Error Detection and Control

Data transmission is traditionally error prone, but computers really prefer data to be completely error-free. There are two ways of handling transmission errors —

1. Error Correction, often called Forward Error Control (FEC). FEC includes extra, redundant, information in the message so that lost data can be reconstructed and any damage repaired.
   • FEC is expensive, especially to handle burst errors rather than occasional bit errors.
   • FEC must be used where the original data is not available, such as deep-space telemetry and data recording
2. Error Detection and retry, or Automatic Repeat Request (ARQ) relies on powerful error detection and retransmission of faulty messages. It is the usual method in data communications, as very good error detection is much simpler than moderate error correction.
   These "checksums" do not *correct* errors, just *detect* them.

3. Always, with both Forward Error Correction and Automatic Repeat Request, we calculate a checksum or similar quantity from the data at the sender and send this checksum with the data.
   At the receiver the checksum is recomputed from the received data and checked for agreement with that transmitted.
   (Sometimes we do the check over (received data + checksum) but the principle is unchanged.)

Forward Error Correction — Hamming Codes

With a codeword of $k = 2^n - 1$ bits, a proper coding of an n-bit error field should be able to indicate one of 2^n conditions–

No error at all, OR which of the $k = 2^n - 1$ bits has a single error.

With $k = 2^n - 1$ bits in the codeword and n bits used for checking, there are clearly $i = k–n$ bits available for information.

This leads to the family of possible Single Error Correcting (SEC) codes, described by their (codeword_length, data_length)

n	k	i	description
2	3	1	(3, 1)
3	7	4	(7, 4)
4	15	11	(15, 11)
5	31	26	(31, 26)
6	63	57	(63, 57)

A 2-out-of-3 majority code

The "usual" Hamming code

we use a version of this one

There are many codes that fit this specification, but the simplest, oldest, and usual introduction is the Hamming code.

But first look at parity.

- The parity bit is an extra bit which is added to the information bits and adjusted so that the overall bit count is either even or odd.
- All error correction uses parity bits, usually many of them, each checking some of the bits so that the pattern of bits where the parity fails indicates the position of the error. It is accurately described by the following ditty

A message of content and clarity,
Has gotten to be quite a rarity –
To combat the terror
Of serious error,
Use bits of appropriate parity!

To handle 8-bit data we must use the (15,11) code, discarding 3 data bits to get a (12, 8) code.

1. Take a 12-bit "word" with bits numbered 1 to 12 (NOT 0 to 11)
2. Allocate bits 2^i to parity bits (1, 2, 4, 8), leaving the rest to data
3. Generate parities according to the table.

	1	2	3	4	5	6	7	8	9	10	11	12
	p1	*p2*	*m1*	*p3*	*m2*	*m3*	*m4*	*p4*	*m5*	*m6*	*m7*	*m8*
group 1	X		X		X		X		X		X	
group 2		X	X			X	X			X	X	
group 4				X	X	X	X					X
group 8								X	X	X	X	X

Parity bit 4 checks the parity of all positions whose binary bit number has the bit for $2^2 = 4$, and similarly for the other parity bits.

To illustrate, the 8-bit information word 01001010 expands first to *pp*0*p*100*p*1010, creating space for the parity bits.
Assume odd parity (which ensures that there is at least 1 1-bit).

	1	2	3	4	5	6	7	8	9	10	11	12
	p1	*p2*	0	*p3*	1	0	0	*p4*	1	0	1	0
group 1	x		x		x		x		x		x	
group 2		x	x			x	x			x	x	
group 4				x	x	x	x					x
group 8								x	x	x	x	x

Then,
*p*1 checks *p*1, *m*1, *m*2, *m*4, *m*5, *m*7	0, 1, 0, 1, 1	*p*1 = 0
*p*2 checks *p*2, *m*1, *m*3, *m*4, *m*6, *m*7	0, 0, 0, 0, 1	*p*2 = 0
*p*3 checks *p*3, *m*2, *m*3, *m*4, *m*8	1, 0, 0, 0	*p*3 = 0
*p*4 checks *p*4, *m*5, *m*6, *m*7, *m*8	1, 0, 1, 0	*p*4 = 1

The final codeword is 000<u>0</u>100<u>1</u>1010. (parity bits underlined)

Now assume an error 000011011010 (bit 6)

Without error correction, we extract the information bits — 01101010

which is NOT equal to the original 01001010

Now doing the error correction by counting 1s under each mask; as the count should be odd, generate a syndrome bit=1 if count is even.

	0	0	0	0	1	1	0	1	1	0	1	0	count	syn-drome
Group 1	x		x		x		x		x		x		3	0
Group 2		x	x			x	x			x	x		2	1
Group 4				x	x	x	x					x	2	1
Group 8								x	x	x	x	x	3	0

- Groups 2 and 4 fail; the error is in bit 2+4 = 6.
- Correct the received data to 000010011010
  Delete the parity bits to get 01001010
  and compare with the original 01001010 which is correct.

- The number locating the error is known as the *syndrome*, and is usually zero if no error.
- We can add a single overall parity bit to get a SECDED code (Single Error Correcting, Double Error Detecting).
- If two bits are corrupted, the overall parity is still OK, but at least one internal parity fails, signalling an uncorrectable error.
- Going to codes which will correct multiple errors is a much more difficult problems, far beyond this class.
- Compact discs have very powerful error correction
  (BER is the bit error rate in reading from the disc)

Maximum completely correctable burst	4000 data bits
Uncorrected errors	< 1 in 750 hrs @ BER $=10^{-3}$
	undetectable @ BER $= 10^{-4}$

Error Detection with Retry (ARQ)

- Data communications errors tend to be rare and occur in bursts.
- Both aspects make it very difficult to design good error correction codes.
- It is more efficient to use very powerful error detection, with retransmission (Automatic Repeat Request — ARQ).
- None of these checksums (1s complement, Fletcher, Adler or CRC) is *designed* to correct errors (although some can do that to a very limited extent, as in ATM cell headers); they are designed as error *detectors*. (The Hamming code as described is a Single-Error-Correcting code [SEC]; with an extra parity bit it becomes a Double-Error-Detecting code as well [SEC-DED], but no more.)
- All are designed to give very good error detection, with very few errors escaping detection (undetected errors often less than 1 in 10^9).
- The checksums all cover entire messages, not individual characters.
- Generally we ignore character parity completely from now on.

Additive Checksums

These codes work by performing *arithmetic addition* on the words of the message (longitudinal, or message, parity is a *logical addition*).

The simplest is the TCP/IP checksum, which is the 1's complement sum of all of the (16 bit) words of the message — assume 32-bit arithmetic.

```
sum += word;                // do the addition
while (sum > 0xFFFF)        // if beyond 16-bits -
  sum = (sum & 0xFFFF)      // isolate 16 low bits
      + (sum >> 16);        // and add the "overflow"
```

It is simple, but not very good.

1. An error in one bit has a rather local effect on the checksum
2. All words are treated equally; it is insensitive to transpositions

Better checksums give different weights to different message positions.

Sum-of-sums checksums.

These maintain two sums in parallel; for each character (or word) compute

```
sum1 += word;          // sum of the words so far
sum2 += sum1;          // this is the sum-of-sums
```

- "**sum1**" is clearly the sum of all the words so far.
- For a message of 1…k characters,

  sum2 = $k\,w_0 + (k–1)\,w_1 + (k–2)\,w_3 + \ldots\ 2\,w_{k–1} + w_k$

The *Fletcher checksum* (OSI Transport Layer – level 4) has sums modulo 255

$$s1 = (s1 + d_i) \bmod 255, \text{ and} \qquad s2 = (s2 + s1) \bmod 255$$

The checksum is the 16 bit concatenation of $s1$ and $s2$; $(256*s1 + s2)$

The result is correct if *either* $s1 = 0$ *or* $s2 = 0$.

The Fletcher checksum is stated to be nearly as powerful as a CRC-16 check, detecting —

- all single-bit errors
- all double-bit erors
- all but 0.000019% of burst errors up to length 16
- all but 0.0015% of longer burst errors

The *Adler checksum*, for the GZIP compressor, is an improvement on the Fletcher checksum, using 16-bit sums and a prime modulus.

$$s1 = (s1 + d_i) \bmod 65521$$
$$s2 = (s2 + s1) \bmod 65521$$

- The initial values are $s1 = 1$ and $s2 = 0$.
- The checksum is the 32 bits $65536*s1 + s2$

Fletcher Checksum code

```
int s1 = 0,  s2 = 0;                    // initialise checksums
 for (int i = 0; i < nChars; i++)       // scan the characters
   {
   s1 += c[i];                          // add in the character
   while (s1 >= 255)                    // reduce modulo 255
     s1 -= 255;

   s2 += s1;                            // get the sum of sums
   while (s2 >= 255)                    // modulo 255
     s2 -= 255;
   }
```

Modulus Checks

This important class of checks is an extension of simple parity. It can be illustrated by some simple decimal examples, say we check 23145.

1. Choose an agreed modulus m – a *prime number* just less than a power of 10, such as $m = 7$ or $m = 97$.
2. Extend the number by zeros, as many as digits in the modulus.
3. Divide the *extended* number by m and get the remainder r. (231450 mod 7 = 2; 2314500 mod 97 = 80.)
4. Replace the zero extension by $(m–r)$, to give 231455 or 2314517.
5. Encode or transmit this new extended number.
6. To check on reception, divide the received number by the agreed modulus – the remainder should be zero.
7. Discard the last (extension or remainder) digits and deliver the preceding digits as the verifed number.

Polynomials involved in checking

The following checks work on bit vectors, and regard the bits as coefficients of polynomial in some arbitrary variable x.

- $i(x)$ Information — The information or data to be checked
- $g(x)$ Generator — The system-defined divisor polynomial
- $c(x)$ Codeword — What is transmitted; $i(x)$ with $(i(x) \bmod g(x))$ appended
- $e(x)$ Error — The error vector; $e(x) = x^i$ for a single bit error
- $v(x)$ Received — What is received; $v(x) = e(x) + c(x)$

Calculate the syndrome

$$\begin{aligned} s(x) &= v(x) \bmod g(x) \\ &= [e(x) + c(x)] \bmod g(x) \\ &= e(x) \bmod g(x) + c(x) \bmod g(x) \\ &= e(x) \bmod g(x) + 0 \quad \text{by construction} \end{aligned}$$

The syndrome $s(x)$ is a function only of the error vector $e(x)$.

Cyclic Redundancy Checks

- Most checking now uses Cyclic Redundancy Checks, which treats the bits of the message as coefficients of an "*information polynomial*" $i(x)$, divides it by another "*generator*" polynomial $g(x)$ and sends the remainder as the check digits at the end of the message.
- The *principles* are identical to the previous slide, but *details* are different
- The two bit patterns are equivalent

$$1\ 0\ 0\ 1\ 0\ 1 \Leftrightarrow x^5 + x^2 + 1$$

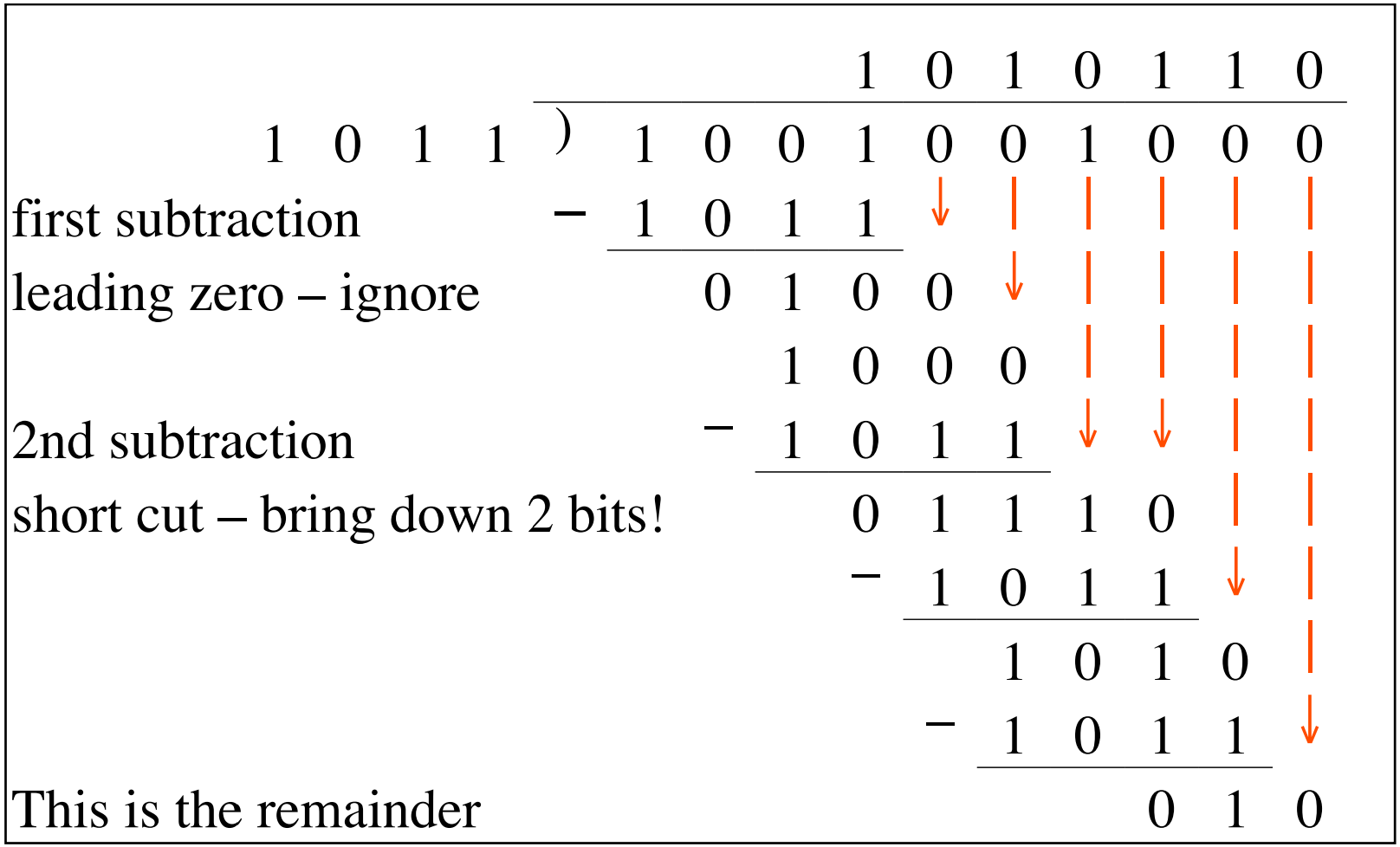
and also $$1\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \Leftrightarrow x^8 + x^2 + x + 1$$

- Arithmetic follows the rules, with no carries

$$(0 - 0) = (1 - 1) = 0$$
$$(0 - 1) = (1 - 0) = 1$$

- A generator $g(x)$ of order r has $r+1$ bits; and *must* have the form $1 \ldots \ldots 1$, with its most-significant and least-significant bits both 1.
- We put r 0 bits after the dividend before performing the division.

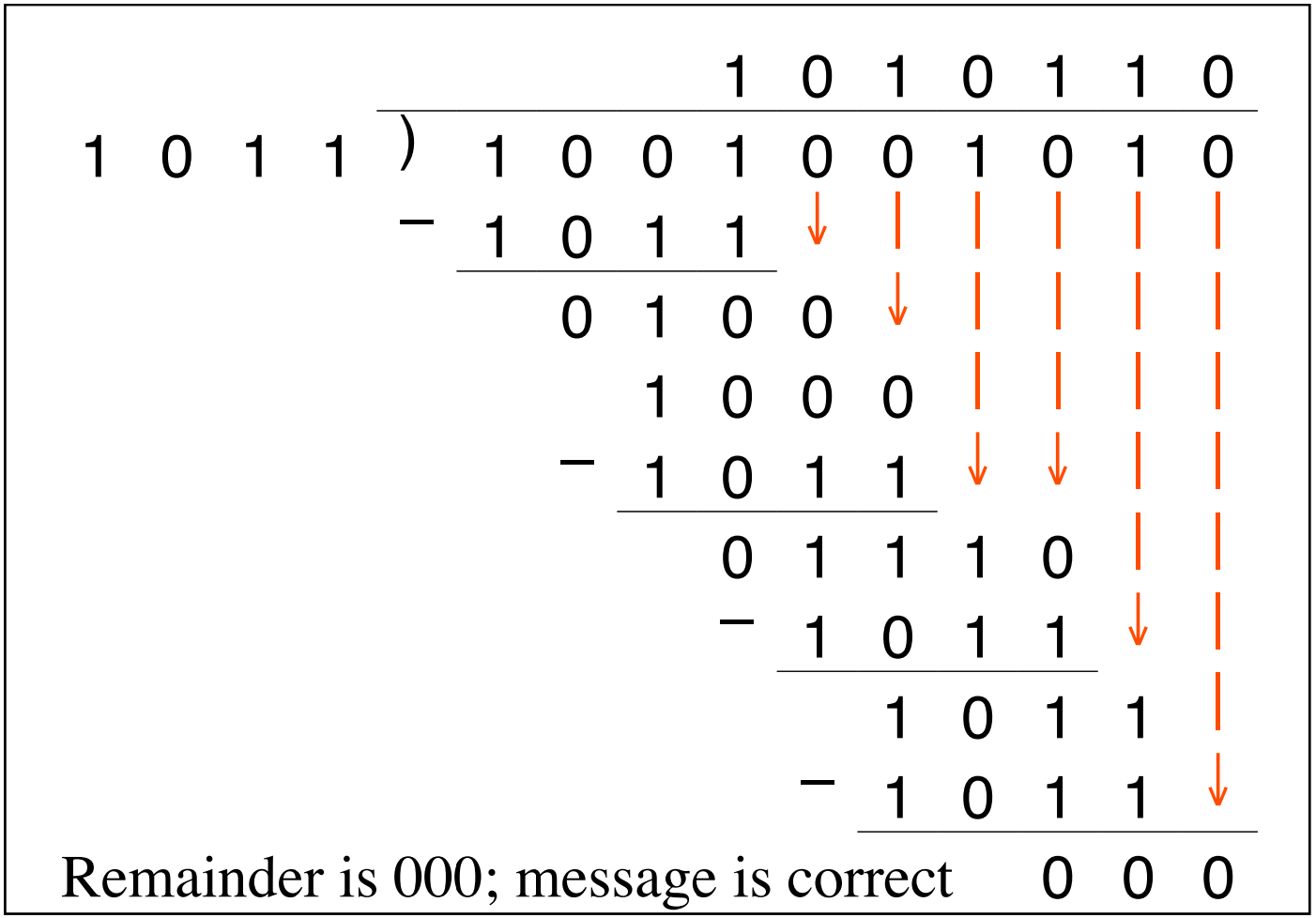
Example – get CRC for $i(x)$ =1001001, with generator $g(x)$ = 1011



Add the remainder to the original $i(x)$; message with checksum is 1001001010 (data [1001001] + remainder [010] = 1001001010).

Check — divide $c(x) = 1001001010$ by $g(x) = 1011$

Divide the received codeword $v(x)$ by $g(x)$, *without* extending by 4 zeros.



The quotient is discarded; only the fact that the remainder is zero or non-zero important.

Check — assume received codeword $v(x) = 1001101010$

Now $v(x) = c(x) + e(x)$, where $e(x) \neq 0$. Here $e(x) = 0000100000$.

```
                        1  0  1  0  0  1  1
                  ------------------------
1  0  1  1  )  1  0  0  1  1  0  1  0  1  0
            -  1  0  1  1
               ----------
               0  1  0  1
                  1  0  1  0
               -  1  0  1  1
               ----------
                  0  0  1  1  0  1
                     -  1  0  1  1
                     ------------
Note correction to quotient         1  1  0  0
                     -  1  0  1  1
                     ------------
Remainder ≠ 0, so error             1  1  1
```

Remember, the only test is whether the remainder is zero or non-zero; its precise value is unimportant and conveys no information.

Construction of generator polynomial

1. A single bit error has $e(x) = x^i$. If $g(x)$ has two or more terms it will never divide $e(x)$ and all single-bit errors will be detected.
2. Two isolated single bit errors have $e(x) = x^i (x^k + 1)$. Find a $g(x)$ which does not divide $x^k + 1$, for k up to message length. Use a computer search; for example $x^{15} + x^{14} + 1$ does not divide $x^k + 1$ for $k < 32768$.
3. If there are an odd number of bits in error, then $e(x)$ has an odd number of bits. As no polynomial with an odd number of bits has $(x+1)$ as a factor, make $g(x) = (x+1)(\ldots\ \ldots)$.
4. A code with r check bits will detect all burst errors of length $< r$.
5. If the burst has length $r + 1$, $r(x)$ will be zero only if $r(x) = g(x)$, and with probability $1/2^{r-1}$.
6. For longer bursts the probability of an undetected error is $1/2^r$.

Standard CRC polynomials

These are all predefined and are agreed between sender and receiver.

CRC-12	$x^{12} + x^{11} + x^3 + x + 1$	Old banking etc codes (6 bit)
CRC-16	$x^{16} + x^{15} + x^2 + x + 1$	North American SDLC (8 bit)
CRC-ITU	$x^{16} + x^{12} + x^5 + x + 1$	ITU standard HDLC (8 bit)
IEEE 802	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	IEEE 802 LAN standards & others
ATM HEC	$x^8 + x^2 + x + 1$	ATM cell headers
ATM AAL3/4	$x^{10} + x^9 + x^5 + x^4 + x + 1$	some ATM traffic

The CRC-16 and CRC-ITU checks can detect —

- all bursts ≤ 16 bits,
- all odd-bit errors and
- 99.998% of error bursts exceeding 17 bits

Misconceptions

Alleged disadvantage	Comment
A disadvantage of CRC-16, Fletcher and 1s complement sums is that they cannot correct an error.	They are not designed to correct errors, but give good error detection
A 1s complement check can detect only a single-bit error.	A16-bit additive checksum is about as good as a 10-bit CRC.
A disadvantage of CRC-16 is that you must send also the generator polynomial	The generator is agreed between sender and receiver and is not transmitted
A disadvantage of CRC-16 is that its generator is difficult to make	It is hard to make, but is then defined in the standard
Some of them are inefficient because we must send the checksum with the message	All of them must send the checksum!

Data Compression

Various techniques —

1. Use a Huffman or similar variable length code and assign short codes to more frequent symbols. Needs a coding dictionary and not very effective.
2. Run-length coding. Many files have lots of spaces (ASCII 0x20); for example replace these runs by DLE *i j*, where *i* and *j* are ASCII decimal digits. (A run of 27 spaces would be replaced by "DLE 2 7".) Especially useful for FAXes with long runs of 0s and 1s (but use a different scheme!)
3. Codebooks. Assign codes to frequent words and then run as extension to 1 above. Excellent in some contexts, but inflexible.
4. Dictionary compressors use the redundancy in text, to replace already known letter sequences or *phrases* by dictionary indices to those phrases. These compressors learn from the text and adapt to it.
5. Relative or differential coding is used for image data; send the changes between frames.

LZW compression

LZW was proposed by Welch as a variant of a compressor originally proposed by Ziv & Lempel, and is the compressor normally used in data communications.

An LZW compressor

- has a dictionary of known phrases or "letter sequences" that it has seen. The dictionary is initialised with all possible character codes, eg 0…255.
- The compressor accepts input symbols as long as the most recent symbols correspond to known phrases in the dictionary.
- As soon as the matching fails, the compressor emits the dictionary index of the known phrase and then creates a new phrase consisting of the known phrase followed by the character which "failed".
- Continue, building a new test string for phrase comparisons.
- The dictionary is initialised to all values, 0…255, NOT as in text

Example of LZW compression

Encoder, input is "**the_theme_that_they_heard**".

the_theme_that_they_heard

Pass	Input	Test String	Emit	Make Entry	Comment
1	t	t			
2	h	th	t	th	
3	e	he	h	he	
4	_	e_	e	e_	
5	t	_t	_	_t	
6	h	th			th exists – continue
7	e	the	th (256)	the	
8	m	em	e	em	
9	e	me	m	me	
10	_	e_			e_ exists
11	t	e_t	e_ (258)	e_t	
12	h	th			th exists
13	a	tha	th (256)	tha	
14	t	at	a	at	
15	_	t_	t	t_	

Dictionary

Index	Contents
0	256
…	ASCII
255	Codes
256	th
257	he
258	e_
259	_t
260	the
261	em
262	me
263	e_t
264	tha
265	at
266	t_
267	_th

LZW example, continued

the_theme_that_they_heard

Pass	Input	Test String	Emit	Make Entry	Comment
15	_	t_	t	t_	
16	t	_t			_t exists
17	h	_th	_t (259)	_th	
18	e	he			he exists
19	y	hey	he (257)	hey	
20	_	y_	y	y_	
21	h	_h	_	_h	
22	e	he			he exists
23	a	hea	he (257)	hea	
24	r	ar	a	ar	
25	d	rd	r	rd	

Dictionary

Index	Contents
256	th
257	he
258	e_
259	_t
260	the
261	er
262	re
263	e_t
264	tha
265	at
266	t_
267	_th
268	hey
269	y_
270	_h
271	hea

Error in *early* copies 2nd Ed : p 198 Fig 3.35, line 19 should be "…with current, c"

LZW Decoding

the_theme_that_they_heard

Pass	Receive codeword	Make Entry	Deliver
1	t		t
2	h	th	h
3	e	he	e
4	_	e_	_
5	(256) th	_t	th
6	e	the	e
7	m	em	m
8	(258) e_	me	e_
9	(256) th	e_t	th
10	a	tha	a
11	t	at	t
12	(259) _t	t_	_t
13	(257) he	_th	he
14	y	hey	y
15	_	y_	_
16	257 (he)	_h	he
17	a	hea	a
18	r	ar	r
19	d	rd	d

Dictionary

Index	Contents
0	256
…	ASCII
255	Codes
256	th
257	he
258	e_
259	_t
260	the
261	em
262	me
263	e_t
264	tha
265	at
266	t_
267	_th
268	hey
269	y_
270	_h
271	hea

Example — compress

gaattcctgagaggagagagtaagcaacttggaa

gaattcctgagaggagagagtaagcaacttggaaaatata

Loop	Buffer	c	What is sent	stored in dictionary	new buffer value	Comment
1	**g**	**a**	g	**ga**	**a**	
2	**a**	**a**	a	**aa**	**a**	
3	**a**	**t**	a	**at**	**t**	
4	**t**	**t**	t	**tt**	**t**	
5	**t**	**c**	t	**tc**	**c**	
6	**c**	**c**	c	**cc**	**c**	
7	**c**	**t**	c	**ct**	**t**	
8	**t**	**g**	t	**tg**	**g**	
9	**g**	**a**			**ga**	ga exists
10	**ga**	**g**	ga (256)	**gag**	**g**	
11	**g**	**a**			**ga**	ga exists
12	**ga**	**g**			**gag**	gag exists
13	**gag**	**g**	gag (264)	**gagg**	**g**	
14	**g**	**a**			**ga**	ga exists
15	**ga**	**g**			**gag**	gag exists
16	**gag**	**a**	gag(264)	**gaga**	**a**	

index	Entry
0 … 255	ASCII codes
256	**ga**
257	**aa**
258	**at**
259	**tt**
260	**tc**
261	**cc**
262	**ct**
263	**tg**
264	**gag**
265	**gagg**
266	**gaga**
267	**ag**
268	**gagt**
269	**ta**
270	**aag**

gaattcctgagaggagagagtaagcaacttggaaaatata

Loop	Buffer	c	What is sent	stored in dictionary	new buffer value	Comment
17	**a**	**g**	a	**ag**	**g**	
18	**g**	**a**			**ga**	ga exists
19	**ga**	**g**			**gag**	gag exists
20	**gag**	**t**	gag (264)	**gagt**	**t**	
21	**t**	**a**	t	**ta**	**a**	
22	**a**	**a**			**aa**	
23	**aa**	**g**	aa (257)	**aag**	**g**	
24	**g**	**c**	g	**gc**	**c**	
25	**c**	**a**	c	**ca**	**a**	
26	**a**	**a**			**aa**	aa exists
27	**aa**	**c**	aa (257)	**aac**	**c**	
28	**c**	**t**			**ct**	ct exists
29	**ct**	**t**	ct (262)	**ctt**	**t**	
30	**t**	**g**	t	**tg**	**g**	
31	**g**	**g**	g	**gg**	**g**	
32	**g**	**a**			**ga**	ga exists
33	**ga**	**a**	256 (ga)	**gaa**	**a**	
34	**a**	**a**			**aa**	aa exists

index	Entry
271	**gc**
272	**ca**
273	**aac**
274	**ctt**
275	**tg**
276	**gg**
277	**gaa**
278	**aaa**
279	**ata**

LZW decoding

The basic steps in LZW decoding are

1. Remember the previous phrase
2. Receive the current index and get its (current) phrase
3. Emit the current phrase
4. Create a new dictionary entry of the previous phrase followed by the first symbol of the current phrase.

There is one nasty problem, illustrated on Shay Table 3.11 p200, and seen on pass 10 of the following example. (Table 5.10, p 235, 3rd Ed)

If the received index is to the phrase which would be created by this step, it looks as though we cannot create this phrase until we know what it is, but we can't know what it is until it is created … … … …

The LZW recursive entry problem

In normal operation we receive an

index *i* for phrase “abc”, followed by an

index *j* for phrase “xyz”.

From receiving index *j* we can create a new dictionary entry “abcx” at index *k*.

- But here the preceding phrase was “pqr” and we know that we must create a new phrase “pqr?”, extending that preceding phrase by one symbol.
- But the old and the new phrases overlap by one symbol because of the way the dictionary entries are made

```
p  q  r  ?
         p  q  r  ?
```

- The phrase to be created must be the preceding phrase, followed by its *first* character.

LZW decoding the previous example

Loop Pass	Prior (String)	Current (string)	Create entry	c (first symbol)	TempString/ Code Pair	What is printed
1	– –	g	– –	**g**		**g**
2	**g**	a	**ga**	**a**		**a**
3	**a**	a	**aa**	**a**		**a**
4	**a**	t	**at**	**t**		**t**
5	**t**	t	**tt**	**t**		**t**
6	**t**	c	**tc**	**c**		**c**
7	**c**	c	**cc**	**c**		**c**
8	**c**	t	**ct**	**t**		**t**
9	**t**	ga(256)	**tg**	**g**		**ga**
10	**ga**	gag(264)	**gag**	**g**		**gag**
11	**gag**	gag(264)	**gagg**	**g**		**gag**
12	**gag**	a	**gaga**	**a**		**a**
13	**a**	gag(264)	**ag**	**g**		**gag**
14	**gag**	t	**gagt**	**t**		**t**
15	**t**	aa(257)	**ta**	**a**		
16	**aa**	g	**aag**	**g**		
17	**g**	c	**gc**	**c**		
18	**c**	aa(257)	**ca**			

index	Entry
0 … 255	ASCII codes
256	**ga**
257	**aa**
258	**at**
259	**tc**
260	**tt**
261	**cc**
262	**ct**
263	**tg**
264	**gag**
265	**gagg**
266	**gaga**
267	**ag**
268	**gagt**
269	**ta**
270	**aag**
271	**gc**
272	
273	

BZIP2 compressor (Burrows-Wheeler Transform)

(not examinable)

1. Write out all cyclic permutations of the input as rows of a matrix
2. Sort the rows into alphabetic order; this collects together similar contexts
3. Transmit the *last symbol* of each row, and the *index* of the first row. (Similar contexts produce similar symbols and often runs of the one symbol.)
4. Use a Move-to-Front or recency recode of each symbol; replace each by the number of different symbols seen since it was last encountered.
5. The output numbers have a very skew distribution — use a Huffman or similar statistical encoder.
6. Some actual frequencies for a text file are —

Value	0	1	2	3	4	5	6	7
Frequency	66.8%	9.0%	4.0%	2.9%	2.3%	1.8%	1.6%	1.4%

Example — "mississippi"

1. Each row has the entire input string
2. The start of each row is the context following the last symbol of the row.
3. The first time "p", "s", "m", "i" are seen their MTF codes can be anything; (assume the ASCII value)

	context	symbol	Index	MTF		context	link
	imississip	p	1	112		i…	5
	ippimissis	s	2	115		i…	7
	issippimis	s	3	0		i…	10
	ississippi	m	4	109		i…	11
→	mississipp	i	5	105		m…	4
	pimississi	p	6	3		p…	1
	ppimississ	i	7	1		p…	6
	sippimissi	s	8	4		s…	2
	sissippimi	s	9	0		s…	3
	ssippimiss	i	10	1		s…	8
	ssissippim	i	11	0		s…	9

Yes, the transformation can be reversed!

1. By counting symbols we know that 4 contexts start with “i”, 1 with “m”, 2 with “p” and 4 with “s”. These are shown in the 6th column of the table.
2. The first “i” context (at 1) corresponds to the first “i” symbol (at 5), the context at 2 to the symbol at 7 and so on, building links as shown in the last column.
3. The initial index is 5, which links to symbol 4, an “m”.
4. Then follow the links 4(m) → 11(i) → 9(s) → 3(s) → 10(i) → 8(s) → 2(s) → 7(i) → 6(p) → 1(p) → 5(i) → (back to start)

Typical compression of text is —

LZW 4.0 – 4.5 bits per character, or 50 – 55% of original size

BZIP 2.0 – 2.5 bits per character, or 25 – 30% of original size