Information Retrieval
Tutorial 3: Index Compression

Professor: Michel Schellekens
TA: Ang Gao

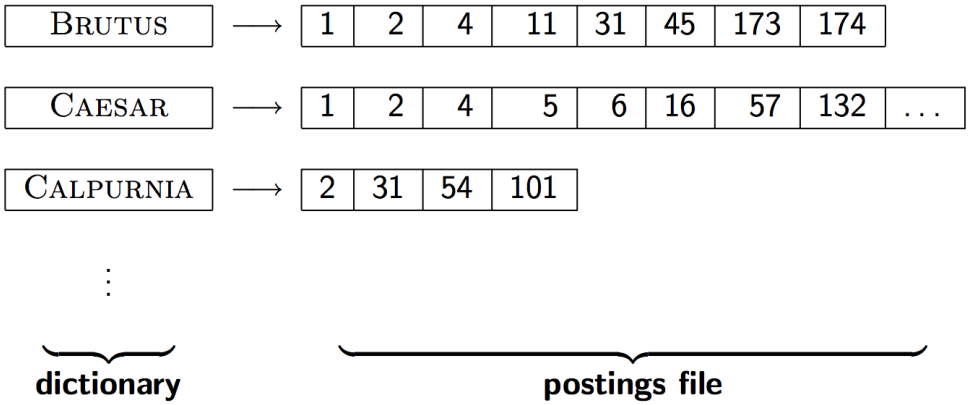
University College Cork

2012-11-09

Outline

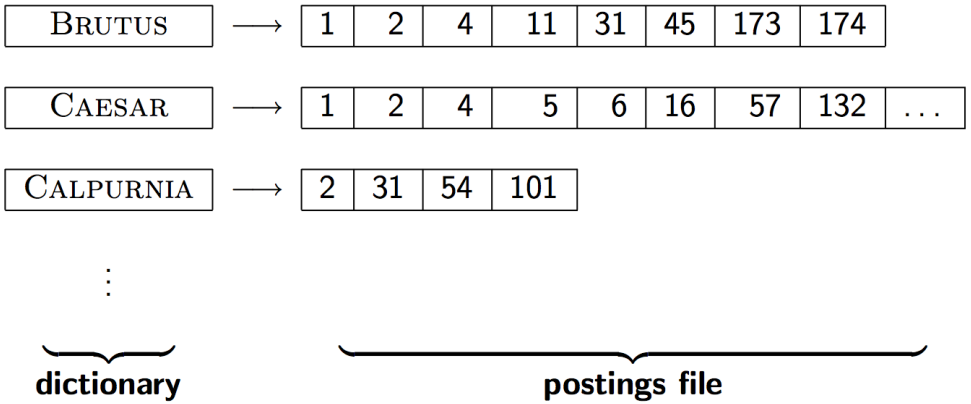
Review

For each term t, we store a list of all documents that contain t.



Review

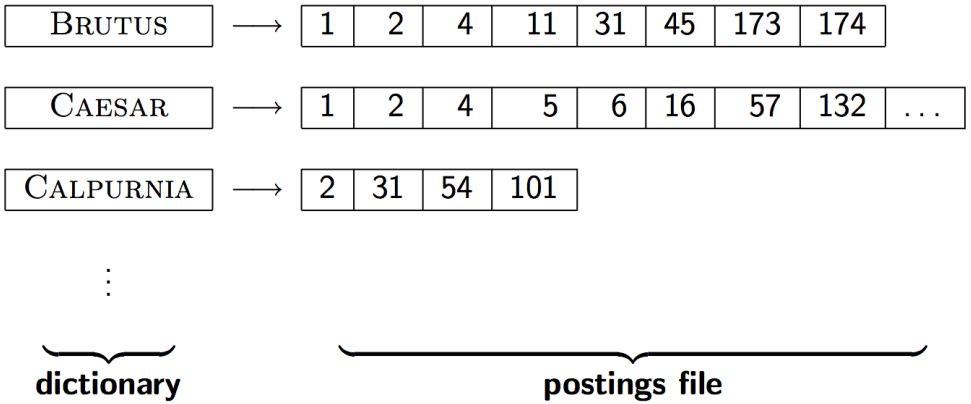
For each term t, we store a list of all documents that contain t.



- Motivation for compression in information retrieval systems

Review

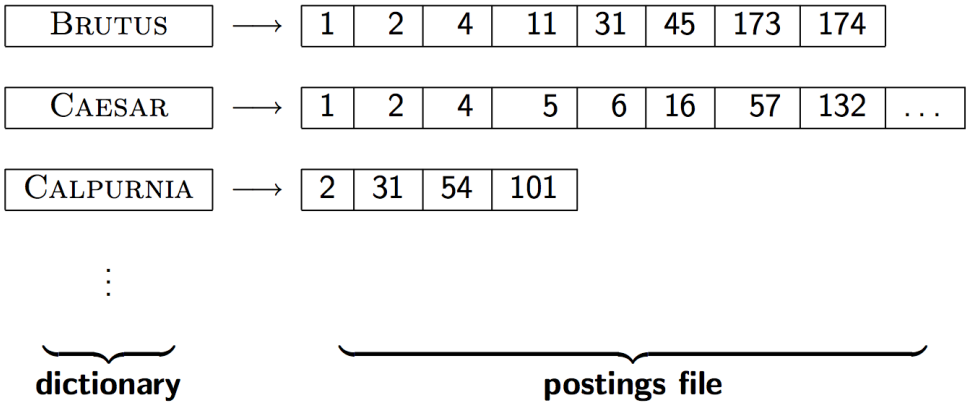
For each term t, we store a list of all documents that contain t.



- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?

Review

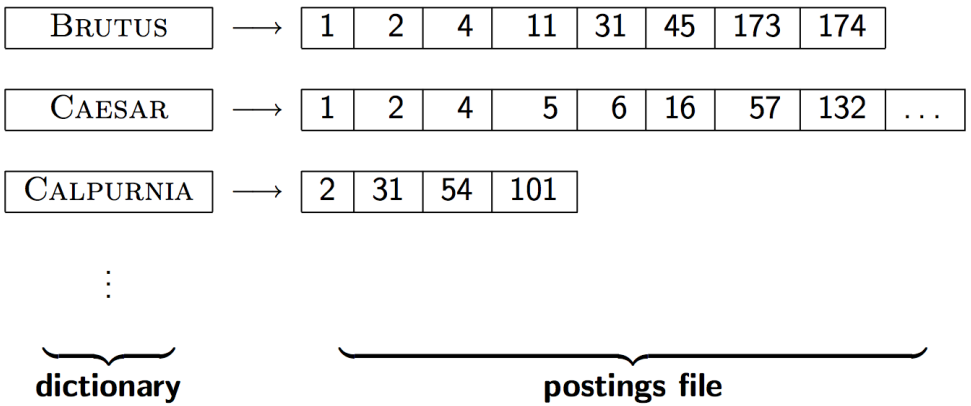
For each term t, we store a list of all documents that contain t.



- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?

Review

For each term t, we store a list of all documents that contain t.



- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics: how are terms distributed in document collections?

Why compression? (in general)

Why compression? (in general)

- Use less disk space (saves money)

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory]
    is faster than
    [read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.

Why compression in information retrieval?

Why compression in information retrieval?

- First, we will consider space for dictionary

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We will devise various compression schemes for dictionary and postings.

Lossy vs. lossless compression

Lossy vs. lossless compression

- Lossy compression: Discard some information

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
- Lossless compression: All information is preserved.

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
- Lossless compression: All information is preserved.
  - What we mostly do in index compression

Model collection: The Reuters collection

symbol	statistic	value
N	documents	800,000
L	avg. # word tokens per document	200
M	word types	400,000
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per word type	7.5
T	non-positional postings	100,000,000

How big is the term vocabulary?

How big is the term vocabulary?

- That is, how many distinct words are there?

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. Thus $M \approx k\sqrt{T}$

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. Thus $M \approx k\sqrt{T}$
- Notice $logM = logk + blogT (y = c + bx)$

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. Thus $M \approx k\sqrt{T}$
- Notice $logM = logk + blogT (y = c + bx)$
- Heaps' law is linear in log-log space.

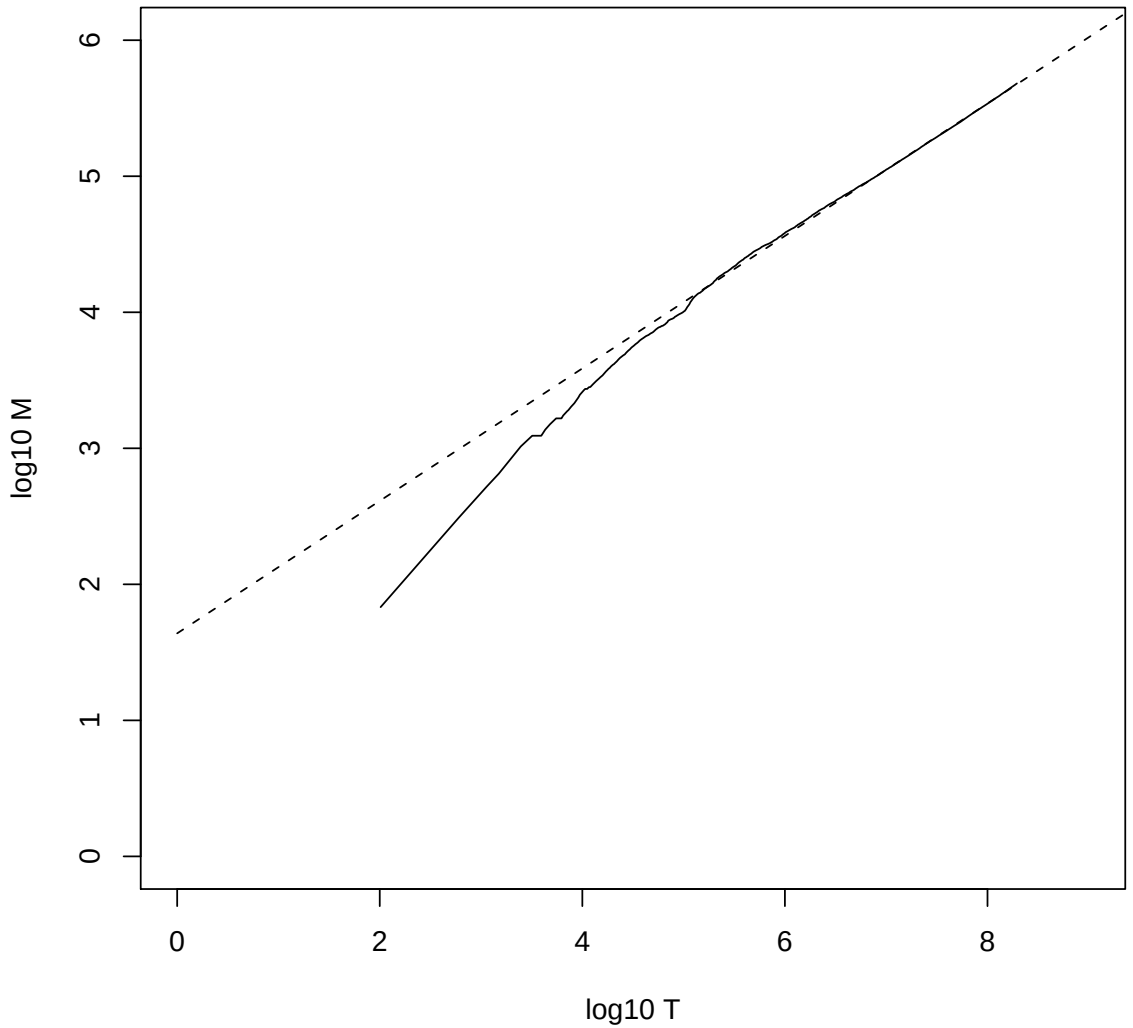
How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. Thus $M \approx k\sqrt{T}$
- Notice $logM = logk + blogT (y = c + bx)$
- Heaps' law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.

How big is the term vocabulary?

- That is, how many distinct words are there?
- In practice,the vocabulary will keep growing with collection size.(eg: names of new people)
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. Thus $M \approx k\sqrt{T}$
- Notice $logM = logk + blogT (y = c + bx)$
- Heaps' law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
  - An empirical finding(Empirical law).

Heaps' law for Reuters



Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit.
Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Empirical fit for Reuters

Empirical fit for Reuters

- Good, as we just saw in the graph.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

- The actual number is 38,365 terms, very close to the prediction.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.

Exercise

1. Compute vocabulary size M

Exercise

1. Compute vocabulary size M
   - 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.

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000$ $T_1 = 10,000$

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000 \;\; T_1 = 10,000$
- $log(3000) = logk + blog(10,000)$

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000$ $T_1 = 10,000$
- $log(3000) = logk + blog(10,000)$
- $log(M_2) = logk + blog(T_2)$ and $M_2 = 30,000$, $T_2 = 1,000,000$

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000$ $T_1 = 10,000$
- $log(3000) = logk + blog(10,000)$
- $log(M_2) = logk + blog(T_2)$ and $M_2 = 30,000$, $T_2 = 1,000,000$
- $log(30,000) = logk + blog(1,000,000)$

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000$ $T_1 = 10,000$
- $log(3000) = logk + blog(10,000)$
- $log(M_2) = logk + blog(T_2)$ and $M_2 = 30,000$, $T_2 = 1,000,000$
- $log(30,000) = logk + blog(1,000,000)$
- thus $logk = log(3000) - 2 \approx 1.477, k \approx 29.99$ and $b = 0.5$

Exercise

1. Compute vocabulary size M
   - 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 (2×10^{10}) pages, containing 200 tokens on average
   - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

- $log(M_1) = logk + blog(T_1)$ and $M_1 = 3000$ $T_1 = 10,000$
- $log(3000) = logk + blog(10,000)$
- $log(M_2) = logk + blog(T_2)$ and $M_2 = 30,000$, $T_2 = 1,000,000$
- $log(30,000) = logk + blog(1,000,000)$
- thus $logk = log(3000) - 2 \approx 1.477, k \approx 29.99$ and $b = 0.5$
- $log(M) = logk + \frac{1}{2} log(20,000,000,000 * 200) = 7.778$ thus $M = 10^{7.778} \approx 6 * 10^7$

Basic knowledge to remember

To binary represent an integer n, number of bits need $= \lfloor log_2(n) \rfloor + 1$

$n = \{2\}_{10} = \{10\}_2$

$n = \{3\}_{10} = \{11\}_2$

$n = \{4\}_{10} = \{100\}_2$

Outline

Dictionary compression

Dictionary compression

- The dictionary is small compared to the postings file.

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

Recall: Dictionary as array of fixed-width entries

Recall: Dictionary as array of fixed-width entries

	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

Space for Reuters: (20+4+4)*400,000 = 11.2 MB

Fixed-width entries are bad.

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS

Fixed-width entries are bad.

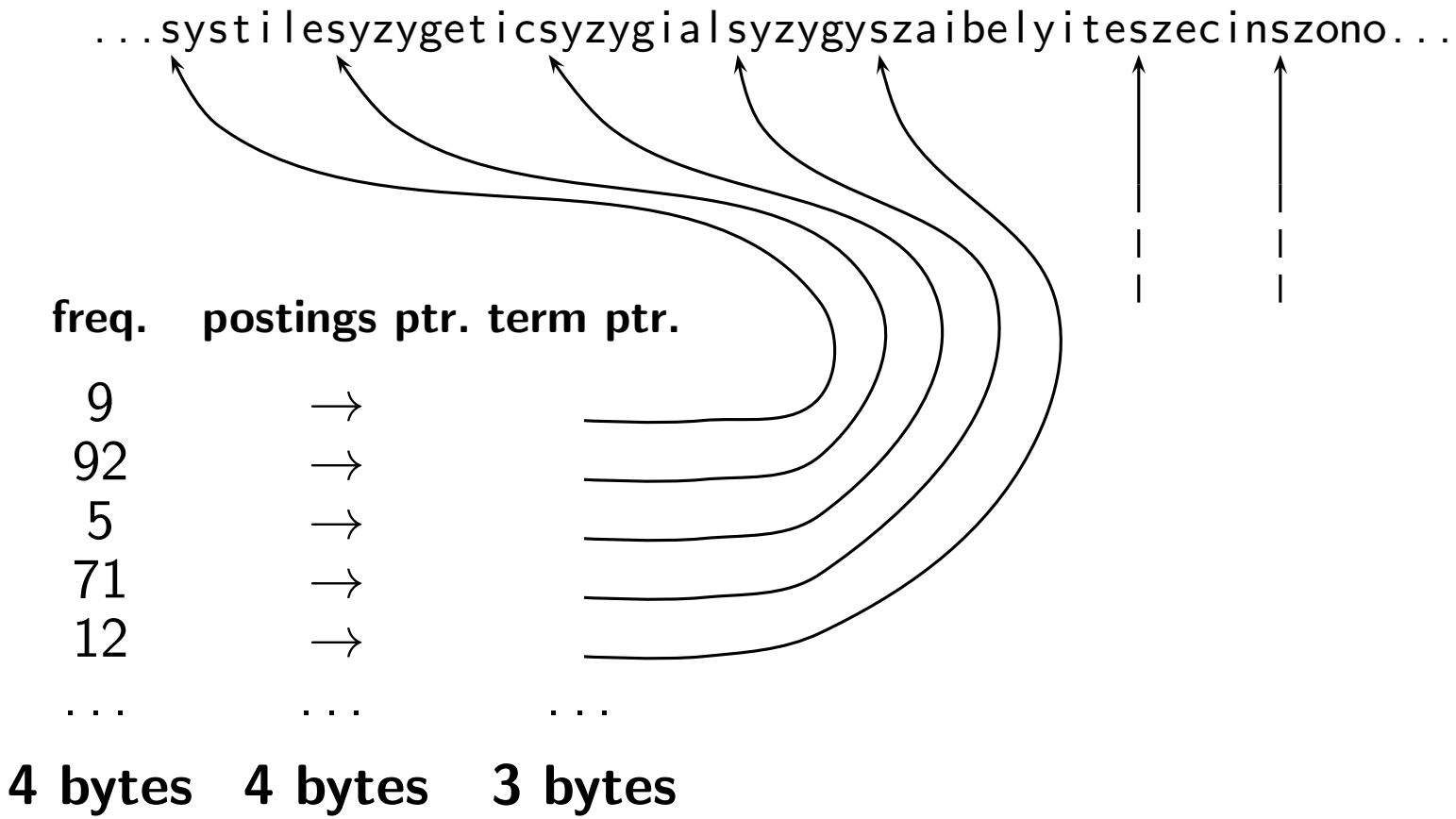
- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

Dictionary as a string

Dictionary as a string



Space for dictionary as a string

Space for dictionary as a string

- 4 bytes per term for frequency

Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list

Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string

Space for dictionary as a string

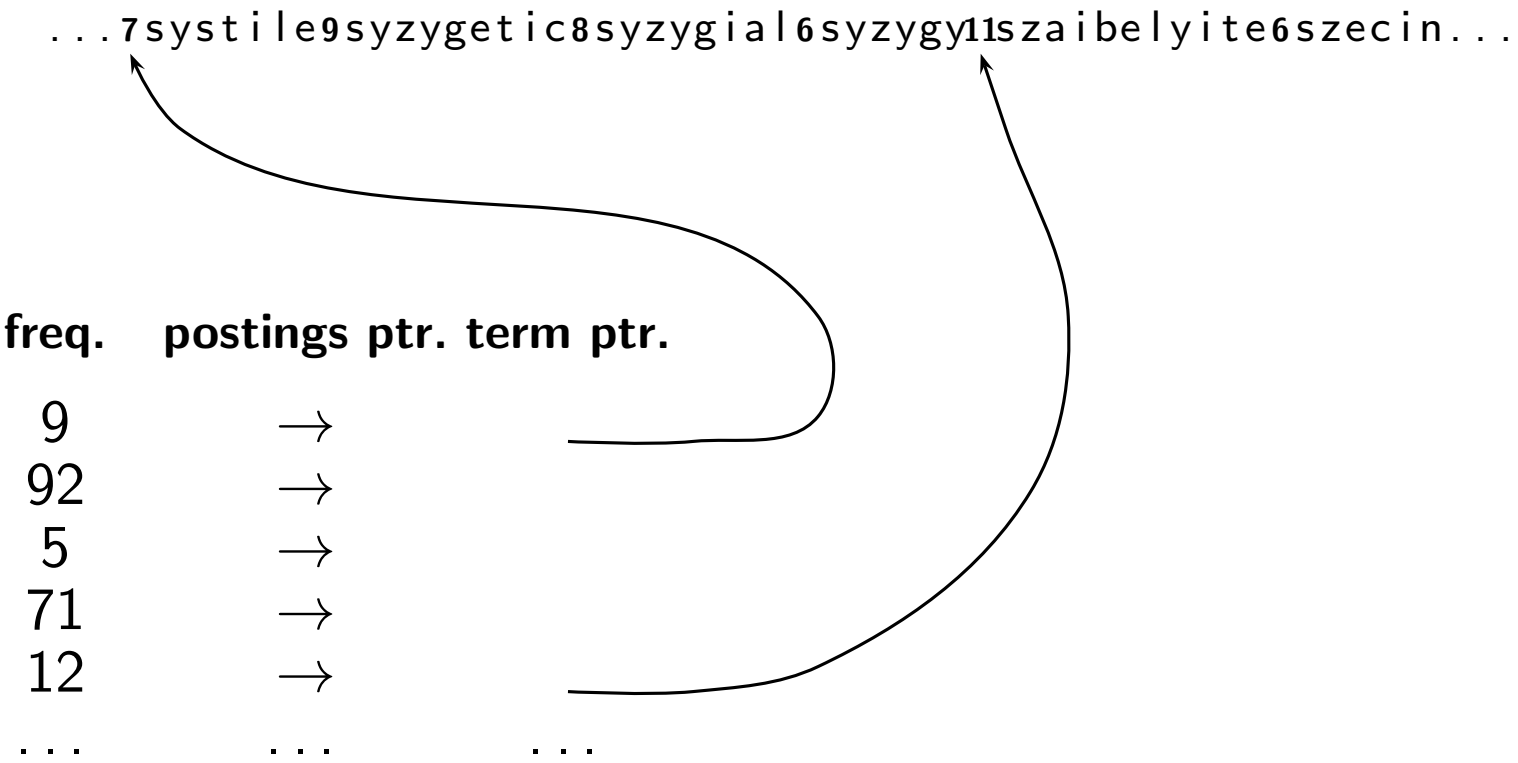
- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400{,}000$ positions)

Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400{,}000$ positions)
- Space: $400{,}000 \times (4 + 4 + 3 + 8) = 7.6$MB (compared to 11.2 MB for fixed-width array)

Dictionary as a string with blocking

Dictionary as a string with blocking



Space for dictionary as a string with blocking

Space for dictionary as a string with blocking

- Example block size $k = 4$

Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking

  ...

Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking . . .
- . . . we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.

Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking . . .
- . . . we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.

Space for dictionary as a string with blocking

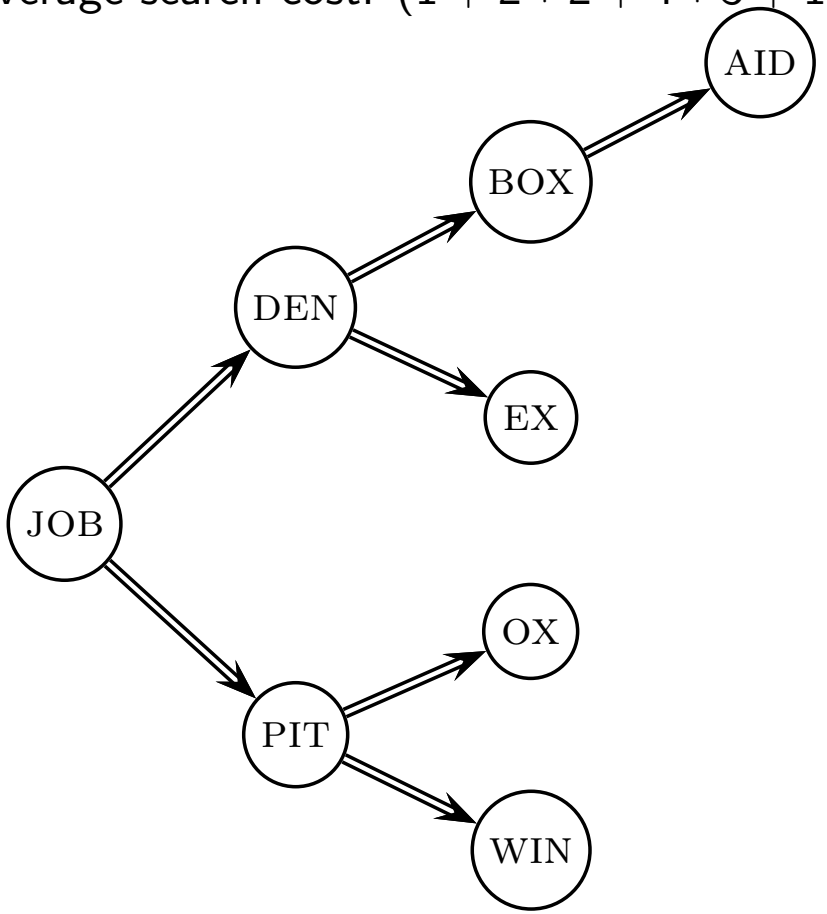
- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking . . .
- . . . we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400{,}000/4 * 5 = 0.5$ MB

Space for dictionary as a string with blocking

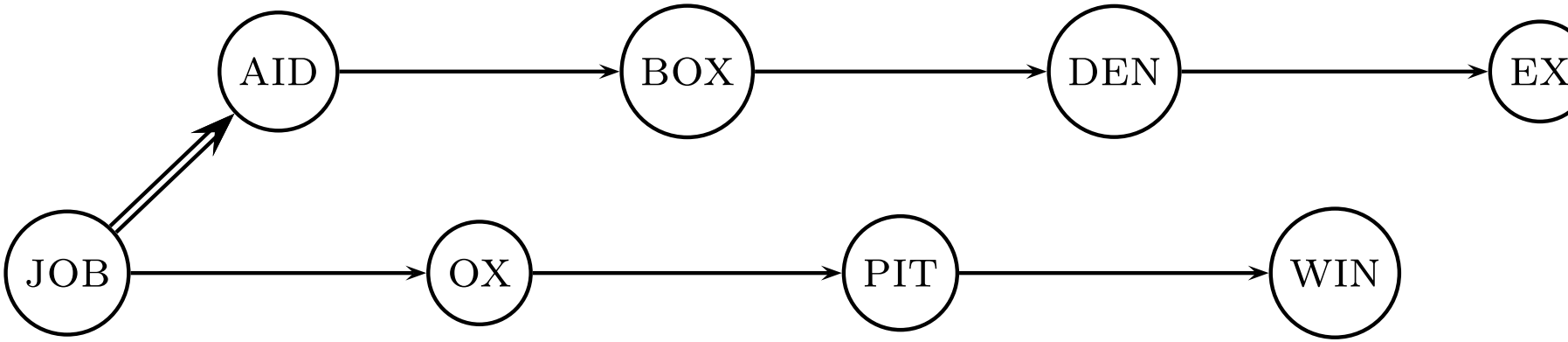
- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking . . .
- . . . we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400{,}000/4 * 5 = 0.5$ MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

Lookup of a term without blocking

Average search cost: $(1 + 2 * 2 + 4 * 3 + 1 * 4)/8 \approx 2.6$ steps



Lookup of a term with blocking: (slightly) slower



Average search cost: $(2 + 3 + 4 + 5 + 1 + 2 + 3 + 4)/8 \approx 3$ steps.

- Question: Can we increase K arbitrarily, is there any problem with it ?

- Question: Can we increase K arbitrarily, is there any problem with it ?
- Ans:We can't increase K arbitrarily, term look up time will go up. If we only have one pointer, we can't do binary search, have to go from beginning to the end to find the term.

Front coding

One block in blocked compression ($k = 4$) . . .

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

. . . further compressed with front coding.

8 a u t o m a t ∗ a **1** ◇ e **2** ◇ i c **3** ◇ i o n

Dictionary compression for Reuters: Summary

Dictionary compression for Reuters: Summary

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

Outline

Postings compression

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly

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.

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.

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 $\log_2 800{,}000 \approx 19.6 < 20$ bits per docID.

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 $\log_2 800{,}000 \approx 19.6 < 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Key idea: Store gaps instead of docIDs

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283154=43

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps : COMPUTER: 283154, 5, 43, ...

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps : COMPUTER: 283154, 5, 43, ...
- Gaps for frequent terms are small.

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps : COMPUTER: 283154, 5, 43, ...
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.

Gap encoding

	encoding	postings list									
THE	docIDs	...		283042		283043		283044		283045	...
	gaps				1		1		1		...
COMPUTER	docIDs	...		283047		283154		283159		283202	...
	gaps				107		5		43		...
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

Variable length encoding

- Aim:

Variable length encoding

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).

Variable length encoding

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).

Variable length encoding

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).
- In order to implement this, we need to devise some form of variable length encoding.

Variable length encoding

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).
- In order to implement this, we need to devise some form of variable length encoding.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

Variable byte (VB) code

Variable byte (VB) code

- Used by many commercial/research systems

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit c.

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit c.
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit c.
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more additional bytes to encode the higher order bits using the same algorithm.

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit c.
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more 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 of the other bytes to 0 ($c = 0$).

VB code examples

docIDs	824		829	215406		
gaps			5	214577		
VB code	00000110	10111000	10000101	00001101	00001100	10110001

Gamma codes for gap encoding

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent n as n 1s with a final 0.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent n as n 1s with a final 0.
  - Unary code for 3 is 1110

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent n as n 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111110

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent n as n 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111110
  - Unary code for 70 is:

11111111111111111111111111111111111111111111111111111111111110

Gamma code

Gamma code

- Represent a gap G as a pair of length and offset.

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 =$ offset

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 =$ offset
- Length is the length of offset.

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 =$ offset
- Length is the length of offset.
- For 13 (offset 101), this is 3.

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 =$ offset
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 =$ offset
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

Exercise

- Compute the variable byte code of 6 and 128

Ans:

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$

Ans:

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$
- Compute the gamma code of 6.

Ans:

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: 00000001, 10000111, 10000010
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

- $6_2 = 110$ VB: 10000110

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

- $6_2 = 110$ VB: 10000110
- $128_2 = 10000000$ VB: $00000001, 10000000$

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

- $6_2 = 110$ VB: 10000110
- $128_2 = 10000000$ VB: $00000001, 10000000$
- $135_2 = 10000111$ and $2_2 = 10$ thus doc_{135} and doc_2

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: $00000001, 10000111, 10000010$
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

- $6_2 = 110$ VB: 10000110
- $128_2 = 10000000$ VB: $00000001, 10000000$
- $135_2 = 10000111$ and $2_2 = 10$ thus doc_{135} and doc_2
- $6_2 = 110$ gamma code: 11010

Exercise

- Compute the variable byte code of 6 and 128
- Decode VB code of documents IDs: 00000001, 10000111, 10000010
- Compute the gamma code of 6.
- Decode gamma code: 110001110001

Ans:

- $6_2 = 110$ VB: 10000110
- $128_2 = 10000000$ VB: 00000001, 10000000
- $135_2 = 10000111$ and $2_2 = 10$ thus doc_{135} and doc_2
- $6_2 = 110$ gamma code: 11010
- $4_2 = 100$ gamma code: 11000 and $9_2 = 1001$ gamma code: 1110001

Length of gamma code

Length of gamma code

- The length of *offset* is $\lfloor \log_2 G \rfloor$ bits.
- The length of *length* is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- γ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$.