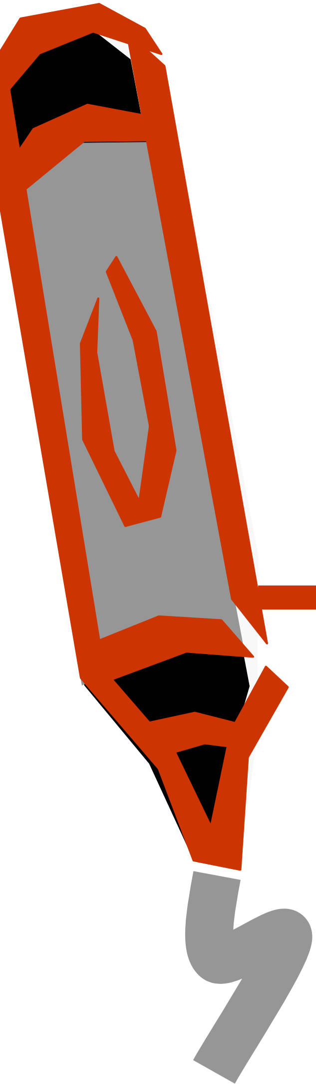


compression

outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

compression

- *Encoding* transforms data from one representation to
- another
- *Compression* is an encoding that takes less space
  - e.g., to reduce load on memory, disk, I/O, network
- *Lossless*: decoder can reproduce message exactly
- *Lossy*: can reproduce message approximately
- *Degree of compression*:
  - (Original - Encoded) / Encoded
  - example: (125 Mb - 25 Mb) / 25 Mb = 400%

compression

- advantages of Compression
- Save space in memory (e.g., compressed cache)
- Save space when storing (e.g., disk, CD-ROM)
- Save time when accessing (e.g., I/O)
- Save time when communicating (e.g., over network)

- Disadvantages of Compression
- Costs time and computation to compress and uncompress
- Complicates or prevents random access
- May involve loss of information (e.g., JPEG)
- Makes data corruption *much* more costly. Small errors may make all of the data inaccessible

compresion

- Text Compression vs Data Compression

- Text compression predates most work on general data compression.

- Text compression is a kind of data compression optimized for text (i.e., based on a language and a language model).
- Text compression can be faster or simpler than general data compression, because of assumptions made about the data.

- Text compression assumes a language and language model
- Data compression learns the model on the fly.

- Text compression is effective when the assumptions are met;
- Data compression is effective on almost any data with a skewed distribution

outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

fixed length compression

- Storage Unit: 5 bits
- If alphabet ≤ 32 symbols, use 5 bits per symbol
- If alphabet > 32 symbols and ≤ 60
  - use 1-30 for most frequent symbols (“base case”),
  - use 1-30 for less frequent symbols (“shift case”), and
  - use 0 and 31 to shift back and forth (e.g., typewriter).
  - Works well when shifts do not occur often.
  - Optimization: Just one shift symbol.
  - Optimization: Temporary shift, and shift-lock
  - Optimization: Multiple “cases”.

fixed length compression : bigrams/digrams

- Storage Unit: 8 bits (0-255)
- Use 1-87 for blank, upper case, lower case, digits and 25 special characters
- Use 88-255 for bigrams (master + combining)
- master (8): blank, A, E, I, O, N, T, U
- combining(21): blank, plus everything but J, K, Q, X, Y Z
- total codes: $88 + 8 * 21 = 88 + 168 = 256$
- Pro: Simple, fast, requires little memory.
- Con: based on a small symbol set
- Con: Maximum compression is 50%.
  - average is lower (33%?).
- Variation: 128 ASCII characters and 128 bigrams.
- Extension: Escape character for ASCII 128-255

fixed length compression : n-grams

- Storage Unit: 8 bits
- Similar to bigrams, but extended to cover sequences of 2 or more characters.
- The goal is that each encoded unit of length > 1 occur with very high (and roughly equal) probability.
- Popular today for:
  - OCR data (scanning errors make bigram assumptions less applicable)
  - asian languages
- two and three symbol words are common
- longer *n*-grams can capture phrases and names

fixed length compression : summary

- Three methods presented. all are
  - simple
  - very effective when their assumptions are correct
- all are based on a small symbol set, to varying degrees
  - some only handle a small symbol set
  - some handle a larger symbol set, but compress best when a few symbols comprise most of the data
- all are based on a strong assumption about the language(English)
- bigram and *n*-gram methods are also based on strong assumptions about common sequences of symbols

outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

restricted variable length codes

- an extension of multicase encodings (“shift key”) where different code lengths are used for each case. Only a few code lengths are chosen, to simplify encoding and decoding.
- Use first bit to indicate case.
- 8 most frequent characters fit in 4 bits (0xxx).
- 128 less frequent characters fit in 8 bits (1xxxxxxx)
- In English, 7 most frequent characters are 65% of occurrences
- Expected code length is approximately 5.4 bits per character, for a 32.8% compression ratio.
- average code length on WSJ89 is 5.8 bits per character, for a 27.9% compression ratio

restricted varible length codes: more symbols

- Use more than 2 cases.
- 1xxx for $2^3 = 8$ most frequent symbols, and
- 0xxx1xxx for next $2^6 = 64$ symbols, and
- 0xxx0xxx1xxx for next $2^9 = 512$ symbols, and
- ...
- average code length on WSJ89 is 6.2 bits per symbol, for a 23.0% compression ratio.

- Pro: Variable number of symbols.
- Con: Only 72 symbols in 1 byte.

restricted variable length codes : numeric data

- 1xxxxxxx for $2^7 = 128$ most frequent symbols
- 0xxxxxxx1xxxxxxx for next $2^{14} = 16{,}384$ symbols
- ...
- average code length on WSJ89 is 8.0 bits per symbol, for a 0.0% compression ratio (!!).
- Pro: Can be used for integer data
  - Examples: word frequencies, inverted lists

restricted variable –length codes : word based encoding

- Restricted Variable-Length Codes can be used on words (as opposed to symbols)
- build a dictionary, sorted by word frequency, most frequent words first
- Represent each word as an offset/index into the dictionary

- Pro: a vocabulary of 20,000-50,000 words with a Zipf distribution requires 12-13 bits per word
  - compared with a 10-11 bits for completely variable length
- Con: The decoding dictionary is large, compared with other methods.

Restricted Variable-Length Codes: Summary

- Four methods presented. all are
  - simple
  - very effective when their assumptions are correct
- No assumptions about language or language models
- all require an unspecified mapping from symbols to numbers (a dictionary)
- all but the basic method can handle any size dictionary

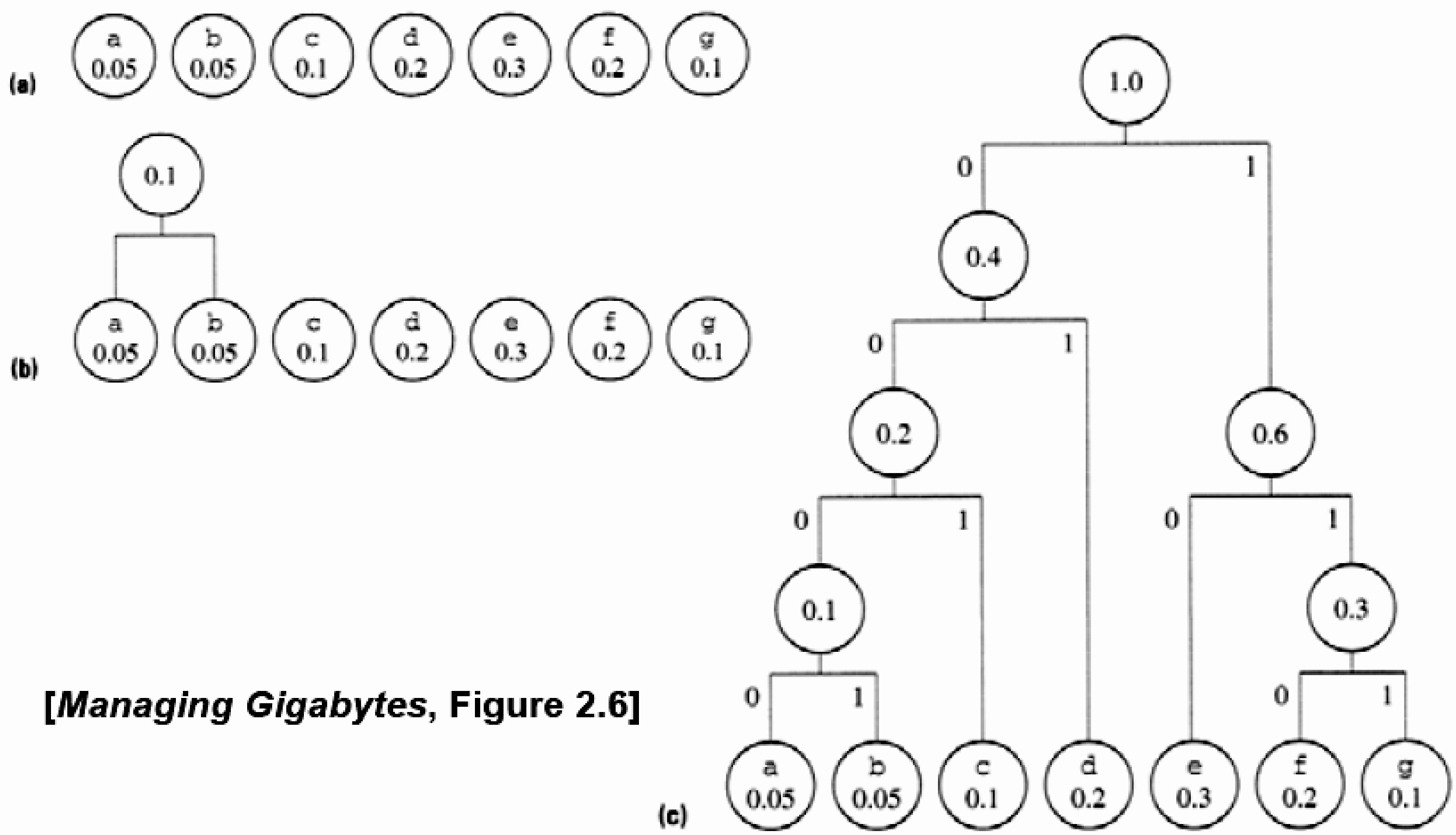
outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

Huffman codes

- Gather probabilities for symbols
  - characters, words, or a mix
- build a tree, as follows:
  - Get 2 least frequent symbols/nodes, join with a parent node.
  - Label least probable branch 0; label other branch 1.
  - $P(\text{node}) = \Sigma_i \, P(\text{child}_i)$
  - Continue until the tree contains all nodes and symbols.
- The path to a leaf indicates its code.
- Frequent symbols are near the root, giving them short codes.
- Less frequent symbols are deeper, giving them longer codes.

Huffman codes



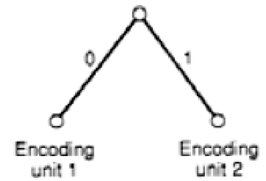
[*Managing Gigabytes*, Figure 2.6]

Huffman codes

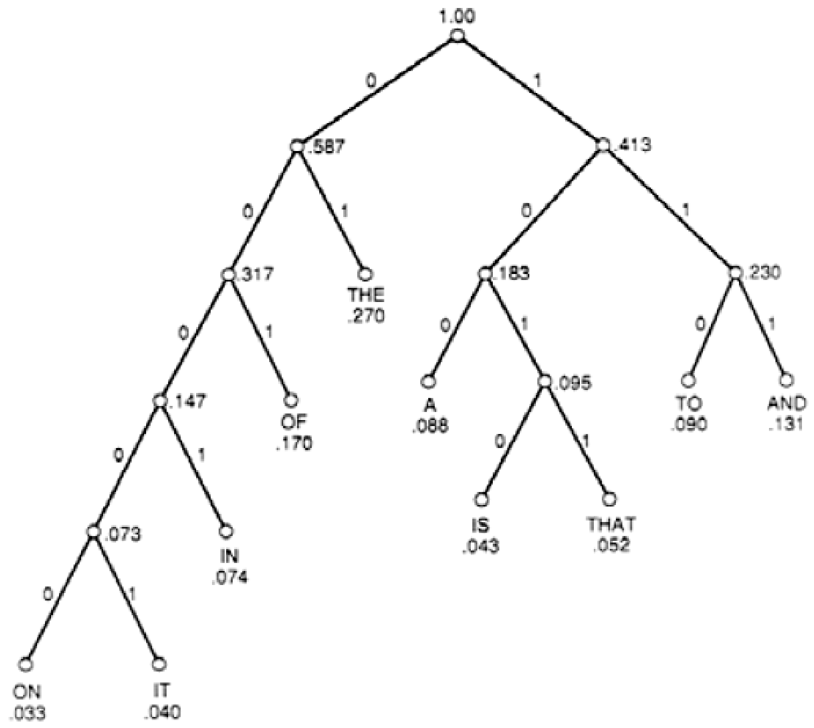
- Huffman codes are “prefix free”; no code is a prefix of another.
- Many codes are not assigned to any symbol, limiting the amount of compression possible.
- English text, with symbols for characters, is approximately 5 bits per character (37.5% compression)
- English text, with symbols for characters and 800 frequent words, yields 4.8-4.0 bits per character (40-50% compression).
- Con: Need a bit-by-bit scan of stream for decoding.
- Con: Looking up codes is somewhat inefficient. The decoder must store the entire tree.
- Traversing the tree involves chasing pointers; little locality.
- Variation: adaptive models learn the distribution on the fly.
- Variation: Can be used on words (as opposed to characters).

Huffman codes

Encoding Unit	Occurrence Probability
the	.270
of	.170
and	.137
to	.099
a	.088
in	.074
that	.052
is	.043
it	.040
on	.033



(a)



Huffman codes

Encoding Unit	Occurrence Probability	Code Value	Code Length
the	.270	01	2
of	.170	001	3
and	.137	111	3
to	.099	110	3
a	.088	100	3
in	.074	0001	4
that	.052	1011	4
is	.043	1010	4
it	.040	00001	5
on	.033	00000	5

Lempel-Ziv

- an adaptive dictionary approach to variable length coding.
- Use the text already encountered to build the dictionary.
- If text follows Zipf's laws, a good dictionary is built.
- No need to store dictionary; encoder and decoder each know how to build it on the fly.
- Some variants: LZ77, Gzip, LZ78, LZW, Unix *compress*
- Variants differ on:
  - how dictionary is built,
  - how pointers are represented (encoded), and
  - limitations on what pointers can refer to.

Lempel Ziv: encoding

- 0010111010010111011011

Lempel Ziv: encoding

- 0010111010010111011011

- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11

Lempel Ziv: encoding

- 0010111010010111011011

- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

Lempel Ziv: encoding

- 0010111010010111011011

- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

- encode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

Lempel Ziv: encoding

- 0010111010010111011011

- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

- encode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

- final string
- 01101100101100011011110100010

Lempel Ziv: decoding

- 011011001011000110111110100010

Lempel Ziv: decoding

- 01101100101100011011110100010

- decode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

Lempel Ziv: decoding

- 01101100101100011011110100010

- decode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

Lempel Ziv: decoding

- 01101100101100011011110100010

- decode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

- decode references
- 0|01 |011|1 |010|0101|11|0110|11

Lempel Ziv: decoding

- 01101100101100011011110100010

- decode the pointers with log(?)bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

- decode references
- 0|01 |011|1 |010|0101|11|0110|11

- original string
- 0010111010010111011011

Lempel Ziv optimality

- LempelZiv compression rate approaches (asymptotic) entropy
  - When the strings are generated by an ergodic source [CoverThomas91].
  - easier proof : for i.i.d sources
    - that is not a good model for English

LempelZiv optimality –i.i.d source

- let $x = \alpha_1\alpha_2 ... \alpha_n$ a sequence of length n generated by a iid source and Q(x) = the probability to see such a sequence

- say LempelZiv breaks into c phrases $x = y_1 y_2 ... y_c$ and call $c_l = \#$ of phrases of length l
then $-\log Q(x) \geq \sum_l c_l \log c_l$
(proof) $\sum_{|y_i|=l} Q(y_i) < 1$ so $\prod_{|y_i|=l} Q(y_i) < (\frac{1}{c_l})^{c_l}$

- if p_i is the source probab for α_i then by law of large numbers x will have roughly np_i occurrences of α_i and then
$logQ(x) = -\log \prod_i p_i^{np_i} \approx n \sum p_i \log p_i = nH_{source}$

- note that $\sum_l c_l \log c_l$ is roughly the LempelZiv encoding length so th einequality reads
$nH \geq\approx LZencoding$ which is to say $H \approx\geq LZrate$.

outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

synchronization

- It is difficult to randomly access encoded text
- With bit-level encoding (e.g., Huffman codes), it is difficult to know where one code ends and another begins.
- With adaptive methods, the dictionary depends upon the prior encoded text.
- Synchronization points can be inserted into an encoded message, from which decoding can begin.
  - For example, pad Huffman codes to the next byte, or restart an adaptive dictionary.
  - Compression effectiveness is reduced, proportional to the number of synchronization points

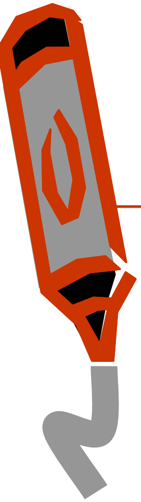
self-syncronizing codes

- In a self-synchronizing code, the decoder can start in the middle of a message and *eventually* synchronize(figure out the code).
- It may not be possible to guarantee *how long* it will take the decoder to synchronize.
- Most variable-length codes are self-synchronizing to some extent
- Fixed-length codes are not self-synchronizing, but boundaries are known (synchronization points).
- adaptive codes are not self-synchronizing.

synchronization

(a)	chillier, but that wasn't to be expected just now.
(b)	chillier, bic that wasn't to be expected just now.
	chillier, bP that wasn't to be expected just now.
	chillier, bft that wasn't to be expected just now.
	chillier, b,t that wasn't to be expected just now.
	chillier, bmt that wasn't to be expected just now.
	chillier, budse, eonasn't to be expected just now.
	chillier, bueea aieonasn't to be expected just now.
	chillier, buh that wasn't to be expected just now.
	chillier, butan, eonasn't to be expected just now.
(c)	chillier, but thaswhrs eree " maem hcL t otaedgsrkeh

(a) Original text, (b) Huffman code with one bit flipped (nine different single bits) and (c) arithmetic coding with one bit flipped [*Managing Gigabytes*, Figure 2.41]



outline

- Introduction
- Fixed Length Codes
  - Short-bytes
  - bigrams / Digrams
  - *n*-grams
- Restricted Variable-Length Codes
  - basic method
  - Extension for larger symbol sets
- Variable-Length Codes
  - Huffman Codes / Canonical Huffman Codes
  - Lempel-Ziv (LZ77, Gzip, LZ78, LZW, Unix *compress*)
- Synchronization
- Compressing inverted files
- Compression in block-level retrieval

compression of inverted files

- Inverted lists are usually compressed
- Inverted files with word locations are about the size of the raw data
- Distribution of numbers is skewed
  - Most numbers are small (e.g., word locations, term frequency)
- Distribution can be made *more* skewed easily
  - Delta encoding: 5, 8, 10, 17 → 5, 3, 2, 7
- Simple compression techniques are often the best choice
  - Simple algorithms nearly as effective as complex algorithms
  - Simple algorithms much faster than complex algorithms
  - Goal: Time saved by reduced I/O > Time required to uncompress

inverted list indexes

- The longest lists, which take up the most space, have the most frequent (probable) words.

- Compressing the longest lists would save the most space.

- The longest lists should compress easily because they contain the least information (why?)

- algorithms:
  - Delta encoding
  - Variable-length encoding
  - Unary codes
  - Gamma codes
  - Delta codes

Inverted List Indexes: Compression

- Delta Encoding ("Storing Gaps")
- Reduces range of numbers.
- Produces a more skewed distribution.
- Increases probability of smaller numbers.
- Stemming also increases the probability of smaller numbers. (Why?)

Inverted List Indexes: Compression

- Variable-Length Codes (Restricted Fixed-Length Codes)

- review the numeric data generalization of restricted variable length codes
- advantages:
  - Effective
  - Global
  - Nonparametric

Inverted List Compression: Unary Code

- Represent a number n ≥ 0 as n 1-bits and a terminating 0.
- Great for small numbers.
- Terrible for large numbers

Inverted List Compression: Gamma Code

- a combination of unary and binary codes

- The unary code stores the number of bits needed to represent *n* in binary.
- The binary code stores the information necessary to reconstruct *n*.
- unary code stores $\lceil \log n \rceil$
- binary code stores n - $2^{\lfloor \log n \rfloor}$
- Example: n = 9
  - log 9 = 3, so unary code is 1110.
  - 9-8=1, so binary code is 001.
  - The complete encoded form is 1110001 (7 bits).
- This method is superior to a binary encoding

Inverted List Compression: Delta Code

- Generalization of the Gamma code

- Encode the length portion of a Gamma code in a Gamma code.
- Gamma codes are better for small numbers.
- Delta codes are better for large numbers.
- Example:
  - For gamma codes, number of bits is 1 + 2 *log n
  - For delta codes, number of bits is:

  log n + 1 + 2 * log(1 + log n)