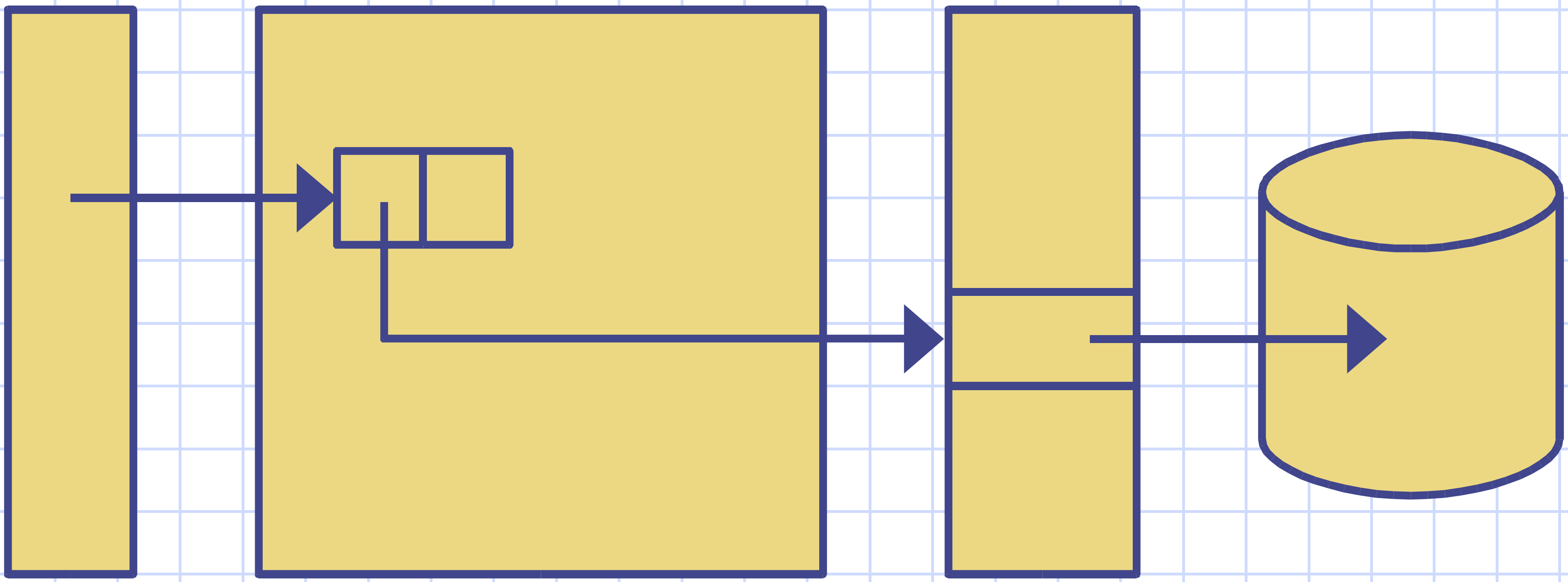
Compression for IR

Lecture 5

IR System Layout



Why Use Compression?

- More storage
  - inverted file and documents use lots of space
  - Compression lets us index more documents
- Faster access
  - A compressed block stores more information
  - Each read brings in more data

Index Compression

- Postings list: list of $< d, f_{t,d} >$
  - or, if we have within-document offsets, $< d; f_{t,d}; o_1..o_{f(t,d)} >$
- 4 bytes for id, counts, offsets expensive
- Can *compress* integers using simple encodings
- Best encoding depends on how values are distributed

Document gaps

- Representing *d* document numbers:
  - keep *d* in ascending order
  - store as sequence of gaps

    3, 5, 20, 21, 23, 76, 77, 78

    becomes: 3, 2, 15, 1, 2, 53, 1, 1
  - Gaps can be efficiently compressed
    - frequent terms have short gaps
    - rare terms have large gaps
- Also for offsets?

Integer Coding

- Nonparametric codes
  - Binary
  - Unary
  - Elias gamma, delta
  - Variable-byte
- Parametric codes
  - Golomb
  - Local Golomb

Unary code

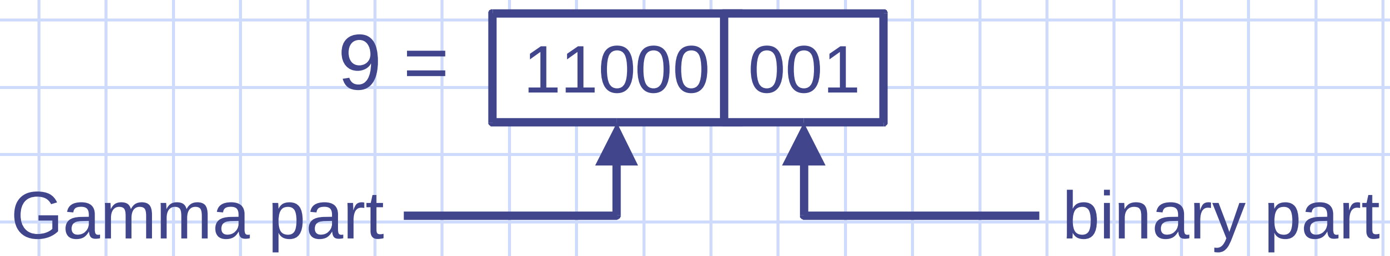
- Encode integer *n* as (n-1) 1's followed by a 0.
  - 1 = 0
  - 3 = 110
  - 9 = 111111110
- Simple and fast
- Smallest numbers have very short codes, but codeword length grows quickly.

Elias' Gamma (γ) code

$$9 = \underbrace{\boxed{1110}}_{\text{unary part}}\underbrace{\boxed{001}}_{\text{binary part}}$$

- Store integer in two parts
  - First part: 1 + floor(lg x) *in unary*
  - Second part: $x - 2^{\text{floor}(\lg x)}$ *in binary* (using floor(lg x) bits)
- Decoding
  - Get unary part c_u (read up through first 0)
  - Get next (c_u-1) bits as binary part
- $\gamma(x) \sim 1 + 2 \lg x$ bits, implying $Pr(x)=1/2x^2$

Elias' Delta (δ) code



- Store integer in two parts
  - First part: $1 + \text{floor}(\lg x)$ *in Gamma code*
  - Second part: $x - 2^{\text{floor}(\lg x)}$ *in binary* (using $\text{floor}(\lg x)$ bits)
- Decoding
  - Decode first gamma-coded part as before
  - Get next (c_u-1) bits as binary part
- $\delta(x) \sim 1 + 2\lfloor \lg \lg 2x \rfloor + \lfloor \lg x \rfloor$ bits
  implying $\Pr(x) = 1/(2x(\lg x)^2)$

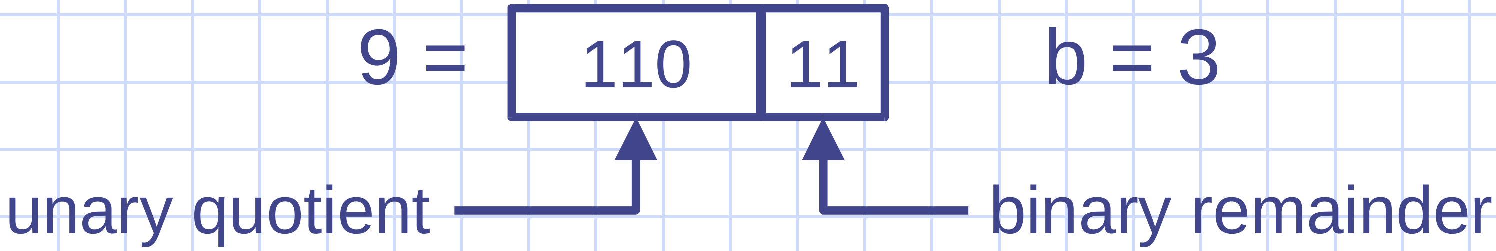
Variable-Byte Code

- Binary, but use minimum number of bytes
- 7 bits to store value, 1 bit to flag if there is another byte
  - $0 < x < 128$: 1 byte
  - $128 < x < 16384$: 2 bytes
  - $16384 < x < 2097152$: 3 bytes
  - Integral byte sizes for easy coding
- Very effective for medium-sized numbers
- A little wasteful for very small numbers

Bernoulli Model

- Idea: use actual density of pointers to parameterize the compression function
- If f = number of pointers, then
  - f / (N * n) = chance that a random document contains some random term
- This implies that if p = probability of a word occurring, then
  - Pr(gap of size x) = $(1\text{-}p)^{x-1}p$
  - ➔ gaps follow a geometric distribution

Golomb code



- For some parameter *b*, integer x represented by
  - quotient part: $\lfloor (x-1)/b \rfloor$ in unary
  - remainder part: x – 1 – qb in binary
- Decoding is simple
  - unary part as usual, binary following *b*

Example Code Values

Gap x	Coding method				
	Unary	Gamma	Delta	Golomb-3	Golomb-6
1	0	0	0	0 0	0 00
2	10	10 0	100 0	0 10	0 01
3	110	10 1	100 1	0 11	0 100
4	1110	110 00	101 00	10 0	0 101
5	11110	110 01	101 01	10 10	0 110
6	111110	110 10	101 10	10 11	0 111
7	1111110	110 11	101 11	110 0	10 00
8	11111110	1110 000	11000 000	110 10	10 01
9	111111110	1110 001	11000 001	110 11	10 100
10	1111111110	1110 010	11000 010	1110 0	10 101

Choosing *b*

- To minimize the average code length

$$b^A = \left\lceil \frac{lg(2-p)}{-lg(1-p)} \right\rceil \approx 0.69 \cdot \frac{N \cdot n}{f}$$

  - Simplification assumes p = f/(N * n) << 1
  - N = number of documents
  - n = number of distinct words
  - f = number of (document, word) pairs

Rice code

- A Rice code is a Golomb code where *b* is a power of 2
- Very fast to decode
  - binary part is the lower *m* bits
  - unary part is in the upper bits
  - easy to implement with shifts and masks

Local Golumb/Rice

- *b* over whole collection may be large
  - average-length not a good target
- Can instead use a *local* model
  - f_t = the number of occurrences of each term
  - (which we store anyway)
  - Compute b^A using $p = f_t/N$ for each term

Bits per entry

Table 3.8 Compression of inverted files in bits per pointer.

Method	Bits per pointer			
	Bible	*GNUbib*	*Comact*	*TREC*
Global methods				
Unary	262	909	487	1918
Binary	15.00	16.00	18.00	20.00
Bernoulli	9.86	11.06	10.90	12.30
γ	6.51	5.68	4.48	6.63
δ	6.23	5.08	4.35	6.38
Observed frequency	5.90	4.82	4.20	5.97
Local methods				
Bernoulli	6.09	6.16	5.40	5.84
Hyperbolic	5.75	5.16	4.65	5.89
Skewed Bernoulli	5.65	4.70	4.20	5.44
Batched frequency	5.58	4.64	4.02	5.41
Interpolative	5.24	3.98	3.87	5.18

Which scheme is best?

- Assuming an index larger than memory
- Smallest index size
  - Delta, Golomb, or Rice for d-gaps
  - Gamma for frequencies
  - Golomb or Rice for offsets
  - Variable-byte for all is competitive
- Fastest query time
  - Golomb d-gaps with Gamma freqs
  - Rice d-gaps with Variable-byte freqs
  - Variable-byte for all best

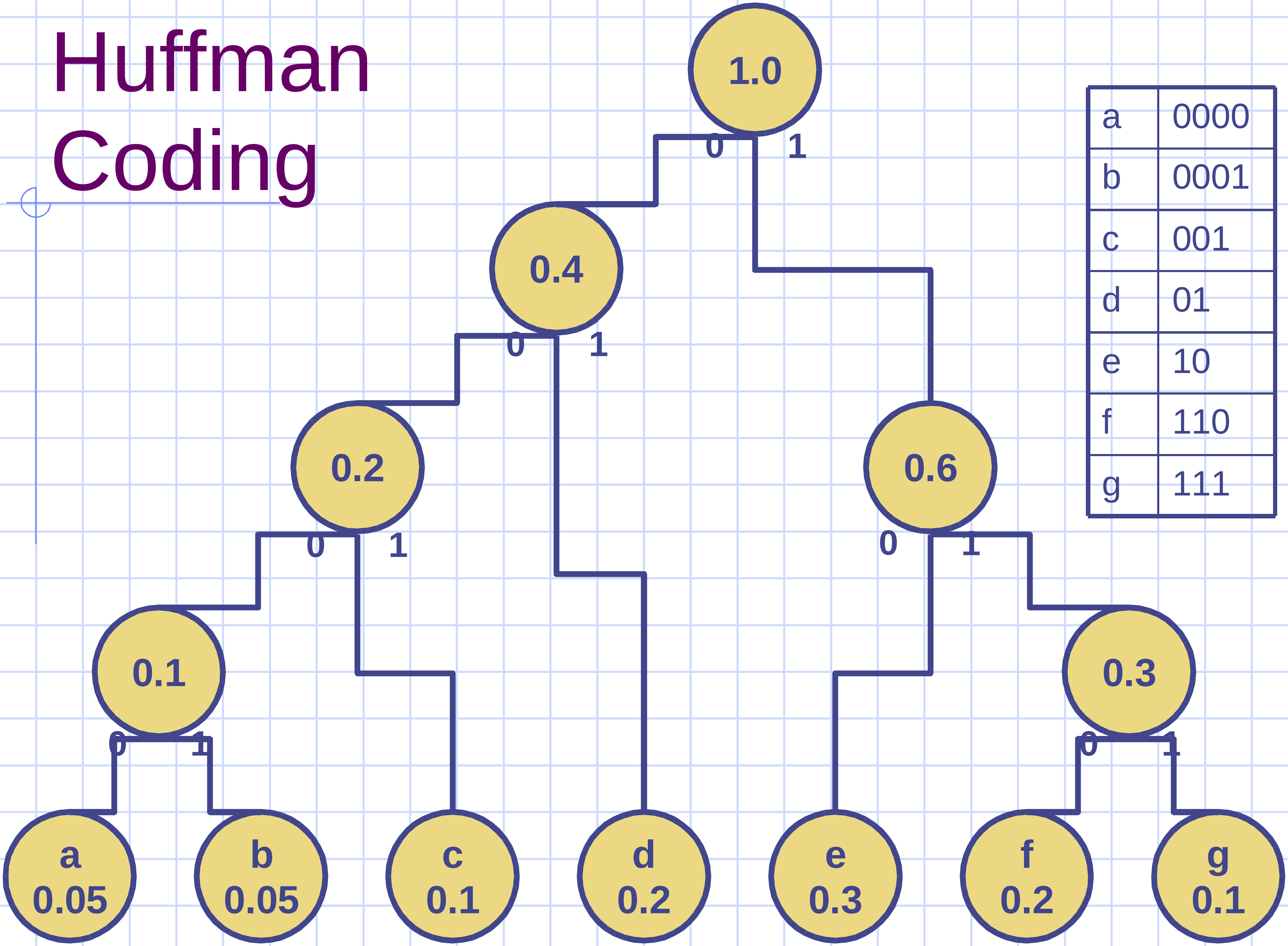
The Moral of the Story

- Index compression is good
  - index is smaller
  - more fits in each block, so it's also faster to read at query time, despite decoding.
  - Variable-byte schemes can even make an in-memory index faster!

Text Compression concepts

- *Alphabet* = set of possible *symbols*
  - words, characters, or fixed-length strings
- Modeling
  - model: probability of each symbol
  - Can be static, semi-static, or adaptive
- Coding
  - Convert symbols into binary digits
  - Use short codes for frequent strings
  - Huffman or arithmetic coding

Huffman Coding



a	0000
b	0001
c	001
d	01
e	10
f	110
g	111

Building the Huffman Tree

1. Input: symbols and their probabilities
2. Loop…
    1. Choose two symbols with smallest P(s)
    2. Join them under a parent node *p*
        - P(p) = P(s1) + P(s2)
    3. Repeat, ignoring nodes that are already children

- Good data structure for this?
- Many possible Huffman trees for a given model

Using a Huffman Code

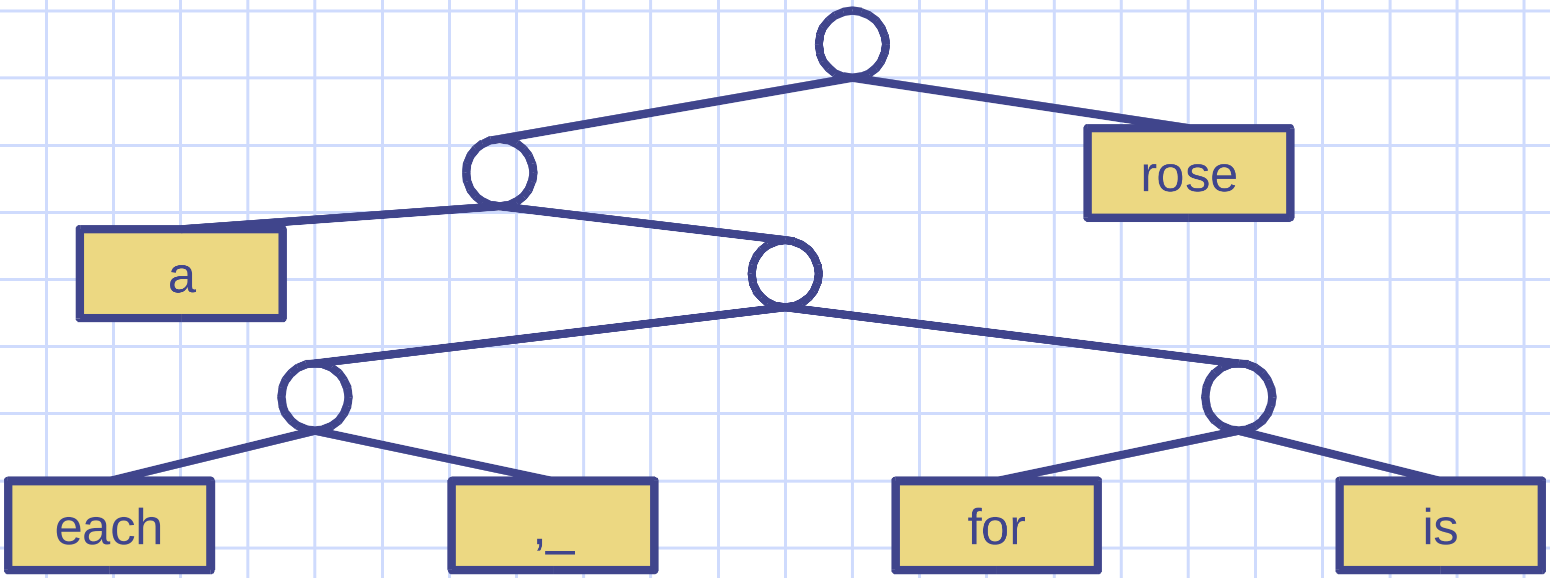
- Encoding
  - look up each symbol in the code table
  - need to output coding tree for decoding
- Decoding
  - start at root of Huffman tree
  - follow appropriate branch for each bit
  - when leaf is reached, output symbol

How good is Huffman?

- Shannon’s Theorem
  - $I(s) = -\lg Pr(s)$ (optimum bits/symbol)
  - $E = \text{sum}[\ Pr(s)\ I(s)\ ] = \text{sum}[-\ Pr(s) \lg Pr(s)\ ]$
- Entropy for example: 2.55 bits/char
- Character-based Huffman gives an average compression rate of about 5 bits/char

Word-based Huffman Coding

- Idea: Use words as symbols
  - Encode words and nonwords separately
  - Or, assume spaces between words
    - encode words and nonspace separators together



Word-based Huffman

- Advantages
  - Compression rate down to ~2 bits/symbol
  - We already know the word frequencies
- Disadvantages
  - The alphabet is VERY large
    - Huffman code tree takes a lot of memory
    - Need the tree in memory for decoding
    - Pointer chasing will cause thrashing

Canonical Huffman Codes

- Same codeword *length* as Huffman code
- But choose codeword *bits* carefully
  - sort words of same codeword length
  - assign codewords in increasing numerical order
  - longer codewords sort first lexically
- Encoding can be determined quickly from
  - codeword length
  - first codeword of that length
  - position in list
- Very compact storage

Canonical Huffman Example

Table 2.2 A canonical Huffman code.

Symbol	Codeword Length	Codeword Bits
100	17	00000000000000000
101	17	00000000000000001
102	17	00000000000000010
103	17	00000000000000011
...	...	...
yopur	17	00001101010100100
youmg	17	00001101010100101
youthful	17	00001101010100110
zeed	17	00001101010100111
zephyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
...	...	...
were	8	10100110
which	8	10100111
as	7	1010100
at	7	1010101
for	7	1010110
had	7	1010111
he	7	1011000
her	7	1011001
his	7	1011010
it	7	1011011
s	7	1011100
said	7	1011101
she	7	1011110
that	7	1011111
with	7	1100000
you	7	1100001
I	6	110001
in	6	110010
was	6	110011
a	5	11010
and	5	11011
of	5	11100
to	5	11101
the	4	1111

(Managing Gigabytes)

Creating a Canonical Code

1. numl[l] <- number of codewords of length l
2. Store value of first code of length l in firstcode[l]

   firstcode[maxlen] = 0

   for l <- (maxlen – 1) downto 1 do

       firstcode[l] <- (firstcode[l+1] + numl[l+1]) / 2
3. nextcode[1..maxlen] <- firstcode[1..maxlen]
4. for i <- 1 to n do
   1. set codeword[i] <- nextcode[l_i]
   2. set symbol[l_i, nextcode[l_i] – firstcode[l_i]] <- i
   3. set nextcode[l_i] <- nextcode[l_i] + 1

Codelengths and Decoding

- Computing codeword lengths is tricky.
- Decoding isn't…

1. set v <- nextinputbit()
   set l <- 1
2. while v < firstcode[l] do
   - set v <- 2*v + nextinputbit()
   - set l <- l + 1
3. Return symbol[l, v – firstcode[l]]