} produce the areas in Figure Ans.64. The three areas have black pixel percentages of 1/16, 2/16, and 16/16, respectively.

$A[x,y] =$	0	1	15
	10001000...	10001000...	11111111...
	00000000...	00000000...	11111111...
	00000000...	00100010...	11111111...
	00000000...	00000000...	11111111...

Figure Ans.64: Ordered Dither: Three Uniform Areas.

H.10: A direct application of Equation (H.2) yields

$$D_{88} = \begin{array}{|r|r|r|r|r|r|r|r|}\hline 0&32&8&40&2&34&10&42\\\hline 48&16&56&24&50&18&58&26\\\hline 12&44&4&36&14&46&6&38\\\hline 60&28&52&20&62&30&54&22\\\hline 3&35&11&43&1&33&9&41\\\hline 51&19&59&27&49&17&57&25\\\hline 15&47&7&39&13&45&5&37\\\hline 63&31&55&23&61&29&53&21\\\hline\end{array}.$$

H.11: A checkerboard pattern. This can be seen by manually simulating the algorithm of Figure H.22b for a few pixels.

H.12: We assume that the test is

if $p \geq 0.5$, then $p := 1$ else $p := 0$; add the error $0.5 - p$ to the next pixel q.

The first pixel is thus set to 1 and the error of $0.5 - 1 = -0.5$ is added to the second pixel, changing it from 0.5 to 0. The second pixel is set to 0 and the error, which is $0 - 0 = 0$, is added to the third pixel, leaving it at 0.5. The third pixel is thus set to 1 and the error of $0.5 - 1 = -0.5$ is added to the fourth pixel, changing it from 0.5 to 0. The results are

$$\begin{array}{|c|c|c|c|c|}\hline .5&.5&.5&.5&.5\\\hline\end{array}\to \begin{array}{|c|c|c|c|c|}\hline 1&0&.5&.5&.5\\\hline\end{array}\to \begin{array}{|c|c|c|c|c|}\hline 1&0&1&0&.5\\\hline\end{array}\to \begin{array}{|c|c|c|c|c|}\hline 1&0&1&0&1\\\hline\end{array}$$

H.13: Direct examination shows that the barons are 62 and 63 and the near-barons are 60 and 61.

H.14: A checkerboard pattern, similar to the one produced by diffusion dither. This can be seen by manually executing the algorithm of Figure H.24 for a few pixels.

H.15: Classes 14, 15, and 10 are barons. Classes 12 and 13 are near-barons. The class numbers in positions (i, j) and $(i, j + 2)$ add up to 15.

I.1: This is straightforward

$$\mathbf{A}+\mathbf{B} = \begin{pmatrix} 8 & 10 & 12\\ 8 & 10 & 12\\ 8 & 10 & 12 \end{pmatrix}, \mathbf{A}-\mathbf{B} = \begin{pmatrix} -6 & -6 & -6\\ 0 & 0 & 0\\ 6 & 6 & 6 \end{pmatrix}, \mathbf{A}\times\mathbf{B} = \begin{pmatrix} 18 & 24 & 30\\ 54 & 69 & 84\\ 90 & 114 & 138 \end{pmatrix}.$$

I.2: Equation (I.4) gives

$$\mathbf{T}^{-1} = \frac{1}{1\cdot 1 - 1\cdot 1}\Bigl(\cdots\Bigr),$$

which is undefined. Matrix $\mathbf{T}$ is thus *singular*; it does not have an inverse! This becomes easy to understand when we think of $\mathbf{T}$ as the coefficients matrix of the system of equations (I.3) above. This is a system of three equations in the three unknowns x, y, and z, but its first two equations are contradictory. The first one says that $x-y$ equals 1, while the second one says that the same $x-y$ equals -2. Mathematically, such a system has a singular coefficients matrix.

I.3: This is easily proved by showing that both dot products $(\mathbf{P}\times\mathbf{Q})\bullet\mathbf{P}$ and $(\mathbf{P}\times\mathbf{Q})\bullet\mathbf{Q}$ equal zero.

$$(\mathbf{P}\times\mathbf{Q})\bullet\mathbf{P} = P_1(P_2Q_3-P_3Q_2)+P_2(-P_1Q_3+P_3Q_1)+P_3(P_1Q_2-P_2Q_1)=0.$$

And similarly for $(\mathbf{P}\times\mathbf{Q})\bullet\mathbf{Q}$.

I.4: In the special case where $\mathbf{i}=(1,0,0)$ and $\mathbf{j}=(0,1,0)$ it is easy to verify that the product $\mathbf{i}\times\mathbf{j}$ equals $(0,0,1)=\mathbf{k}$. The triplet $(\mathbf{i},\mathbf{j},\mathbf{i}\times\mathbf{j}=\mathbf{k})$ thus has the handedness of the coordinate system (it is either right-handed or left-handed, depending on the coordinate system). In a right-handed coordinate system, the right hand rule makes it easy to predict the direction of $\mathbf{P}\times\mathbf{Q}$. The rule is: if your thumb points in the direction of $\mathbf{P}$ and your second finger, in the direction of $\mathbf{Q}$, then your middle finger will point in the direction of $\mathbf{P}\times\mathbf{Q}$. In a left-handed coordinate system, a similar left-hand rule applies.

I.5: They either have the same direction, or they point in opposite directions.

I.6: We are looking for a vector $\mathbf{P}(t)$ that is linear in t and that satisfies $\mathbf{P}(0)=\mathbf{P}_1$ and $\mathbf{P}(1)=\mathbf{P}_2$. It is easy to guess that

$$\mathbf{P}(t)=(1-t)\mathbf{P}_1+t\mathbf{P}_2=t(\mathbf{P}_2-\mathbf{P}_1)+\mathbf{P}_1$$

satisfies both conditions.

I.7: This is not especially hard

$$\begin{aligned}\mathbf{c} =&\frac{2\cdot 1+1\cdot 0+3\cdot(-1)}{1^2+0^2+(-1)^2}(1,0,-1)=(-1/2,0,1/2),\\ \mathbf{d} =&\mathbf{a}-\mathbf{c}=(2.5,1,2.5).\end{aligned}$$

I.8: Because of the wide spread use of computers, the world of science and engineering has moved from analog to digital. Instead of a continuous function $f(t)$ we now have an n-tuple $(f_1,f_2,\ldots,f_n)$, and this is a vector (in n dimensions).

I.9: The multiplication rule yields $(0,1)\times(0,1)=(-1,0)$ or $i\times i=-1$. The strange rule of complex number multiplication results in $i^2=-1$ or $\sqrt{-1}=i$.

I.10: The multiplication rule yields $(a,b)\times(-a,-b)=(0,0)$, which justifies calling $(-a,-b)$ the opposite of (a,b).

I.11: We start with $i = \cos(\pi/2) + i\sin(\pi/2)$, from which we get

$$\sqrt{i} = \left(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2}\right)^{1/2}.$$

By DeMoivre's theorem, this equals

$$\cos\frac{\pi}{4} + i\sin\frac{\pi}{4} = \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}} = \frac{1+i}{\sqrt{2}}. \qquad \text{(Ans.3)}$$

A simple check gives

$$\left(\frac{1+i}{\sqrt{2}}\right)^2 = \frac{1+2i+i^2}{2} = i.$$

(DeMoivre's theorem states that $\sin(nx) + i\cos(nx)$ is one of the values of $(\sin x + i\cos x)^n$.)

I.12: We start with the elegant formula

$$e^{it} = \cos t + i\sin t.$$

Substituting $t = \pi/2$ yields

$$e^{i\pi/2} = i\sin(\pi/2) = i.$$

Both sides are now raised to the ith power, yielding

$$i^i = (e^{i\pi/2})^i = e^{i^2\pi/2} = e^{-\pi/2}.$$

Surprisingly, this is a real number. In the past, calculating many digits of this number (or others like it) involved years of toil. Today, however, the single Matlab statement `vpa('i^i',140)` produces, in less than a second, the 140-digit number

```
0.20787  95763  50761  90854  69556  19834  97877  00338  77841
  63176  96080  75135  88305  54198  77285  48213  97886  00277
  86542  60353  40521  77330  72350  21808  19061  97303  74663
  98700
```

Since $a = \exp(\ln a)$ for any a, we also get

$$e^{\ln i} = i = e^{i\pi/2},$$

from which it is clear that $\ln i = i\pi/2$.

By the way, the exponential function e^z is defined for any complex number $z = x + iy$ by

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots.$$

Leonhard Euler, the great Eighteenth century mathematician, introduced the notation i for $\sqrt{-1}$ in 1777. It is interesting to note that electrical engineers use the notation j instead, since they deal with electrical voltages and currents and find it convenient to reserve i to indicate current.

I.13: The axiom and the production rules stay the same. Only the initial heading and the turn angle change. The initial heading can be either 0 or 90°, and the turn angle either 90° or 270°.

I.14: Such a polynomial depends on three coefficients **b**, **c**, and **d** that can be considered three-dimensional points, and any three points are on the same plane.

I.15:

$$\begin{aligned}\mathbf{P}(2/3) =&(0,-9)(2/3)^3+(-4.5,13.5)(2/3)^2+(4.5,-3.5)(2/3)\\ =&(0,-8/3)+(-2,6)+(3,-7/3)\\ =&(1,1)=\mathbf{P}_3\end{aligned}$$

I.16: We use the relations $\sin 30^\circ=\cos 60^\circ=.5$ and the approximation $\cos 30^\circ=\sin 60^\circ\approx .866$. The four points are $\mathbf{P}_1=(1,0)$, $\mathbf{P}_2=(\cos 30^\circ,\sin 30^\circ)=(.866,.5)$, $\mathbf{P}_3=(.5,.866)$, and $\mathbf{P}_4=(0,1)$. The relation $\mathbf{A}=\mathbf{N}\cdot\mathbf{P}$ becomes

$$\begin{pmatrix}\mathbf{a}\\ \mathbf{b}\\ \mathbf{c}\\ \mathbf{d}\end{pmatrix}=\mathbf{A}=\mathbf{N}\cdot\mathbf{P}=\begin{pmatrix}-4.5&13.5&-13.5&4.5\\ 9.0&-22.5&18&-4.5\\ -5.5&9.0&-4.5&1.0\\ 1.0&0&0&0\end{pmatrix}\begin{pmatrix}(1,0)\\ (.866,.5)\\ (.5,.866)\\ (0,1)\end{pmatrix}$$

and the solutions are

$$\begin{aligned}\mathbf{a}&=-4.5(1,0)+13.5(.866,.5)-13.5(.5,.866)+4.5(0,1)=(.441,-.441),\\ \mathbf{b}&=19(1,0)-22.5(.866,.5)+18(.5,.866)-4.5(0,1)=(-1.485,-0.162),\\ \mathbf{c}&=-5.5(1,0)+9(.866,.5)-4.5(.5,.866)+1(0,1)=(0.044,1.603),\\ \mathbf{d}&=1(1,0)-0(.866,.5)+0(.5,.866)-0(0,1)=(1,0).\end{aligned}$$

The PC is thus $\mathbf{P}(t)=(.441,-.441)t^3+(-1.485,-0.162)t^2+(0.044,1.603)t+(1,0)$. The midpoint is $\mathbf{P}(.5)=(.7058,.7058)$, only 0.2% away from the midpoint of the arc, which is at $(\cos 45^\circ,\sin 45^\circ)\approx(.7071,.7071)$.

I.17: The new equations are easy enough to set up. Using *Mathematica*, they are also easy to solve. The following code

```
Solve[{d==p1,
a al^3+b al^2+c al+d==p2,
a be^3+b be^2+c be+d==p3,
a+b+c+d==p4},{a,b,c,d}];
```

```
ExpandAll[Simplify[%]]
```

(where `al` and `be` stand for α and β, respectively) produces the (messy) solutions

$$
\begin{aligned}
\mathbf{a} &= -\frac{\mathbf{P}_1}{\alpha\beta} + \frac{\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} + \frac{\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta} \\
\mathbf{b} &= \mathbf{P}_1\left(-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3\right)/\gamma + \mathbf{P}_2\left(-\beta+\beta^3\right)/\gamma \\
&\quad + \mathbf{P}_3\left(\alpha-\alpha^3\right)/\gamma + \mathbf{P}_4\left(\alpha^3\beta-\alpha\beta^3\right)/\gamma \\
\mathbf{c} &= -\mathbf{P}_1\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) + \frac{\beta\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} \\
&\quad + \frac{\alpha\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\alpha\beta\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta} \\
\mathbf{d} &= \mathbf{P}_1. \\
&\text{where}\quad \gamma = (-1+\alpha)\alpha(-1+\beta)\beta(-\alpha+\beta).
\end{aligned}
$$

From here, the basis matrix immediately follows

$$
\begin{pmatrix}
-\frac{1}{\alpha\beta} & \frac{1}{-\alpha^2+\alpha^3\alpha\beta-\alpha^2\beta} & \frac{1}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{1}{1-\alpha-\beta+\alpha\beta} \\
\frac{-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3}{\gamma} & \frac{-\beta+\beta^3}{\gamma} & \frac{\alpha-\alpha^3}{\gamma} & \frac{\alpha^3\beta-\alpha\beta^3}{\gamma} \\
-\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) & \frac{\beta}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} & \frac{\alpha}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{\alpha\beta}{1-\alpha-\beta+\alpha\beta} \\
1 & 0 & 0 & 0
\end{pmatrix}
$$

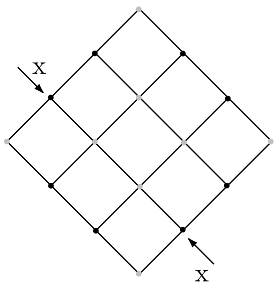
A direct check, again using *Mathematica*, for $\alpha = 1/3$ and $\beta = 2/3$, reduces this matrix to matrix $\mathbf{N}$ of Equation (I.22).

I.18: The missing points will have to be estimated by interpolation or extrapolation from the known points before our method can be applied. Obviously, the fewer points are known, the worse the final interpolation. Note that 16 points are necessary, since a bicubic polynomial has 16 coefficients.

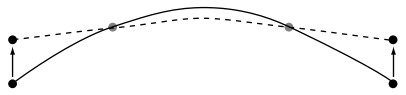
I.19: Figure Ans.65a shows a diamond-shaped grid of 16 equally-spaced points. The eight points with negative weights are shown in black. Figure Ans.65b shows a cut (labeled xx) through four points in this surface. The cut is a curve that passes through pour data points. It is easy to see that when the two exterior (black) points are raised, the center of the curve (and, as a result, the center of the surface) gets lowered. It is now clear that points with negative weights push the center of the surface in a direction opposite that of the points.

Figure Ans.65c is a more detailed example that also shows why the four corner points should have positive weights. It shows a simple symmetric surface patch that interpolates the 16 points

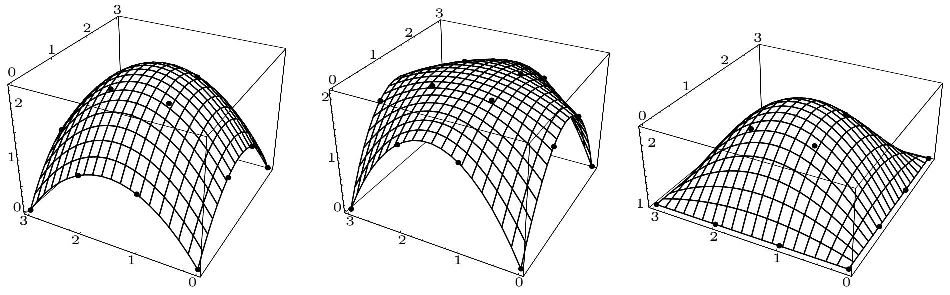
$$
\begin{aligned}
&\mathbf{P}_{00} = (0,0,0), \quad \mathbf{P}_{10} = (1,0,1), \quad \mathbf{P}_{20} = (2,0,1), \quad \mathbf{P}_{30} = (3,0,0), \\
&\mathbf{P}_{01} = (0,1,1), \quad \mathbf{P}_{11} = (1,1,2), \quad \mathbf{P}_{21} = (2,1,2), \quad \mathbf{P}_{31} = (3,1,1),
\end{aligned}
$$



(a)



(b)



(c) (d) (e)

Figure Ans.65: An Interpolating Bicubic Surface Patch.

$$\mathbf{P}_{02} = (0,2,1), \quad \mathbf{P}_{12} = (1,2,2), \quad \mathbf{P}_{22} = (2,2,2), \quad \mathbf{P}_{32} = (3,2,1),$$
$$\mathbf{P}_{03} = (0,3,0), \quad \mathbf{P}_{13} = (1,3,1), \quad \mathbf{P}_{23} = (2,3,1), \quad \mathbf{P}_{33} = (3,3,0).$$

We first raise the eight boundary points from $z = 1$ to $z = 1.5$. Figure Ans.65d shows how the center point $\mathbf{P}(.5, .5)$ gets lowered from $(1.5, 1.5, 2.25)$ to $(1.5, 1.5, 2.10938)$. We next return those points to their original positions and instead raise the four corner points from $z = 0$ to $z = 1$. Figure Ans.65e shows how this raises the center point from $(1.5, 1.5, 2.25)$ to $(1.5, 1.5, 2.26563)$.

the talk of the ordinary Englishman made me sick, I couldn't get enough exercise, and the amusements of London seemed as flat as soda-water that has been standing in the sun.

John Buchan, 1915, *The Thirty-nine Steps*

```
Clear[Nh,p,pnts,U,W];
p00={0,0,0}; p10={1,0,1}; p20={2,0,1}; p30={3,0,0};
p01={0,1,1}; p11={1,1,2}; p21={2,1,2}; p31={3,1,1};
p02={0,2,1}; p12={1,2,2}; p22={2,2,2}; p32={3,2,1};
p03={0,3,0}; p13={1,3,1}; p23={2,3,1}; p33={3,3,0};
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
 {-5.5,9,-4.5,1},{1,0,0,0}};
pnts={{p33,p32,p31,p30},{p23,p22,p21,p20},
 {p13,p12,p11,p10},{p03,p02,p01,p00}};
U[u_]:={u^3,u^2,u,1};   W[w_]:={w^3,w^2,w,1};
(* prt [i] extracts component i from the 3rd dimen of P *)
prt[i_]:=pnts[[Range[1,4],Range[1,4],i]];
p[u_,w_]:={U[u].Nh.prt[1].Transpose[Nh].W[w],
 U[u].Nh.prt[2].Transpose[Nh].W[w], \
 U[u].Nh.prt[3].Transpose[Nh].W[w]};
 g1=ParametricPlot3D[p[u,w], {u,0,1},{w,0,1},
 Compiled->False, DisplayFunction->Identity];
g2=Graphics3D[{AbsolutePointSize[2],
 Table[Point[pnts[[i,j]]],{i,1,4},{j,1,4}]}];
Show[g1,g2, ViewPoint->{-2.576, -1.365, 1.718}]
```

Code For Figure Ans.65.