Variable-length Codes

1- Unary coding

Unary coding, sometimes called thermometer code, is an entropy encoding that represents a natural number, n, with n ones followed by a zero (if natural number is understood as non-negative integer) or with n – 1 ones followed by a zero (if natural number is understood as strictly positive integer). For example 5 is represented as 111110 or 11110. Some representations use n or n – 1 zeros followed by a one. The ones and zeros are interchangeable without loss of generality. Unary coding is both a Prefix-free code and a Self-synchronizing code.

n (non-negative)	n (strictly positive)	Unary code	Alternative
0	1	0	1
1	2	10	01
2	3	110	001
3	4	1110	0001
4	5	11110	00001
5	6	111110	000001
6	7	1111110	0000001
7	8	11111110	00000001
8	9	111111110	000000001
9	10	1111111110	0000000001

Unary coding is an optimally efficient encoding for the following discrete probability distribution

$$\mathrm{P}(n) = 2^{-n} \quad \text{for } n = 1, 2, 3, \ldots.$$

2- Tunstall Code

The main advantage of variable-length codes is their variable lengths. Some codes are short, others are long, and a clever assignment of codes to symbols can produce compression. On the downside, variable-length codes are difficult to work with. The encoder has to construct each code from individual bits and pieces, has to accumulate and append several such codes in a short buffer, wait until n bytes of the buffer are full of code bits (where n must be at least 1), write the n bytes onto the output, shift the buffer n bytes, and keep track of the location of the last bit placed in the buffer. The decoder has to go through the reverse process. It is definitely easier to deal with fixed-size codes, and the Tunstall codes described here are an example of how such codes can be designed. The idea is to construct a set of fixed-size codes, each encoding a variable-length string of input symbols. As a result, these codes are also known as variable-to-fixed (or variable to-block) codes, in contrast to the variable-length codes which are also referred to as fixed-to-variable. Imagine an alphabet that consists of two symbols A and B where A is more common. Given a typical string from this alphabet, we expect substrings of the form AA, AAA, AB, AAB, and B, but rarely strings of the form BB. We can therefore assign fixed-size codes to the following five substrings as follows. AA = 000, AAA = 001, AB = 010, ABA = 011, and B = 100. A rare occurrence of two consecutive *B*s will be encoded by 100100, but most occurrences of B will be preceded by an A and will be coded by 010, 011, or 100. This example is both bad and inefficient. It is bad, because AAABAAB can be encoded either as the four codes AAA, B, AA, B or as the three codes AA, ABA, AB; encoding is not unique and may require several passes to determine the shortest code. This happens because our five substrings don't satisfy the prefix property. This example is inefficient because only five of the eight possible 3-bit codes are used. An n-bit Tunstall code should use all 2n codes. Another point is that our codes were selected without considering the relative frequencies of the two symbols, and as a result we cannot be certain that this is the best code for our alphabet. Thus, an algorithm is needed to construct the best n-bit Tunstall code for a given alphabet of N symbols, and such an algorithm is given in [Tunstall]. Given an alphabet of N symbols, we start with a code table that consists of the symbols. We then iterate as long as the size of the code table is less than or equal to 2n (the number of n-bit codes). Each iteration performs the following step

- Select the symbol with largest probability in the table. Call it S.
- Remove S and include the N substrings S_x where x goes over all the N symbols.

- This step increases the table size by N – 1 symbols (some of them may be substrings). Thus, after iteration k, the table size will be
  N + k(N – 1) elements.
- If N + (k + 1)(N – 1) ≤ 2^n, perform another iteration (iteration k + 1).

Example 2.18

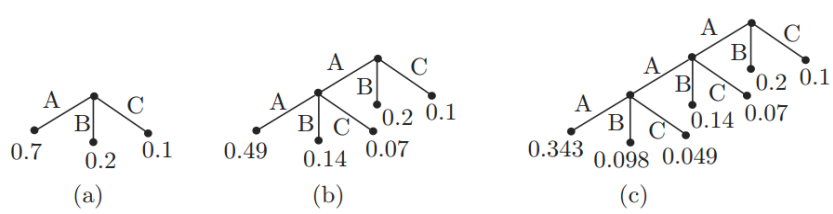


Figure 1.7a

Given an alphabet with the three symbols A, B, and C (N = 3), with probabilities 0.7, 0.2, and 0.1, respectively, we decide to construct a set of 3-bit Tunstall codes (thus, n = 3). We start our code table as a tree with a root and three children (Figure 1.7a). In the first iteration, we select A and turn it into the root of a subtree with children AA, AB, and AC with probabilities 0.49, 0.14, and 0.07, respectively (Figure 1.7b). The largest probability in the tree is that of node AA, so the second iteration converts it to the root of a subtree with nodes AAA, AAB, and AAC with probabilities 0.343, 0.098, and 0.049, respectively (Figure 1.7c). After each iteration we count the number of leaves of the tree and compare it to 23 = 8. After the second iteration there are seven leaves in the tree, so the loop stops. Seven 3-bit codes are arbitrarily assigned to elements AAA, AAB, AAC, AB, AC, B, and C. The eighth available code should be assigned to a substring that has the highest probability and also satisfies the prefix property.

The average bit length of this code is easily computed as

$$\frac{3}{3(0.343+0.098+0.049)+2(0.14+0.07)+0.2+0.1}=1.37 \text{ bits/symbol.}$$

In general, let pi and li be the probability and length of tree node i. If there are m nodes in the tree, the average bit length of the Tunstall code is $n/\sum_{i=1}^{m} p_i l_i$. The entropy of our alphabet is –(0.7 × log2 0.7+0.2 × log2 0.2+0.1 × log2 0.1) = 1.156, so the Tunstall codes do not provide the best compression. The tree of Figure 1.7 is referred to as a parse tree, not a code tree. It is complete in the sense that every interior node has N children. Notice that the total number of nodes of this tree is 3 × 2 + 1 and in general a(N – 1) + 1. A parse tree defines a set of substrings over the alphabet (seven substrings in our example) such that any string of symbols from the alphabet can be broken up (subdivided) into these substrings
(except that the last part may be only a prefix of such a substring) in one way

only. The subdivision is unique because the set of substrings defined by the parse tree is proper, i.e., no substring is a prefix of another substring. An important property of the Tunstall codes is their reliability. If one bit becomes corrupt, only one code will get bad. Normally, variable-length codes are not robust. One bad bit may corrupt the decoding of the remainder of a long sequence of such codes.
It is possible to incorporate error-control codes in a string of variable-length codes, but this increases its size and reduces compression.

3- Byte pair encoding

Byte pair encoding or digram coding is a simple form of data compression in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur within that data. A table of the replacements is required to rebuild the original data.

Example 2.19

Suppose we wanted to encode the data

aaabdaaabac

The byte pair "aa" occurs most often, so it will be replaced by a byte that is not used in the data, "Z". Now we have the following data and replacement table

Z=aa

ZabdZabac

Then we repeat the process with byte pair "ab", replacing it with Y:

ZYdZYac

Y=ab

Z=aa

We could stop here, as the only literal byte pair left occurs only once. Or we could continue the process and use recursive byte pair encoding, replacing "ZY" with "X":

XdXac

X=ZY

Y=ab Z=aa

4- Recursive Phased-In Codes

Any positive integer N can be written uniquely as the sum of certain powers of 2.

Thus, for example, 45 is the sum $2^5 + 2^3 + 2^2 + 2^0$. In general, we can write $N = \sum_{i=1}^{s} 2^{a_i}$, where $a_1 > a_2 > \cdots > a_s \geq 0$ and $s \geq 1$. For N = 45, for example, these values are s = 4 and $a_1 = 5$, $a_2 = 3$, $a_3 = 2$, and $a_4 = 0$. Once a value for N has been selected and the values of all its powers a_i determined, a set of N variable-length recursive phased-in codes can be constructed from the tree shown in Figure 1.8. For each power a_i, this tree has a subtree that is a complete binary tree of height a_i. The individual subtrees are connected to the root by 2s – 2 edges labeled 0 or 1 as shown in the figure.

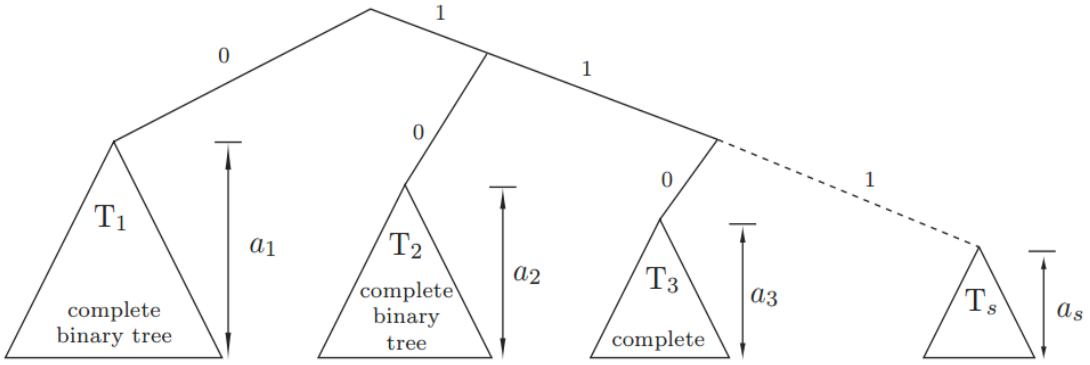


Figure 1.8: A Tree for Recursive Phased-In Codes.

The tree for N = 45 is shown in Figure 1.9. It is obvious that the complete binary tree for a_i has a_i leaves and the entire tree therefore has a total of N leaf nodes. The codes are assigned by sliding down the tree from the root to each leaf, appending a 0 to the code each time we slide to the left and a 1 each time we slide to the right. Some codes are also shown in the figure. Thus, the 45 recursive phased-in codes for N = 45 are divided into the four sets 0xxxxx, 10xxx, 110xx, and 111, where the x's stand for bits. The first set consists of the 32 5-bit combinations prepended by a 0, the second set includes eight 5-bit codes that start with 10, the third set has four codes, and the last set consists of the single code 111. As we scan the leaves of each subtree from left to right, we find that the codewords in each set are in ascending order. Even the codewords in different sets appear sorted, when scanned from left to right, if we append enough zeros to each so they all become six bits long. The codes are

prefix codes, because, for example, once a code of the form 0xxxxx has been assigned, no other codes will start with 0.

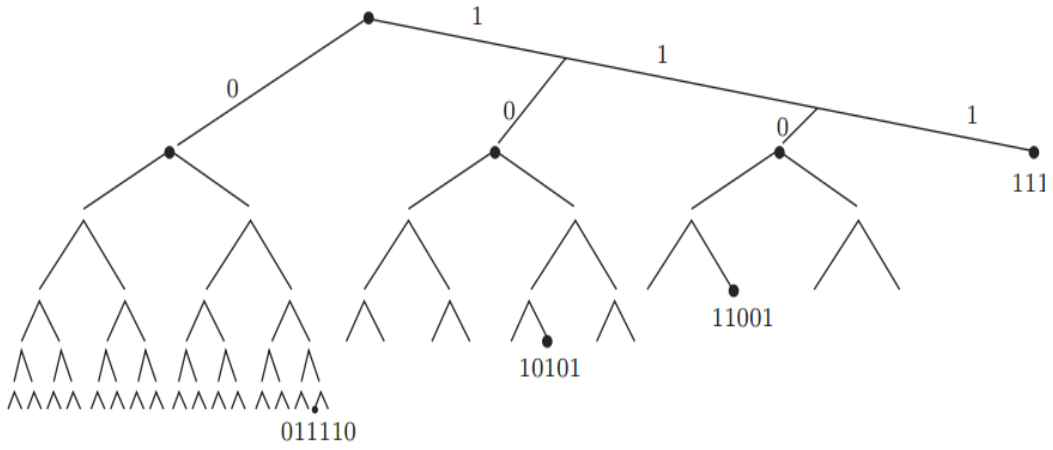


Figure 1.9: A Tree for N = 45

In practice, these codes can be constructed in two steps, one trivial and the other one simple, as follows:

1- (This step is trivial.) Given a positive integer N, determine its powers a_i. Given, for example, 45 = ... 000101101 we first locate its leftmost 1. The position of this 1 (its distance from the right end) is s. We then scan the bits from right to left while decrementing an index i from s to 1. Each 1 found designates a power ai.

2- There is no need to actually construct any binary trees. We build the set of codes for a1 by starting with the prefix 0 and appending to it all the a1-bit numbers. The set of codes for a2 is similarly built by starting with the prefix 10 and appending to it all the a2-bit numbers.

5- Start-Step-Stop Codes

The unary code is ideal for compressing data that consists of integers n with probabilities P(n) ≈ 2−n. If the data to be compressed consists of integers with different probabilities, it may benefit from one of the general unary codes (also known as start-step-stop codes). Such a code, proposed by [Fiala and Greene 89], depends on a triplet (start, step, stop) of nonnegative integer parameters. A set of such codes is constructed subset by subset as follows:

1- Set n = 0.

2- Set a = start + n × step.

3- Construct the subset of codes that start with n leading 1's, followed by a single intercalary bit (separator) of 0, followed by a combination of a bits. There are 2a such codes.
4- Increment n by step.
If n < stop, go to step 2.
If n > stop, issue an error and stop.
If n = stop, repeat steps 2 and 3 but without the single intercalary 0 bit of step 3, and stop.

Example 2.19

The triplet (n, 1, n) defines the standard (beta) n-bit binary codes, as can be verified by direct construction. The number of such codes is easily seen to be

n	$a = 3+n\cdot 2$	nth codeword	Number of codewords	Range of integers
0	3	$0xxx$	$2^3 = 8$	0–7
1	5	$10xxxx$	$2^5 = 32$	8–39
2	7	$110xxxxxx$	$2^7 = 128$	40–167
3	9	$111xxxxxxxx$	$2^9 = 512$	168–679
		Total	680	

Table 2.3: The General Unary Code (3,2,9).

n	$a = 2+n\cdot 1$	nth codeword	Number of codewords	Range of integers
0	2	$0xx$	4	0–3
1	3	$10xxx$	8	4–11
2	4	$110xxxx$	16	12–27
3	5	$1110xxxxx$	32	28–59
	...	...		...
8	10	$\underbrace{11...1}_{8}\underbrace{xx...x}_{10}$	1024	1020–2043
		Total	2044	

n	$a = 10+n\cdot 2$	nth codeword	Number of codewords	Range of integers
0	10	$0\underbrace{x...x}_{10}$	$2^{10} = 1K$	0–1023
1	12	$10\underbrace{xx...x}_{12}$	$2^{12} = 4K$	1024–5119
2	14	$11\underbrace{xx...xx}_{14}$	$2^{14} = 16K$	5120–21503
		Total	21504	

Table 2.6: The General Unary Code (10,2,14).

6- Static Huffman coding

We first look at Huffman's 'bottom-up' approach. Here we begin with a list of symbols as the tree leaves. The symbols are repeatedly combined with other symbols or subtrees, two items at a time, to form new subtrees. The subtrees grow in size by combination on each iteration until the final combination before reaching the root.

In order to easily find the two items with the smallest frequency, we maintain a sorted list of items in descending order. With minor changes, the method also works if an ascending order list is maintained.

Figure 1.10 shows how the tree is built from the leaves to the root step by step. It carries out the following steps in each iteration:

1. Combine the last two items which have the minimum frequencies or probabilities on the list and replace them by a combined item.
2. The combined item, which represents a subtree, is placed accordingly to its combined frequency on the sorted list.

For example, in Figure 1.10(1), the two symbols S and N (in shade) with the least frequencies are combined to form a new combined item SN with a frequency 2. This is the frequency sum of two singleton symbols S and N. The combined item SN is then inserted to the second position in Figure 1.10(2) in order to maintain the sorted order of the list.

Note there may be more than one possible place available. For example, in Figure 1.10(2), SN with a frequency of 2 can also be inserted immediately before symbol I, or before E. In this case, we always place the newly combined item to a highest possible position to avoid it getting combined again too soon. So SN is placed before L.

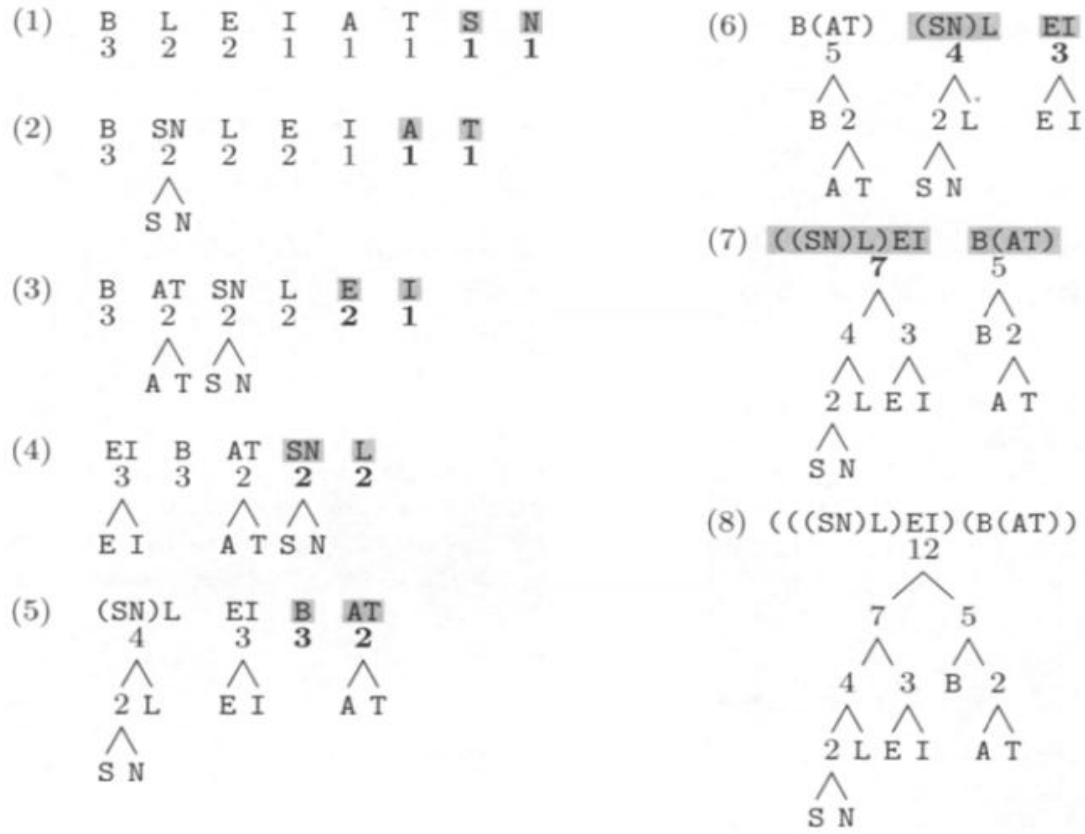


Figure 1.10: Building a Huffman tree

Generalising from the example, we derive the following algorithm for building the tree:

- **Building the binary tree**

  Sort alphabet in descending order $S = (s_1, s_2, \ldots, s_n)$ according to the associated probability distribution.$P = (p_1, p_1, \ldots, p_n)$. Each si represents the root of a subtree. Repeat the following until there is only one composite symbol in S:

  i- If there is one symbol, the tree is the root and the leaf. Otherwise, take two symbols s_i and s_j in the alphabet which have the lowest probabilities pi and pj.

  ii- Remove s_i and s_j from the alphabet and add a new combined symbol (s_i, s_j) with probability p_i + P_j. The new symbol represents the root of a subtree. Now the alphabet contains one fewer symbol than before.

  iii- Insert the new symbol (s_i, s_j) to a highest possible position so the alphabet remains the descending order.

- **Generating the prefix code**

Once we have the binary tree, it is easy to assign a 0 to the left branch and a 1 to the right branch for each internal node of the tree as in Figure 1.11. The 0-1 values marked next to the edges are usually called the weights of the tree. A tree with these 0-i labels is called a weighted tree. The weighted binary tree derived in this way is called a Huffman tree.

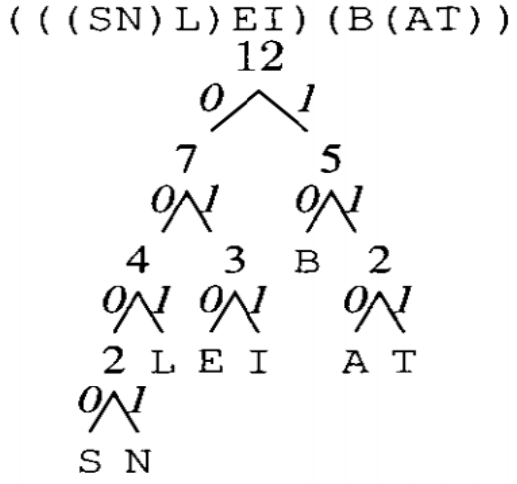


Figure 1.11: A Huffman tree

We then, for each symbol at a leaf, collect the 0 or 1 bit while traversing each tree path from the root to the leaf. When we reach a leaf, the collection of the 0s and Is forms the prefix code for the symbol at that leaf. The codes derived in this way are called Huffman codes.

For example, the collection of the 0s and Is from the root to leaf for symbol E is first a left branch 0, then right branch 1 and finally left branch 0. Therefore, the codeword for symbol E is 010. Traversing in this way for all the leaves, we derive the prefix code (i0 001 010 011 II0 III 0000 0001) for the whole alphabet (B, L, E, I, A, T, S, N) respectively. A prefix code generated in this way is called Huffman code.

7- Shannon-Fano Coding

Shannon-Fano coding, named after Claude Shannon and Robert Fano, was the first algorithm to construct a set of the best variable-size codes. We start with a set of n symbols with known probabilities (or frequencies) of occurrence. The symbols are first arranged in descending order of their probabilities.

The set of symbols is then divided into two subsets that have the same (or almost the same) probabilities. All symbols in one subset get assigned codes that start with a 0, while the codes of the symbols in the other subset start with a 1. Each subset is then recursively divided into two subsubsets of roughly equal probabilities, and the second bit of all the codes is determined in a similar way. When a subset contains just two symbols, their codes are distinguished by adding one more bit to each. The process continues until no more subsets remain.

Table 1.1 illustrates the Shannon-Fano algorithm for a seven-symbol alphabet. Notice that the symbols themselves are not shown, only their probabilities.

The first step splits the set of seven symbols into two subsets, one with two symbols and a total probability of 0.45 and the other with the remaining five symbols and a total probability of 0.55. The two symbols in the first subset are assigned codes that start with 1, so their final codes are 11 and 10. The second subset is divided, in the second step, into two symbols (with total probability 0.3 and codes that start with 01) and three symbols (with total probability 0.25 and codes that start with 00). Step three

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.20	1	0			:10
3.	0.15	0		1	1	:011
4.	0.15	0		1	0	:010
5.	0.10	0		0	1	:001
6.	0.10	0		0	0	1 :0001
7.	0.05	0		0	0	0 :0000

Table 1.1: Shannon-Fano Example

divides the last three symbols into 1 (with probability 0.1 and code 001) and 2 (with total probability 0.15 and codes that start with 000). The average size of this code is 0.25 × 2+0.20 × 2+0.15 × 3+0.15 × 3+0.10 × 3 + 0.10 × 4+0.05 × 4=2.7 bits/symbol. This is a good result because the entropy (the smallest number of bits needed, on average, to represent each symbol) is

$$-\big(0.25\log_2 0.25+0.20\log_2 0.20+0.15\log_2 0.15+0.15\log_2 0.15 \\ +0.10\log_2 0.10+0.10\log_2 0.10+0.05\log_2 0.05\big)\approx 2.67.$$

Example 2.20

Repeat the calculation above but place the first split between the third and fourth symbols. Calculate the average size of the code and show that it is greater than 2.67 bits/symbol.

The code in the table in the answer to Exercise 2.11 has longer average size because the splits, in this case, were not as good as those of Table 2.14. This suggests that the Shannon-Fano method produces better code when the splits are better, i.e., when the two subsets in every split have very close total probabilities. Carrying this argument to its limit suggests that perfect splits yield the best code. Table 2.15 illustrates such a case. The two subsets in every split have identical total probabilities, yielding a code with the minimum average size (zero redundancy). Its average size is 0.25 × 2+0.25 × 2+0.125 × 3+0.125 × 3+0.125 × 3+0.125 × 3=2.5 bits/symbols, which is identical to its entropy. This means that it is the theoretical minimum average size.

	Prob.	Steps			Final
1.	0.25	1	1		:11
2.	0.25	1	0		:10
3.	0.125	0	1	1	:011
4.	0.125	0	1	0	:010
5.	0.125	0	0	1	:001
6.	0.125	0	0	0	:000

The conclusion is that this method produces the best results when the symbols have probabilities of occurrence that are (negative) powers of 2.

- **Compression**

Given a list of symbols, the algorithm involves the following steps:

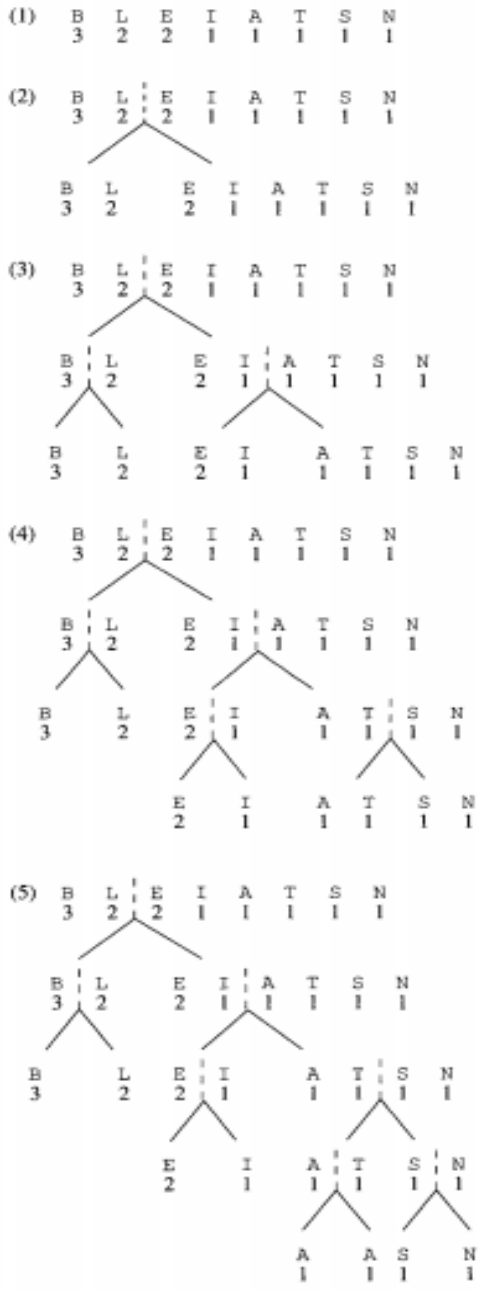
i- Develop a frequency (or probability) table

ii- Sort the table according to frequency (the most frequent one at the top)

iii- Divide the table into two halves with similar frequency counts

iv- Assign the upper half of the list a 0 and the lower half a 1

v- Recursively apply the step of division (3) and assignment (5) to the two halves, subdividing groups and adding bits to the codewords until each symbol has become a corresponding leaf on the tree.



This can be further refined to Shannon-Fano(S) algorithm.

Algorithm 4.5 Shannon-Fano encoding ideas

```
1: if S consists of only two symbols then
2:    add 0 to the codeword of the first symbol and 1 to the second
3: else
4:    if S has more than three symbols then
5:       divide S into 2 subsequence S1 and S2 with minimum probability difference
6:       add a 0 to extend the codeword for each symbol in S1 and a 1 to those in S2
7:       Shannon-Fano(S1)
8:       Shannon-Fano(S2)
9:    end if
10: end if
```

Example 2.20

Suppose the sorted frequency table below is drawn from a source. Derive the Shannon-Fano code.

Symbol	A	B	C	D	E
Frequency	15	7	6	6	5

Solution

1. First division:
   (a) Divide the table into two halves so the sum of the frequencies of each half is as close as possible.

Symbol	A	B	C	D	E
Frequency	15	7	6	6	5
sum(s)	(22)		(17)		

   (b) Assign one bit of the symbol (e.g. upper group 0s and the lower 1s).

Symbol	A	B	C	D	E
Frequency	15	7	6	6	5
sum(s)	(22)		(17)		
codewords	0		1		

2. Second division:
   Repeat the above recursively to each group.

Symbol	A	B,	C	D	E
Frequency	15	7,	6	6	5
sum(s)	(15)	(7),	(6)	(11)	
codewords	00	01,	10	11	

3. Third division:

Symbol	A,	B,	C,	D	E
Frequency	15,	7,	6,	6	5
codewords	00,	01,	10,	110	111

4. So we have the following code (consisting of five codewords) when the recursive process ends:

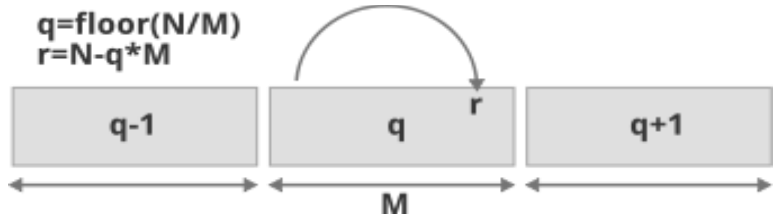
A	B	C	D	E
00	01	10	110	111

8- Golomb coding

Golomb coding is a lossless data compression method using a family of data compression codes invented by Solomon W. Golomb in the 1960s. Alphabets following a geometric distribution will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values.

- **Construction of codes**

Golomb coding uses a tunable parameter M to divide an input value N into two parts: q, the result of a division by M, and r, the remainder. When M=1 Golomb coding is equivalent to unary coding.



Golomb–Rice codes can be thought of as codes that indicate a number by the position of the bin (q), and the offset within the bin (r). The above figure shows the position q, and offset r for the encoding of integer N using Golomb–Rice parameter M. Formally, the two parts are given by the following expression, where x is the number being encoded: $q = \left\lfloor \frac{(x-1)}{M} \right\rfloor$ and $r = x - qM - 1$ The final result looks like: $(q+1)\,r$

Note that r can be of a varying number of bits, and is specifically only b bits for Rice code, and switches between b-1 and b bits for Golomb code (i.e. M is not a power of 2): Let $b = \lceil \log_2(M) \rceil$. If $0 \le r < 2^b - M$, then use b-1 bits to encode r.

If $2^b - M \le r < M$, then use b bits to encode r. Clearly, $b = \log_2(M)$ if M is a power of 2 and we can encode all values of r with b bits

- Simple algorithm

Note below that this is the Rice-Golomb encoding, where the remainder code uses simple truncated binary encoding, also named "Rice coding" (other varying-length binary encodings, like arithmetic or Huffman encodings, are possible for the remainder codes, if the statistic distribution of remainder codes is not flat, and notably when not all possible remainders after the division are used). In this algorithm, if the M parameter is a power of 2, it becomes equivalent to the simpler Rice encoding.

1. Fix the parameter M to an integer value.
2. For N, the number to be encoded, find
    1. quotient = q = int[N/M]
    2. remainder = r = N modulo M
3. Generate Codeword
    1. The Code format : <Quotient Code><Remainder Code>, where
    2. Quotient Code (in unary coding)
        1. Write a q-length string of 1 bits
        2. Write a 0 bit
    3. Remainder Code (in truncated binary encoding)
        1. If M is power of 2, code remainder as binary format.
           So $\log_2(M)$ bits are needed. (Rice code)
        2. If M is not a power of 2, set $b = \lceil \log_2(M) \rceil$
            1. If $r < 2^b - M$ code r as plain binary using b-1 bits.
            2. If $r \geq 2^b - M$ code the number $r + 2^b - M$ in plain binary representation using b bits.

Example 2.20

Set M = 10. Thus $b = \lceil \log_2(10) \rceil = 4$. The cutoff is $2^b - M = 16 - 10 = 6$

Encoding of quotient part	
q	output bits
0	0
1	10
2	110
3	1110
4	11110
5	111110
6	1111110
⋮	⋮
N	$\underbrace{111\cdots111}_{N}0$

Encoding of remainder part			
r	offset	binary	output bits
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	12	1100	1100
7	13	1101	1101
8	14	1110	1110
9	15	1111	1111

For example, with a Rice-Golomb encoding of parameter M = 10, the decimal number 42 would first be split into q = 4,r = 2, and would be encoded as qcode(q),rcode(r) = qcode(4),rcode(2) = 11110,010 (you don't need to encode the separating comma in the output stream, because the 0 at the end

of the q code is enough to say when q ends andr begins ; both the qcode and rcode are self-delimited).

9- Asymmetric Numeral System

Asymmetric Numeral System was introduced by Jarek Duda for several different purposes. We consider here only data compression aspect. Those who want more information can try to read latest Jareks article The first demonstration of ANS coder was published by author atWolfram in March 2008 that means it is not likely that this method will ever be patented because time for filing patent is running out and author confirmed several times that he has no intent to patent this method.

Encoding/decoding table

state	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		

12	27		
13	29		
14	31		

According to the method we need to create single look up table as it is shown above. Encoding may start from any row and from any symbol of the alphabet (A,B,C). If first symbol in the message is B and first row is 1 we take new row number from crossing row 1 and column B, which is 3 and move to the next row. If the next symbol is A we read next row number form the crossing of column A and row 3. This number is 7 and we can go on until the final extend of our table.

Decoding is as elementary as encoding. Having any number, let say 16, we can extract symbol A and previous row 7. Moving back to the 7th element in a table we can extract another symbol A and another new row 3. And finally from element 3 we extract B and row 1, which is the highest point at the table and end of message. Successful encoding and decoding is possible if

- Left column in the table contains all sequential numbers.
- All numbers in columns A,B,C are unique (no repetition).
- Numbers in every row are larger than the number of that row.
- Numbers in all columns are sorted.

The number of row obtained after every step is called state. Having state we can restore the message. This encoder becomes entropy encoder when numbers in columns are selected to match probability of occurrence of correspondent symbol if we divide row number by the value in the column. In this table if we divide row number by any element sitting in column A we have approximately 0.45. The ratio for B is 0.35 and for C is 0.2. The state is growing on each step according to formula **statei = statei-1 / P{symbol}** that means at the end of encoding we shall have number with the length equal to **1 / [P{B} * P{A} * P{A} ...]** which is, obviously, in accordance with entropy. The only part that is missed in this brief introduction is how to manage the state when we reach the bottom of the table. That can be read in the article which also have ANSCoder example. In addition to what is already explained we need to add management of bits output in encoding in the way synchronized

with decoding. Let us get back to our example. This concept is close or similar to the one used in original sample of ANSCoder written by Jarek Duda. According to the concept decoder reads certain number of bits to make the state buffer of the same size before taking of every step. As you can see from the table, in decoding the state can becomes smaller on every step, so if decoder read always any number of bits to make it of the same size we never make the wrong choice and decoding becomes very simple. Read bits, process symbol, read bits, process symbol and so on.
Let us show the decoding process before we explain an encoding one. Presume we have following binary expression to decode:**11000011110** . According to main property of ANS coder we decode backwards. Based on the size of our table we appoint the size of the state buffer 5 bits. On the first step we read last 5 bits of encoded expression shown above, which is **11110** and address the table with this number, which is 30. We extract symbol *C* and obtain new state 6 or in binary format **110** . In order to make our new state buffer match size 5 we read another 2 bits from the expression and get **11000**. which is 24. For state=24 we extract symbol *A* and make new state 11, which is in binary format **1011**. We read one more bit and get (**10110**) 22. Then we do the same thing: we extract B, get state=8, read one more bit, get 16, extract *A*, get 7, read last two bits and get end-of-message marker **11111**. We know that this is end of message because no more bits to read and state match the default marker.

Encoding/decoding table

state	*A*	B	*C*
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30

Encoding process

state before output of bits	31	16	22
output bits	11	0	0
state after bit output	7	8	11
processed symbol	*A*	B	*A*
state after processing	16	22	24

7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

The decoding process is much simpler than encoding. In encoding we need to output bits to provide this simple rule that we used in decoding. The encoding process is shown in the table on the left. We can start from any 5 bit number but we prefer distinguished end/start marker 31 (**11111**). The concept for encoding is a little-bit more complicated. We output bits before processing a symbol. As you can see we can not process 31, there is no row number 31, so we output as many bits as necessary in order to process the symbol. We can find that out because we know what symbol we are processing. It is A, and we need to output two least significant bits, which is**11**. We show the whole process in the table. The number before we output bits is 31. Below this number are two bits that we output to the stream. The result after output state=7. We process 7 with symbol A and get new state=16. Next symbol is B. According to the table we can not process B with state=16, so we need to output one least significant bit **0** and make new state=8. After processing B for the state=8 we get 22 and so on. We have to output least significant bits because we need to reduce the number. After we process our very last symbol C we have state=30 and we output it completely so our output buffer looks like it is shown above.