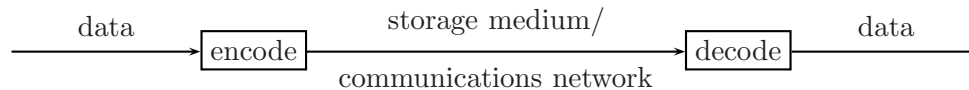


Compression

The Big Picture

We want to be able to store and retrieve data, as well as communicate it with others. In general, this requires **encoding** the data and **decoding** the encoded data:



For the purpose of this lecture, we observe the following constraints:

- We consider only **digital** storage and communication media, in which all information is expressed via a sequence of discrete values (in the simplest case, 0 and 1). This contrasts with **analog** media modeled by continuously varying waveforms.
- We consider only **lossless** storage/transmission in which all information in the original data must be preserved. I.e., for all d , $\text{decode}(\text{encode}(d)) = d$. This contrast with **lossy** approaches that may lose some information (common with images, such as JPEG format, where the data is already an approximation of an analog waveform).
- We consider only **noiseless** storage/transmission in which no errors are introduced between the encoding and decoding phases. In practice, error-correction strategies must be applied to handle errors introduced by real-world “noise”.

Uniform-Length Encoding of Textual Data

For textual data, it is common to encode each character as an 8-bit **byte** using a **uniform-length encoding** known as **ASCII**. Each byte can be written as a decimal integer in the range $[0 .. 255]$. Below is a table showing ASCII values in the range $[0 .. 127]$ and their associated characters:

0: ^@	1: ^A	2: ^B	3: ^C	4: ^D	5: ^E	6: ^F	7: ^G
8: ^H	9: \t	10: \n	11: ^K	12: ^L	13: ^M	14: ^N	15: ^O
16: ^P	17: ^Q	18: ^R	19: ^S	20: ^T	21: ^U	22: ^V	23: ^W
24: ^X	25: ^Y	26: ^Z	27: ^[28: ^\	29: ^]	30: ^^	31: ^_
32: :	33: !	34: "	35: #	36: \$	37: %	38: &	39: '
40: (41:)	42: *	43: +	44: ,	45: -	46: .	47: /
48: 0	49: 1	50: 2	51: 3	52: 4	53: 5	54: 6	55: 7
56: 8	57: 9	58: :	59: ;	60: <	61: =	62: >	63: ?
64: @	65: A	66: B	67: C	68: D	69: E	70: F	71: G
72: H	73: I	74: J	75: K	76: L	77: M	78: N	79: 0
80: P	81: Q	82: R	83: S	84: T	85: U	86: V	87: W
88: X	89: Y	90: Z	91: [92: \	93:]	94: ^	95: _
96: `	97: a	98: b	99: c	100: d	101: e	102: f	103: g
104: h	105: i	106: j	107: k	108: l	109: m	110: n	111: o
112: p	113: q	114: r	115: s	116: t	117: u	118: v	119: w
120: x	121: y	122: z	123: {	124:	125: }	126: ~	127: ^?

For a letter or symbol σ , the notation $\wedge\sigma$ stands for the character specified by pressing the **Control** key and σ key at the same time. The notation $\backslash\text{t}$ stands for the tab character, and $\backslash\text{n}$ for the newline character. Integers in the range $[128 .. 255]$ correspond to other special characters.

Variable Length Encoding of Textual Data

Uniform-length encodings do not take advantage of the fact that different letters have different frequencies. For instance, here is a table of letter frequencies in English (per 1000 letters)¹:

E	130	I	74	D	44	F	28	Y	19	B	9	J	2
T	93	O	74	H	35	P	27	G	16	X	5	Z	1
N	78	A	73	L	35	U	27	W	16	K	3		
R	77	S	63	C	30	M	25	V	13	Q	3		

This suggests a **variable-length encoding** in which frequent letters, like E and T, have shorter representations than infrequent letters. Morse code is an example of a variable length encoding:

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

¹as reported by <http://library.thinkquest.org/28005/flashed/thelab/cryptograms/frequency.shtml>

Information Theory

As implied by variable-length encoding, some strings contains more “information” than others. This is the key idea underlying **information theory**, developed by Claude Shannon in the late 1940s.

The **information content** of a symbol s is defined as

$$I(s) = -\lg(\text{Pr}[s]) \text{ bits}$$

where $(\text{Pr}[s])$ is the predicted probability of a symbol.

For example:

- $I(\text{H})$, where H is getting heads with a fair coin, is $-\lg(1/2) = \lg(2) = 1$ bit.
- $I(\text{H})$, where H is getting heads with a coin that has $\text{Pr}(\text{H}) = 1/16$ is $-\lg(1/16) = \lg(16) = 4$ bits.
- $I(\text{T})$, where T is getting heads with a coin that has $\text{Pr}(\text{T}) = 15/16$ is $-\lg(15/16) = 0.93$ bits.
- $I(\text{e})$, where E is the letter e in English language text = $\text{Pr}(\text{E}) = .13$ is $-\lg(.13) = 2.94$ bits.
- $I(\text{Z})$, where Z is the letter z in English language text = $\text{Pr}(\text{E}) = .13$ is $-\lg(.001) = 9.97$ bits.

The information content of a symbol can depend on the context. Although U has probability 0.027 in general, it has $> .99$ probability following a Q. Although E has probability 0.13 in general, it has close to 0 probability after GH, while T has over .60 probability in this context,

The **entropy** of an alphabet A is the average information content per symbol over the whole alphabet:

$$H(A) = \sum_{s \in A} (\text{Pr}[s] \cdot I(s)) = - \sum_{s \in A} (\text{Pr}[s] \cdot \lg(\text{Pr}[s]))$$

Information content and entropy can depend on the particular text. We will consider the following two examples:

1. "miss mississippi"

The “doodley” poem (from Kurt Vonnegut’s *Cat’s Cradle*):

we do, doodley do, doodley do, doodley do,
what we must, muddily must, muddily must, muddily must,
muddily do, muddily do, muddily do, muddily do,
until we bust, bodily bust, bodily bust, bodily bust.

Huffman Coding

Huffman coding is a technique for choosing variable-length codes for symbols based on their relative probabilities. It was invented by David Huffman in 1952.

Main Encoding Steps:

1. Determine the relative frequency of the symbols in the given text. Symbols may be characters, words, or other strings of characters.
2. Put all symbols into a priority queue weighted by their relative frequency. Dequeue the two lowest frequency elements, and combine them into a **binary trie** whose frequency is the sum of that of the two subtrees. Do this until there is a single **trie**.
3. Make a table mapping each symbol to the bit string path that reaches it from the root of the trie.
4. Encode the text using the table.

Decoding Step: Use bit strings to find each character of text in the encoding trie.

Example: Make the encoding trie for "miss mississippi"

Note: The trie itself must be encoded and transmitted in addition to the encoded text!

Dictionary Methods

Key Idea: Replace substrings in text by **codewords** = indices into a dictionary.

Simple version:

- 8-bit char with high bit 0 stands for self
- 8-bit char with high bit 1 is an index into a dictionary of 128 common multi-character entries (e.g., "st", "tr", "and", "the").

More complex version: Interleave codewords and character strings. E.g., ("m", 17, 17, "ippi"), where 17 is a codeword for "iss".

Which dictionary?

- **static model:** use the same dictionary, regardless of text being coded.
 - *Advantage:* dictionary can be hardwired into encoder/decoder.
 - *Disadvantage:* doesn't model special features of given text.
- **semi-static model:** create dictionary in first pass over text.
 - *Advantage:* dictionary is specialized to text.
 - *Disadvantages:* (1) requires an extra pass and (2) encoder must send dictionary to decoder along with compressed text.
- **adaptive model:** use already processed text as a dictionary that evolves over time.
 - *Advantages:* (1) dictionary is specialized to text; (2) can be done in a single pass; and (3) decoder can reconstruct dictionary from compressed text, so it need not be sent separately.
 - *Disadvantage:* More complex than other methods.

LZ78: An Adaptive Dictionary Method

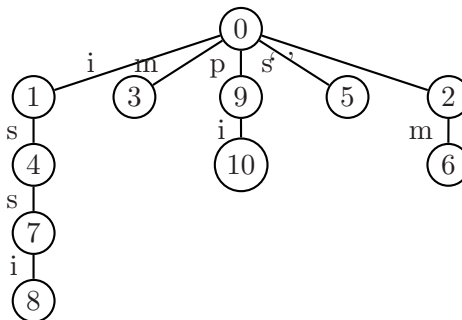
LZ78 is an adaptive dictionary compression method invented by Jacob Ziv and Abraham Lempel in 1978. The more well-known LZW compression algorithm used in the Unix *compress* utility and GIF images is a variant of LZ78.

Idea: Encode text via a sequence of pairs of the form $(codeword, char)$, where *codeword* is an index into a growing dictionary of processed text and *char* is the next character after the string denoted by *codeword*.

Mississippi Example: Here is the LZ78 encoding of "i miss mississippi":

(codeword,char) pairs	dictionary entries
(0, i)	1 \mapsto "i"
(0, $_$)	2 \mapsto " "
(0, m)	3 \mapsto "m"
(1, s)	4 \mapsto "is"
(0, s)	5 \mapsto "s"
(2, m)	6 \mapsto " m"
(4, s)	7 \mapsto "iss"
(7, i)	8 \mapsto "issi"
(0, p)	9 \mapsto "p"
(9, i)	10 \mapsto "pi"

In the encoding phase, the inverse dictionary can be represented efficiently as a **trie** – a multiway-branching tree with nodes bearing values (here, codewords) and edges labeled by keys (here, characters). Below is the trie for the above example:



Doodley Example: Using LZ78, the doodley poem can be encoded with 80 pairs = 160 bytes vs. the 200 uncompressed bytes:

```

[(0, w), (0, e), (0,  $\_$ ), (0, d), (0, o), (0, ' '), (3, d), (5, o),
(4, l), (2, y), (7, o), (6,  $\_$ ), (4, o), (5, d), (0, l), (10,  $\_$ ),
(13, ' '), (11, o), (9, e), (0, y), (11, ' '), (0,  $\_$ ), (1, h), (0, a),
(0, t), (3, w), (2,  $\_$ ), (0, m), (0, u), (0, s), (25, ' '), (3, m),
(29, d), (4, i), (15, y), (32, u), (30, t), (12, m), (33, d), (0, i),
(35,  $\_$ ), (28, u), (37, ' '), (36, d), (34, l), (20,  $\_$ ), (42, s), (31,  $\_$ ),
(42, d), (45, y), (21,  $\_$ ), (49, d), (40, l), (46, d), (5, ' '), (44, d),
(53, y), (51, m), (39, i), (41, d), (55,  $\_$ ), (29, n), (25, i), (15,  $\_$ ),
(1, e), (3, b), (29, s), (31,  $\_$ ), (0, b), (14, i), (41, b), (67, t),
(12, b), (70, l), (46, b), (72, ' '), (66, o), (50,  $\_$ ), (69, u), (37, ' ')]
  
```

LZ77: Another Adaptive Dictionary Method

LZ77 is another adaptive dictionary compression method invented by Ziv and Lempel in 1977. The Linux *gzip* utility is a variant of LZ77.

Idea: Encode text via triples of the form $(back, len, char)$, where:

- *back* is the number of characters back from the current pointer to the start of the dictionary entry.
- *len* is the length of the entry string.
- *char* is the next character in the text after the entry string.

Mississippi Example: Here is the LZ77 encoding of "i miss mississippi":

```
[(0,0,i), (0,0,␣), (0,0,m), (0,0,i) (0,0,s),  
(1,1,␣), (5,4,i), (3,3,p), (1,1,i)]
```

Doodley Example: Using LZ77, the doodley poem can be encoded with 35 triples = 105 bytes vs. the 200 uncompressed bytes:²

```
[(0,0,w), (0,0,e), (0,0,␣), (0,0,d), (0,0,o), (0,0,','), (4,3,o), (3,1,l),  
(0,0,e), (0,0,y), (12,28,␣), (43,1,h), (0,0,a), (0,0,t), (9,1,w), (13,1,␣),  
(0,0,m), (0,0,u), (0,0,s), (8,1,','), (6,3,d), (21,1,i), (27,1,y), (14,34,␣),  
(14,8,d), (80,3,m), (12,34,␣), (11,1,n), (53,1,i), (11,1,␣), (105,3,b), (77,5,b),  
(130,2,i), (26,3,b), (13,29,','.)]
```

²In practice, the compression factor is better than this because the indices themselves are compressed, say via Huffman coding.

Lossless Image Compression

An image is a two-dimensional array of pixels.

- For black-and-white images, consider pixels as 0 or 1.
- For grayscale images, may have gradations between black and white.
- For color images, each pixel has red, green, and blue components, typically 8 bits of information per color.

Uniform Encoding: A simple **bitmap encoding** consists of the width and height of the image followed by a sequence of pixel values (often in row-major order).

Some Image Compression Techniques:

- Use short codewords to index colors in a dictionary. GIF uses a dictionary of 256 colors.
- Use text-based techniques to compress sequences of pixel values. GIF uses standard LZW on pixel bytes.
- Especially in b/w images, can have long sequences of one pixel value. Use **run-length encoding** to shorten these.
- For some images, especially those with noticeable patterns or those composed of geometric shapes like lines, polygons, etc. makes sense to encode image as an **image-generating program**. The decoder “draws” the image by executing the program. This is the big idea behind the PostScript language.

Other Issues

- Tradeoffs between encoding/decoding time and compression factor.
- Most compression techniques do not support random access into compressed text.
- What’s the most effective way to compress a million numbers generated by a random number generator?