5 *Index compression*

Chapter 1 introduced the dictionary and the inverted index as the central data structures in information retrieval. In this chapter, we employ a number of compression techniques for the two data structures which are essential for efficient information retrieval systems.

One benefit of compression is immediately clear. We will need less disk space. As we will see, compression ratios of 1:4 are easy to achieve, potentially cutting the cost of storing the index by 75%.

A more subtle benefit is the increased use of caching. Search systems use some parts of the dictionary and the index much more than others. For example, if we cache the posting list of a frequently used query term w, then responding to that query reduces to a simple memory lookup. With compression, we can fit a lot more information into main memory. Instead of having to expend a disk seek when processing a query with w, we instead access its posting list in memory and decompress it. As we will see below, there are compression schemes with simple and efficient decompression, so that the penalty of having to decompress the posting list is small. As a result, we are able to decrease the response time of the IR system substantially. Since memory is a more expensive resource than disk space, increased speed due to caching – rather than decreased space requirements – is often the prime motivator for compression.

This chapter first gives a statistical characterization of the distribution of the entities we want to compress – terms and postings in large collections (Section 5.1). We then look at compression of the dictionary, using the dictionary-as-a-string method and blocked storage (Section 5.2). Section 5.3 describes two techniques for compressing the postings file, variable byte encoding and γ encoding.

	term types			non-positional postings			tokens		
	size	Δ	cumul.	size	Δ	cumul.	size	Δ	cumul.
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056	-3%	-12%	179,158,204	-0%	-9%
30 stop words	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop words	391,373	-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383	-17%	-33%	63,812,300	-4%	-42%	94,516,599	-0%	-52%

► **Table 5.1** The effect of preprocessing on the number of term types, non-positional postings, and positional postings for RCV1. "Δ" indicates the reduction in size from the previous line, except that "30 stop words" and "150 stop words" both use "case folding" as their reference line. "cumul." is cumulative reduction (from unfiltered). We performed stemming with the Porter stemmer (Chapter 2, page 29).

5.1 Statistical properties of terms in information retrieval

As in the last chapter, we will use Reuters-RCV1 as our model collection (see Table 4.1, page 55). We give some term and postings statistics for the collection in Table 5.1. The number of term types is the main factor in determining the size of the dictionary. The number of non-positional postings is an indicator of the expected size of the non-positional index of the collection. In this chapter, we define a *posting* as a docID in a posting list. For example, the posting list (6; 20, 45, 100) (where 6 is the termID of the list's term) contains 3 postings. Postings in most search systems also contain frequency and position information although we will only cover simple docID postings in this chapter to simplify the discussion (but see references in Section 5.4). The expected size of a positional index is related to the number of positions it must encode. The last three columns of Table 5.1 show how many positions there are in RCV1 for different levels of preprocessing.

POSTING

In general, the statistics in Table 5.1 show that preprocessing affects the size of the dictionary and the number of non-positional postings greatly. Stemming and case folding reduce the number of term types (or distinct terms) by 17% each,[1] and the number of non-positional postings by 4% and 3%, respectively. The treatment of the most frequent words is also important. The *rule of 30* states that the 30 most common words account for 30% of the tokens in written text (31% in the table). Eliminating the 150 commonest terms from indexing (as stop words; cf. Section 2.2.2, page 23) will cut 25–30% of the non-positional postings. But, while a stop list of 150 words reduces the number of postings by a quarter, this size reduction does not carry over to the size

RULE OF 30

1. For the definitions of token and type, see Chapter 2, page 20.

of the compressed index. As we will see later in this chapter, the posting lists of frequent words require only a few bits per posting after compression.

The deltas in the table are in a range typical of large collections. Note, however, that the percentage reductions can be very different for some text collections. For example, for a collection of web pages with a high proportion of French text, stemming or lemmatization would reduce vocabulary size much more than the Porter stemmer does for an English-only collection since French is a morphologically richer language than English.

LOSSLESS COMPRESSION
LOSSY COMPRESSION

The compression techniques we describe in the remainder of this chapter are *lossless*, that is, all information is preserved. Better compression ratios can be achieved with *lossy compression*, which discards some information. Case folding, stemming and stop word elimination are forms of lossy compression. Similarly, the vector space model (Chapter 7) and dimensionality reduction techniques like Latent Semantic Indexing (Chapter 18) create compact representations from which we cannot fully restore the original collection. Lossy compression makes sense when the "lost" information is unlikely ever to be used by the search system. For example, web search is characterized by a large number of documents, short queries, and users who only look at the first few pages of results. As a consequence, we can discard postings of documents that would only be used for hits far down the list. Thus, there are retrieval scenarios where lossy methods can be used for compression without any reduction in effectiveness.

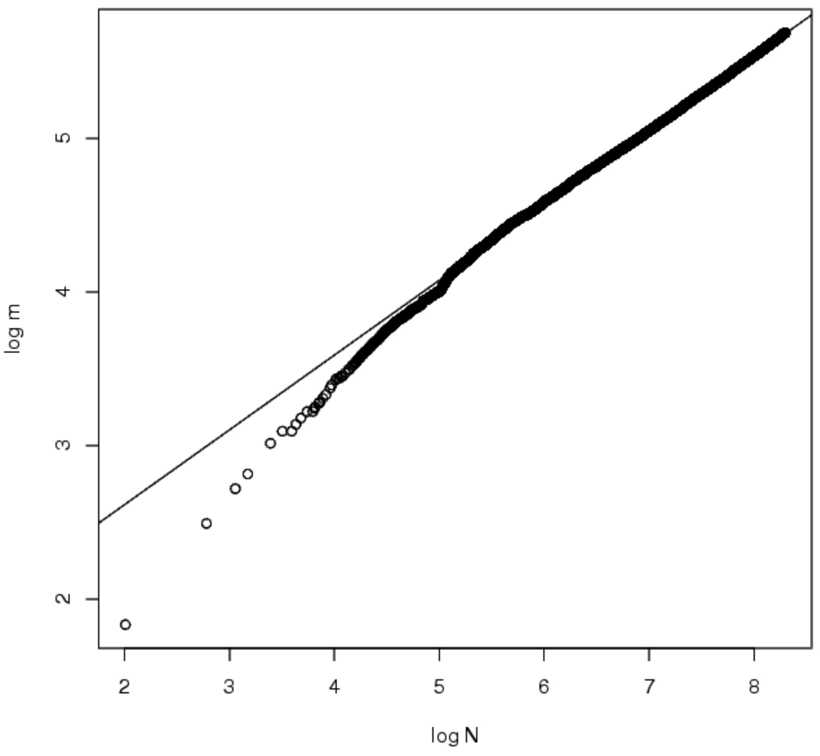
Before introducing techniques for compressing the dictionary, we want to estimate the number of term types M in a collection. It is sometimes said that languages have a vocabulary of a certain size. The second edition of the Oxford English Dictionary (OED) defines more than 600,000 words. But the size of the OED cannot be equated with the size of the vocabulary in IR. The OED does not include most names of people, locations, products and scientific entities like genes. These names still need to be included in the inverted index, so our users can search for them.

HEAPS' LAW

A better way of getting a handle on M is *Heaps' law*, which estimates vocabulary size as a function of collection size:

$$M = kT^b \tag{5.1}$$

where T is the number of tokens in the collection. Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. The motivation for Heaps' law is that the simplest possible relationship between collection size and vocabulary size is linear in log-log space and the assumption of linearity is usually born out in practice as shown in Figure 5.1 for Reuters-RCV1. In this case, the fit is excellent for $T > 10^5 = 100{,}000$, for the parameter values $b = 0.49$ and $k = 44$. For example, for the first 1,000,020 tokens Heaps' law



► **Figure 5.1** Heaps' law. Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit. Thus, $k = 10^{1.64} \approx 44$ and $b = 0.49$. AXIS TITLES: CHANGE LOG N TO LOG 10 T AND LOG m TO LOG 10 M.

predicts 38,323 term types:

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

The actual number is 38,365 term types, very close to the prediction.

The parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed. Case-folding and stemming reduce the growth rate of the vocabulary, whereas including numbers and spelling errors will significantly increase it. Regardless of the values of the parameters for a particular collection, Heaps' law suggests that: (i) the dictionary size will continue to increase with more documents in the collection, rather than a maximum vocabulary size being reached, and (ii) the size of the dictionary will be quite large for large collections. These

	term	freq.	pointer to posting list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	40 bytes	4 bytes	4 bytes

► **Figure 5.2** Storing the dictionary as an array of fixed-width entries.

two hypotheses have been empirically shown to be true of large text collections (Section 5.4). So dictionary compression is important for an effective information retrieval system.

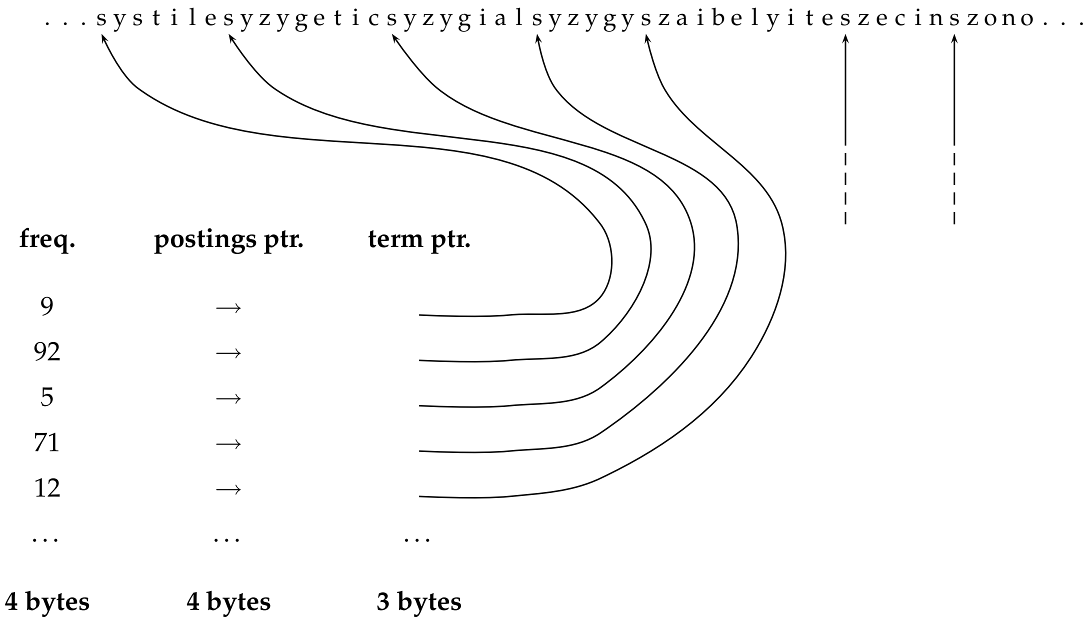
5.2 Dictionary compression

This section presents a series of dictionary representations that achieve increasingly higher compression ratios. Let us first motivate the need for dictionary compression. The dictionary is small compared to the postings file as suggested by Table 5.1. So why compress it if it is responsible for only a small percentage of the overall space requirements of the IR system?

One of the main determinants of response time of an IR system is the number of disk seeks necessary to process a query. If parts of the dictionary are on a disk, then many more disk seeks are necessary in query evaluation. Thus, the main goal of compressing the dictionary is to fit it in main memory, or at least a large portion of it, in order to support high query throughput. While dictionaries of very large collections will fit into the memory of a standard desktop machine, this is not true of many other application scenarios. For example, an enterprise search server for a large corporation may have to index a multi-terabyte collection with a comparatively large vocabulary because of the presence of documents in many different languages. We also want to be able to design search systems for limited hardware such as mobile phones and onboard computers. Other reasons for wanting to conserve memory are fast startup time and having to share resources with other applications. The search system on your PC must get along with the memory-hogging word processing suite you are using at the same time.

5.2.1 Dictionary-as-a-string

The simplest data structure for the dictionary is to store the lexicographically ordered list of all terms in an array of fixed-width entries as shown in Figure 5.2. Assuming a unicode representation, we allocate 2×20 bytes for the

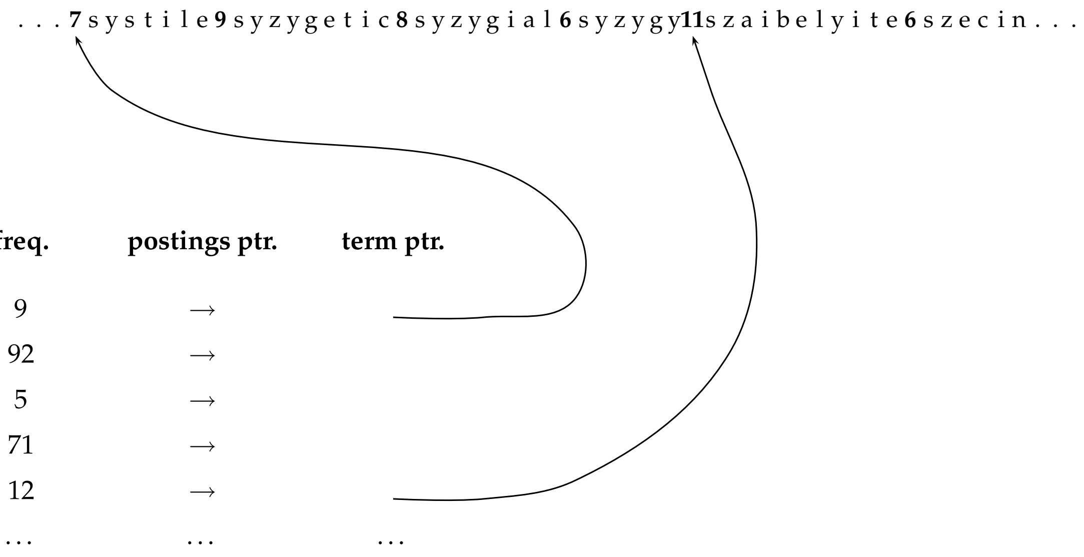


. . . s y s t i l e s y z y g e t i c s y z y g i a l s y z y g y s z a i b e l y i t e s z e c i n s z o n o . . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
. . .
4 bytes
4 bytes
3 bytes

► **Figure 5.3** Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile (frequency 9), syzygetic (frequency 92) and syzygial (frequency 5).

term itself since few terms have more than 20 characters in English, 4 bytes for its frequency and 4 bytes for the pointer to its posting list. (4-byte pointers resolve a 4 GB address space. For large collections like the web, we need to allocate more bytes per pointer.) We look up terms in the array by binary search. For Reuters-RCV1, we need $M \times (2 \times 20 + 4 + 4) = 400{,}000 \times 48 =$ 19.2 MB for storing the dictionary in this scheme.

Using fixed-width entries for terms is clearly wasteful. The average length of a term in English is about 8 characters, so on average we are wasting 12 characters (or 24 bytes) in the fixed-width scheme. Also, we have no way of storing terms with more than 20 characters like hydrochlorofluorocarbons and supercalifragilisticexpialidocious. We can overcome these shortcomings by storing the dictionary terms as one long string of characters, as shown in Figure 5.3. The pointer to the next term is also used to demarcate the end of the current term. As before, we locate terms in the data structure by way of binary search in the (now smaller) table. This scheme saves us 60% compared to fixed-width storage – 24 bytes on average of the 40 bytes we allocated for terms before. However, we now also need to store term pointers. The



. . . **7** s y s t i l e **9** s y z y g e t i c **8** s y z y g i a l **6** s y z y g y **11** s z a i b e l y i t e **6** s z e c i n . . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
. . .

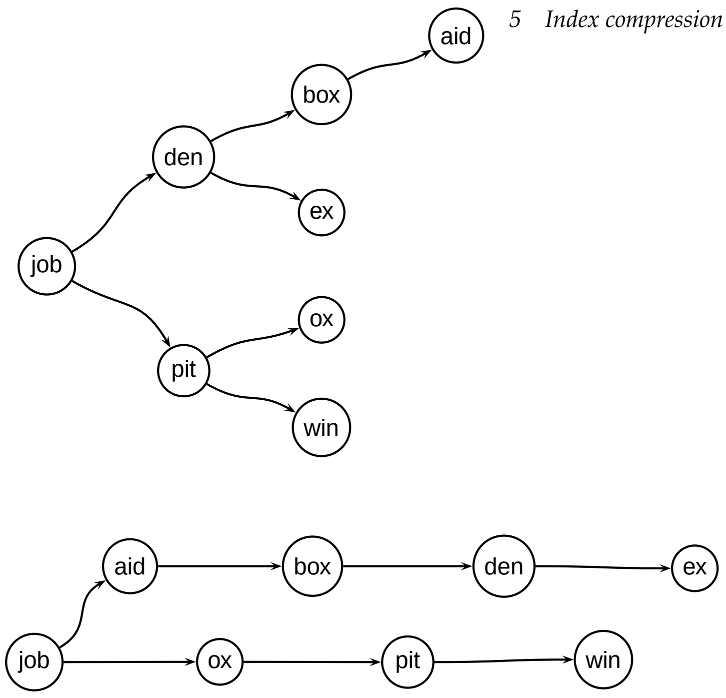
► **Figure 5.4** Blocked storage with four terms per block. The first block consists of systile, syzygetic, syzygial, and syzygy with lengths 7, 9, 8 and 6 characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

term pointers resolve $400{,}000 \times 8 = 3.2 \times 10^6$ positions, so they need to be $\log_2 3{,}200{,}000 \approx 22$ bits or 3 bytes long.

In this new scheme, we need $400{,}000 \times (4 + 4 + 3 + 2 \times 8) = 10.8$ MB for the Reuters-RCV1 dictionary: 4 bytes each for frequency and postings pointer, 3 bytes for the term pointer, and 2×8 bytes on average for the term. So we have reduced the space requirements by almost half from 19.2 MB to 10.8 MB.

5.2.2 Blocked storage

The dictionary can be further compressed by grouping terms in the string into blocks of size k and keeping a term pointer only for the first term of each block (see Figure 5.4). The length of the term is stored in the string as an additional byte at the beginning of the term. We thus eliminate $k - 1$ term pointers, but need an additional k bytes for storing the length of each term. For $k = 4$, we save $(k - 1) \times 3 = 9$ bytes for term pointers, but need an additional $k = 4$ bytes for term lengths. So the total space requirements for the dictionary of Reuters-RCV1 are reduced by 5 bytes per 4-term block, or a total of $400{,}000 \times 1/4 \times 5 = 0.5$ MB to 10.3 MB.



► **Figure 5.5** Search of the uncompressed dictionary (top) and a dictionary compressed by blocking with $k = 4$ (bottom).

By increasing the block size k, we get better compression. However, there is a tradeoff between compression and query processing speed. Searching the uncompressed eight-term dictionary in Figure 5.5 (top) takes on average $(4+3+2+3+1+3+2+3)/8 \approx 2.6$ steps, assuming each term is equally likely to come up in a query. For example, finding the first two terms, aid and box, takes 4 and 3 steps, respectively. With blocks of size $k = 4$, we need $(2+3+4+5+1+2+3+4)/8 = 3$ steps on average, 20% more (Figure 5.5, bottom). By increasing k, we can get the size of the compressed dictionary arbitrarily close to the minimum of $400{,}000 \times (4+4+1+2\times 8) = 10$ MB, but query processing becomes prohibitively slow for large values of k.

One source of redundancy in the dictionary we have not exploited yet is the fact that consecutive entries in an alphabetically sorted list share common prefixes. This observation leads to *front coding* as shown in Figure 5.6. A common prefix is identified for a subsequence of the term list and then referred to with a special character. In the case of Reuters, front coding saves another 2.4 MB.

FRONT CODING

Other schemes with even greater compression rely on minimal perfect hashing, i.e., a hash function that maps M terms onto $[1, \ldots, M]$ without col-

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

$\Downarrow$

. . . further compressed with front coding.
8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ i o n

► **Figure 5.6** Front coding. A sequence of terms with identical prefix ("automat" in this example) is encoded by marking the end of the prefix with $*$ and replacing it with $\diamond$ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

representation	size in MB
dictionary, fixed-width	19.2
dictionary, term pointers into string	10.8
$\sim$, with blocking, $k = 4$	10.3
$\sim$, with blocking & front coding	7.9

► **Table 5.2** Dictionary compression for Reuters-RCV1.

lisions. However, we cannot adapt perfect hashes incrementally – each new term causes a collision and therefore requires the creation of a new perfect hash function. Therefore, they are not very usable in a dynamic environment.

Even with the best compression scheme, it may not be feasible to store the entire dictionary in main memory for very large text collections and for hardware with limited memory. If we have to partition the dictionary onto pages that are stored on disk, then we can index the first term of each page using a B-tree. For processing most queries, the search system has to go to disk anyway to fetch the postings. One additional seek for retrieving the term's dictionary page from disk is a significant, but tolerable increase in the time it takes to process a query.

Table 5.2 summarizes the compression achieved by the four dictionary representation schemes.

5.3 Postings file compression

Recall from Table 4.1 (page 55) that Reuters-RCV1 has 800,000 documents, 200 tokens per document, 6 characters per token on average and 100,000,000 postings where we define a posting in this chapter as a docID in a posting list (i.e., excluding frequency and position information). These numbers

	encoding	posting 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								

► **Table 5.3** Encoding gaps instead of document ids. For example, we store gaps 14, 107, 5, ..., instead of docIDs 283047, 283154, 283159 ... for computer. The first docID is left unchanged (only shown for arachnocentric).

correspond to line 3 ("case folding") in Table 5.1. So the size of the collection is about 800,000 $\times$ 200 $\times$ 6 bytes $=$ 960 MB. Document identifiers are $\log_2 800{,}000 \approx 20$ bits long. Thus, the size of the uncompressed postings file is 100,000,000 $\times$ 20/8 $=$ 250 MB. To reduce its size, we need to devise methods that use fewer than 20 bits per document.

To devise a more efficient representation, we observe that the postings for frequent terms are close together. Imagine going through the documents of a collection one by one and looking for a frequent term like computer. We will find a document containing computer, then we skip a few documents that do not contain it, then there is again a document with the term and so on (see Table 5.3). The key idea is that the *gaps* between postings are short, requiring a lot less space than 20 bits to store. In fact, gaps for the most frequent terms such as the and for are mostly equal to 1. But the gaps for a rare term that occurs only once or twice in a collection (e.g., arachnocentric in Table 5.3) have the same order of magnitude as the docIDs and will need 20 bits. For an economical representation of this distribution of gaps, we need a *variable encoding* method that uses fewer bits for short gaps.

To encode small numbers in less space than large numbers, we look at two types of methods: bytewise compression and bitwise compression. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively.

5.3.1 Variable byte codes

VARIABLE BYTE ENCODING

CONTINUATION BIT

Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a *continuation bit*. It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts. Figure 5.7 gives pseudocode

```
VBENCODE($n$)
1  $bytes \leftarrow empty\_list$
2  while $true$
3  do PREPEND($bytes, n \bmod 128$)
4      if $n < 128$
5        then BREAK
6      $n = n \text{ div } 128$
7  $bytes$[LENGTH($bytes$)]$+= 128$
8  return $bytes$
```

```
VBDECODE($bytes$)
1  $numbers \leftarrow empty\_list$
2  $n = 0$
3  for $i = 1$ to LENGTH($bytes$)
4  do if $bytes[i] < 128$
5       then $n = 128 * n + bytes[i]$
6       else $n = 128 * n + bytes[i] - 128$
7            APPEND($numbers, n$)
8            $n = 0$
9  return $numbers$
```

► **Figure 5.7** Variable byte encoding and decoding. The functions div and mod compute integer division and remainder after integer division, respectively. Prepend adds an element to the beginning of a list.

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

► **Table 5.4** Variable byte (VB) encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

for VB encoding and decoding and Table 5.4 an example of a VB-encoded posting list.[2]

With variable byte compression, the size of the compressed index for Reuters-RCV1 is 116 MB, a more than 50% reduction of the size of the uncompressed index (Table 5.6).

2. Note that the origin is 0 in the table. Since we never need to encode a docID or a gap of 0, in practice the origin is usually 1, so that 10000000 encodes 1, 10000101 encodes 6 (not 5 as in the table) etc. We find the origin-0 version easier to explain and understand.

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
1025		11111111110	0000000001	11111111110,0000000001

► **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.

NIBBLE

The same general idea can also be applied to larger or smaller units: 32-bit words, 16-bit words, and 4-bit words or *nibbles*. Larger words further decrease the amount of bit manipulation necessary at the cost of less effective (or no) compression. Word sizes smaller than bytes get even better compression ratios at the cost of more bit manipulation. In general, bytes offer a good compromise between compression ratio and speed of decompression.

For most information retrieval systems variable byte codes offer an excellent tradeoff between time and space. They are also simple to implement – most of the alternatives referred to in Section 5.4 below are more complex. But if disk space is a scarce resource, we can achieve better compression ratios by using bit-level encodings, in particular two closely related encodings: γ codes, the subject of the next section, and δ codes (Exercise 5.7).

5.3.2 γ codes

UNARY CODE

Variable byte codes use an adaptive number of *bytes* depending on the size of the gap. Bit-level codes adapt the length of the code on the finer grained *bit* level. The simplest bit-level code is *unary code*. The unary code of n is a string of n 1's followed by a 0 (see the first two columns of Table 5.5). Obviously, this is not a very efficient code, but it will come in handy in a moment.

How efficient can a code be in principle? Assuming the 2^n gaps G with $1 \leq G < 2^n$ are all equally likely, the optimal encoding uses n bits for each G. So some gaps ($G = 2^n$ in this case) cannot be encoded with fewer than $\log_2 G$ bits. Our goal is to get as close to this lower bound as possible.

γ ENCODING

A method that is within a factor of optimal is γ *encoding*. γ codes implement variable length encoding by splitting the representation of a gap G into