

INDEX AND DOCUMENT PROPERTIES AND COMPRESSION

Chapter 3 (3.5,3.6,3.7) Chapter 4 (but not 4.5.2)

Chapter 5 (5.4,5.5), Chapter 6

DISCLAIMER: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2016 ; distribution outside this group of students is NOT allowed.*

Introduction: Efficient (or not) key search

Index oriented operations require the use of a **dictionary data structure** that supports operations **Insert**, **Search**, **Delete** applied to a **word** that is a token which is a string of alphanumeric characters of arbitrary length, a **wordID** which results from mapping a **word** to a integer-like numeric values of optimally small bit-size, and a **wordsum** which is a cryptographic fingerprint of a **word** of intermediate bit-size (to that of a **wordID** and that of a **word**). Likewise when those operations applied to a URL, its **docID**, and its **URLsum**.

Some bit manipulations and fundamentals (See also Handout 2)

How many bits do we need to represent	$2^{**2} = 4?$	and	$2^{**3} - 1 = 7?$	3	bits (100 and 111)
How many bits do we need to represent all integers	2^{**2}	..	$2^{**3} - 1 ?$	3	bits
How many bits do we need to represent all integers	0	..	$2^{**3} - 1 ?$	3	bits maximum
How many bits do we need to represent all integers	0	..	$2^{**n} - 1 ?$	n	bits maximum
How many bits do we need to represent all integers	0	..	$2^{**\lg n} - 1 ?$	$\lg n$	bits maximum
How many bits do we need to represent n				?	$\text{FLOOR}(\lg n) + 1 = \text{CEILING}(\lg(n+1))$

How do we map binary number 1011 into a decimal number?

3	2	1	0	
2	2	2	2	
x	x	x	x	
+ 1	0	1	1	

= $1x8 + 0x4 + 1x2 + 1x1 = 8 + 0 + 2 + 1 = 11$

Fingerprinting functions: CRC-32, MD5, SHA256, etc. A **word** or URL that we shall call it x generically can have variable length. Maintaining it multiple times in an index or index's data structures is too space and access time consuming. One prefers to obtain a mapping **key** that maps x to **key(x)** into a shorter and highly likely uniquely bit sequence. If x was an IPv4 address such as 128.235.32.40 then function $key(x) = key(a.b.c.d)$ would be easy to compute by $a \cdot 2^{24} + b \cdot 2^{16} + c \cdot 2^8 + d$, which is an integer between 0 and $2^{32} - 1$. However for arbitrary strings we prefer to use for **key** a cryptographic "fingerprint" (or checksum) function such as SHA-1 (20B output) or SHA-2 variants (eg. SHA256 with 32B and SHA512 with 64B output) or MD5 (16B) or something simpler (such as the error-correcting CRC-32 for a 4B long output). A naive implementation of **key(x)** that maps string ABZ to $65 + 66 + 90 = 221$ i.e. the sum of the ASCII values of A,B,Z is to be avoided! So for the remainder it would be convenient to assume that **wordsum** or **URLsum** are fixed length bit-sequence representations (can be thought of as unsigned or signed integers) of key x through function **key(x)** of 4B (for CRC-32) to 64B (if SHA512 is used). This way an arbitrarily long **word** or URL gets mapped into a fixed-length bit-sequence that uniquely represents it (the chances that two words get mapped to the same bit-sequence is remotely small).

We now offer a brief overview of Dictionary implementations (data structure plus algorithm implementations for Insert, Delete and Search) including some suboptimal ones. Consult your favorite CS114 textbook or equivalent for more details.

A1. Linked List or Array "implemetation" and their uses. A "naive" implementation of a dictionary using a **linked list** or an unordered array or even an ordered array would lead to some operations taking excessive linear time $O(n)$, where n is the number of keys maintained (keys can be words or URLs). The use of an ordered (aka sorted) table however allows the utilization of BinarySearch that can expedite a Search operation (but not Insert or Delete) to $O(\lg n)$. In fact Google@1998 maintained a sorted (sorted by URLsum) table of URLsum and docID tuples in a DOCIndex data structure. For a given URL its fingerprint URLsum was computed and then a binary search on the URLsum was used to obtain the docID corresponding to the input URL. However an ordered or not table suffers from linear-time performance and insertion in such an ordered table was slow (linear-time) and done rarely.

Other alternatives to linked list structures or ordered tables include **balanced tree structures** such as a red-black tree, an AVL tree, 2-3-4 trees, or B-trees, with which one can achieve logarithmic time for all three operations (Insert, Delete, Search) in the worst-case, i.e. $O(\lg n)$ for a data structure with n keys. The extra space for pointers however might become a concern. Google@1998 uses in its DOCIndex a B-tree structure to maintain meta information about documents: documents are indexed by docID and each entry contains metadata about the document such as information about its URL, and also its position in the Repository where its contents can also be retrieved.

A2. Direct Addressing. If the universe of possible key values U is of limited size, say a small number u , several optimizations can be achieved by using a direct addressing scheme. In **direct addressing of an array** we allocate an array of size equal **to the number of possible keys** and store a key in the corresponding table slot of the array in $O(1)$ time. For $U = \{0, \dots, u-1\}$ a universe of key values table $T[0..u-1]$ allocates key k to the k -th entry of the table. x below and in the remainder is a record and $key(x)$ is the key of record x , and $other(x)$ contains other relevant information.

DAddress-Search (T, k)	DAddress-Insert(T,x)	DAdress-Delete(T,x)
return(T(k))	T[key(x)]=x	T[key(x)]=NULL;

The drawback of this scheme is that if u is large, this scheme is uneconomical. Think of x being URLs of length up to 50, with the value of u in the range of 2^{63-50} (63 is the number of lower case, upper case English characters, digits and a space, but not including slashes or colons that form a URL). If however x is an IPv4 address such as 128.235.32.40 then function $key(x) = key(a.b.c.d)$ using $a \cdot 2^{24} + b \cdot 2^{16} + c \cdot 2^8 + d$ requires a table of $2^{32} - 1$ entries of size no more than $u = 4GiB$. However this scheme breaks down for an IPv6 address that consists of 16 bytes and a table of size $u = 2^{128}$ is impossible to get.

Hash Tables

Introduction

The introduction of **hash tables** allows faster constant-time on the average (i.e. practical) dictionary operations: worst case performance of a hash table is still that of an ordinary array. Hash tables, because of their simplicity have many practical applications. For example, a hash table is used in compilers to implement a keyword look-up table and a hash table constitutes an integral part of languages that deal with manipulation of string structures (e.g. Perl, Python). Google@1998 used a hash table to implement its lexicon.

A3. Hash Table. If allocation of a table of size m equal to $m = u$ is not possible because u is too large (cf IPv6 addresses), the alternative to direct addressing is **hashing**. A hash table allocates space **proportional to the (maximum) number of keys that we plan to maintain in the hash table** and not proportional to the number of possible key values. Thus if we only plan to maintain a million IPv6 addresses we need a table may of size approximately two million (for reasons that will be made clear later, a hash table should accommodate more keys than the number of keys that will actually be stored).

A4. Collisions and Fingerprints and Hash Tables. No matter what the choice of $key(x)$ the key value $key(x)$ is too large to be used as an index for a slot into a table with m slots. Thus the 2^{128} IPv6 addresses must be mapped into a table of 2,000,000 entries if we set $m = 2,000,000$. Even if CRC-32 is used to turn an address x of 128-bits into a $key(x)$ of 32-bits, we still need to map the 32-bit $key(x)$ into a index of the range $0, \dots, 1,999,999$ which requires 21 bits only. Thus an auxiliary function h must be used that will map $k = key(x)$ into a slot $h(k)$ of the hash table. **Collisions are then unavoidable.** A collision refers to the situation that two keys that are different i.e. $x_1 \neq x_2$ and even key value i.e. $key(x_1) \neq key(x_2)$, but $h(key(x_1)) = h(key(x_2))$.

A5. A minimization of collisions is a reasonable objective in designing a hash table and choosing a hash function such as key. **Key collision resolution** is, however, essential in hashing.

A6. Some terminology to be used. *For the remainder we shall assume that all keys are not in the “raw” x form; key $k = key(x)$ is available and thus they are binary numbers or their base-10 or base-16 integer representation..*

- 1. The number of keys to be hashed will be denoted by n .*
- 2. The size of the hash table is denoted by m and the slots are indexed $0, \dots, m - 1$.*
- 3. Load factor is the ratio $a = n/m$. As we mentioned before several times m should be chosen $m \approx 2n$ leading to a load factor of 50% .*

Hash Tables

Collision Resolution

B1. Collision Resolution Protocol: Chains (aka linked lists). One elementary scheme in resolving collisions is by **chaining** all keys k_1, k_2 that get hashed into the same slot $h(k_1) = h(k_2)$ form a (doubly) linked list attached or associated with that slot. Though the worst-case performance is $O(n)$ (all keys get into the same chain) for an m -slot table on the average a $O(1 + n/m) = O(1 + a)$ performance can be achieved (the average length of a chain would be about n/m). Thus, if we have 100 keys and 20 slots of a hash table, on the average 5 keys would get mapped to the same slot. This scheme requires extra space for pointers of the linked lists built and is usually avoided. The next protocol is used more often.

B2. Collision Resolution Protocol: Open-Addressing. In **open-addressing** the n keys get mapped inside the m slots of a table of size $m \geq n$. Each slot is wide enough to accommodate a key. For variable length keys a slot contains a pointer to the actual location that stores the variable length key. With reference to a prior discussion it makes sense to use such a hash table T to store **wordID** or **wordsum** that have fixed length but not actual words of variable length. **words** can then be stored in an auxiliary array A compactly (see discussion to follow). A slot contains either a key (eg **wordID**) or a flag (unused key value) that indicates that the slot is available for insertion. The load factor $a = n/m \leq 1$ is at most 1 for open-addressing (and preferably 0.5 if $m \approx 2n$).

0	1	2	3	4	5	6	7	8	9	10

					5	6	16			

h'(k)=k%11
h(k,i)= (h'(k) + i) % 11 [A naive and inefficient hash function]

Figure 1.

In order to insert key k we **repeatedly probe** the hash table until we can find an empty slot in which to put key k . Instead of probing slots in a predetermined order $0, 1, \dots, m-1$, the **probing sequence** depends on k and the choice of h . We thus extend hash function $h(k)$, to include a probe index starting from 0 thus defining for every key k a probe sequence that will be followed to determine where to insert a key (and also to search for a key). In fact a probing sequence that is a permutation of $0, \dots, m-1$ is the best choice.

$$h : U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\} \quad \text{Probing Sequence of key } k : \langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

Thus for key k , we first probe $h(k, 0)$, then if necessary $h(k, 1)$, and so on. If $h(k, m-1)$ is also occupied, then we determine that the hash table is full and this causes an overflow; no available slot will be used instead.

B2.1 Average Case Performance. For an Insertion or an Unsuccessful search (key not in the Hash Table) the performance of open-addressing on the average is $O(1/(1 - a))$. Thus if $a = 0.5$ and the hash table is half-full, on the average two probe operations are required for insertion or to determine that a key is not in the hash table. If $a = 0.9$ and the table is 90% full we need to perform 10 or 11 probes instead. Deletion and Successful search are a bit faster than Insertion or Unsuccessful search. Having said that worst-case performance is $O(m)$ as in the worst-case we might probe all the m slots of the table!

B3. What do we store in Hash table T ? What in an associated array A ? In all the previous examples we never stored into the hash table the key x , i.e. a URL or a word that are of variable length. A URLsum, wordsum, docID, wordID can then be stored (we used k earlier to denote this). A slot can be wide enough to accommodate this k value but also an index (or a pointer) p . Pointer p then points to an array (or memory location) storing the variable-length URL or word. For a variable length record such as a URL, word, the first few fixed bytes store the length l of the URL. If this is too much we can delimit URLs or words with a special NULL symbol (all zero bit sequence) to denote the end of one and beginning of another URL or word. (A NULL is a byte of zeroes for an ASCII or two bytes of zeroes for a UNICODE character). It is inefficient to store within T a URL or word: this is because a slot will have to be made wide enough to accommodate the longest URL possible. Thus if the average URL is 50B long and a slot is 200B wide, 75% of the hash table space will be wasted.

B4. English Dictionary Statistics: Average length of a word in a dictionary. The average length of a word in a dictionary is roughly 7 characters (in fact in a widely used corpus known as the TREC collection, it is a bit larger at 7.36).

B5. English Dictionary Statistics: Average length of a word in a text. The average length of a word in a text is somewhat smaller at 4 characters (about 3.86 for the TREC collection). This is because in a text we have repetitions of short words involving articles such as a, an, prepositions of, for, etc.

B6. Lexicon of words. On the next page we provide some alternative methods to maintaining a lexicon of words using a hash table. One of them is similar to the method Google used in 1998. The lexicon uses an open-addressing hash table scheme. The hash table T is minimally a hash table of pointers. The strings themselves are stored into a separate contiguous array A separated by NULL ('\0') characters. For efficiency purposes lexicons used in say Web-searching are memory resident.

Case study: Three organizations for word or wordIDs in a lexicon

Case T1. Suppose we would like to maintain a table T of n words using a hash table. Collision resolution by open addressing will be used and suggests that $m = 2n$. We choose to store, naively, the **word** in a slot of the hash table! Let the maximum length of a word to be maintained be 32 characters, yet the average length of a word be 7 (see note B4 on previous page). If ASCII encoding gets used, the number of characters is the number of bytes used. Total space of the hash table is $m \cdot 32B = 2n \cdot 32B = 64nB$. (If one has a PC with 512MiB of space, a large capacity for 1998 configurations, then one could barely accommodate $n = 512/64 = 8\text{M}$ words with such a scheme.)

Case T2a. Scheme T2a is a minor variation of **T1**; it maintains inside the hash some form of a pointer (or index) to an auxiliary array A that contains the **word** NULL separated from other words. A NULL character is `\0`. The bitlength of a **pointer** can be 32-bits i.e. 4B; 3B or 2B might be possible depending on the length of A and whether 3B or 2B indexes can then be used! The words themselves are stored in an array A of characters where words are separated. For hash table T this scheme uses $m \times 4B = 2n \times 4B = 8nB$ space if a pointer is 4B. The length of A is $n \cdot 8 = 8n$. This is because A maintains n words of average length 7 (see Case T1); adding the NULL at the end of it, the average word occupies 8B of space. The total space used for T and A is thus $8n + 8n = 16nB$, a fraction (one quarter) of the space used previously. 32M words can be accommodated instead. **Google's Lexicon** in 1998 bears similarity to this organization. A number of $n \approx 14,000,000$ words were maintained in the Google lexicon in less RAM memory.

Case T2b. This scheme is a slight variant of T2a. We maintain a **wordID** in addition to pointer p in a slot of the hash table. If we have n words we need at a minimum $\lg n + 1$ bits for a **wordID**. For $n < 2^{32}$ we would need no more than 4B per wordID. Thus **wordIDs** can consume $m \times 4B = 8n$ more bytes. If the previous scheme used $16nB$ of space this one uses $24nB$ of space. Roughly $n = 512/24 \approx 21\text{M}$ words can be maintained.

Hashing

Duplicate Detection

B7. Fingerprints. We use the term hash function to describe h in collision resolution by open-addressing when we maintain a hash table. The term "hash" or "hash-table" is also used interchangeably to denote a cryptographic fingerprint or cryptographic hash function such as those described on page 1 of this Subject (MD5, SHA-1, SHA256, SHA512) and also the checksum function CRC-32.

B8. Duplicate detection using digital fingerprints. Hashing can be used for fingerprinting to detect and eliminate duplicates. Consider two documents k_1, k_2 . We would like to detect whether they are the same or not. We could go through them character by character and decide whether the i -th character of k_1 is the same as the i -th character of k_2 or not, and this is the problem of string matching. However we could determine whether k_1 is different from k_2 by just checking the file size of k_1, k_2 . If they are different the documents are different. If they are the same, they might contain the same number of "characters", but not necessarily the same characters. Thus in this latter case a character to character comparison might be necessary. Say we have a hash function h that maps k_1 into h_1 i.e. $h_1 = h(k_1)$ and also k_2 into $h_2 = h(k_2)$. Even if k_1, k_2 are long, h_1, h_2 might be of fixed-length (CRC-32, MD5, etc). In such a case **duplicate detection** (i.e. identification that k_2 is a copy of k_1) is easier by comparing the fingerprints h_1 and h_2 rather than k_1 and k_2 . However there is one case (Case 2 in the diagram below), where a **spurious hit** might result. The two documents are different yet their hashes coincide (this was previously called a collision). This happens because a huge universe (of possible documents) gets mapped into a smaller address-space.

```
DuplicateDetection(h1,h2,k1,k2)
1. If h1 != h2
2.   Documents k1 and k2 are different
3. If h1 == h2
4.   Case 1. k1 and k2 can be duplicates of each other
5.   Case 2. k1 and k2 are different but it just happened that h1= h(k1) = h(k2) = h2
```

For a document, a single checksum for the whole document might generate plenty of spurious hits. One better alternative is to split it into parts. Say we have parts $a_1, b_1, c_1, \dots$ for document k_1 and $a_2, b_2, c_2, \dots$ for another document k_2 and generate hashes $h(a_1), h(b_1), \dots$ and $h(a_2), h(b_2), \dots$ for each part instead. Then for k_1, k_2 we have more than one fingerprints, and thus the number of spurious hits will decrease. How would one define the part of a document? It can be just a line of it, or a fixed collection (say, 10) of lines, if k_1, k_2 are text documents, or a fixed number of bytes (say 1000) if they are binary documents.

Hashing

Duplicate Detection: Simhash

B9. Simhash Algorithm. In the previous approach of fingerprinting text documents were viewed as a collection of characters or lines: they can also be viewed as a collection of words instead! Algorithm **Simhash** can be used to further reduce the chances of getting a spurious hit. Algorithm **Simhash** (d) can be described as follows when applied to document d : it generates a vector H containing b bits.

Step 1. Find all n (distinct) words in document d . For each word w_i identified, find its frequency f_i , i.e. generate pairs of words and frequencies (w_i, f_i) .

Step 2. Depending on the number of words found, generate arbitrary vectors of b -bits that will become hashes for each word. Thus for each word w_i one will generate a unique b -bit vector of zeroes and ones h_i that will be the hash for word w_i . Given that one has n words, it is expected that $b \geq \lg n$. (The b -bit hashes will be unique for unique words.)

Step 3. Create a b -word vector of real numbers $V = (v_1, \dots, v_b)$ initialized to zero. Also create a b -word 0-1 vector $H = (H_1, \dots, H_b)$ initialized to zero (this will become the output). Go through each word (for $i = 1$ to n) and through each one of the b bits of the hash for that word (for $j = 1$ to b) and apply one of the two steps below:

Hash-bit is 1. If the j -th bit of h_i is 1, add f_i to v_j .

Hash-bit is 0. If the j -th bit of h_i is 0, subtract f_i from v_j .

Step 4. Go through vector V . If v_j is positive, set $H_j = 1$, else set $H_j = 0$.

Step 5. The hash for the document is the b -vector H of zeros and ones.

Example. Consider a document consisting of data tropical fish data fast fish.

Step 1:	Find frequencies and words	:	data	2	0 1 0 1	b=4
Step 2:	Assign b -bit 0-1 vectors to each word		tropical	1	1 0 1 1	
	(we decided to use $b=4$)		fish	2	0 0 1 1	
			fast	1	1 1 0 1	
Step 3:	$V = (0, 0, 0, 0)$ initially					
	$V = (-2+1-2+1, +2-1-2+1, -2+1+2-1, +2+1+2+1) = (-2, 0, 0, 6)$					
Step 4:	$H =$				$(0, 0, 0, 1)$ is the hash for the document	

Several times during Web Search we realize that information becomes available **redundantly** and in **repetitive form** and this repetition can be eliminated by **compressing information**.

C1. Compression and Web Search. The documents obtained by the crawler are to be stored to the Repository. Compression can help reduce the amount of information maintained by roughly 66-75% or roughly to one-quarter of the original size. The index itself along with its auxiliary data structures can grow in size to be as large as the Corpus it indexes. Thus compressing the index and thus maintaining a compressed index can provide saving of roughly 50-75% .

In Google@1982 the uncompressed text was 150GiB. Compression brought that down to 53GiB. The index was hand-compressed to an approximate size of 64GiB.

C1.1 Compression and the Repository. Google@1998 indexed 150GiB of 25M indexed documents. The use of compression allowed Google to use 53GiB of disk space for roughly 3:1 compression. There are tradeoffs among various compression methods such as the time it takes to compress/decompress a document vs compressed volume size. Google@1998 chose speed over size by utilizing zlib/gzip over bzip. The benefit of this is that decompressing a document can be done faster (using zlib/gzip) than other methods (bzip). In this Subject we touch the subject of data compression by introducing some concepts and discussing a primitive compression algorithm that is known as **Huffman coding**.

C1.2 Compression and the forward index. As we mentioned earlier the forward index, for a given docID contains all the wordIDs that appear in it. With reference to say Subject 2, page 24-26, successive tuples of the forward index can contain repetitions of the same docID. Moreover word offsets within the same document are almost consecutive in the forward index; thus the difference between two adjacent offsets is a small integer value. *There is room to compress both the docID and offset information in the forward index.*

C1.3 Compression and the inverted index. As we mentioned earlier the inverted index, for a given wordID contains all the docIDs that contain it. Furthermore, successive tuples of the inverted index can contain repetitions of the same wordID. And when consecutive wordID values that are different differ by small **gap values**. *There is room to compress the tuples with common wordIDs by factoring them out; there is room to compress the gaps of consecutive wordIDs that differ by small gap values.*

C1.4 Compression in the index: Doclists. For a given **wordID** its tuples contain a sorted order of **docIDs** that describe all the documents into which the **wordID** appears. If no offset or other information is recorded this list is known as the **Doclist**. If the **wordID** is a frequent one in the Corpus, its **Doclist** would be long, and successive **docIDs** would differ by a small integer (**small gap**) since frequent **wordIDs** have long **Doclistss**. Such a long **Doclist** can be compressed by compressing the small gap values between consecutive **docIDs**.

C1.5 Compression in the index: Occurrence Lists. If a word offset is recorded for the occurrence of a given **wordID** in a given **docID**, and this was the case back in Subject 2, a tuple is used that records not only **docID** information but also word offset. In such a case the tuple list is known as the **Occurrence List** of a given **wordID**. In occurrence lists, **docIDs** are also sorted as in **Doclists** and compression demands arise for similar reasons: remove repetitions or compress the small gap values of consecutive **docIDs**.

Thus a **docID** 1223456780 might be followed by a **docID** such as 1223456788 and that one by 1223558780. The first two **docID** differ by 8 and the next two by 2,000. Although the **docIDs** need 4 bytes for a minimal representation the indicated gaps need one or two bytes only. Thus even if **docIDs** are large numbers the **gaps** between two consecutive **docIDs** are small numbers.

C1.6 Gap encoding or Gap compression. Thus index compression involves *compressing the gaps between consecutive docIDs or offsets or even wordIDs*.

We discuss algorithms such as **v-byte encoding**, **Elias- γ** , **Elias- δ** , Golomb-b in this context. Other parts of the index might also get compressed by these or other methods.

C2. Text compression: from T to C . In text compression we are given a file or string T of characters drawn from some alphabet Σ and we want to efficiently encode T into a smaller in size string C over some other alphabet Σ' . The two alphabets can overlap. Moreover Σ' might contain only two characters 0 and 1. This way a character of Σ might get converted into a bitsequence of 0 and 1s of Σ' . Those bitsequences can then be packed into forming bytes (8-bit sequences) and thus eventually C .

C2.1 Compression and Decompression: Encoding and Decoding. By way of **compression** every element (character) of Σ is assigned a distinct and unique bit sequence under a mapping E that is known as **encoding**. The mapped value becomes the **code** of the corresponding character. Furthermore during compression every character of T is mapped through E to generate output C . The encoding E should be such that we can retrieve T using C and the encoding E , along with Σ and Σ' . The retrieval of T from C is done during **decompression** and involves the recovery (i.e. **decoding**) of the original character from the corresponding code.

$$\text{Compression : } C = E(T, \Sigma, \Sigma') \quad \text{Decompression : } T = E(C, \Sigma, \Sigma')$$

C2.2 Compression Ratio. The **compression ratio** is defined to be the ratio of the size of the corresponding uncompressed file over the size of the compressed file i.e. $|T|/|C|$.

C2.3 Codes: ASCII and Unicode. The bit sequence assigned to a given character is called the **code** of that character. Thus the code for A in an ASCII code is 65 in decimal, or 41 in hexadecimal (0x41), or 101 (0o101) in octal. In UNICODE this would be 0x0041 in hexadecimal: two bytes one of which is all-zero and the other the ASCII code of the character. If each character is equally likely to appear in a document (or string) it makes sense to assign the same number of bits (or bytes) in the code of that character. *Since n bits have 2^n distinct values, an alphabet with $|\Sigma|$ characters requires $\lg(|\Sigma| + 1)$ bits.* In English, certain letters are more likely to appear than others (e.g. e is more frequent than w or x). Thus it makes sense to assign fewer bits to more frequent characters and more bits to infrequent characters. This gives rise to **bit-aligned** codes, where one byte of memory can be used to encode one or more characters.

C2.4 Byte-aligned codes. Such codes respect end of byte boundaries and use a fixed number of bytes per code. Thus ASCII (1B) or UNICODE (currently, 2B) character codes are byte-aligned codes; so is v -byte codes (to be discussed).

Several of the codes that we will be discussing in the remainder are bit-aligned codes.

C2.5 Bit-aligned codes. In a bit aligned code, the breaks between one word (code) and another one can occur at any bit position, not necessarily at multiples of 8 bits (byte). This however creates a certain number of problems. Let A,C,G,T (to keep this example short) have "codes" 0,1,10,11 respectively. Then encoding ACACGT into 01011011 causes a number of problems during decompression. The compressed text can also be decompressed as AGTAT!

C2.6 Prefix codes. Thus for bit-aligned codes it is imperative that no code is the **prefix** of another code, and in the previous example, 1, the code for C, is prefix of both 10 and 11, the codes for G,T. Such codes are called **prefix codes** (and imply the prefix-free property)!

C2.7 Statistical coding (Huffman, Arithmetic coding). Huffman coding and other similar methods (e.g. arithmetic coding) are **statistical**-based in the sense that they generate good probability estimates of the appearance of every symbol in the text. Such models assign probabilities to the next symbol to be encoded. Models can be

adaptive which assume initially nothing about the text and build their model/probabilities during compression,

static that they use a-priori probabilities; if a document deviates from that model, they will perform poorly,

semi-static which perform two or more passes of the text, first to collect information and then build the compressed text (e.g. ordinary Huffman coding).

Compression models can also be classified as

- **Character**-based if they use characters as symbols (e.g. ordinary Huffman coding).
- **Word**-based if they use words as symbols.

C2.8 Arithmetic coding computes codes incrementally, one symbol at a time, and decoding from the middle of a compressed file requires a restart. For this reason arithmetic coding is not a very good candidate for text-based document processing as this is used in Information Retrieval/Web-searching. *Huffman coding* is fixed-length bit coding. Each symbol is pre-encoded using a fixed-length number of bits. Decoding a string starting in the middle of a file is possible without having to decode everything from the beginning to the desired word (as long as we know the codes for all characters of course).

C2.9 Character-based and Dictionary-based. Huffman coding is (primarily) *character-based* i.e. it assigns codes to individual characters. *Dictionary-based* methods such as LZ77 (gzip, ARJ, LHarc, PKZip), LZ78/LZW (arc, pak, UNIX compress) are those in which a dictionary of words is being built and thus codes are assigned to words not (necessarily) individual characters. Huffman coding can be dictionary-based; such an approach makes sense if the number of words is small.

C2.10 Lossless vs Lossy compression. Data compression techniques are **lossless** or **noisy/lossy**. For a lossless technique, one can get T from C and E . There are also **noisy or lossy** techniques where full-recovery of T from C is not possible. Thun in decompression $T' = E(C, \Sigma, \Sigma')$ one retrieves T' and not necessarily T : however T' is an approximation of T and contains enough information available in T , but not all of it. Such techniques can be used in image or voice compression. JPEG is a lossy image compression algorithm. GIF is a losless data compression algorithm based on the Lempel-Ziv-Welch (LZW) algorithm. PNG is also lossless (and for sometime was used as a replacement for GIF).

C3.1. Entropy (Charact-based). The amount of information in a document is known as its **entropy**. A document N characters long consists of n different characters; the i -th character has frequency f_i and thus $\sum_{i=1}^n f_i = N$. Let $p_i = f_i/N$. (One may call p_i the a-priori probability of the i -th character appearing in the document or the fraction of the text using the i -th character, and $0 \leq p_i \leq 1$.) The entropy is the defined as

$$\text{(Document of size } N\text{.) Entropy : } E = - \sum_{i=1}^n p_i \lg(p_i) = - \sum_{i=1}^n \frac{f_i}{N} \lg\left(\frac{f_i}{N}\right), \quad \sum_i f_i = N, \quad |\Sigma| = n \text{ distinct characters}$$

C3.2 Entropy of (binary alphabet) documents. Consider a document with N zeroes and ones. If the number of zeroes and ones is the same $N/2$, then the entropy of the document is 1. ($-p_0 \lg p_0 - p_1 \lg p_1$, where $p_0 = p_1 = (N/2)/N = 1/2$, and thus the entropy is $-1/2(-1) - 1/2(-1) = 1$). The entropy of 1 indicates that one bit suffices to encode the characters of the document: indeed one bit is needed to encode a 0 or a 1! ASCII or Unicode encoding is an overkill!

C3.3 256-character (extended) ASCII text. If the document on the other hand contains $n = 256$ characters (think of ASCII or extended ASCII), each one with the same frequency $N/256$ we also get that $p_i = (N/256)/N = 1/256$. Similarly, the entropy of this document is also $-\sum p_i \lg(p_i) = -\sum 1/256 \lg(1/2^8) = -\sum (1/256)(-8) = 8 \sum_{i=1}^{n=256} 1/256 = 8$.

For this case 8 bits are needed per character. For a general n , if each one of the n characters is equally likely to show up in the document/text, we can show that the entropy is going to be $\lg n$. Thus the ASCII encoding assumes a-priori a uniform distribution for the characters of a text.

C3.4 Small Alphabet text. Now assume that the text only contains four characters (aka symbols) A,C,G,T with frequencies $N/2$, $N/4$, $N/8$, and $N/8$ corresponding to “probabilities” (or fractions) of $p_A = 1/2, p_C = 1/4, p_G = 1/8, p_T = 1/8$. The entropy is equal to $1/2 \cdot 1 + 1/4 \cdot 2 + 1/8 \cdot 3 + 1/8 \cdot 3 = 1 + 6/8 = 1.75$ which is the average number of bits of information of the text (per character). We could use

$-\lg(1/2), -\lg(1/4), -\lg(1/8), -\lg(1/8)$ i.e. 1,2,3, and 3 bits to encode each one of the four characters and by doing so we would use $1.75N$ bits altogether or 1.75 bits per character. Thus the entropy is an indicator of the minimal number of binary digits we can use to encode each character of the next on the average. This is much less than standard ASCII coding that uses 7 (or 8) bits per character for a total of $7N$ (or $8N$) bits. Or one that uses two bits for each one of the four characters for a total of $2N$ bits. Because the “probabilities” are inverse powers of two, Huffman coding leads to a code that has this property of (1.75 bits per character on average and equal to the entropy of the text). The code generated is known as Shannon-Fano code and same in bit length to the corresponding Huffman code.

H1. Huffman codes. The primary method that will be presented in this subject for text compression is Huffman coding that yields what is known as Huffman codes. Huffman codes are bit-aligned prefix codes. They lead to lossless compression. Huffman encoding requires that we know in advance the alphabet Σ and also the frequency of each character in the document or text that will be compressed. Each character is represented by a different encoding string or binary character code (or code in short) in such a way that no encoding string is a prefix of another encoding string: prefix codes are being used. Huffman codes result in space savings of 20% to 90% depending on the structure of the input string or file that is to be compressed.

H1.1 Huffman coding performance. Nowadays it is not very often that anyone is using Huffman compression (aka coding) to compress a text or strings of characters. Average savings observed is of 3 bits per character with 5bits being used for character encoding. This is not good enough any more. Lempel Ziv (LZ78) uses only 4.2 bits on the average, and LZ77 a bit less. The UNIX `compress` function uses approximately 3.31 bits and `gzip` close to 2.53. And `bzip2` uses 2.23 bits per character which implies close to 4:1 compression.

H1.2 Huffman coding variants. However instead of doing character compression/encoding we can do word compression/encoding. Huffman can exhibit substantially better performance to the level of 2.95 bits per word.

H2. Huffman decoding (decompression). The important property of prefix codes is that decoding is quite straightforward. Since no code is a prefix of another one, that code at the start of a string or a file is unambiguous and can be removed from the coded file; repeating this process decodes the rest of the string or file uniquely.

H3. Prefix code representation: Binary Tree. A character of the alphabet is a leaf of that tree. A code is a description of the path from the root to the specified character/leaf. Every internal node of the binary tree has a left child and a right child; the left branch is encoded as a 0 and the right branch as an 1. The binary tree representing a prefix code is full with $n - 1$ internal children where n is the number of distinct characters appearing in the text/document/string and correspond to the leaves of the tree.

H4. Building the binary tree of prefix codes. The characters of a file (or a string) that is to be compressed are collected and their frequencies computed. Thus the frequency $f(c)$ of any character c becomes available. Every node of the tree is going to have a right and left child (i.e. the binary tree is full) except for the leaves that have no children. Every node will have a frequency field that records the sum of the frequencies of its left and right children; this is also the sum of the frequencies of the leaves in the subtree of that node. The number of bits required to encode the file with a given prefix code tree is $B(T) = \sum_{c \in C} f(c)d(c)$, where $d(c)$ is the depth of character c in the tree (i.e. its distance from the root), and $f(c)$ is the frequency of character c in the document/file/string. used to obtain the tree.

The algorithm that constructs a Huffman code (i.e. an optimal prefix code) is a greedy algorithm. It builds the tree that corresponds to the code bottom to top. Initially the n characters form individual one-node trees. Any single node is the root of its own tree. The information associated with each node is the character representation (e.g. 'A') of the node and the frequency of the character in the string/file. The algorithm works bottom-up by combining two trees into a single one by creating a root node for the new tree whose left child is the root of one of the two trees and the right child the root of the other tree. The frequency field of the new root is the sum of the frequencies of the roots of the two constituent trees (now, children). There is no character representation associated with such an internal node. Therefore after $n - 1$ such operations the n original trees have been merged into one whose frequency field is the number of characters N of the string/file.

Greedy principle for combining trees. The criterion for choosing the two trees to combine is a “greedy” one: the two trees with the lowest frequency fields at their respective roots are chosen.

H5. Computer Implementation Details. We use a priority queue to maintain the trees, and implement the priority queue with a MINHEAP. The priority of a tree is the frequency of its root.

```
Encoding(Uncompressed ufile, Compressed cfile)
1. Read file, find its characters, compute frequency of each character.
2. Build Huffman tree T from chars and their frequencies.
3. Use T to encode ufile into compressed cfile (might include in cfile information about T)
   // This line 3 is Encode(ufile,T,cfile);
```

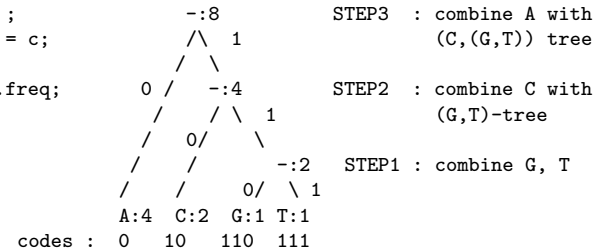
```
Text is AACCGTA
A: 4 C: 2 G: 1 T:1
```

Huffman codes

Encoding and Decoding

Build HuffmanTree(T)

1. For every character c build a one node binary tree with character c as its root ;
Frequency of root is the frequency of its character i.e. c.freq= freq(c); c.val = c;
2. while there are more than one binary trees do {
3. Pick the two trees y and z whose roots have the lowest frequency y.freq and z.freq;
4. Build a new tree with root x;
5. x.left = y ; x.right = z; x.freq = y.freq + z.freq;
6. }
7. return(RootOfOnlyTreeLeft);



Encode(Uncompressed ufile, HuffmanTree T, Compressed cfile)

1. For every character c of ufile {
2. Determine the path from root(T) to the leaf that is that character c
3. Record path (left branch is 0 , right branch is 1)
4. Emit into cfile path 0-1 information
- }

ufile is : AACCAGTA

cfile is : 0 0 10 10 0 110 111 0
ie 00101001101110

Decode(Uncompressed ufile, HuffmanTree T, Compressed cfile)

1. r = root(T);
2. for every bit b of cfile {
3. if b is 0 r=r.left
4. else if b is 1 r=r.right
5. if r is a leaf
6. emit the char value of the leaf into ufile;
7. r= root(T);
8. else
9. continue; // i.e. read one more bit b from cfile
- }

cfile is 00101001101110
 root of T above: read 0 ---> A leaf reached;reset
 root of T again read 0 ---> A leaf reached;reset
 root of T again read 1 ---> -:4
 read 0 ---> C leaf reached;reset
 root of T again read 1 ---> -:4
 read 0 ---> C leaf reached;reset
 root of T again read 0 ---> A leaf reached;reset
 root of T again read 1 ---> -:4
 read 1 ---> -:2
 read 0 ---> G leaf reached;reset
 continue likewise to generate T and then A...

I1. (Inverted) Index: Doclist The inverted index of a corpus consist of a collection of lists for each **word** or **wordID** of the document. In its simplest form this collection of lists is a collection of docIDs. The **Doclist** for a given **wordID** records the docID of all the documents containing that wordI. A Doclist is sorted by docID. The number of **doclists** is the number of **wordIDs** of the corpus. Consider for example a doclist for some **wordID**=20 corresponding to **word**= DATA consisting of [1000, 1002, 1003, 1004, 1010]. Say a docID is 4B i.e. 32bits. This doclist would thus use $4 \cdot 5 = 20B$.

```
Forward Index (docID,wordID) no word offsets
(1000,20)
(1002,20)           generates [20] = [ 1000, 1002, 1003, 1004, 1010]
(1003,20)           the doclist
(1004,20)
(1010,20)
```

I2. (Inverted) Index: Occurrence List. More often that not this **Doclist** is called an **Occurrence List**; an occurrence list contains offset information and other context information about the word (or wordID). It is sometimes called Hits (and rarely, a HitList) or Positions. For a given docID more than one entry may be included in that list. In Subject 2, each element of the Occurrence List of a given wordID was a triplet $\langle \text{docID}, \text{offset}, \text{context} \rangle$ containing the docID in which an instance of wordID appears, a word offset within that document describing the position of wordID in that document, and attribute information that describes the context of the appearance of that instance of wordID (eg fontsize, capitalization etc). During the inversion process Occurrence lists are sorted first by **wordID** and for the same **wordID** by **docID** and then **offset**. The size of such a list can be huge : think for example the length of the occurrence list for a stopword-like word such as **a** or **the** in google. Consider for example an Occurrence List for some **wordID**=20 corresponding to **word**= DATA and the Doclist [1000, 1002, 1010].

```
Forward Index (docID,wordID,offset)
(1000,20,10)
(1000,20,15)           generates [20] = [ (1000,2,[10 15])
(1002,20, 5)           (1002,2,[5 15])
(1002,20,15))          the doclist   (1010,1,[ 7]) ]
(1010,20, 7)
```

I3. Compressing doclists or occurrence lists. Most wordIDs in a Corpus have Doclists that are short. We are not concerned about them. It is long Doclist that cause storage related performance problems. The byte size of a long doclist can be reduced significantly if specialized compression methods are to be used based on the principle that

“small numbers are more likely to occur than large numbers”,

Although the **small** might refer to nominal values such as frequencies or table lengths, it more commonly refers to **differences** (also known as **gaps** or **delta**) of consecutive docIDs or **offsets** (or even wordIDs).

Observation 1. *Frequent words have small/short gaps and very long Doclists.*

Observation 2. *Infrequent words have larger gaps and thus short Doclists and thus they do not need compression!*

Remark: Gap encoding, Delta encoding or δ -encoding. Sometimes the encoding of gaps (differences) of docIDs is known as **gap encoding** or **delta encoding** or **δ encoding**. (The Greek letter δ is known as delta.)

I4. Example: Occurrence List. Suppose we isolate the inverted index of a single wordID appearing in only three documents: 1000, 1002, and 1010. A Doclist would include 3 docIDs but an Occurrence List 5 tuples. Compression is possible by combining entries for the same docID. All entries for the same wordID and docID are grouped together into a single **Hits** that also records the length **Nhits** of **Hits**.

In such a scheme we can gap-encode the docIDs with the exception of the base docID that is not encoded). Nhits are never encoded. Moreover Hit entries (offsets) are also gap-encoded similarly to docIDs.

By default we use 4B integers to represent the information in the uncompressed Occurrence List by default: 44B will be used. For the compressed list only the baseline docID or Hits entry and the Nhits entries need be 4B long. Everything else can be compressed. Thus the first list would use $4 + 4 + (4 + 1) = 13B$, the second one $4 + 4 + (4 + 1) = 13B$ and the third one $4 + 4 + 4 = 12B$. This is a total of 38B vs 44B originally. Thus even for a simple artificial (and not very good) example roughly 15% was shaved off.

	Uncompressed Occurrence List	Compressed Occurrence List
(1000,20,10)		
(1000,20,15)	(1000,2,[10 15])(1002,2,[5,15]),(1010,1,[7])	(1000,2,[10 5])(2,2,[5 10])(8,1,[7])
(1002,20, 5)	[10 15] becomes [10 15-10]= [10 5]	
(1002,20,15))	1002 becomes 1002-1000	
(1010,20, 7)	1010 becomes 1010-1002	

Specialized Compression

Gap encoding and Prefix Computation

I5. Recovery is easy: It involves a prefix computation. For a sequence of numbers $x_1, x_2, \dots, x_n$ the prefix sum computation involves the computation of n prefix sums

$x_1, x_1 + x_2, x_1 + x_2 + x_3, \dots, x_1 + x_2 + x_3 + \dots + x_i, \dots, x_1 + \dots + x_n$. The i -th prefix sum can be computed from the $(i - 1)$ -st prefix sum by adding to it x_i .

The prefix sum of the n numbers	$x_1,$	$x_2,$	$x_3,$	$\dots,$	x_n
is the sequence of the n sums	$x_1,$	$x_1+x_2,$	$x_1+x_2+x_3,$	$\dots,$	$x_1+x_2+\dots+x_n$
	$S_1=x_1,$	$S_2=S_1+x_2,$	$S_3=S_2+x_3,$	$\dots,$	$S_n=S_{n-1} + x_n$

Thus a $[10, 5]$ after a prefix sum becomes $[10, 15]$ whereas $[5, 10]$ becomes $[5, 15]$. Moreover the docID related information 1000, 2, 8 becomes 1000, $1000 + 2$, $1000 + 2 + 8$ i.e. 1000, 1002, 1010.

I6. Prefix Sums in Parallel. A prefix sum computation is very efficient. A parallelization of a prefix sum computation in PC clusters is also possible (Google's MAPREDUCE language/system is built to perform such computations efficiently.) It is known as a parallel prefix sum computation. A parallel prefix sum computation is also known as a **scan, prefix-sum** and can be done for any associative operator not just sum (e.g. multiplication, MIN, MAX, etc).

I7.1 Note (Associativity). Operator $+$ is associative if and only if $(a + b) + c = a + (b + c)$ for all a, b, c .

I7.2 Note (Commutativity). Operator $+$ is commutative if and only if $a + b = b + a$ for all a, b .

Gap Compression

Byte-aligned: v-byte encoding

I8. Variable-byte length encoding using byte-aligned codes. The example in I4 was intentionally simple and fitting. All the gaps generated were small numbers whose values could fit into a single byte. In general to represent a gap in gap-encoding we might need more than one byte: 2B or 3B or even 4B. We have the option of using bit-aligned coding of gaps or byte-aligned. One such latter method is what is known as **variable byte length** or **v-byte** encoding. *It uses short codes for small numbers and longer codes for larger/longer numbers.*

Each code is a sequence of bytes rather than bits i.e. it is a byte-aligned code. So the shortest code is one byte long. If a value cannot fit into one byte, a second byte is used; if it cannot fit into two bytes a third byte is then used and so on.

I8.1 v-byte encoding. The v-byte encoding of a number k represents the number in binary in the low seven bits of a collection of bits. The low seven bits (seven rightmost bits) contain numeric data in binary. The most significant (leftmost) bit is a terminator bit. The last byte of each code has the terminator bit 1; any other byte(s) has it equal to 0. If there is only one byte involved it has the bit set to 1. The method can represent only non-negative integers. Thus a number k which is less than 128 it can be represented by one byte 1AAAAAAA where AAAAAA is the binary representation of k . If a number is 128 or more but less than 2^{14} , it can be represented by two bytes 0BBBBBBB 1CCCCCCC, where BBBBBBBBCCCCCCC is the binary representation of k , and so on (the left-most B is the most significant bit, and the right-most C the least significant bit). Note all but the rightmost byte have a 0 leftmost bit; the rightmost byte has a 1 leftmost bit: this indicates it is the last byte of the sequence identifying a number.

k=0	10000000	in hex 0x80	for binary 00000000 whose hex value is 0x00
k=1	10000001	in hex 0x81	for binary 00000001 whose hex value is 0x01
k=127	11111111	in hex 0xFF	for binary 01111111 whose hex value is 0x7F
k=128	00000001 10000000	in hex 0x01 0x80	for binary 10000000 whose hex value is 0x80

Note: This discussion is from page 148 of the textbook. Note also that on page 150 there is a typo in the textbook. 4th line from the bottom, the sequence ... 1, 2, 6, 11, 180 ... should have been ... 1, 3, 6, 11, 180, ..., to comply with the textbook's "We can't delta-encode the word counts".

I9.1 Occurrence lists and v-byte encoding. Let us use an Occurrence list of pairs: the first element of a pair is a `docID` and the second a word offset within that document. It looks like $(1,1) (1,9) (2,6) (2,7)$

I9.2. Step 1: Generate Hits and NHits. This step is also performed by Google 1998 and involves the grouping performed involving word offsets (also known as positions) within the same document. Thus for each document we will create a triplet $(\text{docID}, \text{Nhits}, [\text{positions}/\text{offsets}]) = (\text{docID}, \text{Nhits}, \text{Hits})$. The end result would be to get something like $(1,2, [1\ 9])$, $(2,2,[6\ 7])$

instead of a long list. In this example the first element of a triplet is a `docID` such as 1 and 2 respectively. The second element is the `Nhits` or the length of the `Hits` to follow i.e. 2 in both cases. Thus the `docID` is stored just once and not for every pair. So this simple method "compresses" the original "hitlist" in a very simple and obvious way. The `Nhits` field is important because it determines how long the `Hits` list is especially if we deparenthesize and debracket the notation used above.

I9.3 Step 2: Flatten out. If we deparenthesize and debracket we get

1, 2, 1, 9, 2, 2, 6, 7

I9.4 Step 3: Gap encoding. For the numbers involved we now use gap-encoding i.e. delta-encoding which means instead of using `x`, `y`, `z` in a list we encode it by using $X=x$, $Y=y-x$, $Z=z-y$ i.e representing it with `X`, `Y`, `Z` instead . In order to retrieve the `x,y,z` from the `X,Y,Z` we need to perform a prefix computation i.e. compute `X`, `X+Y`, `X+Y+Z`. The delta-encoding would apply to `docID` and `Hits` fields but not to `Nhits`. Our example list would then become the first line below, that with the prefix computation involved in the second line could retrieve the original list.

1, 2, 1, 8, 1, 2, 6, 1
since this means
1, 2, 1,1+8,1+1, 2, 6, 6+1

I9.5 Step 4: Convert into Hexadecimal v-byte codes. A *v*-byte encoding would then give in hexadecimal (note that 0x81 is in hexadecimal and it is 10000001 in binary or 129 in decimal).

81, 82, 81, 88, 81, 82, 86, 81 (hexadecimal notation)

J1. Bit-aligned codes. In this section we discuss **bit-aligned codes** as an alternative to byte-aligned codes to address the problem of gap encoding in index compression.

J1 Optimal bit-aligned code: Binary code. The **optimal bit-aligned code** is the binary representation of an integer in the minimal number of bits. However this can not be used in index compression because the resulting code is **NOT a prefix code**. An optimal code to represent k is its binary representation. That code for k uses only $\lfloor \lg k \rfloor + 1$ bits; however it is not a prefix code. This is because the codes for 0, 1 (binary or decimal) are prefixes of the codes for 2,3, which are 10 and 11 respectively, which are also prefixes for the code of 4,5,6,7 which are 100, 101, 110, 111, and so on. Thus if one sees the binary sequence 10010011 one cannot determine if it encodes 4,4,3 or 2,0,1,0,0,3, or 9,0,0,1,1. One must use prefix codes to be able to untangle combined binary sequences of varied lengths.

J2. Bit-aligned prefix codes. Bit-aligned codes that are prefix codes that can be used for gap-encoding are presented below. There are four (primary) classes of such codes.

- Unary codes.
- Elias- γ codes.
- Elias- δ codes.
- Golomb-b codes. (If b is a power of two, they are also known as Rice codes.)

J2.1 Unary codes. The unary code for k uses $(k + 1)$ bits. (If we decide not to represent 0 we can go down to k bits as well.) It is very un-optimal relative to the "optimal" binary code.

J2.2 Elias- γ codes. The Elias- γ code for k will use $2\lfloor \lg k \rfloor + 1$ bits. It is close to the performance of the "optimal" binary code.

J2.3 Elias- δ codes. The Elias- δ code for k will use $2\lfloor \lg (\lfloor \lg k \rfloor + 1) \rfloor + \lfloor \lg k \rfloor + 1$ bits. It uses Elias- γ codes recursively.

Bit-aligned codes

Unary codes

J3. Unary codes. We are to devise methods that can compress small numbers (gaps) efficiently in as little space as possible. A unary code is a bit-aligned code. The unary code of number k (i.e. its representation) is a string consisting of k ones followed by a single zero. (Equivalently one can use k zeros followed by a single one as the representation.) Thus

Number k	Code for k
0	0
1	10
2	110
3	1110

J3.1 Number of bits in unary representation of k is equal to $k + 1$ (includes k ones followed by one zero).

J3.2 Observation. One might observe that for gaps in doclists, if docIDs are unique, there will be no gaps that are zero. Thus one might not have to represent 0 at all.

J3.3 Alternative representation. When a 0 is not to be represented, one could represent k by $(k - 1)$ 1's followed by a single 0. This representation uses one fewer bit than before.

J3.4 Unary codes vs Binary codes. In unary codes the number of bits to store k is $(k + 1)$. This is too much. One could use a **binary** code and thus only need $\lceil \lg(k + 1) \rceil = \lfloor \lg k \rfloor + 1$ bits. (In the following section the floor function variant will be primarily used.) The problem with binary codes, as explained earlier, is that they are not prefix codes.

J3.5 Unary codes vs Hexadecimal or octal codes. Besides unary and binary codes (representations) we have decimal (base-10), octal (base-8) and hexadecimal(base-16) representations or codes.

J3.6 Number of bits in binary representation of k is equal to $\lfloor \lg k \rfloor + 1$ or $\lceil \lg(k + 1) \rceil$. (Both give the same answer.)

J4. Elias- γ . A unary representation uses too much space. Elias- γ (Elias-gamma) codes combine unary and binary codes and provide a variable length bit-aligned coding scheme. The **concatenation** of two strings a and b is putting the characters of b immediately after a . Thus the concatenation of 101 and 000 would give 101000.

The Elias- γ code of natural integer k is represented by the concatenation $w_d w_r$ of w_d and w_r , defined as,

- w_d is k_d in unary, where $k_d = \lfloor \lg k \rfloor$. Thus w_d consists of k_d ones followed by a single 0. The number of bits used for w_d is thus $\lfloor \lg k \rfloor + 1$ or in other words $k_d + 1$. This is also the number of bits in the binary representation of k or the unary representation of k_d !
- w_r is k_r in binary using exactly k_d bits, where $k_r = k - 2^{k_d} = k - 2^{\lfloor \lg k \rfloor}$. k_r is thus what is left of the binary representation of k if one drops the leftmost (most significant) 1 in the representation. The position of that one in the representation is k_d (we count from the rightmost position starting with 0). The number of bits to represent w_r is the number of bits in the binary representation of k_r which is the number of bits in the binary representation of k minus the leftmost bit 1, i.e. $\lfloor \lg k \rfloor + 1 - 1 = \lfloor \lg k \rfloor$. The number of bits used for writing w_r is thus $\lfloor \lg k \rfloor$ i.e. k_d bits.
- Since $w_d w_r$ becomes the Elias- γ code of k , the number of bits used in this representation is $2k_d + 1 = 2\lfloor \lg k \rfloor + 1$.

Example: Find the Elias- γ code of $k = 7$.

Consider $k = 7$. Its unary code is 11111110, i.e. 8 bits. Its binary code is 111, i.e. three bits. Observe that $k_d = \lfloor \lg k \rfloor = \lfloor \lg 7 \rfloor = 2$. The formula for the number of bits in the binary representation of arbitrary k is $\lfloor \lg k \rfloor + 1 = \lfloor \lg 7 \rfloor + 1 = 2 + 1 = 3$, which is correct since $7_{10} = 111_2$ (the subscript shows the base).

1. What is w_d ? Form $k_d = \lfloor \lg k \rfloor = \lfloor \lg 7 \rfloor = 2$. The unary representation of $k_d = 2$ is 110 and thus $w_d = 110$. Number of bits in w_d is $k_d = \lfloor \lg k \rfloor + 1 = 3$, which is the number of bits in the binary representation of $k = 7$.
2. What is w_r ? Form $k_r = k - 2^{k_d} = 7 - 2^2 = 3$. The binary representation of $k_r = 3$ is 11, and thus $w_r = 11$. Note that 11 is 111 if the left-most 1 is thrown away. The left-most one is the second one ($k_d = 2$), since counting starts from the rightmost position and with 0. Note also that the number of bits for w_r is $k_d = 2$.
3. What is the Elias- γ code for $k = 7$? It is $w_d w_r$ i.e. 110 11 which is 11011 without spaces. The total number of bits is $2k_d + 1 = 2\lfloor \lg k \rfloor + 1 = 2\lfloor \lg 7 \rfloor + 1 = 5$.

One more Elias- γ example.

Example: Find the Elias- γ code of $k = 8$.

Consider $k = 8$.

1. **Unary code** for $k = 8$ is 11111110, i.e. of $k + 1 = 9$ bits.
2. **Binary code** for $k = 8$ is 1000, i.e. of $\lfloor \lg 8 \rfloor + 1 = 4$ bits.
3. **Decimal vs binary** for $k = 8$ we have $8_{10} = 1000_2$.
4. w_d **string** for $k = 8$ is k_d in unary, where $k_d = \lfloor \lg k \rfloor = \lfloor \lg 8 \rfloor = 3$ in unary is 1110. That is $w_d = 1110$. Number of bits is $k_d + 1 = \lfloor \lg k \rfloor + 1 = 4$, i.e. the number of bits in the binary representation of $k = 8$.
5. w_r **string** for $k = 8$ is k_r in binary, where $k_r = k - 2^{k_d} = 8 - 2^3 = 0$ in binary with the appropriate number of bits. Note that the number of bits for w_r is $k_d = \lfloor \lg k \rfloor = \lfloor \lg 8 \rfloor = 3$. Thus we write 000 not 0!. Therefore $w_r = 000$.
6. **Elias- γ code** for $k = 8$ is $w_d w_r = 1110000$ consisting of $2k_d + 1 = 2\lfloor \lg k \rfloor + 1 = 2\lfloor \lg 8 \rfloor + 1 = 7$.

Example. Show that the Elias- γ code of $k = 6$ is 11010.

J5. Elias- δ . In the Elias- γ code of k , k_d is represented in unary, and unary encoding uses too many bits. Thus Elias- δ attempts to use Elias- γ encoding of k_d instead of unary, and leave k_r untouched by writing it in binary. The only tiny problem for all the details to work out smoothly is that **we need to Elias- γ encode not k_d but $k_d + 1$ instead**. We use the notation used for Elias- γ to describe the Elias- δ code for k .

(Alarm. The textbook on page 147 uses k_d instead of $k_d + 1$, and is in error.)

The Elias- δ (Elias-delta) code of natural integer k is represented by the concatenation of three strings w_1 , w_2 and w_r , defined as,

- w_1w_2 is the Elias- γ code of $(k_d + 1)$ (and NOT of k_d as stated on **page 147** of the textbook), where $k_d = \lfloor \lg k \rfloor$.
 - w_1 is k_{dd} in unary, where $k_{dd} = \lfloor \lg(k_d + 1) \rfloor$. The number of bits used for w_1 is thus $k_{dd} + 1 = \lfloor \lg(k_d + 1) \rfloor + 1$ which is also the number of bits to represent $k_d + 1$ in binary.
 - w_2 is k_{dr} in binary, where $k_{dr} = (k_d + 1) - 2^{k_{dd}}$ (**TEXTBOOK IS IN ERROR**). The number of bits used for w_2 is thus $k_{dd} = \lfloor \lg(k_d + 1) \rfloor$.
- w_r is k_r in binary, where $k_r = k - 2^{k_d} = k - 2^{\lfloor \lg k \rfloor}$. The number of bits used for w_r is thus $k_d = \lfloor \lg k \rfloor$. (This part is the same w_r used for the Elias- γ code of k ; that is why we used w_r instead of w_3 !)

Since $w_1w_2w_r$ becomes the Elias- δ code of k , the number of bits used in this representation is $(k_{dd} + 1) + k_{dd} + k_d$ which is

$\lfloor \lg(k_d + 1) \rfloor + 1 + \lfloor \lg(k_d + 1) \rfloor + \lfloor \lg k \rfloor$ which is

$2\lfloor \lg(k_d + 1) \rfloor + 1 + \lfloor \lg k \rfloor$, and since

$k_d = \lfloor \lg k \rfloor$, we have that the number of bits is

$2\lfloor \lg(\lfloor \lg k \rfloor + 1) \rfloor + 1 + \lfloor \lg k \rfloor$.

This is roughly $2 \lg \lg k + \lg k$, where $\lg \lg k$ is the logarithm of the logarithm of k i.e. $\lg \lg 16 = \lg 4 = 2$, and $\lg \lg 2^{16} = 4$.

Example: Find the Elias- δ code of $k = 7$.

We have already found the Elias- γ code for $k = 7$ to be 11011. With that we found that $k_d = \lfloor \lg 7 \rfloor = 2$, and $k_r = k - 2^{k_d} = 7 - 2^2 = 3$ and thus $w_r = 11$. We only need to find $w_1 w_2$ that relate to the Elias- γ of $k_d + 1$ which is $k_d + 1 = 2 + 1 = 3$.

1. What is w_1 ? Form $k_d + 1 = \lfloor \lg 7 \rfloor + 1 = 2 + 1 = 3$. The unary representation of $k_{dd} = \lfloor \lg (k_d + 1) \rfloor = \lfloor \lg 3 \rfloor = 1$, is 10. That is $w_1 = 10$, and the number of bits is the number of bits in the binary representation of $k_d + 1 = 3$ which is also 2 (since $3_{10} = 11_2$).

2. What is w_2 ? Form $k_{dr} = k_d + 1 - 2^{k_{dd}} = 2 + 1 - 2^1 = 1$. The binary representation of $k_{dr} = 1$ is 1, in $k_{dd} = \lfloor \lg (k_d + 1) \rfloor = \lfloor \lg 3 \rfloor = 1$ bits. Therefore $w_2 = 1$.

3. What is the Elias- δ code for $k = 7$? It is $w_1 w_2 w_r$ i.e. 10 1 11 which is 10111 without spaces. The total number of bits is $\lfloor \lg (k_d + 1) \rfloor + 1 + \lfloor \lg (k_d + 1) \rfloor + \lfloor \lg k \rfloor$ which is $(1 + 1) + (1) + (2) = 5$ (parentheses show the contribution of the corresponding string).

Example: Find the Elias- δ code of $k = 8$.

We have already found $w_r = 000$ from the Elias- γ code and also $k_d = 3$. Then $k_d + 1 = 4$.

1. $k_{dd} = \lfloor \lg 4 \rfloor = 2$. Thus w_1 is 2 in unary i.e. $w_1 = 110$.

2. $k_{dr} = k_d + 1 - 2^{k_{dd}} = 3 + 1 - 2^2 = 0$. Thus w_2 is 0 in binary using k_{dd} bits. Since $k_{dd} = 2$, we have $w_2 = 00$.

3. What is the Elias- δ code for $k = 8$. It is $w_1 w_2 w_r$ i.e. 110 00 000 i.e. 11000000.

Example. Show that the Elias- δ code of $k = 6$ is 10110.

Bit-aligned codes

Golomb- b codes.

J6. Golomb- b codes. The Golomb- b code of k is the concatenation of w_q and w_r , where q, r are the quotient and the remainder of the division of $k - 1$ by b . Consequently, $0 \leq r < b$. (If b is a power of two Golomb- b codes are known as Rice- b or Rice- 2^b codes.)

- w_q is the unary representation of $q = \lfloor \frac{k-1}{b} \rfloor$,
- w_r is the binary representation of $r = (k - 1) - q * b$ represented in $\lceil \lg b \rceil$ or $\lfloor \lg b \rfloor$ bits as specified below.
(If b is a power of two the ceiling and floor coincide and $\lg b$ bits are used.)

Let $C = \lfloor \lg b \rfloor$, and $B = \lceil \lg b \rceil$.

Case 0. If B is equal to C i.e. b is a power of two, then $B = C$ bits are used. Otherwise, one of Cases 1 or 2 applies.

Case 1. If r is small, i.e. $0 \leq r < 2^B - b$, then C bits are used.

Case 2. If $b > r \geq 2^B - b$, then $B = C + 1$ bits are used with the leftmost bit being 1. Then $r + 2^B - b$ is represented in binary (inclusive of the leftmost 1 bit).

The b in Golomb codes when used for gap encoding of doclists is determined by the corpus properties and the properties of a given doclist. Thus for a corpus of N documents, with p distinct terms and f representing the sum of the lengths of its doclists, a choice for b is $b \approx 0.69(Np)/f$.

If however the length of a doclist of a given term is l , then an alternative choice that is term (index-term) dependent is $b \approx N/l$.

Bit-aligned codes

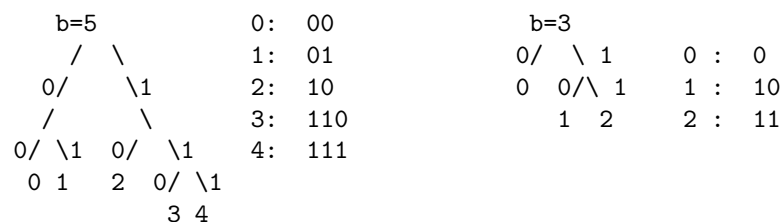
Golomb- b code examples.

Example for ($k = 8, b = 3$). Let $k = 8, b = 3$ and thus $q = 7/3 = 2$ and $r = 7 - 2 * 3 = 1$. Let $B = \lceil \lg b \rceil$. Then $B = 2$. Similarly $C = 1$. The unary representation of $q = 2$ is $w_q = 110$. But $r < 2^B - b$ is not true, since $r \geq 2^2 - 3 = 2^B - b$. Thus we represent r with B bits and its representation is that of $r + 2^B - b = 1 + 4 - 3 = 2$. The binary representation of $r = 1$ is thus $w_r = 10$. Thus the Golomb-3 code of $k = 8$ is 11010.

Example for ($k = 8, b = 4$). Let $k = 8, b = 4$ and thus $q = 7/4 = 1$ and $r = 7 - 1 * 4 = 3$. The unary representation of $q = 1$ is $w_q = 10$. The binary representation of $r = 3$ is $w_r = 11$, since $r - 2^{\lceil \lg b \rceil - 1} = 3 - 2^{2-1} = 1$. Thus the Golomb-4 code of $k = 8$ is 1011.

Example for ($k = 3$ and $b = 5$). We have $q = 0$ and $r = 2$. We have $r < 2^3 - 5$ and thus 010.

The remainders r for a given b can be retrieved by a Huffman tree-like consideration.



D1. What can we say about the distribution of document sizes in the Web? For an English text, or English-based texts, or in general documents expressing a spoken language several pieces of information are known in advance (a-priori). How can we approximate the distribution of size for Web-based document? How can for example the crawler anticipate the amount of space it will use for fetching information, or the indexer to plan ahead of time for the 1 billion Web-based documents that it plant to index? Web-based document size distribution changes between text and image documents and thus depends on the document type. In any case, the size of a document can be approximated by a Pareto distribution.

D2. Pareto distribution. Web-document size can be approximated to follow a **Pareto distribution** with probability density function

$$p(x) = \theta k^\theta / x^{1+\theta}, \text{ and thus } Pr(X \geq x) \approx \left(\frac{k}{x}\right)^\theta \text{ i.e. } Pr(X \leq x) \approx 1 - \left(\frac{k}{x}\right)^\theta.$$

where x is measured in bytes and $k, \theta > 0$ are parameters of the distribution. ($p(x)$ gives the probability that a document is of size x , and $Pr(X \geq x)$ is the probability that the size X of a document is at least x bytes long.)

Between the two parameters k and θ , parameter θ might change very slowly with time. On the other hand parameter k might grow significantly if say, video/audio files become more frequent and thus contribute to the increase of file size.

D2.1. Choosing θ . For text files, a $\theta = 1.36$ is used and smaller values can be used for image files and other binary formats. A typical generic choice for all file types is $\theta = 1.1$.

D2.2. Choosing k . The choice of k varies with time. For example $k = 9.3KB, 18.7KB, 130KB, 400KB$ is a best fit for 1998, 2003, 2009, and 2013 respectively.

D2.3. Example. For $k = 9.3KB$, and $\theta = 1.1$, if we use the expression for $Pr(X \geq x)$ we conclude that 93% of the documents have size no more that 9.3KB.

D2.4 Power Law. Note that the Pareto density function is of the form c/x^θ , for some constant c . Such distributions are said to follow the power law. The degree of a vertex of the Web graph also seems to follow a power law distribution with $\theta = 2.1 - 2.5$.

Document Properties

Word-based Entropy and Zipf's Law

D3. Entropy (word-based). One can express entropy and thus the information available by a text in terms of words rather than characters. Consider for example a document that consists not of N characters but we instead use as a metric number of words. Let it have N words. Let the number of distinct words in the document be n . As before let f_i be the frequency of the i -th word and let $p_i = f_i/N$. The (word-based) entropy measure for this document is $E = -\sum_{i=1}^n p_i \lg(p_i)$.

D4. Zipf's Law. One important law that applies to words appearing in documents is Zipf's law. It states that the i -th most frequent word has frequency proportional to its rank and thus $F_i = f/i$ if the rank of a word is i . Note that rank 1 is attained by the most frequent word. For a document with n distinct words appearing a total of N times ($N \geq n$), the least frequent word will have rank n (bottom of the list) if all ranks are distinct.

Definition 1 (Zipf's Law) *The i -th most frequent word in a text has a frequency F_i such that $F_i = f/i$, where f is the frequency of the most frequent word (since $F_1 = f/1 = f$). Moreover $p_i = F_i/N = (f/N)/i = A/i$, with $A \approx 0.1$.*

$$F_i \times i = f \quad \text{and} \quad p_i = F_i/N = A/i, \quad \text{where} \quad A = f/N$$

The series $H(n, 1) = H_n = 1/1 + 1/2 + 1/3 + \dots + 1/n$ is known as the harmonic series of order n . A more general form of this series is $H(n, \theta) = 1/1^\theta + 1/2^\theta + \dots + 1/n^\theta$. It is known that $H_n \approx \ln n + \gamma$, where γ is Euler's constant and $\gamma \approx 0.5772$.

Since the text has in total N words of which n are distinct, we conclude that $N = f/1 + f/2 + f/3 + f/4 + \dots + f/n = f(1 + 1/2 + 1/3 + \dots + 1/n)$. Using the properties of the harmonic series we conclude that $N = f/1 + f/2 + \dots + f/n = fH_n = f(\ln n + \gamma)$.

In other words, $A = f/N = 1/(\ln n + \gamma)$. For English texts where the number of distinct words after stemming is roughly 250,000, we expect A to be around 0.08 – 0.1.

D4.1 Note. A variation of Zipf's Law is that for example in a company, 10% of the customers generate 90% of the service requests. (Or similarly 10% of the students in a class ask 90% of the questions.)

D5.1 Question 1. *What is the rank r_m of a word that appears m times ?*

Answer to Question 1. A word that appears m times has frequency m , i.e. $F_t = m$, for some rank t . Thus using $F_t \times t = f$ and solving for t we get the answer.

$$\text{Rank of word appearing } m \text{ times : } \text{rank} = t = \frac{f}{m} = \frac{f}{N} \cdot \frac{N}{m} = \frac{AN}{m}.$$

D5.2 Question 2. *What is the "highest" rank, i.e. the rank of the lowest frequency word?*

Answer to Question 2. The lowest frequency word has frequency equal to 1. Thus $f_i = 1$. Then by way of $f_i \times i = f$ and $f_i = 1$, we have $i = f$. Thus **the highest rank of a word is f** .

D5.3 Question 3 (for information). *What is the "lowest" rank, i.e. the rank of the highest frequency word?*

Answer to Question 3. The lowest rank is 1!

In question 1 we identified the rank of a word that appears m times. There might be more than one words that appear m times. We pick the last one of them to determine the rank. Thus rank r_m is to include all words with ranks $1, \dots, r_m$ that appear in the corpus m or more times. Also rank r_{m+1} is equivalent to that all words with ranks $1, \dots, r_{m+1}$ appear $m+1$ or more in the corpus. Thus

D5.4 Question 4. *How many words appear exactly m times?*

Answer to Question 4. The number of words appearing m times is the number of words appearing at least m times minus the number of words appearing at least $m+1$ times. The former, by Question 1 is AN/m and the latter by Question 1 as well is $AN/(m+1)$ respectively. Thus the number of words appearing exactly m times is

$$AN/m - AN/(m+1) = \frac{AN}{m(m+1)} = \frac{f}{m(m+1)}.$$

D5.5 Question 5. *What is the fraction of words appearing m times?*

Answer to Question 5. By Question 4 the number of words appearing m times is $f/(m(m+1))$. Moreover by Question 2 the highest rank of a word is f . Thus the fraction of words appearing m times is $f/(m(m+1))$ divided by f i.e. $1/(m(m+1))$.

D5.6 Question 6. *How many words appearing once in the text? What is the fraction of words appearing once in a text?* This is known as **apax legomena** (Greek for **spoken once**).

Answer to Question 6. The number of words appearing exactly once is by Question 4 for $m = 1$ equal to $f/(1 \cdot (1+1))$ i.e. $f/2$. By Question 5 for $m = 1$ as well we obtain that one-half of all words appear only once!

D5.7 Question 7. *What is the fraction of words appearing once, twice, thrice, in a text?*

Answer to Question 7. Apply Question 5 for $m = 1, m = 2, m = 3$. The result is that 75% of the word appear no more than three times.

Conclusion 1. Most words have short Doclists.

Conclusion 2. Very few words have very long Doclists.

D6. Reformulations of Zipf's Law. A reformulation of Zipf's Law is to make it look closer to a Pareto distribution in the denominator. One could consider the previous formulation a simplified version of the standard formulation shown below if one substitutes 1 for θ below.

Definition 2 (Zipf's Law (standard)) *The i -th most frequent word in a text has a frequency F_i such that $F_i = f/i^\theta$, where f is the frequency of the most frequent word (and thus $F_1 = f/1 = f$).*

Then $N = f + f/2^\theta + \dots = fH(n, \theta)$, and thus $f = N/H(n, \theta)$ and equivalently the i -th ranked word has frequency $F_i = N/(H(n, \theta) \cdot i^\theta)$.

Let $H(n, \theta; q)$ be a variation of the Harmonic series H_n or $H(n, \theta)$ such that $H(n, \theta; q) = 1/(1+q)^\theta + 1/(2+q)^\theta + \dots + 1/(n+q)^\theta$. Then, we can obtain a Mandelbrot's distribution in which the term i^θ becomes $(i+q)^\theta$ instead.

Definition 3 (Mandelbrot's distribution.) *The i -th ranked word has frequency given by $F_i = \frac{f}{(i+q)^\theta} = \frac{N}{H(n, \theta; q) \cdot (i+q)^\theta}$.*

A negative binomial distribution can be used to express the fraction of the documents containing a word k times.

Definition 4 (Negative binomial distribution.) *The fraction of the documents containing a word k times is given by $F(k) = \binom{\theta+k-1}{k} p^k (1+p)^{-\theta-k}$, where $\theta = 0.42$ and $p = 9.24$.*

Document Properties

Heaps Law on vocabulary size

D7. Heaps's Law revisited (from Subject 3). The size of a vocabulary t can be inferred quite reliably from the size of the document N (we continue using the notation introduced earlier; N was W back in Subject 3). In order for the result to apply (even empirically), one must assume that **no typing errors occur** that could potentially bloat out the vocabulary size.

Definition 5 (Heaps' Law.) *In an English text with N words, the size t of its vocabulary (i.e. distinct words) is approximately given by $t \approx k \cdot N^\beta$. For English texts, $10 \leq k \leq 100$, and $0.4 \leq \beta \leq 0.6$. (Note that in Subject 3, N was denoted by a W .)*

Several times β is considered to be equal to 0.5 and with this value chosen, Heaps' Law takes the form $t \approx 10 \cdot \sqrt{N}$.

D7.1 Some text collections. A set of interesting statistics for the Bible and the TREC collection available at <http://trec.nist.gov/data.html> is the following table.

(Note that the terminology in the table below is different than the one we have been using in this section.)

Text Collection	Bible	TREC
No of docs : n	31k	741k
No of terms: F	884k	333M
No of disti terms: t	8k	535k
No index pointers: f	701k	134M
Size	4.3MB	2070MB

Document Properties

English texts

D7.2 English text characteristics. We give below a collection of observations that apply to English texts and values for β and θ as those used in Heaps' and Zipf's law. A rule to remember is that $\beta\theta \approx 1$.

Average English Word Length (All)	:	4.8 - 5.3 characters	--> w
Average English Word Length (exlude stopwords)	:	6.0 - 7.0 characters	
Average English Word Length (distinct words)	:	8.0 - 9.0 characters	--> W
Shortening of text due to stemming	:	5.0 - 6.0 characters, a 30% reduction	

D7.3 Heaps' Law parameters for various texts.

Text Collection	k	β	$1/\beta$	θ	w	W
AP89	62.95	0.45	2.2			
AP	26.80	0.46	2.1	1.87	6.3	8.0
DOE	10.80	0.52	1.9	1.70	6.4	8.4
WSJ	43.50	0.43	2.3	1.87	5.2	7.4

Other statistics that can be derived from the textbook (Chapter 4, page 78 and onwards) include the following. The most frequent words of the AP89 collection are **the**, **of**, **to**, **a**, and **and** with $p_i = f_i/N$ ranging from 6.4% to 2.32%. (Note that because these words are sorted by rank already, there is not difference in Table 4.2 of the textbook whether one uses f_i or F_i .) Although ip_i varies from 0.065 to 0.120, for the words with rank 11 through rank 50 the approximation $ip_i \approx 0.093$ applies consistently.

I. Floors and Ceilings.

A1. **The floor function** $\lfloor x \rfloor$: denotes the largest integer smaller than or equal to x .

A2. **The ceiling function** $\lceil x \rceil$: denotes the smallest integer greater than or equal to x .

Examples for floor and ceiling.

$$\lfloor 3.5 \rfloor = 3, \lfloor -3.5 \rfloor = -4, \text{ and } \lfloor 3.0 \rfloor = 3.$$

$$\lceil 3.5 \rceil = 4, \lceil -3.5 \rceil = -3, \text{ and } \lceil 3.0 \rceil = 3.$$

II. Exponentials.

Let a, m, n be real numbers such that $a \neq 0$.

A3. $a^0 = 1, \quad a^1 = a, \quad a^{-1} = 1/a.$

A4. $a^m \cdot a^n = a^{n+m}, \quad a^m / a^n = a^{m-n}.$

A5. $(a^m)^n = (a^n)^m = a^{(mn)}.$

III. Base-two Logarithms.

A6. The base-2 logarithm of x , denoted by $\lg x$ (or sometimes $\log_2 x$), is the real number y such that $2^y = x$.

A7. $\lg^k n = (\lg n)^k.$

A8. $\lg \lg n = \lg (\lg n).$

A9. For all $a > 0, b > 0, c > 0$ and n we have that

A9a. $a = 2^{\lg a},$

A9b. $\lg(ab) = \lg a + \lg b,$

A9c. $\lg(a/b) = \lg a - \lg b,$

A9d. $\lg a^n = n \lg a,$

A9f. $\ln a = \frac{\lg a}{\lg e},$ where $\ln a$ is the natural logarithm of a base e .

Exercise 1 *Give unary, Elias- γ , Elias- δ , and Golomb-3 representations for (decimal) integer 12.*

Exercise 2 *How many words account for 50% of a text?*

Exercise 3 *What is the frequency of 90% of the words of a text?*