Hashing, Data Compression, and Coding

Review data-structures and algorithms (pages 2-11,18-27)

Chapter 3 (3.5,3.6,3.7), Chapter 5 (5.4)

Introduction: Efficient (or not) key search

Index-based applications require the use of an **dictionary data structure** that supports operations **Insert, Search and Delete**. Such operations can convert a `wordID` into a `word` or the "fingerprint" or "checksum" `fword` of a `word` into a `wordID`. Or they are related to the conversion between a `URL` and its `docID` or the "fingerprint" `fURL` of the `URL` itself and its `docID`.

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 linear time $O(n)$ in the worst-case, where n is the number of keys maintained (keys can be words or URLs or other index-related entities). This is too much. The use of an ordered table however, allows the utilization of BinarySearch that can expedite the search operation (but not Insert and Delete) to $O(\lg n)$. In fact Google 1998 uses in its `DOCIndex` binary search on a `DOCIndex` table sorted by `fURL` to determine the `docID` of a given URL. But for most uses, this use of an ordered array is too slow. Updating this table (i.e. inserting or deleting keys) take linear $O(n)$ time that makes sense only if it is done in "batch mode" where several records are deleted or inserted at once, not one at a time!

Other alternatives 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) 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 to maintain information about documents: they are indexed by `docID` and through that key (`wordID`), information about the URL and also its position in the Repository can also be retrieved.

A2. Direct Addressing. 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 $U = \{0, \ldots u-1\}$ 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 x are URLs of length 50, the value of u can grow too large to $2^{63 \cdot 50}$ (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 $key(x) = key(a.b.c.d)$ is easy to compute of $a \cdot 2^{24} + b \cdot 2^{16} + c \cdot 2^{8} + d$, it is an integer between 0 and $2^{32} - 1$ and a table of size $u = 4GiB$ might work miracles with direct addressing. However this scheme breaks down for an IPv6 address that consists of 16 bytes, not just 4 and thus require a table with $u = 2^{128}$ size, an impossible number.

Introduction

The introduction of **hash tables** allows faster dictionary operations ON THE AVERAGE but NOT in the worst-case; the worst case performance of a hash table is that of a linked list! However the average case performance under realistic assumptions is **expected constant time** $O(1)$. 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 constitutes an integral part of languages that deal with manipulation of string structures (e.g. Perl, Python).

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), direct addressing is not an option as it wastes space. An alternative to direct addressing is **hashing**. A hash table allocates space **proportional to the (maximum) number of keys that we plan to store in the hash table** NOT TO THE TOTAL NUMBER OF POSSIBLE KEY VALUES. Thus if we only plan to maintain a million IPv6 addresses we need a table may of size approximately one million (in fact for reasons that will be made clear later, twice as much, i.e. two million is a reasonable estimate for a hash table size to maintain 1,000,000 keys).

```
How many bits do we need to represent 2**2   = 4?        3   bits (100)
How many bits do we need to represent 2**3-1 = 7?        3   bits (111)
How many bits do we need to represent 2**2 .. 2**3 -1 ?  3   bits
How many bits do we need to represent 0    .. 2**3 -1 ?  3   bits maximum
How many bits do we need to represent 0    .. 2**n -1 ?  n   bits maximum
How many bits do we need to represent 0    .. 2**lgn-1?  lgn bits maximum
How many bits do we need to represent n               ?  FLOOR(lgn)+1 = 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      = 8 + 0 + 2 +1  = 11
```

Fingerprinting functions: CRC-32, MD5, SHA256, etc. Although function `key(x)` was easily available for converting IPv4 address $x = a.b.c.d$ into an integer value, if x is a string of characters (ASCII or UNICODE values) something more elaborate is to be used. Such a `key` function is then known as a "fingerprint" function and can be an checksum such as CRC-32 (4B long) or SHA-1 (20B) or the SHA-2 variants (eg. SHA256, SHA512) or MD5 (16B). An arbitrary string (eg a URL and rarely a long chemistry or biology-related word) of 50 or so characters (50B for ASCII or 100B for UNICODE) thus gives a fingerprint of 16B-32B. If the x is a `word` or a `URL` the result $key(x)$, i.e. the fingerprint of x is denoted by `fword` or `fURL` respectively. $key(x)$ might also use something simple (and a very poor choice of a fingerprint): thus string ABZ might get converted into $65 + 66 + 90 = 221$, the sum of the ASCII values of A,B,Z.

B1. Collisions and Fingerprints and Hash Tables. No matter what the choice of $key(x)$ the end result is that a Universe of u values gets mapped into a Universe of fewer m values where $m \ll u$. Thus the 2^{128} IPv6 addresses get mapped into a table of 2,000,000 entries i.e. $m = 2,000,000$. **Collisions** then are unavoidable: that is there could be two keys $x_1 \neq x_2$ $key(x_1) = key(x_2)$ even if $x_1 \neq x_2$. Let us call $k_1 = key(x_1)$ and $k_2 = key(x_2)$.

Several times the use of say SHA256 might be uneconomical: 256 bits or 32B for a universe of $u = 2^{256}$ possible finger print values is still too much. We then need apply one (secondary) more `hash function` **h** that maps k_1, k_2 into a hash index, and then use this hash index as table index into a table of $m = 2,000,000$ entries! Thus even if 256-bit SHA256 fingerprints k_1 and k_2 are different from each other it is still possible that $h(k_1) = h(k_2)$ indeed! Collisions are unavoidable (automotive or key-based).

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

B2. 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 $0, \ldots, m-1$.

3. Load factor is the ratio $a = n/m$. As we mentioned before several times (but now always) m should be chosen $m \approx 2n$.

B3.1. Collision Resolution Protocol: Chains (aka linked lists). One scheme in resolving collisions is by **chaining** all keys k_1, k_2 that get hashed into the same slot $h(k_1) = h(k_2)$ in a (doubly) linked list attached to 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).

B3.2. Collision Resolution Protocol: Open-Addressing. In **open-addressing** we always have $n \leq m$ and we store the keys into the hash table itself. (By keys with might mean the `wordID`s but not necessarily the words themselves.) A slot contains either a key or a flag (unused key value) that indicates that the slot is available. The Load factor $a = n/m \leq 1$ is thus at most 1 for open-addressing.

```
   0   1   2   3   4   5   6   7   8   9  10
---------------------------------------------   h'(k)=k%11
|   |   |   |   |   |   |   |   |   |   |   |   h(k,i)= (h'(k) + i) % 11
|   |   |   |   |   |  5|  6| 16|   |   |   |
---------------------------------------------   Figure 1.
```

In order to insert k we **repeatedly probe** the hash table until we can find an empty slot in which to put the key. Instead of probing slots in a predetermined order $0, 1, \ldots, m-1$, the **probing sequence** depends on k. We thus extend hash function $h(k)$, to include a probe index/number starting from 0

$$h : U \times \{0, \ldots m-1\} \rightarrow \{0, \ldots m-1\}$$

thus defining for every key k a probe sequence,

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

that is required to be a permutation of $0, \ldots, m-1$, or there may be some slots that may be missed during the probing.

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 full, then we determine that the hash table is full and this causes an overflow.

B3.2.1 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 $a = 0.5$ i.e. the hash table is half-full, on the average two probe operations are required for insertion or to determine a key is not there (in the hash table). If $a = 0.9$ i.e. the table is 90% full, this becomes close to 10 or 11 instead. Deletion and Successful search are a bit faster. Having said that worst-case performance is $O(m)$ as in the worst-case we might probe all the m slots of the table!

B4. What do we store in the Hash table? In all the previous examples we never stored into the hash table the key x, i.e. we never store directly a `URL` or a `word`. This is because a `URL, word` have arbitrary lengths. In a hash-table we prefer to stored fixed-length information. A `fURL, fword, docID, wordID` in the form of k is thus stored instead! Next to it there might be a room for an index (or pointer) p. Thus index or pointer p might point to the memory location storing the variable-length URL or word. The first few bytes (usually 4B) of that location might store the length of the URL or word. Alternatively we can pack all arbitrarily long URLs, words into a very long array that is separated with a special symbol i.e. NULL value whose ASCII code or UNICODE code is a zero (1B in the former, 2B in the latter case).

If instead of storing k `fURL, fword, docID, wordID` in the hash table we decide to store x itself, we would need to allocate space in a hash table wide enough to accommodate the longest word or URL i.e. the longest x! This is not very efficient as a single slot of a hash table would need to be 200B to 400B long for a URL and most of the space of a slot then and most of the time would get wasted!

B5. Google's DocIndex: docID and k coexisting. With respect to Google 1998 (Brin and Page, 1997/1998 paper), one of the tables in DocIndex stores URLs (i.e. x) and docIDs (i.e. k) for a particular document and a variety of data associated with it such as the status of the document, a pointer to the Google repository holding the document (the "cached" copy of the Web document), the document checksum (`fURL` or not) established by Google's internal checksum function (which can be one of the previously listed functions) and different from the URL checksum (most likely `fURL`), statistics about the document, the URL itself and its title (if any). This table is in fact a B-tree.

Another table related to DocIndex stores a pair of docID (k) and URL checksum (`fURL`) and allows through binary search on the URL's checksum to determine the docID of a given URL. This is possible because this table is sorted by URL checksum. From time to time this table is merged with a table of (new) URL checksums. Some of them have no associated docID available in the table, and thus the URL associated with those URL checksums get assigned a new docID. This is a sorted table, each slot of which can accommodate a URL checksum plus a `docID`.

B6. Average length of a word in a dictionary/lexicon/vocabulary. 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 7.36).

B7. Average length of a word in a corpus/text. The average length of a word in a text is somewhat smaller at 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, etc.

B8. 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 1997/1998.

Case study: Three organizations for `word` or `wordID`s in a lexicon

One application of a hash table is to maintain a vocabulary (or lexicon). The lexicon (vocabulary) may be implemented by a more elaborate version of an open-addressing scheme. The hash table itself (similar to the table used for open-addressing) might be a hash table of pointers. The strings themselves are stored into a separate contiguous memory separated by NULL (`'\0'`) characters. For efficiency purposes vocabularies used in say Web-searching are memory resident.

Case T1. Suppose we would like to maintain a table of n words using a hash table. Collision resolution by open addressing will be used. For efficiency reasons we use $m = 2n$, i.e. the table should be at least twice as large as the number of words that will be maintained in it. We choose to store x itself i.e. `word` inside 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. If ASCII encoding gets used the number of charactes is also 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 1997 configurations, then one could barely accommodate $n = 512/64 = 8$M words with such a scheme.)

Case T2a. Scheme T2a is a minor variation of **T1**; it maintains inside the hash table not the string itself, but only some-form-of-a-pointer to strings that are (represent) `word`s. The bitlength of a `pointer` can be 32-bits i.e. 4B. However if an `index` to a table is used instead, we can get away with few extra bits and can thus use 24bits, or 16bits or fewer than the 32 bits of a pointer. The words themselves are stored in an array A of characters where words are separated by NULL characters i.e. \0. For hash table T this scheme used $4m = 4 \cdot 2n = 8nB$ 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 assumptions). Adding the NULL at the end, this becomes an 8. The total space used for T and A is thus $8n + 8n = 16nB$, a fraction (one quarter) of the space used previously. This 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.

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 `wordID`s can consume $4m = 8n$ more bytes. If the previous scheme used $16n$B of space this one uses $24n$B of space. Roughly $n = 512/24 \approx 21$M words can be maintained.

Duplicate Detection

In computing the terms "hash function", "hash" (the result of the application of a fingerprinting function), "checksum", or "fingerprint" might refer to the same thing in a variety of contexts. Thus converting x into a k involved a checksum (CRC-32) or a "hash" (MD5, SHA-1, SHA-2), and converting k into $h(k)$ the application of a hash function. k is a checksum, or hash of x, and so is $h(k)$ of k. They are also fingerprints of x and k respectively.
B9. Fingerprints. Hashing can be used for fingerprinting to detect and eliminate duplicates.
B10. Duplicate detection. 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. A variation of this applies to URLs. 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 small 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 $a_1, b_1, c_1, \ldots$ for document k_1 and $a_2, b_2, c_2, \ldots$ for another document k_2 and generate hashes $h(a_1), h(b_1), \ldots$ and $h(a_2), h(b_2), \ldots$ 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.

Duplicate Detection: Simhash

The structure of the document will determine what constitutes a part. In the previous approach of fingerprinting text documents were viewed as a collection of characters: 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 $V = (v_1, \ldots, v_b)$ initialized to zero. Also create a b-word 0-1 vector $H = (H_1, \ldots, H_b)$ 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
```

C1.1 Why compression? Repository maintenance Google in 1997 required close to 150GB to store the corpus (its approximately 24,000,000 documents) that it indexed. With the use of compression this could go down to one third to one fourth of that size. There are tradeoffs among various compression methods such as the time it takes to compress/decompress a document vs compressed volume size. Google for example prefers the former (zlib/gzip) over the latter (bzip), and because of this the compressed volume was approximately 50GB, whereas using bzip a smaller size could have been achieved. 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 concept of compression by introducing some concepts and discussing a primitive compression algorithm that is known as Huffman coding.

C1.2 Why compression? Index compression The index for a given word is a long list (array) of `docIDs` that describe all the documents containing the given word. If the list contains no other information (eg word offset, context) it is sometimes called **doclist**. If a word appears in a document multiple times there might be multiple consecutive instances of the same `docID`. More often than not the list is sorted by `docID`. Thus a very common word might appear in all or almost all of the documents of the collection. Thus its doclist would be quite long; moreover successive docIDs are going to be consecutive or differ by a very small number in a sorted doclist. Thus a docID 23456780 might be followed by a docID quite close to 23456780 (as in 23456781, 23456782, ... , 23556790). Thus even if docIDs are large numbers the **gaps** between two consecutive docIDs are small numbers. Thus index compression involves compressing the gaps between consecutive docIDs. We discuss algorithms such as **v-byte encoding**, **Elias**$-\gamma$, **Elias**$-\delta$, Golomb-b in this context. Other parts of the index might also get compressed by these or other methods.

C2. Text compression and decompression. In the remainder we deal with the problem of text compression. In text compression we are given a string T of characters drawn from some alphabet Σ and we want to efficiently encode string T into a smaller (binary) string C. The way **compression** is performed is by first assigning (i.e. encoding) a distinct bit sequence (called the code) to each element (character) of Σ and then converting the characters of T using this character encoding into C. Moreover we should be able from C and the information available about Σ (i.e. the encoding of its characters) in C to recover T during the **decompression** of the text.

The process that converts T into C is called **encoding or compression**; **decoding or decompression** is the inverse process.

Codes: Bit-aligned, Byte-aligned, Prefix and Lossy vs Lossless compression

C2.1. Codes. The bit sequence assigned to a given character is called the **code** of that character. Thus the code for `A` in ASCII is 65 in decimal, or 41 (0x41) in hexadecimal, or 101 (0o101) in octal. In UNICODE this would be in hexadecimal `0x0041` i.e. two bytes one of which is zero and the other the ASCII value of the character. Since n bits have 2^n distinct values, an alphabet with $\mid \Sigma \mid$ characters requires $\lg \mid \Sigma \mid$ bits.

If each character is equally likely to appear in a document (or string) not much can be done. In real-life however this is not the case. 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. In a **bit-aligned** code the same byte can be used to encode one or more or part of one or more characters.

C2.2 Byte-aligned codes. These codes respect end of byte boundaries and use a fixed amount of bytes per code. Thus ASCII or UNICODE character codes are byte-aligned codes; in ASCII a character is assigned one byte, but in UNICODE at least two (and currently, only two) bytes.

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

C2.3 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`!

C2.4 Prefix codes. Thus for bit-aligned codes it is imperative that no code is the **prefix** of another code, and in our 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)!

C2.5 Lossless vs Lossy compression. Data compression techniques are **lossless** or `noisy/lossy`. For a lossless technique, and compressed text C, we can fully recover the original uncompressed T from C. There are also **noisy or lossy** techniques where full-recovery of T from C is not possible. 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. Entropy. The amount of information in a document is known as its **entropy**. Thus for a document whose n different characters have frequencies $f_1, f_2, \ldots$ of total length N, let us define the fractions $p_i = f_i/N$ which might correspond to the a-priori probability of occurrence in a document of the corresponding character. Then the entropy of the document is $-\sum_{i=1}^{n} p_i \lg(p_i)$, where n is the number of different characters and $\sum_i f_i = N$ is the length of the document in characters.

C3.1 Example (a): Binary documents. Consider a document with N zeroes and ones (the only two characters). If the number of zeroes and ones is the same $N/2$, then the entropy of the document is $-p_0 \lg p_0 - p_1 \lg p_1$, where $p_0 = p_1 = (N/2)/N = 1/2$, and thus $-1/2(-1) - 1/2(-1) = 1$. 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!

C3.2 Example (b). 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 $\lg n$.

C3.3 Example (c). 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 $1/2, 1/4, 1/8, 1/8$. We could use
$-\lg 1/2, -\lg 1/4, -\lg 1/8, -\lg 1/8$ bits for each one of the characters i.e. 1,2,3, and 3, to get an entropy 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 per character. Thus in a document with N characters (and frequencies as specified) we can encode it using only $1.75N$ bits altogether. This is much less than standard ASCII coding that uses 7 (or 8) bits per character for a total of $7N$ (or $8N$).

Huffman codes

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 text. 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 A greedy algorithm. Huffman's algorithm for encoding and decoding a text is a **greedy algorithm** that during its course of execution makes a decision/choice that is the best possible decision at the time (that's the greedy principle in algorithm design).

H1.2 Huffman coding performance. Nowadays it is not very often that anyone is using Huffman compresison (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.

H1.3 Huffman coding variants. However instead of doing character compression/encoding we can do word compression/encoding, Huffman can exhibit substantiallhy 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.

Huffman coding

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 frequenciess 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)                                     Text is AACCAGTA
1. Read file, find its characters, compute frequency of each character.                  A: 4  C: 2  G: 1 T:1
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);
```

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);
```

```
               -:8          STEP3  : combine A with
               /\  1                 (C,(G,T)) tree
              /  \
           0 /   -:4        STEP2  : combine C with
            /  / \    1              (G,T)-tree
           /   0/    \
          /    /       -:2   STEP1 : combine G, T
         /    /     0/  \ 1
         A:4  C:2  G:1 T:1
codes :  0   10   110  111
```

```
   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...
```

Text Compression
Arithmetic Coding, Dictionary-based methods

H6. Statistical 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.

H7. Compression Ratio. The **compression ratio** is defined to be the ratio of the size of the compressed file over the size of the corresponding uncompressed file.

H8.1 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).

H8.2 Arithmetic coding computes the codes incrementally, one symbol at a time, and decoding from the middle of a compressed file requires to start things from the beginning. 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.

H8.3 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.

H9. Compression models. Compression models assign probabilities to the next symbol to be coded. 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.

Occurrence lists: Gap or delta encoding

I1. (Inverted) Index: A 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: for a given `wordID` its **doclist** records the docID of all the documents containing that wordID. More often than not, this list is sorted by docID. The number of `doclist`s is the number of `wordID`s of the corpus.

I2. (Inverted) Index: An occurrence list. More often that not this `doclist` is called an **occurrence list**; an occurrence list containing offset information is sometimes called Hits (and rarely, a HitList) or Positions. More than one entry for a given docID may be included in that list. Each element of the occurrence list of a given wordID can be viewed as a triplet
$\langle \mathbf{docID}, \mathbf{offset}, \mathbf{context} \rangle$
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). Occurrence lists are sorted first by `docID` and for the same `docID` by `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.

I3. Compressing doclists or occurrence lists. However for several `wordID`s the corresponding doclists are short. The size of a `doclist` can be reduced significantly if specialized compression methods are used based on the principle that

" small numbers are more likely to occur than large numbers",

where large vs small refers not to the value of a docIDs (for example) or the number of bits needed to write it down (or store it) but to differences (in value) of two consecutive docIDs in a sorted list of of docIDs. The difference between consecutive docIDs is known as a **gap** or **delta** (for difference/gap).

Observation 1. Frequent words have small/short gaps.

Observation 2. Infrequent words have longer/larger gaps. But infrequent words have short doclists as well! (And thus they do not need to be compressed!)

Gap encoding, Delta encoding or $\delta-$encoding. Sometimes this encoding of gaps of docIDs is also called **gap encoding** or **delta encoding** or δ **encoding**.

Occurrence lists: Gap or delta encoding

I4. Example: Doclist. Consider for example a doclist consisting of $[1000, 1002, 1003, 1004, 1006]$.
Each one of the docIDs can use 4 or 8 bytes for storage (32-bit or 64-bit representation of a docID). Thus this doclists would use $5 \cdot 8 = 40B$ bytes or 20B depending on the representation. However, beyond the baseline (first) docID that will be stored using that number of bits (i.e. `1000`), for the remaining docIDs we can only use docID differences (**gaps**) instead. Thus a way to store the doclist above is to store it in the form $[1000, 2, 1, 1, 2]$.
The **gaps** are sometimes referred to as the `delta`s. Given that the difference between consecutive docIDs is a small number, we can use a single byte (or less) to represent gaps. For this particular example, 8B can still be used for 1000, but 1B for the remaining four gaps for a total of 12B. Compare this to the 40B original storage requirements. For a 32-bit docID the original requires 20B and the latter gap-encoding only 8B providing 60% savings at a minimum.

I5. Example: Occurrence List. Suppose we isolate the inverted index of a single wordID appearing in only two documents: in one it appears thrice at offsets, 2,3,5 and in the other twice at offsets 5 and 7. The doclist is of length 2 but the ocurrence list is of length 5. Elementary compression can be performed if all entries for the same `wordID` and `docID` are grouped together into **Hits** recording in addition its length i.e. `Nhits`.

Furthermore we can gap-encode the `docID`s. The base `docID` is not encoded. Thus 10000 remains as it. The follow-up docID is gap-encode relative to the previous one. The difference between 10001 and 10000 is recorded. Nhits is not compressed. The basse hit entry of Hits is not encoded however, hit 3 is encoded as $3-2$ and 5 as $5-3$. Likewise 7 is encoded as $7-5$.

For the "compressed hitlist" [2 1 2] we can decompress it by generating all prefixes of it as in $2, 2+1, 2+1+2$ which give $2, 3, 5$, as needed. Likewise for [52] and also [100001]

```
  Say wordID=2 (for data) appears in two documents with docID 10000 and 10001
  The forward index is
(10000, 2, 2)                                 doclist for wordID=2 is length=2:  10000 ---> 10001
(10000, 2, 3)
(10000, 2, 5)       inversion --> same  --->      occurrence list          docID Nhits Hits      docID Nhits  Hits
(10001, 2, 5)                                              for wordID=2 is  10000,3,    [2 3 5] , 10001, 2,    [5 7]
(10001, 2, 7)


                                              Compress occurence list as follows
                                                                  10000,3,    [2 1 2]    1,     2,   [5 2]
                                                              or   10000,3,2,1,2,1,2,5,2
```

I6. Recovery is easy: It involves a prefix computation.

```
     x1 x2 x3 x4 ....  xn
x1                     =  S1
x1 + x2                =  S2 = S1 + x2
x1 + x2 + x3           =  S3 = S2 + x3
x1 + x2 + x3 + x4      =  S4 = S3 + x4
...
x1 + x2 + x3 + ... + xn  =  Sn = Sn-1 + xn
```

For a sequence of numbers $x_1, x_2, \ldots, x_n$ the prefix sum computation involves the computation of $x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots, x_1 + x_2 + x_3 + \ldots + x_i, \ldots, x_1 + \ldots + x_n$. The i-th prefix sum can be computed from the $(i-1)$-st prefix sum by adding to it x_i.

Thus, from `1000, 2, 1, 1, 2` we can compute the prefix sums $1000, 1000+2, 1000+2+1, 1000+2+1+1, 1000+2+1+1+2$ to recover the docIDs of the corresponding doclist.

I7. 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.) A parallel prefix sum computation is also known as a `scan, prefix-sum` and can be done for any associative operator not just sum (eg multiplication, MIN, MAX, etc).

I8.1 Note (Associativity). For an operator + and operands, a, b, c, the operator is associative if and only if $(a + b) + c = a + (b + c)$ for all a, b, c.

I8.2 Note (Commutativity). For an operator + and operands, a, b, the operator is commutative if and only if $(a + b = b + a$ for all a, b.

Byte-aligned: v-byte encoding

I8. Variable-byte length encoding using byte-aligned codes. The example in I4 and I5 was intentionally simple and fitting. All the gaps 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. 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 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 `AAAAAAA` 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 `BBBBBBBCCCCCCC` 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
```

Practical Issues: v-byte encoding

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 1997 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 `(docID,Nhits,[positions/offsets]) = (docID,Nhits,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 `81` is `10000001` in binary).

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

Elias and Golomb codes

For the remainder $\lg k$ is the binary (or base-two) logarithm of k, and $\lceil x \rceil$ is the ceiling function and $\lfloor x \rfloor$ is the floor function (see the penultimate page, ie. around page 30, of this Subject for definitions).

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, but does not work! The **optimal bit-aligned code** for a (non-negative) integer k is its binary representation! It cannot be used for gap encoding however, because it 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.

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 tries to reach the performance of the "optimal" binary code, using unary codes.

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 is a bootstrapping of Elias-γ codes, where the use of unary codes is replaced by (recursively) using Elias-γ 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 lefmost 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 `111111110`, 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`.

Elias-δ.

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,

- $w_1 w_2$ is the Elias-γ code of (k_d+1) (and NOT of k_d as stated on **page 147** ofthe 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 $w_1 w_2 w_r$ 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 $\lg \lg 2^{16} = 4$.

Examples for Elias-δ encoding.

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`.

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 $b \approx N/l$.

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 $r - 2^{\lfloor \lg b \rfloor - 1} = 3 - 2^{2-1} = 1$. 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.

```
   b=5           0:  00              b=3
    /  \         1:  01             0/  \ 1     0 :  0
  0/    \1       2:  10             0  0/\ 1    1 :  10
  /      \       3:  110               1  2     2 :  11
0/ \1  0/  \1    4:  111
 0 1   2  0/ \1
           3 4
```

Refresh
Logs and Exponentials

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.
$\lfloor 3.5 \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, and $\lfloor 3.0 \rfloor = 3$.
$\lceil 3.5 \rceil = 4$, $\lceil -3.5 \rceil = -3$, and $\lceil 3.0 \rceil = 3$.

II. Exponentials.

Let a, m, n be real numbers such that $a \neq 0$.
A3. $a^0 = 1, \qquad a^1 = a, \qquad a^{-1} = 1/a.$
A4. $a^m \cdot a^n = a^{n+m}, \qquad a^m/a^n = a^{m-n}.$
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.

Exercises
Practice makes perfect

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