Lecture #39: Compression

Credits: This presentation is largely taken from CS61B lectures by Josh Hug.

Compression and Git

- Git creates a new object in the repository each time a changed file or directory is committed.
- Things can get crowded as a result.
- To save space, it *compresses* each object.
- Every now and then (such as when sending or receiving from another repository), it packs objects together into a single file: a "packfile."
- Besides just sticking the files together, uses a technique called *delta compression.*

Delta Compression

- Typically, there will be many versions of a file in a Git repository: the latest, and previous edits of it, each in different commits.
- Git doesn't keep track explicitly of which file came from where, since that's hard in general:
  - What if a file is split into two, or two are spliced together?
- But, can *guess* that files with same name and (roughly) same size in two commits are probably versions of the same file.
- When that happens, store one of them as a pointer to the other, plus a list of changes.

Delta Compression (II)

- So, store two versions

V1	V2
My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I’ve recovered my forgotten moral senses,	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I’ve recovered my forgotten moral senses, and don’t give twopence halfpenney for any consequences.

as

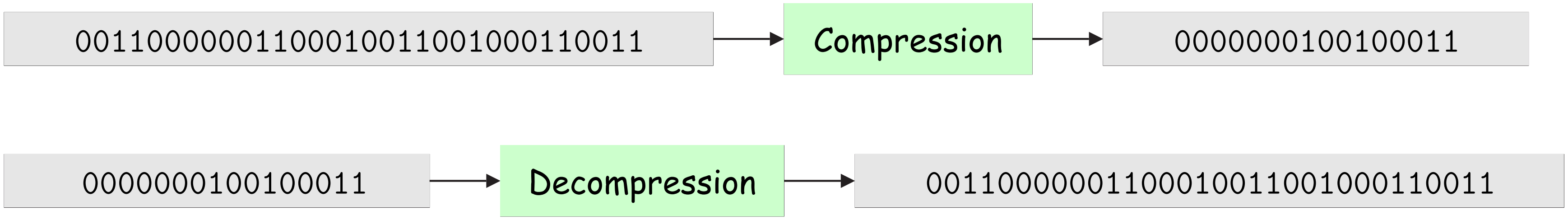
V1	V2
[Fetch 1st 6 lines from V2]	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I’ve recovered my forgotten moral senses, and don’t give twopence halfpenney for any consequences.

Two Unix Compression Programs

```
$ gzip -k lect37.pic.in      # The GNU version of ZIP
$ bzip2 -k lect37.pic.in     # Another compression program
$ ls -l lect37.pic*
#                        Size
#                       (bytes)
-rw-r--r-- 1 hilfingr lisp 14794 Apr 25 11:35 lect37.pic.in
-rw-r--r-- 1 hilfingr lisp  5426 Apr 25 11:35 lect37.pic.in.bz2 # Roughly 1/3 size
-rw-r--r-- 1 hilfingr lisp  5529 Apr 25 11:35 lect37.pic.in.gz
$ gzip -k lect37.pdf
$ ls -l lect37.pdf*
-rw-r--r-- 1 hilfingr lisp 79932 Apr 27 11:21 lect37.pdf
-rw-r--r-- 1 hilfingr lisp 66021 Apr 27 11:21 lect37.pdf.gz   # Roughly 83% size
$ gunzip < lect37.pic.in.gz > lect37.pic.in.ungzip # Uncompress
$ diff lect37.pic.in lect37.pic.in.ungzip
$                # No difference from original (lossless)
$ gzip < lect37.pic.in.gz > lect37.pic.in.gz.gz
$ ls -l lect37.pic*gz
-rw-r--r-- 1 hilfingr lisp 5529 Apr 25 11:35 lect37.pic.in.gz
-rw-r--r-- 1 hilfingr lisp 5552 Apr 27 11:31 lect37.pic.in.gz.gz
$                # Compressing twice doesn't help.
```

Compression and Decompression

- A *compression algorithm* converts a stream of symbols into another, smaller stream.
- It is called *lossless* if the algorithm is *invertible* (no information lost).
- A common symbol is the bit:



- Here, we simply replaced the 8-bit ASCII bit sequences for digits (where, for example, the single character '0' is encoded as `0x30=0b00110000`) with 4-bit (*binary-coded decimal*).
- Call these 4-bit sequences *codewords*, which we associate with the *symbols* in the original, uncompressed text.
- Can do better than 50% compression with English text.

Example: Morse Code

- Compact, simple to transmit.
- Actually use three symbols:
  dih, dah, and pause.
  Pauses go between codewords.

A	●▬	U	●●▬
B	▬●●●	V	●●●▬
C	▬●▬●	W	●▬▬
D	▬●●	X	▬●●▬
E	●	Y	▬●▬▬
F	●●▬●	Z	▬▬●●
G	▬▬●		
H	●●●●		
I	●●	0	▬▬▬▬▬
J	●▬▬▬	1	●▬▬▬▬
K	▬●▬	2	●●▬▬▬
L	●▬●●	3	●●●▬▬
M	▬▬	4	●●●●▬
N	▬●	5	●●●●●
O	▬▬▬	6	▬●●●●
P	●▬▬●	7	▬▬●●●
Q	▬▬●▬	8	▬▬▬●●
R	●▬●	9	▬▬▬▬●
S	●●●		
T	▬		

Prefix Free Codes

- Morse code needs pauses between codewords to prevent ambiguities.
- Otherwise,

  ▬●●●●▬▬●●●●

  could be DEATH, BABE, or BATH.
- The problem is that Morse code allows many codewords to be *prefixes* of other ones, so that it's difficult to know when you have come to the end of one.
- Alternative is to devise *prefix-free codes*, in which no codeword is a prefix of another.
- Then one always knows when a codeword ends.

Prefix-Free Examples

Encoding A

space	1
E	01
T	001
A	0001
O	00001
I	000001
. . .	

Encoding B

space	111
E	010
T	1000
A	1010
O	1011
I	1100
. . .	

- For example, "I ATE" is unambiguously

  0000011000100101 in Encoding A, or

  110011110101000010 in Encoding B.

- What data structures might you use to. . . Encode? Decode?

Prefix-Free Examples

Encoding A

space	1
E	01
T	001
A	0001
O	00001
I	000001
. . .	

Encoding B

space	111
E	010
T	1000
A	1010
O	1011
I	1100
. . .	

- For example, “I ATE“ is unambiguously

  0000011000100101 in Encoding A, or

  1100111101010000010 in Encoding B.

- What data structures might you use to. . .
  Encode? Ans: HashMap or array Decode?

Prefix-Free Examples

Encoding A

space	1
E	01
T	001
A	0001
O	00001
I	000001
. . .	

Encoding B

space	111
E	010
T	1000
A	1010
O	1011
I	1100
. . .	

- For example, “I ATE” is unambiguously

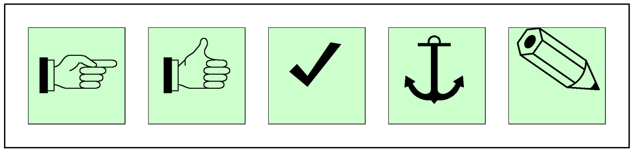
  00000110001001101 in Encoding A, or

  1100111101010000010 in Encoding B.

- What data structures might you use to. . .
  Encode? Ans: HashMap or array Decode? Ans: Trie

Shannon-Fano Coding

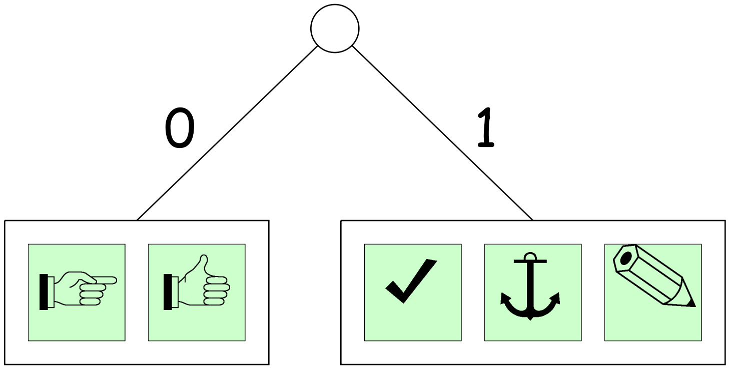
Symbol	Frequency	Encoding
☞	0.35	
👍	0.17	
✓	0.17	
⚓	0.16	
✎	0.15	



- Count frequencies of all characters in text to be compressed.
- Break grouped characters into two groups of roughly equal frequency.
- Encode left group with leading 0, right group with leading 1.
- Repeat until all groups are of size 1.

Shannon-Fano Coding

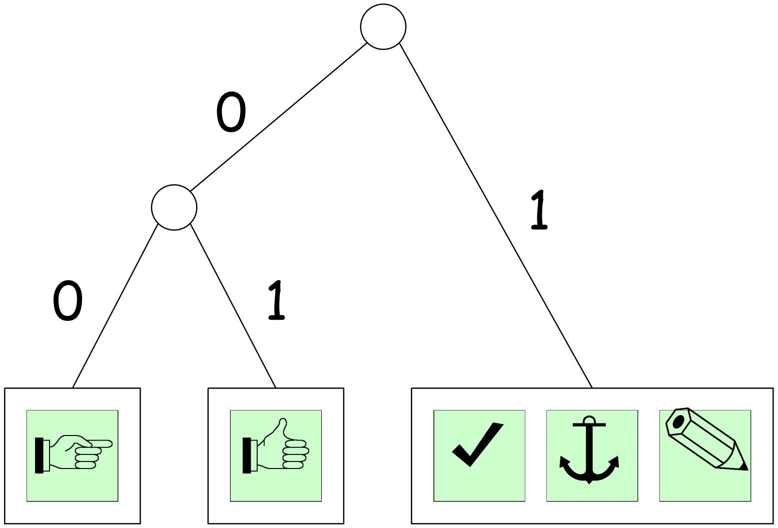
Symbol	Frequency	Encoding
☞	0.35	0...
👍	0.17	0...
✓	0.17	1...
⚓	0.16	1...
✎	0.15	1...



- Count frequencies of all characters in text to be compressed.
- Break grouped characters into two groups of roughly equal frequency.
- Encode left group with leading 0, right group with leading 1.
- Repeat until all groups are of size 1.

Shannon-Fano Coding

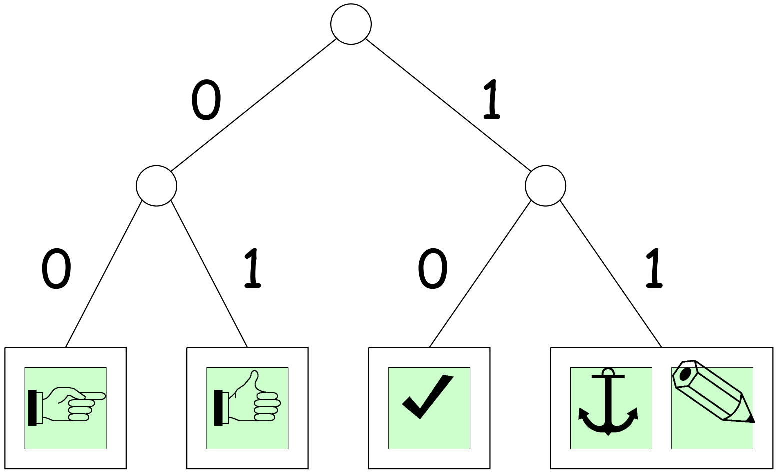
Symbol	Frequency	Encoding
☞	0.35	00
👍	0.17	01
✓	0.17	1. . .
⚓	0.16	1. . .
✎	0.15	1. . .



- Count frequencies of all characters in text to be compressed.
- Break grouped characters into two groups of roughly equal frequency.
- Encode left group with leading 0, right group with leading 1.
- Repeat until all groups are of size 1.

Shannon-Fano Coding

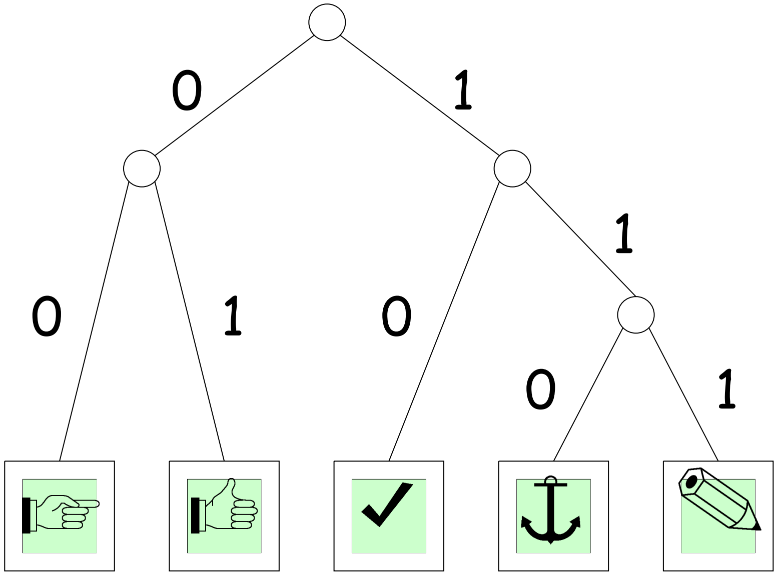
Symbol	Frequency	Encoding
☞	0.35	00
👍	0.17	01
✓	0.17	10
⚓	0.16	11...
✎	0.15	11...



- Count frequencies of all characters in text to be compressed.
- Break grouped characters into two groups of roughly equal frequency.
- Encode left group with leading 0, right group with leading 1.
- Repeat until all groups are of size 1.

Shannon-Fano Coding

Symbol	Frequency	Encoding
☞	0.35	00
👍	0.17	01
✓	0.17	10
⚓	0.16	110
✎	0.15	111

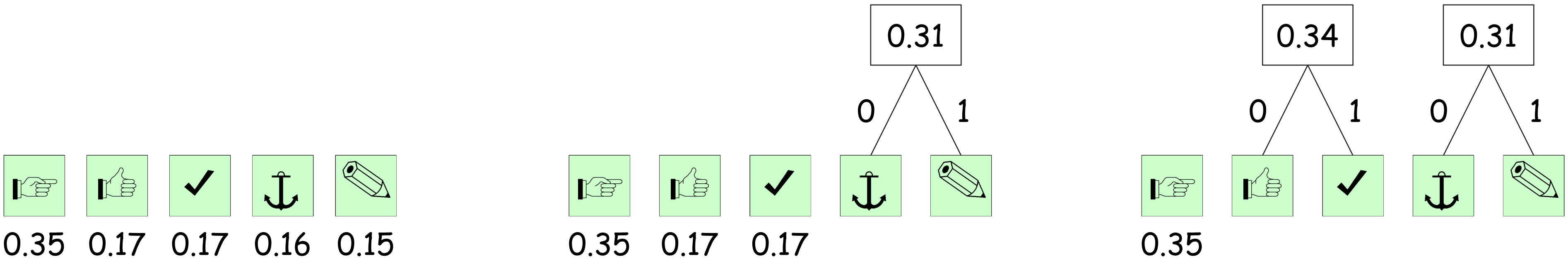


- Count frequencies of all characters in text to be compressed.
- Break grouped characters into two groups of roughly equal frequency.
- Encode left group with leading 0, right group with leading 1.
- Repeat until all groups are of size 1.

Can We Do Better?

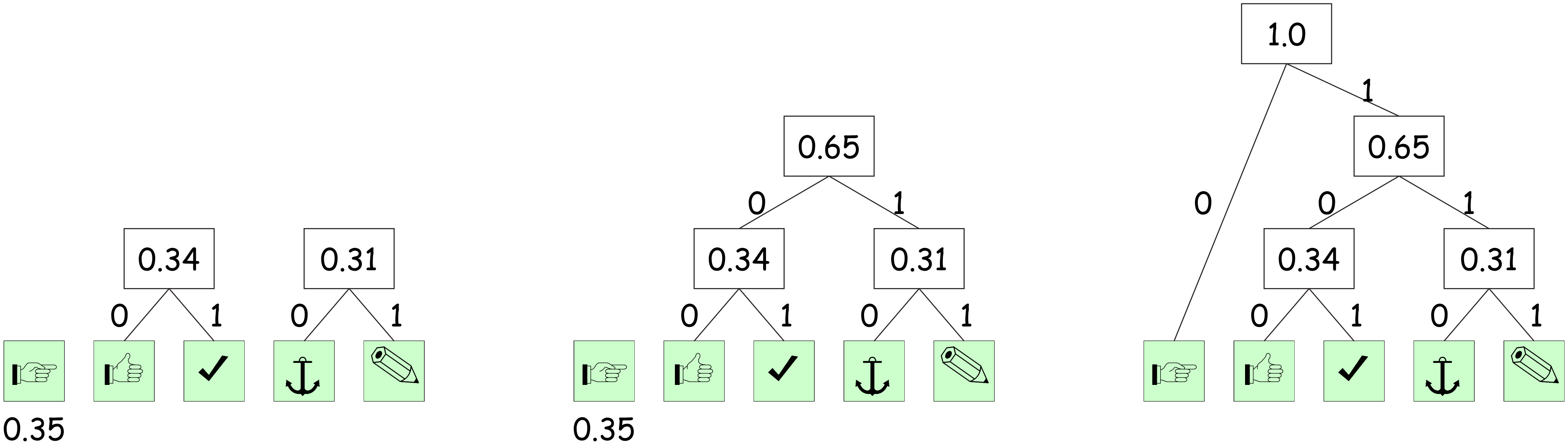
- We'll say an encoding of symbols to codewords that are bitstrings is *optimal* for a particular text if it encodes the text in the fewest bits.
- Shannon-Fano coding is good, but not optimal.
- The optimal solution was found by an MIT graduate student, David Huffman, in a class taught by Fano. The students were given the choice of taking the final or solving this problem (i.e., finding the encoding and a proof of optimality).
- The result is called *Huffman coding*.
- That's right: Fano assigned a problem he hadn't been able to solve. Professors do that occasionally.
- See also this article.

Huffman Coding



- Put each symbol in a node labeled with the symbol's relative frequency (as before).
- Repeat the following until there is just one node:
  - Combine the two nodes with smallest frequencies as children of a new single node whose frequency is the sum of those of the two nodes being combined.
  - Let the edge to the left child be labeled '0' and to the right be labeled '1'.
- The resulting tree shows the encoding for each symbol: concatenate the edge labels on the path from the root to the symbol.

Huffman Coding



- Put each symbol in a node labeled with the symbol's relative frequency (as before).
- Repeat the following until there is just one node:
  - Combine the two nodes with smallest frequencies as children of a new single node whose frequency is the sum of those of the two nodes being combined.
  - Let the edge to the left child be labeled '0' and to the right be labeled '1'.
- The resulting tree shows the encoding for each symbol: concatenate the edge labels on the path from the root to the symbol.

Comparison

Symbol	Frequency	Shannon-Fano	Huffman
☞	0.35	00	0
👍	0.17	01	100
✓	0.17	10	101
⚓	0.16	110	110
✎	0.15	111	111

For this case, Shannon-Fano coding takes a weighted average of 2.31 bits per symbol, while Huffman coding takes 2.3.

LZW Coding

- So far, we have used systems with one codeword per symbol.
- To get better compression, must encoded *multiple* symbols per codeword.
- This will allow us to code strings such as

  ```
  bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
  ababababababababababababababababababababababa
  abcdabcdeabcdefabcdefgabcdefghabcdefghiabcd
  ```

  (each 43 characters long) in space that can be less than
  43 × weighted average symbol length.
- In LZW coding, we create new codewords as we go along, each corresponding to substrings of the text:
  - Start with a trivial mapping of codewords to single symbols.
  - After outputting a codeword that matches the longest possible prefix, X, of the remaining input, add a new codeword Y that maps to the substring X followed by the next input symbol.

Example of LZW encoding

- Start with a trivial mapping of codewords to single symbols.
- After outputting a codeword that matches the longest possible prefix, X, of the remaining input, add a new codeword Y that maps to the substring X followed by the next input symbol.

Consider the following text as an example:

```
B="aababcabcdabcdeabcdefabcdefgabcdefgh"
```

We'll compute $C(B)$, the encoding of B. Our codewords will consist of 8-bit ASCII codes (0x00–0x7f).

LZW Step 0: Initial state

B = $\boxed{\text{a}}$ababcabcdabcdeabcdefabcdefgabcdefgh

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>

C(B) =

LZW Step 1

B = $\boxed{\text{a}}$ababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table is ‘a’, so output 0x61,
- And add $\boxed{\text{a}}$a to the table with a new code.

Code	**String**
`0x61`	`a`
`0x62`	`b`
`0x63`	`c`
...	...
`0x7e`	`~`
`0x7f`	`<DEL>`
`0x80`	`aa`

C(B) = 0x61

LZW Step 2

B = a$\boxed{a}$babcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is still 'a', so output 0x61,
- And add $\boxed{a}$b to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab

C(B) = 0x6161

LZW Step 3

B = aababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is ‘b’, so output 0x62,
- And add ba to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba

C(B) = 0x616162

LZW Step 4

B = aab ab cabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now ‘ab’, so output 0x81 (half as many bits as ‘ab’).
- And add abc to the table with a new code.

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc

C(B) = 0x61616281

LZW Step 5

B = aabab$\boxed{\text{c}}$abcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now ‘c’, so output 0x63
- And add $\boxed{\text{c}}$a to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca

C(B) = 0x6161628163

LZW Step 6

B = aababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now ???, so output ???
- And add ??? to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	???

C(B) = 0x6161628163??

LZW Step 6

B = aababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now 'abc', so output 0x83
- And add `abcd` to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd

C(B) = 0x616162816383

LZW Step 7

B = aababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now ‘d’, so output 0x64
- And add ‘da’ to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd
0x86	da

C(B) = 0x61616281638364

LZW Step 7

B = aababcabcdabcdeabcdefabcdefgabcdefgh

- Best prefix match in the table for remaining input is now ‘d’, so output 0x64
- And add ‘da’ to the table with a new code.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd
0x86	da

C(B) = 0x61616281638364

- What’s next?
- What is the complete encoding? (When reviewing, try to figure it out before looking at the next slide.)

LZW Final State

B = aababcabcdabcdeabcdefabcdefgabcdefgh (200 bits)

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd
0x86	da
0x87	abcde
0x88	ea
0x89	abcdef
0x8a	fa
0x8b	abcdefg
0x8c	ga
0x8d	abcdefgh

$C(B) =$ 0x616162816383648565
876689678b68
(120 bits)

To think about: How might you represent this table to allow easily finding the longest prefix at each step?

Decompression

- Because each different input creates a different table, it would seem that we need to provide the generated table in order to decode a message.
- Interestingly, though, we don't!
- Suppose that, starting with the same initial table we did before, with codes 0x00–0x7f already assigned, we're given

  $$C(B) = \texttt{0x}616162816383$$

  and wish to find B.
- We can see it starts with `aab`. What's next?

Code	**String**
`0x61`	`a`
`0x62`	`b`
`0x63`	`c`
...	...
`0x7e`	`~`
`0x7f`	`<DEL>`

Reconstructing the Coding Table (I)

- Idea is to reconstruct the table as we process each codeword in $C(B)$.
- Let $S(X)$ mean "the symbols encoded by codeword X," and let Y_k mean character k of string Y.
- For each codeword, X, in $C(B)$, add $S(X)$ to our result.
- Whenever we decoded two consecutive codewords, X_1 and X_2, *add* a new codeword that maps to $S(X_1) + S(X_2)_0$
- Thus, we recapitulate a step in the compression operation that created $C(B)$ in the first place.
- Since we go from left to right, the table will (almost) always already contain the mapping we need for the next codeword.

LZW Decompression, Step 1

C(B) = 0x$\boxed{61}$616281638364

- S(0x61) is ‘a’ in the table, so add it to B.
- Don’t have a previous codeword yet, so don’t add anything to the table.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>

B = a

LZW Decompression, Step 2

C(B) = 0x61$\boxed{\texttt{61}}$6281638364

- S(0x61) is ‘a’ in the table, so add it to B.
- We have two codewords—S(0x61)=‘a’ twice—so add ‘aa’ to the table as a new codeword

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa

B = aa

LZW Decompression, Step 3

C(B) = 0x6161$\boxed{62}$81638364

- S(0x62) is 'b' in the table, so add it to B.
- We have two codewords—S(0x61)='a' and S(0x62)='b'—so add 'ab' to the table as a new codeword.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab

B = aab

LZW Decompression, Step 4

C(B) = `0x616162`$\boxed{\texttt{81}}$`638364`

- S(`0x81`) is ‘ab’ in the table, so add it to B.
- We have two codewords—S(`0x62`)=’b’ and S(`0x81`)=’ab’—so add ‘ba’ to the table as a new codeword.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba

B = aabab

LZW Decompression, Step 5

C(B) = 0x61616281$\boxed{63}$8364

- S(0x63) is 'c' in the table, so add it to B.
- We have two codewords—S(0x81)='ab' and S(0x63)='c'—so add 'abc' to the table as a new codeword.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc

B = aababc

LZW Decompression, Step 6

C(B) = 0x6161628163$\boxed{83}$64

- S(0x83) is ??? in the table, so add it to B.
- We have two codewords—S(???)=??? and S(???)=???—so add ??? to the table as a new codeword.

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
???	???

B = aababc???

LZW Decompression, Step 6

C(B) = 0x6161628163$\boxed{83}$64

- S(0x83) is 'abc' in the table, so add it to B.
- We have two codewords—S(0x63)='c' and S(0x83)='abc'—so add 'ca' to the table as a new codeword.

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca

B = aababcabc

LZW Decompression, Step 7

C(B) = 0x616162816383$\boxed{64}$

- S(0x64) is ‘d’ in the table, so add it to B.
- We have two codewords—S(0x83)=‘abc’ and S(0x64)=‘d’—so add ‘abcd’ to the table as a new codeword.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd

B = aababcabcd

Reconstructing the Coding Table (II)

- In a previous slide, I said "... the table will (almost) always already contain the mapping we need..."
- Unfortunately, there are cases where it doesn't.
- Consider the string B='`cdcdcdc`' as an example.
- After we encode it, we end up with

Code	**String**
`0x61`	`a`
`0x62`	`b`
`0x63`	`c`
...	...
`0x7e`	`~`
`0x7f`	`<DEL>`
`0x80`	`cd`
`0x81`	`dc`
`0x82`	`cdc`

C(B) = 0x63648082

- But decoding causes trouble...

Tricky Decompression, Step 1

C(B) = `0x`$\boxed{\texttt{63}}$`648082`

- S(`0x63`) is 'c' in the table, so add it to B.
- Don't have a previous codeword yet, so don't add anything to the table.

Code	**String**
`0x61`	`a`
`0x62`	`b`
`0x63`	`c`
...	...
`0x7e`	`~`
`0x7f`	`<DEL>`

B = c

Tricky Decompression, Step 2

$C(B) = \texttt{0x63}\boxed{\texttt{64}}\texttt{8082}$

- S(0x64) is ‘d’ in the table, so add it to B.
- We have two codewords—S(0x63)=’c’ and S(0x64)=’d’—so add ‘cd’ to the table as a new codeword

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	cd

B = cd

Tricky Decompression, Step 3

C(B) = `0x6364`$\boxed{\texttt{80}}$`82`

- S(`0x80`) is 'cd' in the table, so add it to B.
- We have two codewords—S(`0x64`)='d' and S(`0x80`)='cd'—so add 'dc' to the table as a new codeword

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	cd
0x81	dc

B = cdcd

Tricky Decompression, Step 4

C(B) = `0x636480`$\boxed{\texttt{82}}$

- Oops! S(`0x82`) is not yet in the table. What now?

Code	**String**
`0x61`	`a`
`0x62`	`b`
`0x63`	`c`
...	...
`0x7e`	`~`
`0x7f`	`<DEL>`
`0x80`	`cd`
`0x81`	`dc`
`0x82`	`???`

B = cdcd???

- Problem is that we could look *ahead* while coding, but can only look *behind* when decoding.
- So must figure out what 0x82 is going to be by looking back.

Tricky Decompression, Step 4 (Second Try)

C(B) = `0x636480`$\boxed{\texttt{82}}$

- S(`0x82`)=Z (to be figured out).
- Previously decoded S(0x80)="cd" and now have S(0x82)=Z, so will add "cdZ_0" to the table as S(0x82).
- So Z starts with S(0x80) and therefore Z_0 must be 'c'!
- Thus S(0x82) = S(0x80)+Z_0 = 'cdc'.

Code	**String**
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	cd
0x81	dc
0x82	cdc

B = cdcdcdc

LZW Algorithm

- LZW is named for its inventors: Lempel, Ziv, and Welch.
- Was widely used at one time, but because of patent issues became rather unpopular (especially among open-source folks).
- The patents expired in 2003 and 2004.
- Now found in the *.gif* files, some PDF files, the BSD Unix `compress` utility and elsewhere.
- There are numerous other (and better) algorithms (such as those in `gzip` and `bzip2`).
- The presentation here is considerably simplified.
  - We used fixed-length (8-bit) codewords, but the full algorithm produces variable-length codewords using (!) Huffman coding (compressing the compression).
  - The full algorithm clears the table from time to time to get rid of little-used codewords.

Some Thoughts

- Compressing a compressed text doesn't result in much compression.
- Why must it be impossible to keep compressing a text?
- A program that takes no input and produces an output can be thought of as an encodings of that output.
- Leading to the following question: Given a bitstream, what is the length of the shortest program that can produce it?
- For any specific bitstream, there is a specific answer!
- This is a deep concept, known as Kolmogorov Complexity.

Some Thoughts

- Compressing a compressed text doesn't result in much compression.
- Why must it be impossible to keep compressing a text?
- Otherwise you'd be able to compress any number of different messages to 1 bit!
- A program that takes no input and produces an output can be thought of as an encodings of that output.
- Leading to the following question: Given a bitstream, what is the length of the shortest program that can produce it?
- For any specific bitstream, there is a specific answer!
- This is a deep concept, known as Kolmogorov Complexity.

More Thoughts

- It's actually weird that one can compress much at all.
- Consider a 1000-character ASCII text (8000 bits), and suppose we manage to compress it by 50%.
- There are 2^{8000} distinct messages in 8000 bits, but only 2^{4000} possible messages in 4000 bits.
- That is, no matter what one's scheme, one can encode only 2^{-4000} of the possible 8000-bit messsages by 50%! Yet we do it all the time.
- Reason: Our texts have a great deal of redundancy (aka low *information entropy*).
- Texts with high entropy—such as random bits, previously compressed texts, or encrypted texts—are nearly incompressible.

Git

- Git Actually uses a different scheme from LZW for compression: a combination of LZ77 and Huffman coding.
- LZ77 is kind of like delta compression, but within the same text.
- Convert a text such as

  `One Mississippi, two Mississippi`

  into something like

  `One Mississippi, two <11,7>`

  where the `<11,7>` is intended to mean "the next 11 characters come from the text that ends 7 characters before this point."
- We add new symbols to the alphabet to represent these (length, distance) inclusions.
- When done, Huffman encode the result.

Lossy Compression

- For some applications, like compressing video and audio streams, it really isn't necessary to be able to reproduce the exact stream.
- We can therefore get more compression by throwing away some information.
- Reason: there is a limit to what human senses respond to.
- For example, we don't hear high frequencies, or see tiny color variations.
- Therefore, formats like JPEG, MP3, or MP4 use *lossy compression* and reconstruct output that is (hopefully) imperceptibly different from the original at large savings in size and bandwidth.
- You can see more of this in EE120 and other courses.

Wrapping Up

- Lossless compression saves space (and bandwidth) by exploiting redundancy in data.
- Huffman and Shannon-Fano coding represent individual symbols of the input with shorter codewords.
- LZW and similar codes represents multiple symbols with shorter codewords.
- Both adapt their codewords to the text being compressed.
- Lossy compression both uses redundancy and exploits the fact that certain consumers of compressed data (like humans) can't really use all the information that could be encoded.