Overview of this lecture

- **Organizational**
  - Your experiences with ES3 Efficient List Intersection

- **Compression**
  - Motivation saves space and query time
  - Codes Elias, Golomb, Variable-Byte
  - Entropy Shannon's famous theorem
  - **Exercise Sheet 4:** three nice proofs → part of Shannon's theorem + optimality of Golomb + size of inverted index

We take a break from implementation work this week.
Experiences with ES3  1/2

Summary / excerpts

- Interesting exercise, many liked performance tweaking
- Less work than ES2 again
- Lack of programming practice in Java or C++
- People who started late took much longer
- Some found it hard to make an improvement
- Given code already used native arrays and "while" trick
- Some of you had large variation between runs
- Coding while watching the US election results is even worse than lack of sleep, etc.
Results

- Three inverted lists of different lengths
  
  **them**  1,717,305 postings  
  **existence**  162,511 postings  
  **bielefeld**  5,257 postings  

- Query **them+bielefeld**, list length ratio = **327**
  
  Any of galloping, skip ptrs, bin. search give large speedup  

- Query **existence+bielefeld**, list length ratio = **31**
  
  Skipping helps, but not too much  

- Query **them+existence**, list length ratio = **11**
  
  Skipping costs more than it helps, switch to tuned baseline
Compression 1/6

- **Motivation**

  - Inverted lists can become very large

  Recall: length of an inverted list of a word = total number of occurrences of that word in the collection

  For example, in the English Wikipedia:

  - **them:** 1,717,305 occurrences
  - **year:** 2,052,964 occurrences
  - **one:** 4,022,417 occurrences

  - Compression potentially saves space *and* time
Compression  2/6

- Index in **memory**
  - Then compression saves memory (obviously)
  - Also: the index might be too large to fit into memory without compression, and with compression it does

  Fitting in memory is good because reading from memory is much much **much** faster than reading from disk

  Transfer rate from memory $\approx 2$ GB / second

  Transfer rate from disk $\approx 50$ MB / second
Index on **disk:**

- Then compression saves disk space (obviously)
- But it also saves query time, here is a realistic example:

  Disk transfer time: 50 MB / second
  Compression rate: Factor 5
  Decompression time: 30 MB / second
  Inverted list of size: 50 MB

Reading uncompressed: 1.0 seconds → 50 MB
Reading compressed: 0.2 seconds → 10 MB
Decompressing: 0.3 seconds → 50 MB

Reading compressed + decompression **twice faster** compared to reading uncompressed
Gap encoding

- Example inverted list (doc ids only):
  
  3, 17, 21, 24, 34, 38, 45, ..., 11876, 11899, 11913, ...

- Numbers small in the beginning, large in the end, using an int for each id would be **4 bytes per id**

- Alternative: store differences from one item to next:
  
  +3, +14, +4, +3, +10, +4, +7, ..., +12, +23, +14, ...

- This is called **gap encoding**

- Works as long as we process the lists from left to right

- Now we have a sequence of mostly (but not always) small numbers ... how do we store these in little space?
Compression 5/6

- Binary representation

- We can write number $x$ in binary using $\lfloor \log_2 x \rfloor + 1$ bits

<table>
<thead>
<tr>
<th>$x$</th>
<th>binary</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
</tr>
</tbody>
</table>

$\lfloor \log_2 1 \rfloor + 1 = 1$
$\lfloor \log_2 2 \rfloor + 1 = 2$
$\lfloor \log_2 3 \rfloor + 1 = 2$
$\lfloor \log_2 4 \rfloor + 1 = 3$

- This encoding is optimal in a sense ... see later slides

- So why not just (gap-)encode like this and concatenate:

$$+3, +14, +4, \ldots \rightarrow 11, 1110, 100, \ldots \rightarrow 111110100\ldots$$
Prefix-free codes, definition

- Decode bit sequence from the last slide: \(111110100\)
  
  This could be: \(+3, +14, +4\) \(\rightarrow\) \(11, 1110, 100\)

  Could also be: \(+7, +6, +4\) \(\rightarrow\) \(111, 110, 100\)

  Or: \(+3, +3, +2, +4\) \(\rightarrow\) \(11, 11, 10, 100\)

- Problem: we have no way to tell where one code ends and the next code begins

  Equivalently: some codes are prefixes of other codes

- In a **prefix-free code**, no code is a prefix of another

  Then decoding from left to right is unambiguous!
Elias-Gamma … from 1975

- Write $\lfloor \log_2 x \rfloor$ zeros, then $x$ in binary like on slide 9
- Prefix-free, because the number of initial zeros tells us exactly how many bits of the code come afterwards
- Code for $x$ has a length of exactly $2 \cdot \lfloor \log_2 x \rfloor + 1$ bits

\[
\begin{array}{c|c}
1 & 1 \\
2 & 010 \\
3 & 011 \\
4 & 00100 \\
\vdots & \vdots \\
10 & 0001010 \\
\end{array}
\]
Elias-Delta … also from 1975

- Write \( \lfloor \log_2 x \rfloor + 1 \) in Elias-Gamma, followed by \( x \) in binary (like on slide 9) but **without** the leading 1.

- Elias-Delta is also prefix-free and the length of the code length is \( \lfloor \log_2 x \rfloor + 2 \log_2 \log_2 x + O(1) \) bits.

\[
\begin{array}{cccccc}
1 & & & 1 & & \\
2 & 010 & 0 & & & \\
3 & 0101 & & & & \\
4 & 0110 & 0 & & & \\
\vdots & & & & & \\
10 & 00100 & 010 & & & \\
\end{array}
\]

Elias-Gamma Codes

\[
\begin{array}{cccc}
1 & 1 & & \\
2 & 010 & & \\
3 & 011 & & \\
4 & 00100 & & \\
\vdots & & & \\
10 & 0001010 & & \\
\end{array}
\]
Golomb (not Gollum) ... from 1966

- Comes with an integer parameter $M$, called \textit{modulus}
- Write $x$ as $q \cdot M + r$, where $q = x \div M$ and $r = x \mod M$
- The code for $x$ is then the concatenation of:
  - $q$ written in unary with 0s
  - A single 1 (as a delimiter)
  - $r$ written in binary

Solomon Golomb
1932 – 2016
- Idea: use **whole bytes**, in order to avoid the (expensive) bit fiddling needed for the previous schemes

  VB often used in practice, for exactly that reason

- Use one bit of each byte to indicate whether this is the last byte in the current code or not

- VB is also used for **UTF-8 encoding** ... see later lecture

\[ x = 501 = 3 \cdot 128 + 117 \]

Write a 2 bytes

\[ 000001111110101 \]

should be 501

[Diagram of encoding process]
Motivation

- Which code compresses the best?
  
  It depends!

  But on what?

- Roughly: it depends, on the relative frequency on the numbers / symbols we want to encode

  For example, in natural language, an "e" is much more frequent than a "z"

  So we should encode "e" with less bits than "z"

- The next slides will make this more precise
Entropy

- **Intuitively:** the information content of a message = the optimal number of bits to encode that message
- **Formally:** defined for a discrete random variable \( X \)
  
  Without loss of generality range of \( X = \{1, \ldots, m\} \)

  Think of \( X \) as generating the symbols of the message

  Then the **entropy** of \( X \) is written and defined as

  \[
  H(X) = - \sum p_i \log_2 p_i \quad \text{where } p_i = \text{Prob}(X = i)
  \]

  - Example 1: one \( p_i = 1 \) all other 0, then \( H(X) = 0 \)
  - Example 2: all \( p_i = 1/m \), then \( H(X) = \log_2 m \)
Shannon's source coding theorem ... from 1948

- Let $X$ be a random variable with finite range

- For an arbitrary prefix-free (PF) encoding, let $L_x$ be the length of the code for $x \in \text{range}(X)$

(1) For any PF encoding it holds: $\mathbb{E} L_X \geq H(X)$

(2) There is a PF encoding with: $\mathbb{E} L_X \leq H(X) + 1$

where $\mathbb{E}$ denotes the expectation

**In words:** no code can be better than the entropy, and there is always a code as good
Central Lemma ... to prove the source coding theorem

- Denote by $L_i$ the length of the code for the $i$-th symbol, then

  (1) Given a PF code with lengths $L_i \Rightarrow \sum_i 2^{-L_i} \leq 1$
  (2) Given $L_i$ with $\sum_i 2^{-L_i} \leq 1 \Rightarrow \text{exists PF code with length } L_i$

- Note: $\sum_i 2^{-L_i} \leq 1$ is known as "Kraft's inequality"

- Intuitively: not all $L_i$ can be small ... small $L_i \rightarrow$ large $2^{-L_i}$

  For example, the lemma says that a prefix-free code where
  three $L_i = 1$ is not possible, because $2^{-1} + 2^{-1} + 2^{-1} > 1$
Proof of central lemma, part (1)

- Show: given PF code with lengths $L_i$ then $\Sigma_i 2^{-L_i} \leq 1$
- Consider the following random experiment:
  
  Generate a random binary sequence, and pick each bit independent from all other bits
  
  Stop when you have a valid code, or when no more code is possible ... well-defined for PF codes only!

- Let $C_i = \text{the event that code } i \text{ is generated} \rightarrow \Pr(C_i) = 2^{-L_i}$
- Then $\Pr(C_1) + \ldots + \Pr(C_m) = \Pr(C_1 \cup \ldots \cup C_m) \leq 1$
- And the left-hand side is just $\Sigma_i 2^{-L_i}$
Proof of central lemma, part (2)

- To show: \( L_i \) with \( \sum_i 2^{-L_i} \leq 1 \) \( \Rightarrow \) exists PF code with lengths \( L_i \)

- Complete binary tree of depth \( M = \max L_i \) ... has \( 2^M \) leaves

- Mark all left edges 0, and all right edges 1

- Consider the code lengths \( L_i \) in sorted order, smallest first

- Then iterate: pick subtree with \( 2^M - L_i \) leaves that does not overlap with already picked subtrees ... path to that subtree gives code for symbol \( i \) and sum \( 2^M - L_i = 2^M \cdot \sum_i 2^{-L_i} \leq 2^M \)
Proof of source coding theorem, part (1)

- To show: for any PF encoding \( E L_X \geq H(X) \)

- By definition of expectation: \( E L_X = \sum_i p_i \cdot L_i \) \hspace{1cm} (1)

- By Kraft's inequality: \( \sum_i 2^{-L_i} \leq 1 \) \hspace{1cm} (2)

- Using Lagrange, it can be shown that, under the constraint (2), (1) is minimized for \( L_i = \log_2 1/p_i \)

- Then \( E L_X = \sum_i p_i \cdot L_i \geq \sum_i p_i \cdot \log_2 1/p_i = H(X) \)

This is Exercise 1 from ES4

Perfect exercise to practice Lagrangian optimization and deepen understanding of the source coding theorem
Proof of source coding theorem, part (2)

- Show: there is a PF encoding with $E L_X \leq H(X) + 1$

- Let $L_i = \lceil \log_2 1/p_i \rceil$, then $\sum_i 2^{-L_i} \leq 1$

  Note that rounding is necessary because the code length must be an integer, and that we need to round upwards, so that Kraft's inequality holds.

- By the central lemma, part (2), there then exists a PF code with code lengths $L_i$

- By definition of expectation: $E L_X = \sum_i p_i \cdot L_i$

- Hence $E L_X = \sum_i p_i \cdot \lceil \log_2 1/p_i \rceil \leq \sum_i p_i \cdot (\log_2 1/p_i + 1)$

  $= \sum_i p_i \cdot \log_2 1/p_i + \sum_i p_i = H(X) + 1$
Entropy-optimal codes

- Consider a PF code with $L_i = \text{code length for symbol } i$ and $p_i = \text{probability for symbol } i$
- We say that the code is optimal for distribution $p_i$ if

$$L_i \leq \log_2 \frac{1}{p_i} + 1$$

Then $\mathbf{E} L_X \leq H(X) + 1$ and by Shannon's theorem this is the best we can hope for.

For the optimality proof from Exercise 2 from ES4, it suffices that you show $L_i \leq \log_2 \frac{1}{p_i} + \mathbf{O}(1)$
Universal codes

- A prefix-free code is called *universal* if for every probability distribution over the symbols to be encoded

\[ \mathbb{E} L_X = O( H(X) ) \]

That is, the expected code length is within a constant factor of the optimum for any distribution

- Elias-Gamma, Elias-Delta, Golomb, and Variable-Byte are all universal in this sense

For a finer distinction, the definition of optimality from the previous slide is better

\[ \mathbb{E} L_X \leq H(X) + 1 \quad \text{versus} \quad \mathbb{E} L_X = O( H(X) ) \]
Entropy-optimality of Elias-Gamma

- Recall: code length for Elias-Gamma is $L_i = 2 \lfloor \log_2 i \rfloor + 1$
- For which probability distribution is this entropy-optimal?
- We need $L_i = 2 \lfloor \log_2 i \rfloor + 1 \leq \log_2 1/p_i + 1$
- This suggests something like $p_i \approx 1/\iota^2$ because:
  \[ p_i = 1/\iota^2 \rightarrow \log_2 1/p_i = \log_2 \iota^2 = 2 \cdot \log_2 \iota \]
- We have to take care that the $p_i$ sum to 1, hence let $p_i = 1/\iota^2$ for $\iota \geq 2$, and $p_1$ such that $\sum_i p_i = 1$
  That is, numbers $\iota \geq 2$ occur with probability $1/\iota^2$
  Note that $\sum_{\iota=1}^{\infty} 1/\iota^2 = \pi^2/6 = 1.6449...$
Optimality of Golomb

- Consider the following random experiment for the generation of an inverted list $L$ of length $m$:
  
  Include each document in $L$ with probability $p = m/n$, independently of each other, where $n = \#\text{documents}$

- Let $X$ be a fixed gap in this inverted list, then
  
  $\Pr(X = x) = (1 - p)^{x - 1} \cdot p =: p_x$ for $x = 1, 2, 3, ...$

  Exercise 2 from ES4: Golomb is optimal for this distrib.

- Bottom line: Golomb is optimal for gap-encoded lists
  
  But not practical, because of the bit fiddling, see slide 14
References

- **Textbook**
  
  Section 5: Index compression  
  Section 5.3: Postings file compression  
  some codes only

- **Wikipedia**

  http://en.wikipedia.org/wiki/Elias_gamma_coding  
  http://en.wikipedia.org/wiki/Elias_delta_coding  
  http://en.wikipedia.org/wiki/Golomb_coding  
  http://en.wikipedia.org/wiki/Variable-width_encoding  
  http://en.wikipedia.org/wiki/Kraft_inequality