Information Retrieval

INFO 4300 / CS 4300

- Indexing
  - Inverted indexes
  - ➡ Compression
  - Index construction
  - Ranking model

But first...

- Simple in-memory indexer

```
procedure BuildIndex(D)                    ▷ D is a set of text documents
    I ← HashTable()                        ▷ Inverted list storage
    n ← 0                                  ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                       ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

Compression

- Inverted lists are very large
  - e.g., 25-50% of collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have ***good compression ratios*** and are **easy to decompress**
- ***Lossless*** compression – no information lost

Compression

- **Basic idea**: Common data elements use short codes while uncommon data elements use longer codes
  - Example: coding numbers
    - » number sequence: $0, 1, 0, 3, 0, 2, 0$
    - » possible encoding: $00\ 01\ 00\ 10\ 00\ 11\ 00$
    - » encode 0 using a single 0: $0\ 01\ 0\ 10\ 0\ 11\ 0$
    - » only 10 bits, but...

Compression Example

- ***Ambiguous*** encoding – not clear how to decode
  - » another decoding:

$$0\ 01\ 01\ 0\ 0\ 11\ 0$$

  - » which represents:

$$0, 1, 1, 0, 0, 3, 0$$

  - » use unambiguous code:

Number	Code
0	0
1	101
2	110
3	111

  - » which gives:

$$0\ 101\ 0\ 111\ 0\ 110\ 0$$

Delta Encoding

- **Word count** data is good candidate for compression
  - – many small numbers and few larger numbers
  - – encode small numbers with small codes
- Frequency of **document numbers** in inverted lists is less predictable
  - – but differences between numbers in an ordered list (e.g. an inverted list) are smaller and more predictable
- ***Delta encoding***:
  - – encodes differences between document numbers (*d-gaps*)

Delta Encoding

- Inverted list (doc #s without counts)

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

- Differences between adjacent numbers

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

- Differences for a high-frequency word are easier to compress (many small d-gaps), e.g.,

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$$

- Differences for a low-frequency word are large, e.g.,

$$109, 3766, 453, 1867, 992, \ldots$$

Bit-Aligned Codes

- Breaks (i.e. spaces) between encoded numbers can occur after any bit position
- ***Unary*** code (base-1 encoding)
  - – Encode k by k 1s followed by 0
  - – 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

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 is ambiguous

Elias-γ Code

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$ length of offset
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$ offset
    - » k_d is # of binary digits needed to encode offset
      - ◆ Represent in unary code
    - » k_r
      - ◆ Represent in binary

k = 13
k_d = 3 k_d (unary) 1110
k_r = 5 k_r (binary) 101

1110 101

Elias-γ Code

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$ length of offset
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$ offset

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Elias-γ Code: alternate explanation

- To encode a number *k*,
  - Encode k in binary
  - Compute a ***length – offset*** pair
  - Offset: (for k>0) drop initial 1 from binary form of k
  - Length: # of bits needed to represent offset

k = 13 k (binary) 1101
k_r (binary) 101
k_d= 3 (unary) 1110

1110 101

▶ **Table 5.5** Some examples of unary and γ codes. Unary codes are only shown for the smaller numbers. Commas in γ codes are for readability only and are not part of the actual codes.

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111

Decoding

- Read the unary code up to the 0
  - Tells us how long the offset is
- Read the offset
- Append a 1 to the front
- Convert to base-10

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 $2 \lfloor \log_2 k \rfloor + 1$ 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-δ Code

- Split k_d into:
  - $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
  - $k_{dr} = (k_d + 1) - 2^{\text{floor} (\log\_2 (k\_d + 1))}$
  - encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111