Data Compression

We've seen several instances in this course of data stored in files. Some of the formats for storing data were more compact than others, and this was often seen as an advantage.

Data Compression

encoding information in a format that takes up less space.

Topic	***Folk & Zoellick***
• A Few Definitions	
• Fixed Length Codes	§ 5.1.2
◦ RLE	§ 5.1.2
• Variable Length Codes	§ 5.1.3
◦ Huffman Codes	§ 5.1.3
• String Matching	
◦ LZW	§ 5.1.5
• Other Stuff	
◦ JPEG	
◦ MPEG	
◦ Fractal Compression	

There's also a cute Java tutorial at
`http://www.cs.sfu.ca/cs/CC/365/li/squeeze/`

and more than you'd ever want to know at
`http://www.faqs.org/faqs/compression-faq/`

Definitions

Given a sequence of bytes *S* (characters, symbols):

S' = encode(*S*)

S'' = decode(*S'*)

Data Compression

S' is smaller than *S*

Lossless Compression

S'' = *S*

Lossy Compression

S'' ~ *S*

Code

A map from one set *A* of symbols (or values) to another set *B* of symbols (or values).

Each symbol in *B* is also referred to as a *code* for its corrseponding symbol in *A*.

Fixed Length Code

All the codes in set *B* are the same size (number of bits).

Variable Length Code

The codes in set *B* differ in size (number of bits).

Fixed Length Codes

The ASCII code maps some displayable symbols onto one-byte (8-bit) numbers, reserving 256 possible codes.

Q: But what if we only need to use the upper case alphabetic characters, the digits 0-9 and the punctuation characters?

A: We can do it with only 6 bits instead of 8!

	000000
!	000001
"	000010
#	000011
$	000100
%	000101
&	000110
’	000111
(	001000
)	001001
*	001010
+	001011
,	001100
-	001101
.	001110
/	001111

0	010000
1	010001
2	010010
3	010011
4	010100
5	010101
6	010110
7	010111
8	011000
9	011001
:	011010
;	011011
<	011100
=	011101
>	011110
?	011111

@	100000
A	100001
B	100010
C	100011
D	100100
E	100101
F	100110
G	100111
H	101000
I	101001
J	101010
K	101011
L	101100
M	101101
N	101110
O	101111

P	110000
Q	110001
R	110010
S	110011
T	110100
U	110101
V	110110
W	110111
X	111000
Y	111001
Z	111010
[	111011
\	111100
]	111101
^	111110
_	111111

Run Length Encoding

Replace repetitive sequences of symbols with three bytes:

1. a special RLE byte
2. the repeated symbol (once only)
3. the number of repetitions

Here's an example from the *Celebrities* database, giving a celebrity's name and annual income:

```
Ken Barker.........$8000
Wayne Gretzky......$6000000
Paul Kariya........$8500000
Eric Lindros.......$11000000
Michael Jackson....$100000000
Oprah Winfrey......$400000000
```

Run Length Encoding (cont.)

Assuming the character '#' does not appear anywhere in the file, we can use it as the special RLE character.

Now here's the *Celebrities* data, compressed using *Run Length Encoding*:

```
Ken Barker#.9$8000
Wayne Gretzky#.6$6#06
Paul Kariya#.8$85#05
Eric Lindros#.7$11#06
Michael Jackson#.4$1#08
Oprah Winfrey#.6$4#08
```

The *compression ratio* is a modest 5:4.

Q: What kinds of files would RLE be good for?

A:

Variable Length Codes

Variable Length Code

The mapped-to codes differ in size (number of bits) depending on the source symbol.

Usually, variable length codes use fewer bits to encode symbols that occur frequently, and more bits to encode symbols that occur rarely.

E	·
T	—
I	· ·
A	· —
N	— ·
M	— —
S	· · ·
U	· · —
R	· — ·
W	· — —
D	— · ·
K	— · —
G	— — ·

O	— — —
H	· · · ·
V	· · · —
F	· · — ·
L	· — · ·
P	· — — ·
J	· — — —
B	— · · ·
X	— · · —
C	— · — ·
Y	— · — —
Z	— — · ·
Q	— — · —

Q: What problems are presented by this particular code?

A:

Huffman Codes

Each symbol in the input is encoded as a pattern of bits. The number of bits varies depending on the frequency of the symbol in the input:

- The most frequent symbols are represented with the fewest bits; the least frequent symbols are represented with the most bits.
- The frequency of a symbol in the input can be thought of as the *probability* that the symbol occurs in any one position in the input.

Suppose our input is: `this is a test`

Symbol	***Frequency***	***Probability***	***Huffman Code***
	3	3/14	10
s	3	3/14	01
t	3	3/14	00
i	2	2/14	110
a	1	1/14	1110
e	1	1/14	11111
h	1	1/14	11110

Ok, but where do those codes come from?

Huffman Encoding

Here's the algorithm for Huffman encoding:

```
1. Put all the symbols in the input into a list (the
   symbol_list) ordered according to their probability

2. As long as there are symbols left in the symbol_list
    a. create a tree with the bottom symbol in
        symbol_list as the left subtree and the
       next-to-bottom symbol as the right subtree
    b. the probability of the parent is the sum of the
       probabilities of the children
    c. label the left branch 0 and the right branch 1
    d. add the parent to symbol_list in the correct
       position according to its probability

3. The Huffman code for each original symbol is given by
   tracing down from the root of the tree to each leaf,
   collecting branch labels along the way
```

I think we need a picture...

Huffman Walkthrough

```
this is a test
```

s	*p*	*s*	*p*	*s*	*p*	*s*	*p*	*s*	*p*	*s*	*p*
	3/14										
s	3/14										
t	3/14										
i	2/14										
a	1/14										
e	1/14										
h	1/14										

s	*c*
s	
t	
i	
a	
e	
h	

Huffman Decoding

In order to decode a Huffman encoded file, you have to know the Huffman code for this particular file.

- We *could* store the code with the file, but if there are 256 symbols, the code for the least frequent symbol could have as many as 128 bits!
- If we store the symbols and their probabilities instead, the decoder can recreate the tree, and therefore the code (as long as the decoder uses the exact same algorithm as the encoder).

Decoding consists of reading the encoded message bit-by-bit and following the path from the root of the tree until we hit a leaf node.

Let's decode this message:

```
00111101100110110011011101000111110100
```

Q: Does this code suffer from the same problem as the Morse code? (*I.e.*, is it ambiguous?)

A:

Lempel and Ziv (and Welch)

In 1977, Abraham Lempel and Jacob Ziv published a compression algorithm based on repeating strings of characters. That algorithm (LZ77) is still used in most commercial compression software (zip, arj, zoo, etc.).

In 1978, Lempel and Ziv published another algorithm (LZ78) also based on string matching. When some string of characters repeats, it is put into a "dictionary". Each string in the dictionary can be identified by its position in the dictionary.

In 1984, Terry Welch published a variant of LZ78 called LZW. The Unix `compress` command is based on a variant of LZW (called LZC).

Enough with the history lesson already, let's boogie!

LZW

The ASCII table is an example of how you can encode character symbols as single-byte (8-bit) values.

But if we had more bits, we could associate *pairs* of characters (or even bigger things) with a single code. For example,

- If we had 9 bits, we would have 512 codes available
- Use the first 256 codes (0-255) for the standard ASCII mappings (65=’A’, 66=’B’, etc.)
- Then use codes 256-511 for bigger things (256=’the’, 257=’file’, etc.).

Storing the most frequently repeated strings in a text as single entries in the dictionary allows for significant reduction in the number of bits required to represent the text. From the example above...

- 'the' would be represented with 9 bits instead of 3×8=24 bits
- 'file' would be represented with 9 bits instead of 4×8=32 bits

LZW Encoding

Here's the algorithm for LZW encoding:

```
1. Start with a dictionary of 4096 entries (each entry
   can be identified using a 12-bit number)

2. Set w to ''

3. As long as there are more characters to read
    a. read a character k
    b. if wk is in the dictionary
          set w to wk
       else
          output the code for w
          add wk to the dictionary
          set w to k

4. output the code for w
```

Variant: What do you do when the dictionary is full?

LZW Encoding Walkthrough

```
this_is_his_history
```

c	*s*
0	
1	
2	
.	.
.	.
.	.
65	A
66	B
.	.
.	.
.	.
103	g
104	h
105	i
.	.
.	.
.	.
255	

c	*s*
256	
257	
258	
259	
260	
261	
262	
263	
264	
265	
266	
267	
268	
269	
.	.
.	.
.	.

w	*k*	*output*

LZW Decoding

Here's the coolest part:

- You don't need to store the dictionary in the encoded file, the decoder will recreate it as it decodes!

```
1. Start with a dictionary of 4096 entries (the first
   256 are standard ASCII, the rest are empty).

2. Read a code k from the encoded file

3. Output dict(k)

4. Set w to dict(k)

5. As long as there are more codes to read
    a. read a code k
    b. if k is in the dictionary
          output dict(k)
          add w + first character of dict(k) to the dict
       else
          add w + first char of w to the dict and output
       it
    c. Set w to dict(k)
```

LZW Decoding Walkthrough

```
this_<258>_<257><259><263>tory
```

c	*s*
0	
1	
2	
.	.
.	.
.	.
65	A
66	B
.	.
.	.
.	.
103	g
104	h
105	i
.	.
.	.
.	.
255	

c	*s*
256	
257	
258	
259	
260	
261	
262	
263	
264	
265	
266	
267	
268	
269	
.	.
.	.
.	.

w	*k*	*output*