Overview

https://www.youtube.com/watch?v=RV5aUr8sZD0
https://www2.cs.duke.edu/csed/curious/compression/lzw.html

- Variation of Lemple-Ziv (LZ77 & LZ78)
- Reduce transmitted data via codes in a dictionary
- Represent repeating patterns of chars as codewords in dictionary
- Dictionary is dynamic (constructed during encoding & decoding processes, not sent with message)
- Used in Unix and GIF compression

Encoding

Summarized from pg 963 of Data Communications textbook

Steps

1. Initialize dictionary with the alphabet and string buffer with the first character
2. Character by character, check if the string buffer + the character is in the dictionary
    a. **If it is**; append the char to the string buffer
    b. **If it isn't**; add the string buffer + character to the dictionary, encode the string buffer, reset the string buffer with the character
3. Repeat step 2 until there is no more input (remember to process the last string in the buffer)

Pseudocode

```
LZW-encode (input):
init dict w/ alphabet
string buffer S = first character //tracks longest encodable sequence
while(more characters in message){
      c = next character
      if S+c in dict: //!Longest table entry yet
            S = S+c
      else if S+c ! in dict: //S = longest table entry
            add S+c to dict
            encode S (add to output) //index of S in dictionary
            S=c
}
encode S (add to output)//whatever is left in the buffer
Output compressed message
```

Decoding

Summarized from pg 964 of Data Communications textbook

Steps

1. Initialize dictionary with the alphabet
2. Read first codeword then get the first character by decoding with the dictionary
3. Add the first character to output
4. Codeword by Codeword, set string buffer to decoded previous codeword and check if the the codeword is in the dictionary
    a. **If it is**; decode the codeword with the dictionary, add the string buffer + the first character of decoded codeword to the dictionary, output decoded codeword
    b. **If it isn't**; output and add the string buffer + first character of string buffer to dict *(**rare**; only happens when calling a table entry that's still being processed (ie. when substring begins & ends w same char), detailed example in duke resource)*
5. Repeat step 2 until there is no more input

Pseudocode

```
LZW-decode (input):
init dict w/ alphabet
cw = first codeword //will be 1 char
Add dict[cw] to output
while(more cw in message){
      String buffer S = dict[cw] //decoded previous codeword
      cw = next codeword
      if cw in dict:
            Add S + first symbol of dict[cw] to dict
            Add dict[cw] to output
      else if cw ! in dict: //rare case
            Add S+ first symbol of S to dict and output
}
Output decompressed message
```

Set Implementation Specs

Command Names: `./compress-lzw` & `./decompress-lzw`

Table Size: 4096 codes (ie.12 bits)

- Once table is filled, begin overwriting from first non ascii table codeword (256)
- Always 12 bits per codeword (instead of changing number of bits; for sake of simplicity)

Alphabet Initialization: codes 0-255 reserved for ascii values (single characters)

- Initialize at beginning of each process & table resets

Input/Output format: (example values not to set specs)

- **Compress**: character string (8 bits) -> 12 bits
    - *Worked example on next page*
    - Input: “TOBEORNOTTOBEORTOBEORNOT”
    - Output:
        - *201525151814152027293136303234*
        - 20 15 2 5 15 18 14 15 20 27 29 31 36 30 32 34
        - 000000010100 000000001111 000000000010 …
- **Decompress**: 12bits -> character string (8 bits)
    - *Worked example on next page*
    - *note: read 12 bits at a time (to elim need of a delimiter)*
    - Input: *201525151814152027293136303234*
    - Output:
        - “TOBEORNOTTOBEORTOBEORNOT”

Example **(example values not to set specs)**

Encode "TOBEORNOTTOBEORTOBEORNOT**" (from wikipedia)**

1. Init table (**note**: we will init from 0 to 255 for all ascii values)

Symbol	Binary	Decimal	Symbol	Binary	Decimal
#	000000000000	0	**M**	000000001101	13
A	000000000001	1	**N**	000000001110	14
B	000000000010	2	**O**	000000001111	15
C	000000000011	3	**P**	000000010000	16
D	000000000100	4	**Q**	000000010001	17
E	000000000101	5	**R**	000000010010	18
F	000000000110	6	**S**	000000010011	19
G	000000000111	7	**T**	000000010100	20
H	000000001000	8	**U**	000000010101	21
I	000000001001	9	**V**	000000010110	22
J	000000001010	10	**W**	000000010111	23
K	000000001011	11	**X**	000000011000	24
L	000000001100	12	**Y**	000000011001	25

2. Encode

   @ comment for 32: we'll use standard 12 bits instead of changing # bits

Current Sequence	Next Char	Output		Extended Dictionary	Comments
		Code	Bits		
NULL	T				
T	O	20	10100	27: TO	27 = first available code after 0 through 26
O	B	15	01111	28: OB	
B	E	2	00010	29: BE	
E	O	5	00101	30: EO	
O	R	15	01111	31: OR	
R	N	18	10010	32: RN	32 requires 6 bits, so for next output use 6 bits
N	O	14	001110	33: NO	
O	T	15	001111	34: OT	
T	T	20	010100	35: TT	
TO	B	27	011011	36: TOB	
BE	O	29	011101	37: BEO	
OR	T	31	011111	38: ORT	
TOB	E	36	100100	39: TOBE	
EO	R	30	011110	40: EOR	
RN	O	32	100000	41: RNO	
OT	#	34	100010		# stops the algorithm; send the cur seq

Output: *201525151814152027293136303234* (in 12 bits per number)

- 20 15 2 5 15 18 14 15 20 27 29 31 36 30 32 34
- 000000010100 000000001111 000000000010 …

Decode: *20152515181415202729313630323 4* **(from wikipedia)**

1. Init table (**note**: we will init from 0 to 255 for all ascii values)

Symbol	Binary	Decimal	Symbol	Binary	Decimal
#	000000000000	0	**M**	000000001101	13
A	000000000001	1	**N**	000000001110	14
B	000000000010	2	**O**	000000001111	15
C	000000000011	3	**P**	000000010000	16
D	000000000100	4	**Q**	000000010001	17
E	000000000101	5	**R**	000000010010	18
F	000000000110	6	**S**	000000010011	19
G	000000000111	7	**T**	000000010100	20
H	000000001000	8	**U**	000000010101	21
I	000000001001	9	**V**	000000010110	22
J	000000001010	10	**W**	000000010111	23
K	000000001011	11	**X**	000000011000	24
L	000000001100	12	**Y**	000000011001	25

2. Decode

@ comment for 32: we'll use standard 12 bits instead of changing # bits

Input		Output Sequence	New Dictionary Entry		Comments
Bits	Code		Full	Conjecture	
10100	20	T		27: T?	
01111	15	O	27: TO	28: O?	
00010	2	B	28: OB	29: B?	
00101	5	E	29: BE	30: E?	
01111	15	O	30: EO	31: O?	
10010	18	R	31: OR	32: R?	created code 31 (last to fit in 5 bits)
001110	14	N	32: RN	33: N?	so start reading input at 6 bits
001111	15	O	33: NO	34: O?	
010100	20	T	34: OT	35: T?	
011011	27	TO	35: TT	36: TO?	
011101	29	BE	36: TOB	37: BE?	36 = TO + 1st symbol (B) of
011111	31	OR	37: BEO	38: OR?	next coded sequence received (BE)
100100	36	TOB	38: ORT	39: TOB?	
011110	30	EO	39: TOBE	40: EO?	
100000	32	RN	40: EOR	41: RN?	
100010	34	OT	41: RNO	42: OT?	

Output: "TOBEORNOTTOBEORTOBEORNOT"

D’Arcy clarifications

Use this section to keep the set mutually informed of decisions, responses from D’Arcy.
If you add something here help everyone out: *notify the set on discord that a clarification has been added*

Here’s a format we can use:

Question: Should we delay output of decode until parity of *the entire file* is established?
D’Arcy’s response: yes

example:
A file has a bad parity bit in the middle of the file.
Your decode should only output an error message (“Bad parity, or something”)

Test cases

Test Case Tracker