11
Codes

11.1 Representing the Data

You have reached the last chapter and it deals with *codes*, which share some common properties with data structures as auxiliary tools in many algorithms and programs. Studying these codes will also give an opportunity to conclude this work by reviewing some of the data structures introduced in previous chapters.

The purpose of codes is to bridge the communication gap between humans and machines. Our civilization has generated over the years some quite sophisticated natural languages, yet our computers insist on talking only binary, forcing us to translate whatever information we wish to share with a computer, be it a command or a piece of data, into some binary equivalent. This translation is often called an *encoding*, the translated elements are *code words* and their set form a *code*.

In the simplest scenario, the elements to be encoded are just the letters of some alphabet, like {a, b, c, . . . , y, z}, but one may need to encode also infinite sets as the integers {0, 1, 2, . . .}, or letter pairs {aa, ab, . . .}, or sets of words {the, of, which, . . .}. Ultimately, the set $\mathcal{A}$ to be encoded may be of any nature, as long as there is a well defined way to break a given file into a sequence of elements of $\mathcal{A}$. We shall refer to $\mathcal{A}$ as an *alphabet* and call its elements *letters* or *characters*, even in the more involved cases, so these terms should not be understood in their restrictive sense.

The choice of a code will be guided by the intended application and expected properties. In many situations, it will be the simplicity of the processing of the code that will be considered as its main advantage, leading to the usage of some standard *fixed length* code, for which all the code words consist of the same number of bits. One of the popular such codes is the American Standard

Code for Information Interchange (ASCII), for which each code word is eight bits long, providing for the encoding of $2^8 = 256$ different elements.

The encoding and decoding processes for fixed length codes are straightforward: to encode, just concatenate the code words corresponding to the letters of the message, to decode, break the encoded string into blocks of the given size, and then use a decoding table to translate the code words back into the letters they represent. For example, the ASCII representation of the word `ascii` is

$$01100001011100110110001101101001011010 01,$$

which can be broken into

$$01100001 \,|\, 01110011 \,|\, 01100011 \,|\, 01101001 \,|\, 01101001.$$

However, it is not always possible to use a fixed length encoding, for example, when the set $\mathcal{A}$ of elements is potentially infinite, and even when it is possible, it may be wasteful, for example when we aim for storage savings via data compression.

Of the many possible applications, we shall briefly address only on the following topics.

(i) Compression codes: trying to reduce the number of necessary bits to encode the data without losing a single bit of information
(ii) Universal codes: efficiently encoding the integers or other infinite alphabets
(iii) Error-correcting codes: recovering the original data even in the presence of erroneous bits
(iv) Cryptographic codes: dealing with the possibility of two parties to exchange messages, while hiding their contents from others, even if the encoded messages are accessible to all

This list is far from covering all the possible applications of codes, and even for those mentioned, it is just the tip of an iceberg: each of the preceding topics is on its own the subject of many books and entire courses.

11.2 Compression Codes

The frequencies of the characters in a typical text written in some natural language are not uniformly distributed, as can be seen in Table 11.1, showing the probabilities, in percent, of some of the characters for English, French and German. The order from left to right is by decreasing frequency in English.

Table 11.1. *Distribution of characters in natural languages*

	E	T	A	O	···	J	X	Q	Z
English	12.7	9.1	8.2	7.5		0.2	0.2	0.1	0.1
French	14.7	7.2	7.6	5.4		0.5	0.4	1.4	0.1
German	17.4	6.2	6.5	2.5		0.3	0.03	0.02	1.1

It therefore does not seem economical to allot the same number of bits to each of the characters. If one is willing to trade the convenience of working with a fixed length code for getting a representation that is more space efficient but harder to process, one could use *variable length* codes. One can then assign shorter code words to the more frequent characters, even at the price of encoding the rare characters by longer strings, as long as the *average* code word length is reduced. Encoding is just as simple as with fixed length codes and still consists in concatenating the code word strings. There are however a few technical problems concerning the decoding that have to be dealt with.

Not every set of binary strings can be considered as a useful code. Consider, for example, the five code words in column (a) of Figure 11.1. A string of 0s is easily recognized as a sequence of As, and the string 11001100 can only be parsed as BABA. However, the string 01001110 has two possible interpretations:

$$0 \,\big|\, 1001 \,\big|\, 110 = \mathsf{ADB} \qquad \text{or} \qquad 010 \,\big|\, 0 \,\big|\, 1110 = \mathsf{CAE}.$$

The existence of such a string disqualifies the given code, because it violates a basic property without which the code is useless, namely that the encoding should be reversible. We shall thus restrict attention to codes for which *every* binary string obtained by concatenating code words can be parsed only into the original sequence of code words. Such codes are called *uniquely decipherable* (UD).

A 0	A 11	A 11	A 1
B 110	B 110	B 011	B 00
C 010	C 1100	C 0011	C 011
D 1001	D 1101	D 1011	D 0101
E 1110	E 11000	E 00011	E 0100
	UD	prefix	complete
Non-UD	non-prefix	non-complete	
(a)	(b)	(c)	(d)

Figure 11.1. Examples of codes.

There are efficient algorithms to check the unique decipherability of a given code, even though infinitely many potential concatenations have to be considered a priori. A necessary condition for a code to be UD is that its code words should not be too short, and more precisely, any binary UD code with code word lengths $\{\ell_1, \ldots, \ell_n\}$ satisfies

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1. \tag{11.1}$$

For example, the sums for the codes (a) to (d) of Figure 11.1 are 0.875, 0.53125, 0.53125 and 1, respectively. Case (a) is also an example showing that the condition in eq. (11.1) is not sufficient for a code to be UD.

On the other hand, it is not always obvious how to decode, even if the given code is UD. The code in column (b) of Figure 11.1 is UD, but consider the encoded string 11011111110: a first attempt to parse it as

$$110 \,\big|\, 11 \,\big|\, 11 \,\big|\, 11 \,\big|\, 10 = \mathsf{BAAA10} \tag{11.2}$$

would fail, because the tail 10 is not the prefix of any code word; hence only when trying to decode the fifth code word do we realize that the first one is not correct, and that the parsing should rather be

$$1101 \,\big|\, 11 \,\big|\, 11 \,\big|\, 110 = \mathsf{DAAB}. \tag{11.3}$$

11.2.1 Prefix Codes

Unlike in the example of eq. (11.3), we should be able to recognize a code word as soon as all its bits are processed, that is, without any delay; such codes are called *instantaneous*. Instantaneous codes have the *prefix property*, and are hence also called *prefix codes*. A code is said to have the prefix property if none of its code words is a prefix of any other. For example, the code in Figure 11.1(a) is not prefix because the code word for A (0) is a prefix of the code word for C (010). Similarly, the code in (b) is not prefix, since all the code words start with 11, which is the code word for A. On the other hand, codes (c) and (d) are prefix.

The equivalence between instantaneous and prefix codes is easy to see. Suppose a prefix code is given and that a code word x is detected during the decoding of a given string. There can be no ambiguity in this case as we had in the decoding attempt in eq. (11.2), because if there were another possible interpretation y which can be detected later, like in eq. (11.3), it would imply that x is

a prefix of y, contradicting the prefix property. Conversely, if a code is instantaneous and a code word x is detected, then x cannot be the prefix of any other code word.

The conclusion is that the prefix property is a sufficient, albeit not necessary, condition for a code to be UD, and we shall henceforth concentrate on prefix codes. In fact, no loss is incurred by this restriction: it can be shown that given any UD code with code word lengths $\{\ell_1, \ldots, \ell_n\}$, one can construct a prefix code with the same set of code word lengths. As example, note that the prefix code (c) of Figure 11.1 has the same code word lengths as code (b). In this special case, the code words of code (c) are obtained from those of code (b) by reversing the strings; now every code word terminates in 11, and the substring 11 occurs only as suffix of any code word, thus no code word can be the proper prefix of any other. Incidently, this also shows that code (b), which is not prefix, is nevertheless UD. We shall return to this example in Section 11.3.2.

A connecting link between this chapter on codes and Chapter 4, which dealt with trees, is the following natural one-to-one correspondence between binary prefix codes and binary trees. We assign labels to both edges and vertices of a binary tree in the following way:

(i) Every edge pointing to a left child is assigned the label 0, and every edge pointing to a right child is assigned the label 1.
(ii) The root of the tree is assigned the empty string.
(iii) Every vertex v of the tree below the root is assigned a binary string which is obtained by concatenating the labels on the edges of the path leading from the root to vertex v.

It follows from the construction that the string associated with vertex v is a prefix of the string associated with vertex w if and only if v is on the path from the root to w. Thus the set of strings associated with the *leaves* of any binary tree satisfies the prefix property and may be considered as a prefix code. Conversely, given any prefix code, one can easily construct the corresponding binary tree. For example, the tree corresponding to the code $\{11, 001, 0100, 0101\}$ is depicted in Figure 11.2, in which the leaves labeled by the code words have been emphasized.

Based on this bijection between general binary trees and prefix codes, we can consider their sets as equivalent. In particular, we might be interested in the subset of codes corresponding to *complete* binary trees, see their definition and properties in Section 4.3. There are good reasons to consider complete trees: if not all the internal nodes of the tree have two children, like certain nodes in Figure 11.2, one could replace certain code words by shorter ones, without violating the prefix property, i.e., build another UD code with strictly smaller

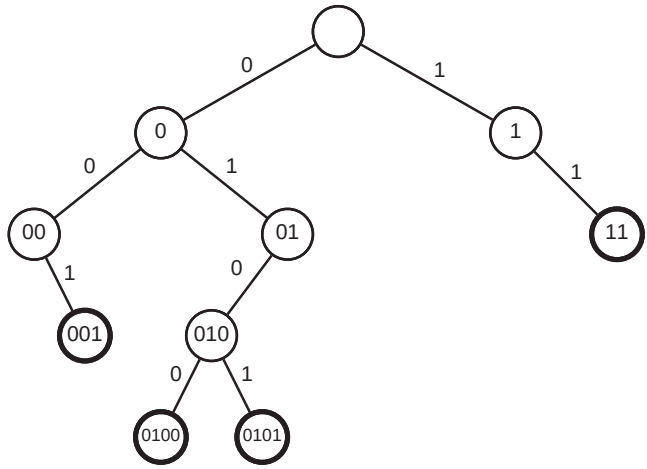

Figure 11.2. Tree corresponding to the code {11, 001, 0100, 0101}.

average code word length. For example, the nodes labeled 1 and 00 have only a right child, so the code words 11 and 001 could be replaced by 1 and 00, respectively; similarly, the vertex labeled 01 has only a left child, so the code words 0100 and 0101 could be transformed by deleting their third bit from the left, yielding 010 and 011, respectively.

As alternative, instead of replacing code words by shorter ones, one could add more code words to the code: if 0100 and 0101 remain in the tree, one could add, for instance, 011. After these amendments, the tree of Figure 11.2 is transformed into a complete one corresponding to the code in Figure 11.1(d), which is, accordingly, called a *complete code*. A code is complete if and only if the lengths $\{\ell_i\}$ of its code words satisfy eq. (11.1) with equality, i.e., $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. This has been shown in Theorem 4.1. An equivalent definition is that a complete binary prefix code is a set C of binary code words which is a binary prefix code, but the addition of any binary string x turns the set into a code $C \cup \{x\}$ which is not UD.

11.2.2 Huffman Coding

Figure 11.3 summarizes schematically how the class of codes under consideration has been restricted in several steps. Starting with a restriction to general UD codes, passing to instantaneous, or equivalently, prefix codes, and finally to complete prefix codes, since we are interested in good compression performance. Even this last set of complete codes is large enough to focus further,

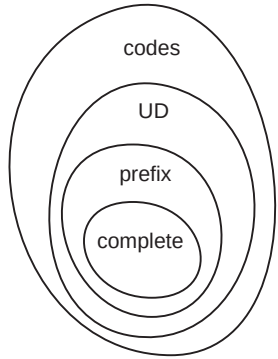

Figure 11.3. Progressively restricting the set of codes.

aiming for adapting an *optimal* code to a given distribution of the character frequencies.

The general problem can thus be stated as follows: we are given a set of n nonnegative weights $\{w_1, \ldots, w_n\}$, which are the frequencies of occurrence of the letters of some alphabet. The problem is to generate a complete binary variable-length prefix code, consisting of code words with lengths ℓ_i bits, $1 \leq i \leq n$, with optimal compression capabilities, i.e., such that the *total length* of the encoded text

$$\sum_{i=1}^{n} w_i \ell_i \tag{11.4}$$

is minimized.

If one forgets about the interpretation of the ℓ_i as code word lengths, and tries to solve the minimization problem analytically without restricting the ℓ_i to be integers, but still keeping the constraint that they must satisfy the equality $\sum_{i=1}^{n} 2^{-\ell_i} = 1$, one gets

$$\ell_i = -\log_2 \left(\frac{w_i}{W} \right) = -\log_2 p_i,$$

where $W = \sum_{i=1}^{n} w_i$ is the sum of the frequencies and thus the total length of the file, and $p_i = w_i/W$ is the relative frequency or probability of the ith letter. This quantity is known as *the information content* of a symbol with probability p_i, and it represents the exact number of bits in which the symbol should ideally be encoded. Note that this number is not necessarily an integer. Returning to the sum in (11.4), we may therefore conclude that the lower limit of the total

size of the encoded file is given by

$$-\sum_{i=1}^{n} w_i \log_2 p_i = W\Big(-\sum_{i=1}^{n} p_i \log_2 p_i\Big).$$

The quantity $H = -\sum_{i=1}^{n} p_i \log_2 p_i$, which is the weighted average of the information contents, has been defined by C. E. Shannon as the *entropy* of the probability distribution $\{p_1, \ldots, p_n\}$, and it gives a lower bound on the average code word length.

Returning to our problem, the lower bound does not really help, because we have to satisfy the additional constraint that the code word lengths have to be integers. In 1952, D. Huffman proposed the following algorithm which solves the problem in an optimal way.

(i) If $n = 1$, the code word corresponding to the only weight is the null-string.
(ii) Let w_1 and w_2, without loss of generality, be the two smallest weights.
(iii) Solve the problem recursively for the $n-1$ weights $w_1 + w_2$, $w_3, \ldots, w_n$; let α be the code word assigned to the weight $w_1 + w_2$.
(iv) The code for the n weights is obtained from the code for $n-1$ weights generated in point 3 by replacing α by the two code words $\alpha 0$ and $\alpha 1$.

In the obvious implementation of the code construction, the weights are first sorted and then every weight obtained by combining the two which are currently the smallest, is inserted in its proper place in the sequence so as to maintain order. This yields an $O(n^2)$ time complexity. One can reduce the time complexity to $O(n \log n)$ by using two queues, as we saw in Section 2.2.1. As alternative, the weights could be used to build a min-heap in time $O(n)$ (see Chapter 7). Each iteration consists then of two min-extractions and one insertion, all of which can be done in $O(\log n)$, for a total of $O(n \log n)$.

As example, suppose we wish to build a Huffman code for certain countries according to the number of their Nobel laureates. The ordered frequencies are given in Figure 11.4 next to the name of the countries. In each iteration, the two smallest numbers are removed from the list, and their sum is inserted in the proper position, until only two frequencies remain. If 437 is assigned the code word $\alpha = 0$, then the code words for 241 and 196 are 00 and 01, and the other code words are derived similarly in a backward scan of the columns. The final code appears in the leftmost column of Figure 11.4, and the corresponding Huffman tree, showing the frequencies in its nodes, is in Figure 11.5.

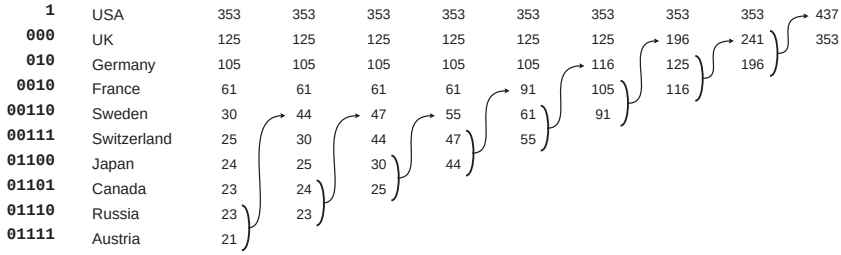

Figure 11.4. Huffman code on Nobel laureates distribution.

The decompression of a Huffman encoded string, or actually of any binary string S, which is the encoding of some text according to a prefix code C, can be conveniently performed by repeatedly scanning the tree T corresponding to C. As initialization, a pointer p points to the root of T and an index i is used to identify the currently scanned bit of the encoded string. We assume that the leaves of the tree store the corresponding characters in their *value* fields. In the tree of Figure 11.5, the "characters" are country names and appear underneath the corresponding leaves. For example, the encoding of the countries of the Nobel Prize laureates in Chemistry and Physics for 2015 would be 000110110001101,

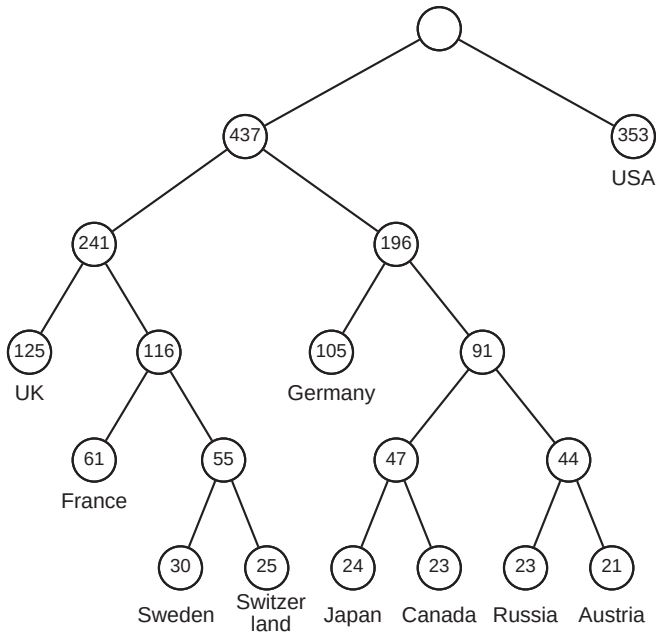

Figure 11.5. Huffman tree of the code of Figure 11.4.

```
Decode(S, T)
    p ⟵ root(T)
    for i ← 1 to |S|
        if S[i] = 0 then
                p ⟵ left(p)
        else    p ⟵ right(p)
        if p is a leaf then
                output value(p)
                p ⟵ root(T)
```

Figure 11.6. Decoding a string S using the tree T of the corresponding code.

to be decoded as UK, USA, USA, Japan, Canada. The formal decoding, with parameters S and T, is given in Figure 11.6.

The fact that Huffman's construction yields an optimal tree is far from being obvious. The proof is based on the following claims concerning optimal trees in general. Let T_1 be an optimal tree for a set of $n \geq 2$ weights $\{w_1, \ldots, w_n\}$. To simplify the description, we shall refer to the w_i as probabilities, but the claims are true for any set of weights. Denote the average code word length by $M_1 = \sum_{i=1}^{n} w_i l_i$, where l_i is the length of the code word assigned to weight w_i.

Claim 11.1. There are at least two elements on the lowest level of T_1.

Proof Suppose there is only one such element and let $\gamma = a_1 \cdots a_m$ be the corresponding binary code word. Then by replacing γ by $a_1 \cdots a_{m-1}$ (i.e., dropping the last bit) the resulting code would still be prefix, and the average code word length would be smaller, in contradiction with T_1s optimality. ∎

Claim 11.2. The code words c_1 and c_2 corresponding to the smallest weights w_1 and w_2 have maximal length (the nodes are on the lowest level in T_1).

Proof Suppose the element with weight w_2 is on level m, which is not the lowest level ℓ. Then there is an element with weight $w_x > w_2$ at level ℓ. Thus the tree obtained by switching w_x with w_2 has an average code word length of

$$M_1 - w_x\ell - w_2 m + w_x m + w_2\ell = M_1 - (\ell - m)(w_x - w_2) < M_1,$$

which is impossible since T_1 is optimal. ∎

Claim 11.3. Without loss of generality one can assume that the smallest weights w_1 and w_2 correspond to sibling nodes in T_1.

Proof Otherwise one could switch elements without changing the average code word length. ∎

Theorem 11.1. Huffman's algorithm yields an optimal code.

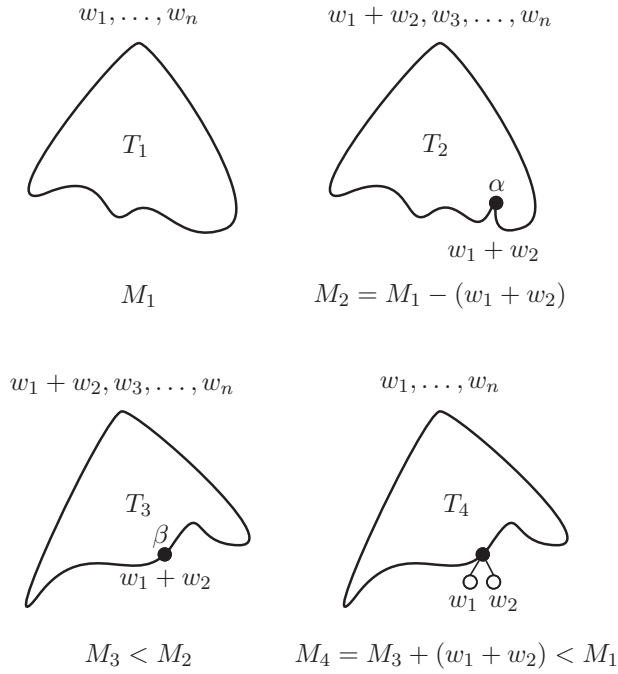

Figure 11.7. Proof of the optimality of Huffman codes.

Proof By induction on the number of elements n. For $n = 2$, there is only one complete binary prefix code, which therefore is optimal, namely $\{0, 1\}$; this is also a Huffman code, regardless of the weights w_1 and w_2.

Assume the truth of the theorem for $n - 1$. Let T_1 be an optimal tree for the weights $\{w_1, \ldots, w_n\}$ as the tree mentioned in the earlier claims. We shall show that a Huffman tree built for the same set of weights yields the same average code word length M_1, which shows that it is optimal.

It will be convenient to follow the description of the following various trees referring to their schematic in Figure 11.7. The weights appear above the trees, and the average code word length below them.

Consider the tree T_2 obtained from T_1 by replacing the sibling nodes corresponding to w_1 and w_2 by their common parent node α, indicated by the black dot in Figure 11.7, to which the weight $w_1 + w_2$ is assigned. Thus the average code word length for T_2 is $M_2 = M_1 - (w_1 + w_2)$. There is no reason to believe that T_2 should be optimal for the weights $(w_1 + w_2), w_3, \ldots, w_n$. We started from an optimal tree T_1, made some very local transformation on its structure, affecting only a few nodes, and assigned, seemingly arbitrarily, $n - 1$

new weights to the $n-1$ leaves of T_2. Why should such a minor modification of the tree keep the optimality of the tree we started with?

Nonetheless, we shall prove that this is precisely the case.

Claim 11.4. T_2 is optimal for the weights $(w_1 + w_2), w_3, \ldots, w_n$.

Proof If not, let T_3 be a better tree with average code word length $M_3 < M_2$. Note that while T_2 is almost identical to T_1, the shape of T_3 can be entirely different from that of T_2 and thus also from that of T_1, as shown in Figure 11.7. Let β be the leaf in T_3 corresponding to the weight $(w_1 + w_2)$. Consider the tree T_4 obtained from T_3 by adding children to β, thereby transforming it into an internal node, and assigning the weight w_1 to β's left child and w_2 to its right child. Then the average code word length for T_4 is

$$M_4 = M_3 + (w_1 + w_2) < M_2 + (w_1 + w_2) = M_1,$$

but this is impossible, since T_4 is a tree for n elements with weights $w_1, \ldots, w_n$ and T_1 is optimal among all those trees, so T_4 cannot have a smaller average code word length. ∎

We may now return to the proof of the theorem. Using the inductive assumption, the tree T_2, which is optimal for $n-1$ elements, has the same average code word length as the Huffman tree for these weights. However, the Huffman tree for $w_1, \ldots, w_n$ is obtained from the Huffman tree for $(w_1 + w_2), w_3, \ldots, w_n$ in the same way as T_1 is obtained from T_2. Thus the Huffman tree for the n elements has the same average code word length as T_1, hence it is optimal. ∎

11.3 Universal Codes

Huffman's solution is not always applicable, for example, when the set to be encoded is the set of the integers. Of course, the specific set we deal with will always be finite, however, we may sometimes wish to prepare a fixed code for all the possible elements, so the set for which code words have to be generated may not be bounded. In that case, using the standard binary representation of an integer is not possible: the number 13 can be represented as 1101 in binary, but we have to decide if to use 1101 as a code word, or maybe rather 01101 or 0001101. The set of the code words without leading zeros $\{1, 10, 11, 100, 101, \ldots\}$ is not a prefix code and is not even UD! It can be turned into a UD code by adding leading zeros to form a fixed length code, but how many bits should then be used?

Let us reconsider the problem of encoding a string by means of a dictionary we studied in Section 3.4.2. Given is a text T and a dictionary D, and the

problem is to parse T into a sequence of elements of D, so that T can be replaced by the correspondent sequence of pointers to D. We mentioned that the problem of finding a good dictionary D is difficult, but J. Ziv and A. Lempel suggested that T itself could be used as the dictionary! All one needs to do is to replace substrings s of T by pointers to earlier occurrences of s in T. The form of these pointers will be pairs (off, len), where off gives the offset of the occurrence, that is, how many characters do we have to search backward, and len is the number of characters to be copied. Here is an example, due to F. Schiller:

`von-der-Stirne-heiß-rinnen-muß-der-Schweiß-` $\cdots$

could be replaced by

`von-der-Stirne-heiß-rin`(11,2)(23,2)`mu`(11,2)(27,5)`chw`(23,4) $\cdots$

The question is how to encode the numbers off and len, since they could, theoretically, be as large as the length of the text up to that point. One usually sets some upper bound on off, which effectively means that we are looking for previous occurrences only within some fixed sized window preceding the current position. This size might be, say, 16K, so that $\log_2 16K = 14$ bits suffice to encode any offset. Similarly, one could impose an upper limit on len, of, say, 256 characters, so that it can be encoded in 8 bits. But most of the copied items will be shorter, thus using always the maximum is wasteful. Moreover, a longer copy item is not unrealistic, because overlaps are allowed. For instance, a string of k identical characters `ccc`$\cdots$`c` (often 0s or blanks) can be encoded by `c`$(1, k-1)$.

The solution we seek should thus be able of encoding any integer in some systematic way. A simple way could be a *unary* encoding, using the code words

$$1, 01, 001, 0001, 00001, 000001, \ldots, \tag{11.5}$$

that is, the length of the nth code word will be n, which is not reasonable for most applications. We shall rather aim at getting a code word of length $O(\log n)$ to represent n. Infinite code word sets with such a property have been defined by P. Elias as being *universal* codes.

11.3.1 Elias Codes

The two best known codes devised by Elias have been called γ and δ. To build the code word for an integer $n \geq 1$ in Elias's γ code, start by using its

standard binary representation without leading zeros, $B(n)$. The length of $B(n)$ is $\lfloor \log_2 n \rfloor + 1$ bits. Consider $B(n)$ without its leading 1-bit, and precede this string by an encoding of its length, using the unary encoding of eq. (11.5). For example, $B(100) = 1100100$, so the encoding of (decimal) 100 will be the suffix of length 6 of $B(100)$, preceded by six zeros and a 1-bit: 0000001100100. $B(1) = 1$, hence the suffix is empty and the string to be pre-pended is 1, thus the γ code word for 1 is 1 itself.

For decoding, the code word is scanned from its left end up to the first occurrence of a 1. If m zeros have been read, we know that the length of the code word is $2m + 1$, of which the rightmost $m + 1$ are the standard binary representation of the sought integer. Some sample code words of Elias's γ code appear in the second column of Table 11.2.

Because of the unary encoding of the length, the γ code is wasteful for large integers, which led to the definition of the δ code. We again start, as for γ, with $B(n)$ without its leading 1-bit, but instead of using a unary code for the length part, the length will be encoded using the γ code. This time, it will be the length of $B(n)$ including the leading 1-bit, because otherwise, the length to be encoded for the integer 1 would be zero, and the γ and δ codes are only defined for $n > 0$. Taking the same example as before, the suffix of length 6 of $B(100) = 1100100$ will be preceded by the γ code word for $|B(100)| = 7$, which is 00111, thus the δ code word for 100 is 00111100100.

Decoding the δ code word of some integer x is done in two stages. First a γ code word is detected. Since γ is a prefix code, this can be done, e.g., by some tree traversal, and no separating bit is needed. If this code word represents the integer m, then $m - 1$ more bits $b_1 b_2 \cdots b_{m-1}$ have to be read, and x is the integer such that $B(x) = 1 b_1 b_2 \cdots b_{m-1}$. For example, if the given code word is 0001010111101000, its only prefix representing a γ code word is 0001010, which is the γ encoding of the integer ten. We then read the next 9 bits, precede them by a leading 1-bits, and get 1111101000, which is the binary representation of (decimal) 1000. Sample code words of Elias's δ code appear in the third column of Table 11.2.

The length of the δ code word for n, $2\lfloor \log_2(\lfloor \log_2 n \rfloor + 1) \rfloor + 1 + \lfloor \log_2 n \rfloor$, is therefore asymptotically smaller than that of the corresponding γ code word, which is $2\lfloor \log_2 n \rfloor + 1$, but that does not mean that δ codes are always preferable to γ codes. The integers to be encoded do generally not appear with a uniform distribution, and it will often happen that the smaller values are much more frequent than larger ones. Yet, it is precisely for these small values that γ has shorter code words than δ, and there are probability distributions on the integers for which the *average* code word length will be shorter for γ than for δ.

Table 11.2. *Elias's γ, Elias's δ, and Fibonacci codes*

Index	Elias's γ	Elias's δ	Fibonacci
1	1	1	11
2	010	0100	011
3	011	0101	0011
4	00100	01100	1011
5	00101	01101	00011
6	00110	01110	10011
7	00111	01111	01011
8	0001000	00100000	000011
9	0001001	00100001	100011
15	0001111	00100111	0100011
16	000010000	001010000	0010011
17	000010001	001010001	1010011
100	0000001100100	00111100100	00101000011
128	000000010000000	00010000000000	00010001011
1000	00000000011111101000	0001010111101000	0000010000000011

It should also be mentioned that encoding the integers is not restricted to numerical applications only. Consider a large corpus of textual data written in some natural language. A popular way to encode it is by generating a code for all the different *words*, rather than just for the different characters. This extended "alphabet" may consist of hundreds of thousands of different terms, and using Huffman coding according to their probabilities of occurrence in the text yields quite good compression performance. The advantage of using this particular dictionary is that large texts are often the heart of some *Information Retrieval* system, as those mentioned in Section 8.1.1. Such systems need the list of the different terms anyway in their inverted files, so there is no additional overhead in using the list also for compression.

It will often be convenient not to generate a new optimal Huffman code after each update of the text, but to use a code that is fixed in advance. The encoding algorithm would then be as follows:

(i) Sort the different terms by nonincreasing frequency.
(ii) Assign the *i*th code word of some universal code, which is the encoding of the integer *i*, to the *i*th term of the sorted list.

For example, the most frequent words are, in order,

for English: `the, be, to, of, and, a, in, that, have, I, ...`
for French: `le, de, un, être, et, à, il, avoir, ne, je, ...`
for German: `der, die, und, in, den, von, zu, das, mit, sich, ...`

The first terms in English, `the`, `be`, and `to`, could then be encoded with a γ code by 1, 010, and 011, respectively.

11.3.2 Fibonacci Codes

An alternative to Elias codes with interesting features is to base the code on the Fibonacci numbers we have seen when studying the depth of AVL trees in Section 5.2. The standard representation of an integer as a binary string is based on a numeration system whose basis elements are the powers of 2. If the number B is represented by the k-bit string $b_{k-1}b_{k-2}\cdots b_1b_0$, then

$$B = \sum_{i=0}^{k-1} b_i 2^i.$$

But many other possible binary representations do exist, and let us consider those using the Fibonacci sequence as basis elements. Recall that we have defined the Fibonacci sequence in eq. (5.7) as

$$\{F(0), F(1), F(2), F(3), \ldots\} = \{0, 1, 1, 2, \ldots\},$$

but the sequence of basis elements should be $\{1, 2, 3, 5, \ldots\}$, thus it should start with index 2.

Any integer B can be represented by a binary string of length $r-1$, $c_rc_{r-1}\cdots c_2$, such that

$$B = \sum_{i=2}^{r} c_i F(i).$$

The representation will be unique if one uses the following procedure to produce it: given the integer B, find the largest Fibonacci number $F(r)$ smaller or equal to B; then continue recursively with $B - F(r)$. For example, $31 = 21 + 8 + 2$, so its binary Fibonacci representation would be 1010010. As a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s.

This property can be exploited to devise an infinite code whose set of code words consists of the Fibonacci representations of the integers: to assure the code being UD, each code word is prefixed by a single 1-bit, which acts like a comma and permits to identify the boundaries between the code words. The first few elements of this code would thus be

$$\{u_1, u_2, \ldots\} = \{\mathbf{1}1, \mathbf{1}10, \mathbf{1}100, \mathbf{1}101, \mathbf{1}1000, \mathbf{1}1001, \ldots\},$$

where the separating 1 is put in boldface for visibility. A typical compressed text could be **1**1001**1**100**1**101**11**101, which is easily parsed as $u_6u_3u_4u_1u_4$. Though being UD, this is not a prefix code, in fact, this is the code in the example in Figure 11.1(b). In particular, the first code word 11, which is the only one containing no zeros, complicates the decoding, because if a run of several such code words appears, the correct decoding of the code word preceding the run depends on the parity of the length of the run, as we have seen in eqs. (11.2) and (11.3).

To overcome this problem, the example in Figure 11.1(c) suggests to reverse all the code words, yielding the set

$$\{v_1, v_2, \ldots\} = \{11, 011, 0011, 1011, 00011, 10011, \ldots\},$$

which is a prefix code, since all code words are terminated by 11 and this substring does not appear anywhere in any code word, except at its suffix. This code is known as *Fibonacci code* and a sample of its code words appears in the last column of Table 11.2.

One of the advantages of a Fibonacci code is its robustness against errors. If a bit gets lost, or an erroneous bit is picked up, or some bit value is swapped, the error will not propagate as it might for other variable or even fixed length codes. For Fibonacci, at most three code words can be affected, and mostly only one or two. For example, if the emphasized, third bit of $v_3v_1v_2 = \texttt{0011-11-011}$ is turned into a zero, the string would be interpreted as $\texttt{00011-1011} = v_5v_4$.

The length of a Fibonacci code word is related to the fact that $F(i) \simeq \phi^i/\sqrt{5}$, where $\phi = 1.618$ is the golden ratio, see eq. (5.9). It follows that the number of bits needed to represent an integer n using the Fibonacci code is $1.4404 \log_2 n$, similarly to the formula for the depth of an AVL tree we saw in eq. (5.2). The Fibonacci code word for n is thus about 44% longer than the minimal $\log_2 n$ bits needed for the standard binary representation using only the significant bits, but it is shorter than the $2 \log_2 n$ bits needed for the corresponding Elias γ code word. On the other hand, even though the number of bits is increased, the number of 1-bits is smaller on the average: while in a standard binary representation, about half of the bits are 1s, it can be shown that the probability of a 1-bit in the Fibonacci code words is only about $\frac{1}{2}\left(1 - \frac{1}{\sqrt{5}}\right) = 0.2764$, so even when multiplied by 1.44, this gives an expected number of only $0.398 \log_2 n$ 1-bits, rather than about $0.5 \log_2 n$. This property has many applications.

11.4 Error Correcting Codes

Our next topic considers a different application of codes: if the previous two sections focused on producing an encoding that is inexpensive in the number

of necessary bits, we now concentrate rather on the correctness of the data. The assumption is that for some reason, the decoder might get an erroneous string, which is not identical to the one produced by the encoder. We still wish to be able to recover the original message, if possible. For simplicity, we assume that there is only a *single* wrong bit, which corresponds to a scenario of a so low probability p for an error, that the possibility of two or more errors might be neglected.

Often the knowledge about the occurrence of an error, rather than its exact location, is sufficient. This can be achieved by adjoining a single bit, often called *parity bit*, consisting of the XOR, or equivalently, the sum modulo 2, of the given n data bits. For example, if we wish to encode the number 1618, its 11-bit standard binary representation would be 11001010010, but we would add as the twelfth bit the XOR of these bits, which is 1, getting

$$1618 \quad \longrightarrow \quad 110010100101,$$

where the parity bit has been emphasized. To decode such an *Error detection code*, we again apply XOR, but on all the bits, including the parity bit. We should get 0. If not, one of the bits has been changed, possibly the parity bit itself. There is no way to know which one of the bits is in error, but the very fact that we know that some error has occurred is enough for the decoder to ask the encoder to resend the data.

11.4.1 A Sequence of Error Correcting Codes

A simple way to enable even correction, not only detection, is to transmit every bit three times. The original data can then be recovered, even if there is a single error, by using the majority rule for every bit-triple. But one can do better than adding $2n$ check-bits to n data bits.

Organize the n data bits into a square with side length $\sqrt{n}$ bits, and add a parity bit for each column and each row. A single error in any of the n data bits can then be located by intersecting the row and the column corresponding to the only affected parity bits. We thus got error correction at the price of additional $2\sqrt{n}$ bits.

A further step would then be to rearrange the data into a cube of side length $\sqrt[3]{n}$. Three vectors of parity bits are then needed, each of length $\sqrt[3]{n}$ and each corresponding to a partition of the cube into planes according to another dimension. Each parity bit will now be the XOR of one of the 2-dimensional planes, that is, of $\sqrt[3]{n^2}$ bits. A single error in one of the data bits will affect exactly three of the parity bits, and the location of the erroneous bit is at the intersection of

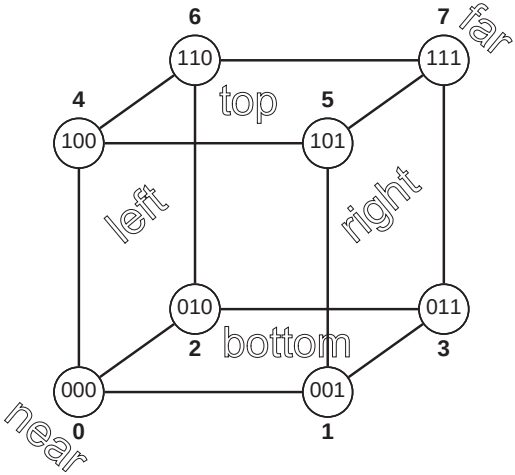

Figure 11.8. Layout of eight data bits in a three-dimensional hypercube with six parity bits.

the three corresponding planes. With such a layout, only $3\sqrt[3]{n}$ additional bits are needed.

For the general case, the data bits are arranged into a k-dimensional hypercube with side length $n^{1/k}$. The number of vectors of parity bits will be k, and each bit will be the XORing of all the data bits of a $(k-1)$-dimensional hyperplane, all the bits of the same vector corresponding to the $n^{1/k}$ disjoint $(k-1)$-dimensional hyperplanes forming the original hypercube, and each of the k vectors corresponding to such a partition in a different dimension. The number of parity bits is $kn^{1/k}$. Figure 11.8 shows a 3-dimensional hypercube with side length 2, and its partition into two 2-dimensional planes in each of the three dimensions: left–right, near–far, top–bottom.

What value should be chosen for k? Note that this is almost an identical derivation to the one in Section 10.1, leading from a simple selection sort to heapsort. The same function $kn^{1/k}$ had to be optimized, and the solution was to use $k = \log_2 n$. This corresponds to a $\log_2 n$-dimensional cube of side length 2 bits, which is the smallest possible side length. The number of parity bits should thus be $2 \log_2 n$.

Figure 11.8 is the hypercube corresponding to $n = 8$, each vertex corresponding to one of the n data bits and being labeled by the 3-bit binary representation of its index i, $0 \le i < 2^3$; vertices are connected by edges if and only if they correspond to numbers differing exactly by a single bit in their binary representations.

For the parity bits, the left–right dimension corresponds to the first (rightmost) bit in the binary representation, that is, left is the XORing of all the data bits having a 0 in the first bit of the binary representation of their index, and right is the XORing of the complementing set of those with 1 in the first bit. Similarly, the near–far dimension corresponds to the second bit, so near is the XORing of the bits indexed 0, 1, 4, and 5 (0 in the second bit), and far is the XORing of the complementing set. Finally, the top–bottom dimension corresponds to the third (leftmost) bit. Summarizing, each of the $2 \log_2 n = 6$ parity bits is obtained by XORing a subset of $\frac{n}{2}$ bits, as follows:

left	0,2,4,6	right	1,3,5,7
near	0,1,4,5	far	2,3,6,7
bottom	0,1,2,3	top	4,5,6,7.

Here is an example of how the parity bits can be used to correct a single error. Suppose the data bit indexed 6 is flipped. This will have an effect on all the parity bits that include 6 in their lists: left, far, and top. Rearranging the bits from left to right and reconverting to the corresponding 0 and 1 values, one gets: top, far, left = 110, which is the binary representation of 6, the index of the wrong bit.

If the side of the hypercube is longer than 2 bits, we need to know the index of the projection of the erroneous bit on each of the dimensions. But if the side length is just 2, we are left with a binary choice, as the possible indices in each dimension are just 0 or 1. Therefore, it suffices to keep only the parity bits corresponding to 1 values, that is right, far, and top. This corresponds to indicating only the 1-bits of the index of the wrong bit, which is enough to recover it, unless its index is 0 and thus has no 1-bits. In addition, one also needs to deal with the case in which no error has occurred. To solve both problems, one might reduce the set of data bits and index them only by the nonzero values. This method of adding $\log_2 n$ parity bits defined according to the binary representation of the indices of the n data bits is known as *Hamming code*. R. Hamming also suggested to store the parity bits interleaved with the data bits at the positions whose indices are powers of 2.

11.4.2 Hamming Code

Here is a more standard definition of the Hamming code. Given are $n = 2^m - 1$ bits of data, for some $m \geq 2$. According to Hamming's scheme, they are indexed from 1 to n, and the bits at positions with indices that are powers of 2 serve as parity bits, so that in fact only $n - m$ bits carry data. The ith parity bit, which will be stored at position 2^{i-1} for $i = 1, 2, \ldots, m$, will be set so that the

XOR of some selected bits will be zero. For the parity bit at 2^{i-1}, the selected bits are those whose indices, when written in standard binary notation, have a 1-bit in their ith bit from the right. That is, the parity bit stored in position $2^0 = 1$ is the XOR of the bits in positions $3, 5, 7, \ldots$, the parity bit stored in position $2^1 = 2$ is the XOR of the bits in positions $3, 6, 7, 10, 11, \ldots$, and so on.

An easy way to remember this procedure is by adding a fictitious first 0-bit, indexed 0, and then scanning the resulting binary string as follows. To get the first parity bit, the one stored in position $2^0 = 1$, compute the XOR of the sequence of bits obtained by repeatedly skipping one bit and taking one bit. This effectively considers all the bits with odd indices (the first of which, with index 1, is the parity bit itself). In general, to get the second, third,. . ., ith parity bit, the one stored in position 2^{i-1}, for $i = 1, 2, \ldots, m$, compute the XOR of the sequence of bits obtained by repeatedly skipping 2^{i-1} bits and taking 2^{i-1} bits.

Continuing our earlier example, consider $n = 15 = 2^4 - 1$ bits, of which only $15 - 4 = 11$ contain data, and assume the data is the standard 11-bit binary representation of 1618, namely 11001010010. This will first be stored as a string in which the bit positions indexed by powers of 2 are set to 0 and the 11 information bits are filled, from left to right, in the remaining positions. Adding the 0-bit at position 0 yields

$$_0\,\mathbf{0}\,\mathbf{0}\,1\,\mathbf{0}\,1\,0\,0\,\mathbf{0}\,1\,0\,1\,0\,0\,1\,0, \tag{11.6}$$

where the zeros in the positions reserved for the parity bits have been emphasized, and the leftmost 0 is smaller to recall that this bit does not carry information. The parity bits in positions 1, 2, 4 and 8 are then calculated, in order, by XORing the underlined bits in the following lines:

$$_0\,\underline{\mathbf{0}}\,0\,\underline{1}\,0\,\underline{1}\,0\,\underline{0}\,0\,\underline{1}\,0\,\underline{1}\,0\,\underline{0}\,\underline{1}\,\underline{0}\,,$$

$$_0\,0\,\underline{\mathbf{0}}\,\underline{1}\,0\,1\,\underline{0\,0}\,0\,1\,\underline{0\,1}\,0\,0\,\underline{1\,0}\,,$$

$$_0\,0\,0\,1\,\underline{\mathbf{0}\,1\,0\,0}\,0\,1\,0\,1\,\underline{0\,0\,1\,0}\,,$$

$$_0\,0\,0\,1\,0\,1\,0\,0\,\underline{\mathbf{0}\,1\,0\,1\,0\,0\,1\,0}\,,$$

which yields, respectively, the bits 0, 1, 0, 1. The final Hamming code word is therefore

$$_0\,\mathbf{0}\,\mathbf{1}\,1\,\mathbf{0}\,1\,0\,0\,\mathbf{1}\,1\,0\,1\,0\,0\,1\,0. \tag{11.7}$$

Figure 11.9 is an example of how the Hamming code is able to recover from a single bit-flip. Suppose the data is the 11-bit representation of 1618, as given, with the 4 parity bits, in eq. (11.7), and suppose there is an error at bit

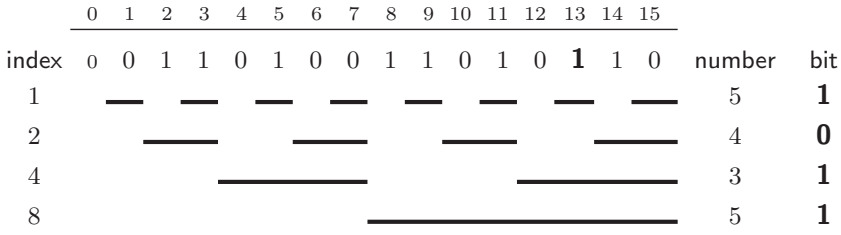

Figure 11.9. Example of correcting a single bit error in position 13.

position 13, that is, the 0-bit there has turned into a 1-bit. The bits appear in the upper line of the body of the table, while the indices appear in the header line. The leftmost column is the index of the given parity bit, and for each of these, the underlined bits are those taken into account when calculating the parity bit. The column headed number gives the number of 1-bits in the set of underlined bits, and the last column is their parity (1 if odd, 0 if even).

The bits in the last column, read bottom up to correspond to the bits indexed 8421, are 1101, which is the standard binary representation of the number 13, the index of the error. We thus know that this bit is wrong, can correct it, and end up with the original data, as requested. If there is no error, all the calculated parity bits would be 0, yielding 0000 in the last column – an indication that the correct data has been received.

11.5 Cryptographic Codes

Cryptography is a fascinating field of research and has undergone radical changes in the past few decades. It is beyond the scope of this book, and we shall just give very simple and historical examples to show the existence of cryptographic codes. It should be mentioned that many of the modern cryptographic methods are based on advanced mathematical tools, and in particular on prime numbers, which we have seen in Chapter 7 in the context of hashing. The relevant prime numbers are huge, using hundreds of bits, like, for example, $2^{393} - 93$, for which sophisticated programs are needed to even know that they are really prime.

Methods for encoding information so that its true content can be understood only by an intended partner, but remains hidden to any occasional or malicious eavesdropper, have been used for thousands of years. Their usefulness and even necessity are obvious, not only to keep military secrets, but modern life, and in particular commerce and banking related transactions we wish to perform from

our wireless phones, would be jeopardized if it were not possible to keep the data confidential.

The general paradigm is to be given two complementing encoding and decoding functions $\mathcal{E}$ and $\mathcal{D}$, both depending on some key K that is supposed to be kept secret. A message M, called the *cleartext* in the particular jargon of the cryptographers, is encrypted by applying

$$C = \mathcal{E}_K(M),$$

thereby producing the *ciphertext* C, which, in itself, should look like some random bunch of symbols and, at least apparently, not convey any information. Nonetheless, whoever is in possession of the secret key K may apply the inverse function for decoding, and get

$$\mathcal{D}_K(C) = \mathcal{D}_K(\mathcal{E}_K(M)) = M,$$

that is, reproduce the original message M.

The challenge is to find appropriate pairs of functions $\mathcal{E}$ and $\mathcal{D}$. Modern cryptography even considers *public key* encryption, in which the encoding function $\mathcal{E}$ is not hidden, but known to everybody, and yet its corresponding inverse function $\mathcal{D}$ can only be derived by somebody who knows the secret key K. These keys are chosen long enough, say of 1000 bits, so that an attempt to try all the possible variants, 2^{1000} in our example, may safely be ruled out.

Before the advent of the modern methods, that are often based on the difficulty to solve certain computationally hard problems, the encoding and decoding functions were much simpler. One of the oldest examples is a so-called *Caesar's code*: indexing the alphabet $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \ldots, \mathsf{Z}\}$ by $\{0, 1, 2, \ldots, 25\}$, the secret key K is just one of these numbers, and

$$\mathcal{E}_K(x) = (x + K) \bmod 26,$$

that is, in the encoding, each character is shifted cyclically K positions forward in the alphabet. For example, if $K = 5$ is chosen, then the encoding of the cleartext

SECRETKEY

would be the ciphertext

XJHWJYPJD. (11.8)

If it is known that a Caesar's code is used, an enemy could try all the possible shifts, since there are only 26 possibilities, until a reasonable cleartext is obtained. Moreover, the enemy would then not only know the specific message at hand, but also the key K, which enables the decoding also of subsequent text

enciphered in the same way. This action of revealing or guessing the secret key is called *breaking* the code, and it is one of the objectives of the enemy.

Caesar's code is a particular case of a more general class called a *substitution cipher*, in which a permutation of the alphabet is given, and each character is replaced by the one with the same index in this permutation. In this case, the secret key is the used permutation or its index, and it cannot be found by exhaustive search, since the number of possibilities is $26! \simeq 4 \cdot 10^{26}$. Nevertheless, a substitution cipher is not really secure, because if a long enough text is given, one could analyze the frequency patterns of the occurrences of the characters, and compare it with the known probabilities in the given language. For example, even for a short text like that displayed in eq. (11.8), the most frequent character is J, which we would guess to be a substitute for the most frequent character in English, which is E.

The idea of Caesar's code has been taken one step further by B. Vigenère, who suggested using a different shift for every character of the text, according to the key K which is now chosen as some secret string. So if the secret key is SECRETKEY, the first nine characters will be shifted cyclically by 18, 4, 2, 17, 4, 19, 10, 4, and 24 positions, respectively. The shifts for the subsequent characters of the cleartext are obtained by repeating this sequence as often as necessary. In the resulting ciphertext, the same letter can represent different cleartext characters, depending on their position, so that a frequency analysis is not possible. However, other attempts to break the code may be successful in what is called a *cryptographic attack*.

An example of encoding using a Vigenère cipher is given in Table 11.3. The chosen cleartext encodes the fact that we are at the end of this book. In the spirit of enciphering the information, let us first translate this idea into French, using the string

C'est fini.

It is common practice to translate everything to upper case and to remove blanks and punctuation signs. We then have to choose a key K. This is very similar to an almost daily request we are confronted with, namely of choosing a *password* for the numerous applications on our computers and cellular phones. Most of these passwords are somehow related to our names, birth dates or telephone numbers, which is why hackers can so easily break into our digital accounts. Many applications therefore urge us to use "stronger" passwords, and often even suggest some randomly generated ones, which, in principle, are impossible to guess. So let us use here also such a random string, for example, RKAOGWOS. The resulting ciphertext appears in the last line of the table, and should be void of any meaning (is it?).

Table 11.3. *Example of a Vigenère encoding*

Cleartext	C	E	S	T	F	I	N	I
Secret key	R	K	A	O	G	W	O	S
Ciphertext	T	O	S	H	L	E	B	A

Exercises

11.1 For each of the codes Elias-γ, Elias-δ, and Fibonacci,

(a) derive a formula for the number n_j of code words of length j, for all possible values of j;
(b) use n_j to prove that the codes are complete by showing that

$$\sum_{i=1}^{\infty} 2^{-\ell_i} = \sum_{j=1}^{\infty} n_j 2^{-j} = 1,$$

where ℓ_i is the length in bits of the ith code word, as used in eq. (11.1).

11.2 Show that the average code word length L for a Huffman code built for the set of probabilities $\{p_1, p_2, \ldots, p_n\}$ satisfies

$$H \leq L \leq H + 1,$$

where $H = -\sum_{i=1}^{n} p_i \log_2 p_i$ is the entropy as defined in Section 11.2.2.

11.3 In a prefix code no code word is the prefix of any other, and one could similarly define a suffix code in which no code word is the suffix of any other. The set of reversed code words of any prefix code is thus a suffix code. A code that has both the prefix and the suffix properties is called an *affix* code, in particular, every fixed length code is an affix code.

(a) Give an example of a variable length affix code. Note that it might not be possible to find an affix code if the alphabet is too small. **Hint:** Try a code with nine code words.
(b) Show that the number of different variable length affix codes is infinite.
(c) Affix codes are called *never-self-synchronizing* codes. Show why.

11.4 Suppose that instead of using a binary alphabet $\{0, 1\}$ to encode a message, one could use some alphabet $\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ with r symbols, for $r > 2$.

(a) Extend Huffman's algorithm to produce an optimal r-ary code for a given set of probabilities $\{p_1, p_2, \ldots, p_n\}$, by working with r-ary trees in which every node can have up to r children.

(b) Consider the following algorithm. Given a set of probabilities $\{p_1, p_2, \ldots, p_n\}$, produce first an optimal 4-ary code, then replace, in each code word, the four symbols $\{\alpha_1, \ldots, \alpha_4\}$ by $\{00, 01, 10, 11\}$, respectively. Show that the resulting binary code is not always optimal, and formulate a condition on the distribution for which this algorithm does produce an optimal code.

11.5 Show how, by the addition of a single bit, one can turn a Hamming code from an error-correcting code for a single error into a code capable to correct a single error and to detect a double error.

11.6 What is the shape of the Huffman tree if the weights of the characters are the first n nonzero Fibonacci numbers $1, 1, 2, \ldots, F(n)$?

11.7 This extends exercise 4.4 of the chapter on trees. Given are n ordered integer sequences $A_1, \ldots, A_n$, of lengths $\ell_1, \ldots, \ell_n$, respectively. We wish to merge all the sequences into a single ordered sequence, but without constraints on adjacency, so that at each stage, any pair of sequences may be merged. The number of steps needed to merge a elements with b elements is $a + b$. Find an optimal way to merge the n sets, that is, such that the number of comparisons is minimized.