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
	$\cdots$	$\cdots$		$\cdots$
8	10	$\underbrace{11\ldots1}_{8}\underbrace{xx\ldots x}_{10}$	1024	1020–2043
		Total	2044	

Table 2.5: The General Unary Code (2,1,10).

The ***number of different general unary codes*** is:

$$\frac{2^{\text{stop}} + 2^{\text{step}} - 2^{\text{start}}}{2^{step} - 1}$$

Notice that this expression increases exponentially with parameter "stop," so large sets of these codes can be generated with small values of the three parameters.

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.

Shannon-Fano coding Steps:

1. The ***symbols*** are first arranged in ***descending*** order of their probabilities.

2. The set of symbols is then ***divided*** into ***two subsets*** that have the ***same*** (or ***almost*** the same) ***probabilities***.

3. ***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***.

4. ***Each subset*** is then ***recursively*** divided into ***two sub-subsets*** of ***roughly equal probabilities***, and the ***second bit*** of ***all the codes*** is determined in a ***similar way***.

5. When a ***subset*** contains just ***two symbols***, their codes are ***distinguished*** by adding ***one more bit to each***.

6. The process ***continues*** until ***no more subsets*** remain.

Example

Table 2.14 illustrates the Shannon-Fano algorithm for a seven-symbol alphabet. Notice that the symbols themselves are not shown, only their probabilities.

	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 2.14: Shannon-Fano Example.

Solution:

- 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 ***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:

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

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***).

The ***entropy is:***

$$2(-0.25\log_2 0.25) + 4(-0.125 log_2 0.125) = \mathbf{2.508}\ \boldsymbol{bits/symbol}$$

The ***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

Table 2.15: Shannon-Fano Balanced Example.

The ***conclusion*** is that this method produces the best results when the symbols have ***probabilities of occurrence that are (negative) powers of 2***.
The ***Shannon-Fano*** method is ***easy to implement*** but the ***code it produces*** is generally ***not as good*** as that produced by the ***Huffman method***.

Move-to-Front Coding

The idea of MtF is to encode a symbol with a '0' as long as it is a recently repeating symbol. In this way, if the source contains a long run of identical symbols, the run will be encoded as a long sequence of zeros.

 Initially, the alphabet of the source is stored in an array and the index of each symbol in the array is used to encode a corresponding symbol. On each iteration, a new character is read and the symbol that has just been encoded is moved to the front of the array.

Algorithm

Input: set of alphabet and input sequence.

Output: Encoded sequence

Step 1: Store the source alphabet in an array.

Step 2: Repeat steps 3-5 until the end of the input sequence.

Step 3: Read a new character from the input sequence.

Step 4: Encode the character by its array's index.

Step 5: Move the character to the front of the array.

Step 6: Terminate.

Example: Suppose that the following sequence of symbols is to be compressed:

DDCBEEEFGGAA from a source alphabet (A, B, C, D, E, F, G). Show how the MtF method works.

Encoding

Initially, the alphabet is stored in an array:

0	1	2	3	4	5	6
A	B	C	D	E	F	G

1. Read D, the first symbol of the input sequence. Encode D by index 3 of the array, and then move D to the front of the array:

0	1	2	3	4	5	6
D	A	B	C	E	F	G

2. Read D. Encode D by its index 0, and leave the array unchanged because D is already at the front position of the array.

0	1	2	3	4	5	6
D	A	B	C	E	F	G

3. Read C. Encode C by its index 3, and move C to the front:

0	1	2	3	4	5	6
C	D	A	B	E	F	G

4. Read B. Encode it by 3 and move B to the front:

0	1	2	3	4	5	6
B	C	D	A	E	F	G

5. Read E. Encode it by 4 and move E to the front:

0	1	2	3	4	5	6
E	B	C	D	A	F	G

⋮

and so on.

This process continues until the entire string is processed. Hence the encoding is 3, 0, 3, 3, 4, ⋯ .

In this way, the more frequently occurring symbols are encoded by 0 or small decimal numbers.

Decoding

Read the following codes: 3, 0, 3, 3, 4, · · · .
Initially,

0	1	2	3	4	5	6
A	B	C	D	E	F	G

1. Read 3, decode it to D, and move it to the front of the array:

0	1	2	3	4	5	6
D	A	B	C	E	F	G

2. Read 0, decode it to D, and do nothing since D is already at the front.

0	1	2	3	4	5	6
D	A	B	C	E	F	G

3. Read 3, decode it to C, and move it to the front.

0	1	2	3	4	5	6
C	D	A	B	E	F	G

⋮

and so on.

Arithmetic Coding Algorithm:

The main steps of arithmetic coding Algorithm:

1. Start by defining the "current interval" as [0, 1).
2. Repeat the following two steps for each symbol *s* in the input stream:
2.1. Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities.
2.2. Select the subinterval for *s* and define it as the new current interval.
3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval (i.e., any number inside the
current interval).

The next example is a little more involved. We show the compression steps for the short string "SWISS_MISS"

As more symbols are being input and processed, Low and High are being updated according to

NewHigh:=OldLow+Range*HighRange(X);
NewLow:=OldLow+Range*LowRange(X);