Introduction to

Information Retrieval

Lecture 5: Index Compression
Last lecture – index construction

- Sort-based indexing
  - Naïve in-memory inversion
  - Blocked Sort-Based Indexing
    - Merge sort is effective for disk-based sorting (avoid seeks!)

- Single-Pass In-Memory Indexing
  - No global dictionary
    - Generate separate dictionary for each block
  - Don’t sort postings
    - Accumulate postings in postings lists as they occur

- Distributed indexing using MapReduce

- Dynamic indexing: Multiple indices, logarithmic merge
Today

- Collection statistics in more detail (with RCV1)
  - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression
Why compression (in general)?

- Use less disk space
  - Saves a little money
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - \([\text{read compressed data } | \text{ decompress}]\) is faster than \([\text{read uncompressed data}]\)
- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use
Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too

- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory

- We will devise various IR-specific compression schemes
### Recall Reuters RCV1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Documents</td>
<td>800,000</td>
</tr>
<tr>
<td>L</td>
<td>Avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>M</td>
<td>Terms (= word types)</td>
<td>(~400,000)</td>
</tr>
<tr>
<td>Avg. # bytes per token (incl. spaces/punct.)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Avg. # bytes per token (without spaces/punct.)</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>Avg. # bytes per term</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>Non-positional postings</td>
<td>100,000,000</td>
<td></td>
</tr>
</tbody>
</table>
## Index parameters vs. what we index
(details *IIR* Table 5.1, p.80)

<table>
<thead>
<tr>
<th>size of</th>
<th>word types (terms)</th>
<th>non-positional postings</th>
<th>positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dictionary</td>
<td>non-positional index</td>
<td>positional index</td>
</tr>
<tr>
<td>Size (K)</td>
<td>Δ%</td>
<td>cumul %</td>
<td>Size (K)</td>
</tr>
<tr>
<td>Unfiltered</td>
<td>484</td>
<td>109,971</td>
<td>197,879</td>
</tr>
<tr>
<td>No numbers</td>
<td>474</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>Case folding</td>
<td>392</td>
<td>-17</td>
<td>-19</td>
</tr>
<tr>
<td>30 stopwords</td>
<td>391</td>
<td>-17</td>
<td>-19</td>
</tr>
<tr>
<td>stemming</td>
<td>322</td>
<td>-17</td>
<td>-33</td>
</tr>
</tbody>
</table>

Exercise: give intuitions for all the ‘0’ entries. Why do some zero entries correspond to big deltas in other columns?
Lossless vs. lossy compression

- **Lossless compression:** All information is preserved.
  - What we mostly do in IR.

- **Lossy compression:** Discard some information

- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.

- Chap/Lecture 7: Prune postings entries that are unlikely to turn up in the top $k$ list for any query.
  - Almost no loss quality for top $k$ list.
Vocabulary vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode 😊
Vocabulary vs. collection size

- Heaps’ law: \( M = kT^b \)
- \( M \) is the size of the vocabulary, \( T \) is the number of tokens in the collection
- Typical values: \( 30 \leq k \leq 100 \) and \( b \approx 0.5 \)
- In a log-log plot of vocabulary size \( M \) vs. \( T \), Heaps’ law predicts a line with slope about \( \frac{1}{2} \)
  - It is the simplest possible relationship between the two in log-log space
  - An empirical finding ("empirical law")
Heaps’ Law

For RCV1, the dashed line
\[ \log_{10} M = 0.49 \log_{10} T + 1.64 \]
is the best least squares fit.

Thus, \( M = 10^{1.64} T^{0.49} \) so \( k = 10^{1.64} \approx 44 \) and \( b = 0.49 \).

Good empirical fit for Reuters RCV1!

For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms

Fig 5.1 p81
Exercises

- What is the effect of including spelling errors, vs. automatically correcting spelling errors on Heaps’ law?

- Compute the vocabulary size $M$ for this scenario:
  - Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
  - Assume a search engine indexes a total of $20,000,000,000$ ($2 \times 10^{10}$) pages, containing 200 tokens on average.
  - What is the size of the vocabulary of the indexed collection as predicted by Heaps’ law?
Zipf’s law

- Heaps’ law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf’s law: The $i$th most frequent term has frequency proportional to $1/i$.
- $\text{cf}_i \propto 1/i = K/i$ where $K$ is a normalizing constant
- $\text{cf}_i$ is **collection frequency**: the number of occurrences of the term $t_i$ in the collection.
Zipf consequences

- If the most frequent term (the) occurs $cf_1$ times
  - then the second most frequent term (of) occurs $cf_1/2$ times
  - the third most frequent term (and) occurs $cf_1/3$ times ...
- Equivalent: $cf_i = K/i$ where $K$ is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship
Zipf’s law for Reuters RCV1
Compression

- Now, we will consider compressing the space for the dictionary and postings
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes
DICTIONARY COMPRESSION
Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn’t in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important
Dictionary storage - first cut

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq.</th>
<th>Postings ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td></td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td></td>
</tr>
</tbody>
</table>

Dictionary search structure

20 bytes

4 bytes each
Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can’t handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.

- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn’t this the number to use for estimating the dictionary size?

- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?

- Short words dominate token counts but not type average.
Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.

<table>
<thead>
<tr>
<th>Freq.</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total string length = $400\,K \times 8B = 3.2\,MB$

Pointers resolve 3.2M positions: $\log_2 3.2M = 22\,\text{bits} = 3\,\text{bytes}$
Space for dictionary as a string

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 ➔ 7.6 MB (against 11.2MB for fixed width)

Now avg. 11 bytes/term, not 20.
Blocking

- Store pointers to every $k$th term string.
  - Example below: $k=4$.
- Need to store term lengths (1 extra byte)

<table>
<thead>
<tr>
<th>Term</th>
<th>Freq.</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>systile</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syzygetic</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syzygial</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syzygy</td>
<td>126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>szaibelyite</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lose 4 bytes on term lengths.

Save 9 bytes on 3 pointers.
Net

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - $3 \times 4 = 12$ bytes,

now we use $3 + 4 = 7$ bytes.

Shaved another $\sim 0.5$ MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB. We can save more with larger $k$.

Why not go with larger $k$?
Exercise

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 4$, 8 and 16.
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons:
\[
\frac{(1+2\cdot2+4\cdot3+4)}{8} \approx 2.6
\]

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?
Dictionary search with blocking

- Binary search down to 4-term block;
  - Then linear search through terms in block.

- Blocks of 4 (binary tree), avg. = \( \frac{1+2 \cdot 2+2 \cdot 3+2 \cdot 4+5}{8} = 3 \) compares
Exercise

- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4, 8$ and $16$. 
Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last $k-1$ in a block of $k$)

\[\text{8automata} \quad \text{8automate} \quad \text{9automatic} \quad \text{10automation}\]

Begins to resemble general string compression.
Front Encoding [Witten, Moffat, Bell]

- Complete front encoding
  - (prefix-len, suffix-len, suffix)
- Partial 3-in-4 front encoding
  - No encoding/compression for the first string in a block
  - Enables binary search

Assume previous string is “auto”

<table>
<thead>
<tr>
<th>String</th>
<th>Complete Front Encoding</th>
<th>Partial 3-in-4 Front Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>8, automata</td>
<td>4, 4, mata</td>
<td>, 8, automata</td>
</tr>
<tr>
<td>8, automate</td>
<td>7, 1, e</td>
<td>7, 1, e</td>
</tr>
<tr>
<td>9, automatic</td>
<td>7, 2, ic</td>
<td>7, 2, ic</td>
</tr>
<tr>
<td>10, automation</td>
<td>8, 2, on</td>
<td>8, , on</td>
</tr>
</tbody>
</table>
## RCV1 dictionary compression summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed width</td>
<td>11.2</td>
</tr>
<tr>
<td>Dictionary-as-String with pointers to every term</td>
<td>7.6</td>
</tr>
<tr>
<td>Also, blocking $k = 4$</td>
<td>7.1</td>
</tr>
<tr>
<td>Also, Blocking + front coding</td>
<td>5.9</td>
</tr>
</tbody>
</table>
POSTINGS COMPRESSION
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.
Postings: two conflicting forces

- A term like *arachnocentric* occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \sim 20$ bits.

- A term like *the* occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case
Postings file entry

- We store the list of docs containing a term in increasing order of docID.
  - *computer*: 33, 47, 154, 159, 202 ...
- **Consequence**: it suffices to store gaps.
  - 33, 14, 107, 5, 43 ...
- **Hope**: most gaps can be encoded/stored with far fewer than 20 bits.
Three postings entries

<table>
<thead>
<tr>
<th>Word</th>
<th>docIDs</th>
<th>postings list</th>
<th>encoding</th>
<th>gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td>...</td>
<td>283042</td>
<td>283043</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>283044</td>
<td>283045</td>
<td>1</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>...</td>
<td>283047</td>
<td>283154</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td></td>
<td>283159</td>
<td>283202</td>
<td>5</td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td>252000</td>
<td>500100</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>252000</td>
<td>248100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable length encoding

- **Aim:**
  - For *arachnocentric*, we will use ~20 bits/gap entry.
  - For *the*, we will use ~1 bit/gap entry.

- **If the average gap for a term is** $G$, **we want to use** ~$\log_2 G$ **bits/gap entry.**

- **Key challenge:** encode every integer (gap) with about as few bits as needed for that integer.

- This requires a *variable length encoding*

- Variable length codes achieve this by using short codes for small numbers
Variable Byte (VB) codes

- For a gap value $G$, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store $G$ and dedicate 1 bit in it to be a continuation bit $c$
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode $G$’s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$. 
### Example

<table>
<thead>
<tr>
<th>docIDs</th>
<th>824</th>
<th>829</th>
<th>215406</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaps</td>
<td></td>
<td>5</td>
<td>214577</td>
</tr>
<tr>
<td>VB code</td>
<td>00000110 10111000</td>
<td>10000101</td>
<td>00001101 00001100 10110001</td>
</tr>
</tbody>
</table>

Postings stored as the byte concatenation:

00000110 10111000 00000110 00001101 10000101 00001101 00001100 10110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Hex(824) = 0x0338
Hex(214577) = 0x00034631
Other variable unit codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
  - Used by many commercial/research systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).
- There is also recent work on word-aligned codes that pack a variable number of gaps into one word (e.g., simple9)
Simple9

- Encodes as many gaps as possible in one DWORD
- 4 bit selector + 28 bit data bits
  - Encodes 9 possible ways to “use” the data bits

<table>
<thead>
<tr>
<th>Selector</th>
<th># of gaps encoded</th>
<th>Len of each gap encoded</th>
<th>Wasted bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>28</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>14</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>9</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0101</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>3</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>2</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>28</td>
<td>0</td>
</tr>
</tbody>
</table>
Unary code

- Represent \( n \) as \( n \) 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is
  \[
  111111111111111111111111111111111111111110.
  \]
- Unary code for 80 is:
  \[
  11111111111111111111111111111111111111111111
  11111111111111111111111111111111111111111110
  \]
- This doesn’t look promising, but....
Bit-Aligned Codes

- Breaks between encoded numbers can occur after any bit position
- **Unary code**
  - Encode $k$ by $k$ 1s followed by 0
  - 0 at end makes code unambiguous

<table>
<thead>
<tr>
<th>Number</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
</tr>
<tr>
<td>5</td>
<td>111110</td>
</tr>
</tbody>
</table>
 Unary and Binary Codes

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- Binary is more efficient for large numbers, but it may be ambiguous
Elias-γ Code

- To encode a number $k$, compute
  - $k_d = \lceil \log_2 k \rceil$
  - $k_r = k - 2^{\lceil \log_2 k \rceil}$

- $k_d$ is number of binary digits, encoded in unary

<table>
<thead>
<tr>
<th>Number ($k$)</th>
<th>$k_d$</th>
<th>$k_r$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>11010</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>1110111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>1111000000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>1111111110111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>111111111101111111</td>
</tr>
</tbody>
</table>
Elias-δ Code

- Elias-γ code uses no more bits than unary, many fewer for \( k > 2 \)
  - 1023 takes 19 bits instead of 1024 bits using unary
- In general, takes \( 2 \left\lfloor \log_2 k \right\rfloor + 1 \) bits
- To improve coding of large numbers, use Elias-δ code
  - Instead of encoding \( k_d \) in unary, we encode \( k_d + 1 \) using Elias-γ
  - Takes approximately \( 2 \log_2 \log_2 k + \log_2 k \) bits
Elias-ε Code

- Split \((k_d + 1)\) into:
  \[
  k_{dd} = \left\lfloor \log_2(k_d + 1) \right\rfloor \\
  k_{dr} = (k_d + 1) - 2^{\left\lfloor \log_2(k_d + 1) \right\rfloor}
  \]
- encode \(k_{dd}\) in unary, \(k_{dr}\) in binary, and \(k_r\) in binary

<table>
<thead>
<tr>
<th>Number ((k))</th>
<th>(k_d)</th>
<th>(k_r)</th>
<th>(k_{dd})</th>
<th>(k_{dr})</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10 0 1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>10 1 10</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>110 00 111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>110 01 0000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>3</td>
<td>0</td>
<td>1110 000 1111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>3</td>
<td>2</td>
<td>1110 010 111111111</td>
</tr>
</tbody>
</table>
# Generating Elias-gamma and Elias-delta codes in Python
#

import math

def unary_encode(n):
    return "1" * n + "0"

def binary_encode(n, width):
    r = ""
    for i in range(0, width):
        if ((1<i) & n) > 0:
            r = "1" + r
        else:
            r = "0" + r
    return r

def gamma_encode(n):
    logn = int(math.log(n,2))
    return unary_encode( logn ) + " " + binary_encode(n, logn)

def delta_encode(n):
    logn = int(math.log(n,2))
    if n == 1:
        return "0"
    else:
        loglog = int(math.log(logn+1,2))
        residual = logn+1 - int(math.pow(2, loglog))
        return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

if __name__ == "__main__":
    for n in [1,2,3, 6, 15,16,255,1023]:
        logn = int(math.log(n,2))
        loglogn = int(math.log(logn+1,2))
        print n, "d_r", logn
        print n, "d_dd", loglogn
        print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
        print n, "delta", delta_encode(n)
        #print n, "gamma", gamma_encode(n)
        #print n, "binary", binary_encode(n)
Gamma code properties

- $G$ is encoded using $2\lceil\log G\rceil + 1$ bits
  - Length of offset is $\lfloor\log G\rfloor$ bits
  - Length of length is $\lceil\log G\rceil + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$

- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
- Gamma code is parameter-free
Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost
Shannon Limit

- Is it possible to derive codes that are optimal (under certain assumptions)?
- What is the optimal average code length for a code that encodes each integer (gap lengths) independently?
- Lower bounds on average code length: Shannon entropy
  - $H(X) = - \sum_{x=1}^{n} \Pr[X=x] \log \Pr[X=x]$
- Asymptotically optimal codes (finite alphabets): arithmetic coding, Huffman codes
Global Bernoulli Model

- Assumption: term occurrence are Bernoulli events
- Notation:
  - \( n \): # of documents, \( m \): # of terms in vocabulary
  - \( N \): total # of (unique) occurrences
- Probability of a term \( t_j \) occurring in document \( d_i \): \( p = \frac{N}{nm} \)
- Each term-document occurrence is an independent event
- Probability of a gap of length \( x \) is given by the geometric distribution
  \[ \Pr[X = x] = (1 - p)^{x-1} \cdot p \]
Golomb Code

- Golomb Code (Golomb 1966): highly efficient way to design optimal Huffman-style code for geometric distribution
  - Parameter $b$
  - For given $x \geq 1$, computer integer quotient
  - and remainder

- Assume $b = 2^k$
  - Encode $q$ in unary, followed by $r$ coded in binary
  - A bit complicated if $b \neq 2^k$. See wikipedia.

- First step: $(q+1)$ bits
- Second step: $\log(b)$ bits

It can also be deemed as a generalization of the unary code.
Golomb Code & Rice Code

- How to determine optimal $b^*$?
- Select minimal $b$ such that
  \[(1 - p)^b + (1 - p)^{b+1} \leq 1\]
- Result due to Gallager and Van Voorhis 1975: generates an optimal prefix code for geometric distribution
- Small $p$ approximation:
  \[b^* \approx \ln 2/p = 0.69 \cdot \text{avg_val}\]
- Rice code: only allow $b = 2^k$
Local Bernoulli Model

- If length of posting lists is known, then a Bernoulli model on each individual inverted list can be used
- Frequent words are coded with smaller b, infrequent words with larger b
- Term frequency need to be encoded (use gamma-code)
- Local Bernoulli outperforms global Bernoulli model in practice (method of practice!)
## RCV1 compression

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>with blocking, ( k = 4 )</td>
<td>7.1</td>
</tr>
<tr>
<td>with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
<tr>
<td>collection (text, xml markup etc)</td>
<td>3,600.0</td>
</tr>
<tr>
<td>collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>Term-doc incidence matrix</td>
<td>40,000.0</td>
</tr>
<tr>
<td>postings, uncompressed (32-bit words)</td>
<td>400.0</td>
</tr>
<tr>
<td>postings, uncompressed (20 bits)</td>
<td>250.0</td>
</tr>
<tr>
<td>postings, variable byte encoded</td>
<td>116.0</td>
</tr>
<tr>
<td>postings, g-encoded</td>
<td>101.0</td>
</tr>
</tbody>
</table>
Google’s Indexing Choice

- Index shards partition by doc, multiple replicates
- Disk-resident index
  - Use outer parts of the disk
  - Use different compression methods for different fields: Rice\(_k\) (a special kind of Golomb code) for gaps, and Gamma for positions.
- In-memory index
  - All positions; No docid
    - Keep track of document boundaries
  - Group-variant encoding
    - Fast to decode

Source: Jeff Dean’s WSDM 2009 Keynote
Other details

- Gap = docidₙ - docidₙ₋₁ - 1
- Freq = freq – 1
- Pos_Gap = posₙ - posₙ₋₁ - 1

Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection
- However, we’ve ignored positional information
- Hence, space savings are less for indexes used in practice
  - But techniques substantially the same.
Resources for today’s lecture

- *IIR* 5
- *MG* 3.3, 3.4.
  - Variable byte codes
  - Word aligned codes