12.4 Text Compression and the Greedy Method

In this section, we consider an important text processing task, ***text compression***. In this problem, we are given a string X defined over some alphabet, such as the ASCII or Unicode character sets, and we want to efficiently encode X into a small binary string Y (using only the characters 0 and 1). Text compression is useful in any situation where we are communicating over a low-bandwidth channel, such as a modem line or infrared connection, and we wish to minimize the time needed to transmit our text. Likewise, text compression is also useful for storing collections of large documents more efficiently, in order to allow for a fixed-capacity storage device to contain as many documents as possible.

The method for text compression explored in this section is the ***Huffman code***. Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters (with 7 bits in the ASCII system and 16 in the Unicode system). A Huffman code, on the other hand, uses a variable-length encoding optimized for the string X. The optimization is based on the use of character ***frequencies***, where we have, for each character c, a count $f(c)$ of the number of times c appears in the string X. The Huffman code saves space over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.

To encode the string X, we convert each character in X from its fixed-length code word to its variable-length code word, and we concatenate all these code words in order to produce the encoding Y for X. In order to avoid ambiguities, we insist that no code word in our encoding is a prefix of another code word in our encoding. Such a code is called a ***prefix code***, and it simplifies the decoding of Y in order to get back X. (See Figure 12.8.) Even with this restriction, the savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

Huffman's algorithm for producing an optimal variable-length prefix code for X is based on the construction of a binary tree T that represents the code. Each node in T, except the root, represents a bit in a code word, with each left child representing a "0" and each right child representing a "1." Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the nodes in the path from the root of T to v. (See Figure 12.8.) Each external node v has a ***frequency***, $f(v)$, which is simply the frequency in X of the character associated with v. In addition, we give each internal node v in T a frequency, $f(v)$, that is the sum of the frequencies of all the external nodes in the subtree rooted at v.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

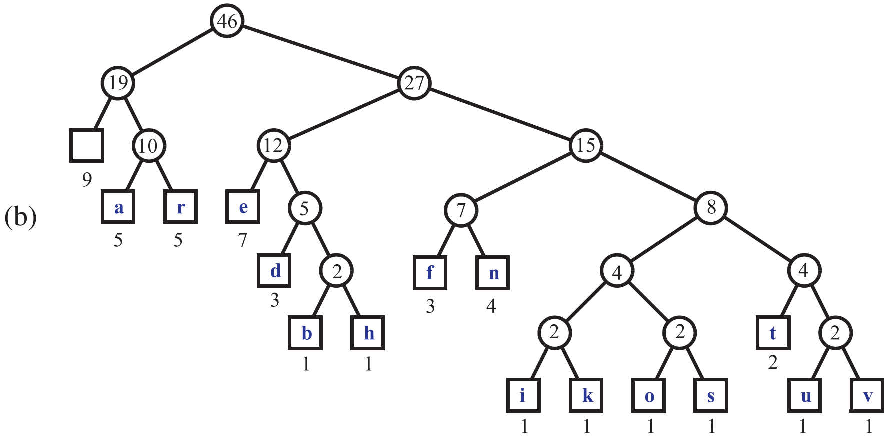


Figure 12.8: An example Huffman code for the input string X = `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for “a” is 010, and the code for “f” is 1100.

12.4.1 The Huffman-Coding Algorithm

The Huffman-coding algorithm begins with each of the d distinct characters of the string X to encode being the root node of a single-node binary tree. The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree. It repeats this process until only one tree is left. (See Code Fragment 12.9.)

Each iteration of the **while** loop in Huffman’s algorithm can be implemented in $O(\log d)$ time using a priority queue represented with a heap. In addition, each iteration takes two nodes out of Q and adds one in, a process that is repeated $d-1$ times before exactly one node is left in Q. Thus, this algorithm runs in $O(n+d\log d)$ time. Although a full justification of this algorithm’s correctness is beyond our scope, we note that its intuition comes from a simple idea—any optimal code can be converted into an optimal code in which the code words for the two lowest-frequency characters, a and b, differ only in their last bit. Repeating the argument for a string with a and b replaced by a character c, gives the following.

Proposition 12.7: *Huffman’s algorithm constructs an optimal prefix code for a string of length n with d distinct characters in $O(n+d\log d)$ time.*

```
Algorithm Huffman($X$):
    Input: String $X$ of length $n$ with $d$ distinct characters
    Output: Coding tree for $X$
    Compute the frequency $f(c)$ of each character $c$ of $X$.
    Initialize a priority queue $Q$.
    for each character $c$ in $X$ do
        Create a single-node binary tree $T$ storing $c$.
        Insert $T$ into $Q$ with key $f(c)$.
    while $Q$.size() $> 1$ do
        $f_1 \leftarrow Q$.min()
        $T_1 \leftarrow Q$.removeMin()
        $f_2 \leftarrow Q$.min()
        $T_2 \leftarrow Q$.removeMin()
        Create a new binary tree $T$ with left subtree $T_1$ and right subtree $T_2$.
        Insert $T$ into $Q$ with key $f_1 + f_2$.
    return tree $Q$.removeMin()
```

Code Fragment 12.9: Huffman-coding algorithm.

12.4.2 The Greedy Method

Huffman's algorithm for building an optimal encoding is an example application of an algorithmic design pattern called the ***greedy method***. This design pattern is applied to optimization problems, where we are trying to construct some structure while minimizing or maximizing some property of that structure.

The general formula for the greedy method pattern is almost as simple as that for the brute-force method. In order to solve a given optimization problem using the greedy method, we proceed by a sequence of choices. The sequence starts from some well-understood starting condition, and computes the cost for that initial condition. The pattern then asks that we iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible. This approach does not always lead to an optimal solution.

But there are several problems that it does work for, and such problems are said to possess the ***greedy-choice*** property. This is the property that a global optimal condition can be reached by a series of locally optimal choices (that is, choices that are each the current best from among the possibilities available at the time), starting from a well-defined starting condition. The problem of computing an optimal variable-length prefix code is just one example of a problem that possesses the greedy-choice property.

Huffman Code

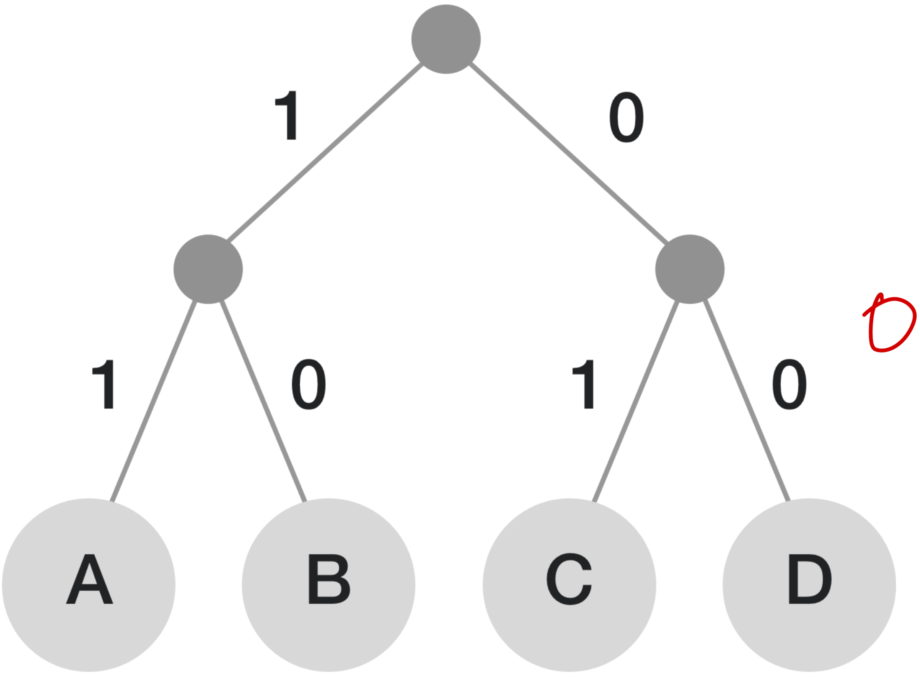
Huffman coding is an efficient method of compressing data without losing information. In computer science, inform encoded as bits—1's and 0's. Strings of bits encode the information that tells a computer which instructions to carry games, photographs, movies, and more are encoded as strings of bits in a computer. Computers execute billions of i per second, and a single video game can be billions of bits of data. It is easy to see why efficient and unambiguous ir encoding is a topic of interest in computer science.

Huffman coding provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear message. Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as n encoded as longer strings. Since the frequencies of symbols vary across messages, there is no one Huffman coding t work for all messages. This means that the Huffman coding for sending message X may differ from the Huffman cod send message Y. There is an algorithm for generating the Huffman coding for a given message based on the frequer symbols in that particular message.

Huffman coding works by using a frequency-sorted binary tree to encode symbols.

TRY IT YOURSELF

Use the encoding described in the tree below to encode "CAB".



○ 110100

[✓ 011110]

○ 001111

○ 100100

0 1 1 1 1 0

Information Encoding

In information theory, the goal is usually to transmit information in the fewest bits possible in such a way that each ei unambiguous. For example, to encode A, B, C, and D in the fewest bits possible, each letter could be encoded as “1”. with this encoding, the message “1111” could mean “ABCD” or “AAAA”—it is ambiguous.

Encodings can either be fixed-length or variable-length.

A **fixed-length encoding** is where the encoding for each symbol has the same number of bits. For example:

A	00
B	01
C	10
D	11

A **variable-length encoding** is where symbols can be encoded with different numbers of bits. For example:

A	000
B	1
C	110
D	1111

EXAMPLE

Explain why the encoding scheme below is a poor encoding.

A	B	C	D
0	1	11	10

Let's try a few encodings:

- Write the binary encoding of the message BAC. $\implies$ 1011
- Write the binary encoding of the message DC. $\implies$ 1011
- Write the binary encoding of the message DDC. $\implies$ 101011
- Write the binary encoding of the message BADBB. $\implies$ 101011

As you can see from the encodings of the messages above, there are cases where the same binary string encodes c messages. This encoding scheme is ambiguous because after we encode one message, its binary representation is unique to that message only. For example, with the encoding above, "BB" is encoded as "11" but "11" is also the en "C". □

TRY IT YOURSELF

Which of the choices cannot be encoded as 01110 under the following encoding scheme?

A	0
B	10
C	1

○ ADB

○ ACCCA

[✓ ○ ADBA]

○ ADCA

D	11

○ ACCB

Note: Recall from Huffman encoding that ambiguous encodings are generally bad encodings.

EXAMPLE

The encoding below is unambiguous, but inefficient since there exists a shorter (and still unambiguous) encoding letters. Give an equivalent but shorter encoding.

A	B	C	D
0000	0101	1010	1111

Each letter has the same number of bits that encodes it. Therefore, we know that every 4 bits in a string will encode Unlike the example above (with variable-length encoding), we do not have to worry about ambiguity because each has a unique binary assignment of the same length. For example, if B were encoded as 1 and C were encoded as 11, seeing a message 11 we could not tell if the sender intended to send BB or C. With the fixed-length encoding above no possible overlap between encodings: 0000 will always be read as an A, and 1010 will always be read as a C.

However, these encodings are twice as long as they need to be: there is redundant information since the four-bit e are just the same two-bit encoding repeated.

To represent x symbols, we need $\log_2(x)$ bits because each bit has two possible values: 1 or 0. Therefore, with x bit represent 2^x different symbols.

This problem has 4 symbols, so we only need $\log_2(4) = 2$ bits to do this. Therefore, our encoding can be as follow

A	B	C	D
00	01	10	11

How many bits are needed to represent x different symbols (using fixed-length encoding)?

To represent x symbols with a fixed-length encoding, $\log_2(x)$ bits are needed because each bit has two possible v 0. Therefore, with x bits, 2^x different symbols can be represented.

Huffman Code

Huffman code is a way to encode information using variable-length strings to represent symbols depending on how they appear. The idea is that symbols that are used more frequently should be shorter while symbols that appear mo can be longer. This way, the number of bits it takes to encode a given message will be shorter, on average, than if a fi code was used. In messages that include many rare symbols, the string produced by variable-length encoding may b than one produced by a fixed-length encoding.

As shown in the above sections, it is important for an encoding scheme to be unambiguous. Since variable-length er are susceptible to ambiguity, care must be taken to generate a scheme where ambiguity is avoided. Huffman coding greedy algorithm to build a prefix tree that optimizes the encoding scheme so that the most frequently used symbol shortest encoding. The prefix tree describing the encoding ensures that the code for any particular symbol is never a the bit string representing any other symbol. To determine the binary assignment for a symbol, make the leaves of th correspond to the symbols, and the assignment will be the path it takes to get from the root of the tree to that leaf.

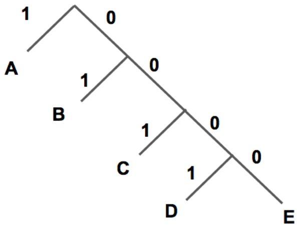
TRY IT YOURSELF

What encoding do we get from the following Huffman tree?

○ Encoding #1

○ Encoding #2

○ Encoding #3



Symbol	Encoding #1
A	0000
B	01
C	101
D	1
E	0000

Symbol	Encoding #2
A	1
B	01
C	001
D	0001
E	0000

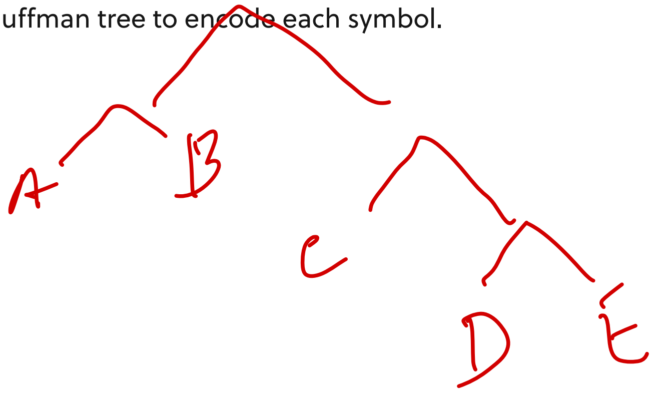
✓

Symbol	Encoding #3
A	1
B	1
C	1
D	1
E	0

The Huffman coding algorithm takes in information about the frequencies or probabilities of a particular symbol occ begins to build the prefix tree from the bottom up, starting with the two least probable symbols in the list. It takes th symbols and forms a subtree containing them, and then removes the individual symbols from the list. The algorithm probabilities of elements in a subtree and adds the subtree and its probability to the list. Next, the algorithm searche and selects the two symbols or subtrees with the smallest probabilities. It uses those to make a new subtree, removes original subtrees/symbols from the list, and then adds the new subtree and its combined probability to the list. This r until there is one tree and all elements have been added.

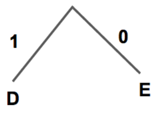
EXAMPLE

Given the following probability table, create a Huffman tree to encode each symbol.



Symbol	Probability
A	0.3
B	0.3
C	0.2
D	0.1
E	0.1

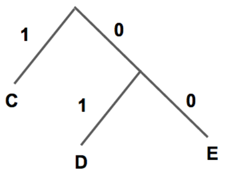
The two elements with the smallest probability are D and E. So we create the subtree:



And update the list to include the subtree DE with a probability of $0.1 + 0.1 = 0.2$:

Symbol	Probability
A	0.3
B	0.3
C	0.2
DE	0.2

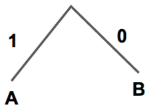
The next two smallest probabilities are DE and C, so we create the subtree:



And update the list to include the subtree CDE with a probability of $0.2 + 0.2 = 0.4$:

Symbol	Probability
A	0.3
B	0.3
CDE	0.4

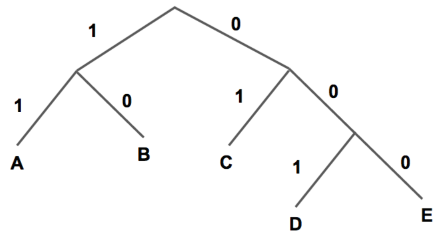
The next two smallest probabilities are A and B, so we create the subtree:



And update the list to include the subtree AB with a probability of $0.3 + 0.3 = 0.6$:

Symbol	Probability
AB	0.6
CDE	0.4

Now, we only have two elements left, so we build the subtree:



The probability of ABCDE is 1, which is expected since one of the symbols will occur.

Here are the encodings we get from the tree:

Symbol	Encoding
A	11
B	10
C	01
D	001
E	000

The Huffman Coding Algorithm

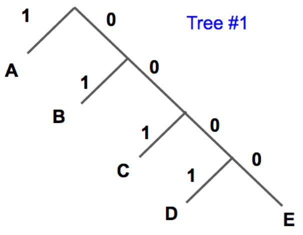
- Take a list of symbols and their probabilities.
- Select two symbols with the lowest probabilities (if multiple symbols have the same probability, select two arbitr
- Create a binary tree out of these two symbols, labeling one branch with a "1" and the other with a "0". It doesn't which side you label 1 or 0 as long as the labeling is consistent throughout the problem (e.g. the left side should be 1 and the right side should always be 0, or the left side should always be 0 and the right side should always be
- Add the probabilities of the two symbols to get the probability of the new subtree.
- Remove the symbols from the list and add the subtree to the list.
- Go back through the list and take the two symbols/subtrees with the smallest probabilities and combine those ir new subtree. Remove the original symbols/subtrees from the list, and add the new subtree to the list.

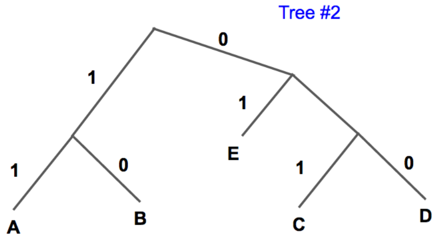
- Repeat until all of the elements are combined.

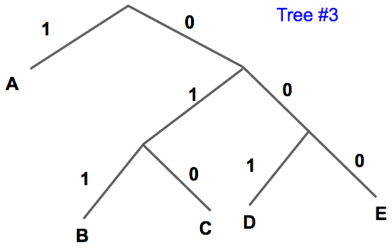
TRY IT YOURSELF

What is the correct Huffman tree for the following symbol/probability set?

Symbol	A	B	C	D	E
Probability	0.6	0.1	0.1	0.1	0.1







○ Tree 2

○ Tree 1

○ Tree 3

Huffman coding is optimal for encoding single characters, but for encoding multiple characters with one encoding, c compression methods are better. Moreover, it is optimal when each input symbol is a known independent and identi distributed random variable having a probability that is the inverse of a power of two. [1]

See Also

- Information Theory
- Entropy (Information Theory)
- Error-correcting Codes
- Greedy Algorithms

References

1. , . *Huffman Coding*. Retrieved July 20, 2016, from https://en.wikipedia.org/wiki/Huffman_coding

Cite as: Huffman Code. *Brilliant.org*. Retrieved 11:51, October 26, 2023, from https://brilliant.org/wiki/huffman-encoding/

Wikipedia
The Free Encyclopedia

Huffman coding

In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code is **Huffman coding**, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".[1]

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.[2] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods - it is replaced with arithmetic coding[3] or asymmetric numeral systems[4] if a better compression ratio is required.

History

In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.[5]

In doing so, Huffman outdid Fano, who had worked with Claude Shannon to develop a similar code. Building the tree from the bottom up guaranteed optimality, unlike the top-down approach of Shannon–Fano coding.

Terminology

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol). Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Problem definition

Informal description

Given

A set of symbols and their weights (usually proportional to probabilities).

Find

A prefix-free binary code (a set of codewords) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root).

Formalized description

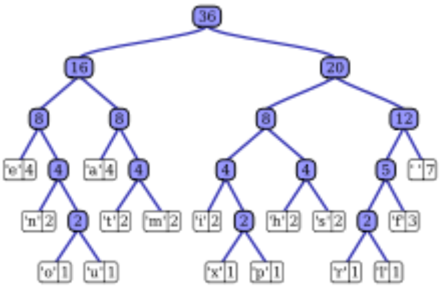
Input.
Alphabet $A = (a_1, a_2, \dots, a_n)$, which is the symbol alphabet of size n.
Tuple $W = (w_1, w_2, \dots, w_n)$, which is the tuple of the (positive) symbol weights (usually proportional to probabilities), i.e. $w_i = \mathrm{weight}(a_i),\ i \in \{1, 2, \dots, n\}$.

Output.
Code $C(W) = (c_1, c_2, \dots, c_n)$, which is the tuple of (binary) codewords, where c_i is the codeword for a_i, $i \in \{1, 2, \dots, n\}$.

Goal.
Let $L(C(W)) = \sum_{i=1}^{n} w_i \,\mathrm{length}(c_i)$ be the weighted path length of code C. Condition: $L(C(W)) \leq L(T(W))$ for any code $T(W)$.

Example

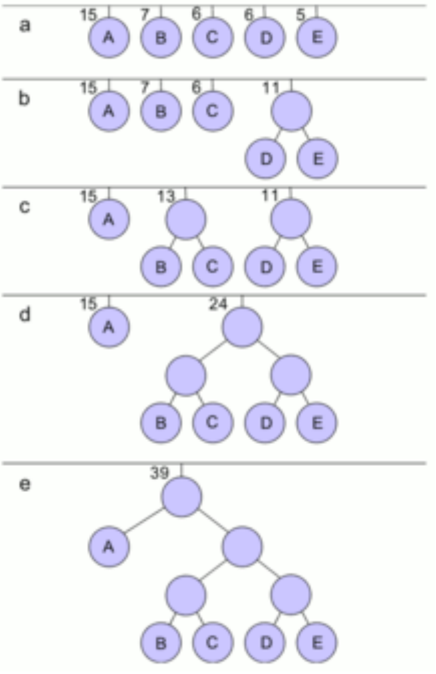
We give an example of the result of Huffman coding for a code with five characters and given weights. We will not verify that it minimizes *L* over all codes, but we will compute *L* and compare it to the Shannon entropy *H* of the given set of weights; the result is nearly optimal.



Huffman tree generated from the exact frequencies of the text "this is an example of a huffman tree". Encoding the sentence with this code requires 135 (or 147) bits, as opposed to 288 (or 180) bits if 36 characters of 8 (or 5) bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.) The frequencies and codes of each character are:

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000

u	1	00111
x	1	10010



Constructing a Huffman Tree

Input (*A*, *W*)	Symbol (a_i)	a	b	c	d	e	**Sum**
	Weights (w_i)	0.10	0.15	0.30	0.16	0.29	= 1
Output *C*	Codewords (c_i)	`010`	`011`	`11`	`00`	`10`	
	Codeword length (in bits) (l_i)	3	3	2	2	2	
	Contribution to weighted path length (l_i w_i)	0.30	0.45	0.60	0.32	0.58	*L*(*C*) = 2.25
Optimality	Probability budget (2^{-l_i})	1/8	1/8	1/4	1/4	1/4	= 1.00
	Information content (in bits) ($-\log_2 w_i$) ≈	3.32	2.74	1.74	2.64	1.79	
	Contribution to entropy (-w_i $\log_2 w_i$)	0.332	0.411	0.521	0.423	0.518	*H*(*A*) = 2.205

For any code that is *biunique*, meaning that the code is uniquely decodeable, the sum of the probability budgets across all symbols is always less than or equal to one. In this example, the sum is strictly equal to one; as a result, the code is termed a *complete* code. If this is not the case, one can always derive an equivalent code by adding extra symbols (with associated null probabilities), to make the code complete while keeping it *biunique*.

As defined by Shannon (1948), the information content *h* (in bits) of each symbol a_i with non-null probability is

$$h(a_i) = \log_2 \frac{1}{w_i}.$$

The entropy H (in bits) is the weighted sum, across all symbols a_i with non-zero probability w_i, of the information content of each symbol:

$$H(A) = \sum_{w_i > 0} w_i h(a_i) = \sum_{w_i > 0} w_i \log_2 \frac{1}{w_i} = -\sum_{w_i > 0} w_i \log_2 w_i.$$

(Note: A symbol with zero probability has zero contribution to the entropy, since $\lim_{w \to 0^+} w \log_2 w = 0$. So for simplicity, symbols with zero probability can be left out of the formula above.)

As a consequence of Shannon's source coding theorem, the entropy is a measure of the smallest codeword length that is theoretically possible for the given alphabet with associated weights. In this example, the weighted average codeword length is 2.25 bits per symbol, only slightly larger than the calculated entropy of 2.205 bits per symbol. So not only is this code optimal in the sense that no other feasible code performs better, but it is very close to the theoretical limit established by Shannon.

In general, a Huffman code need not be unique. Thus the set of Huffman codes for a given probability distribution is a non-empty subset of the codes minimizing $L(C)$ for that probability distribution. (However, for each minimizing codeword length assignment, there exists at least one Huffman code with those lengths.)

Basic technique

Compression

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol and optionally, a link to a **parent** node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a **weight**, links to **two child nodes** and an optional link to a **parent** node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n-1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

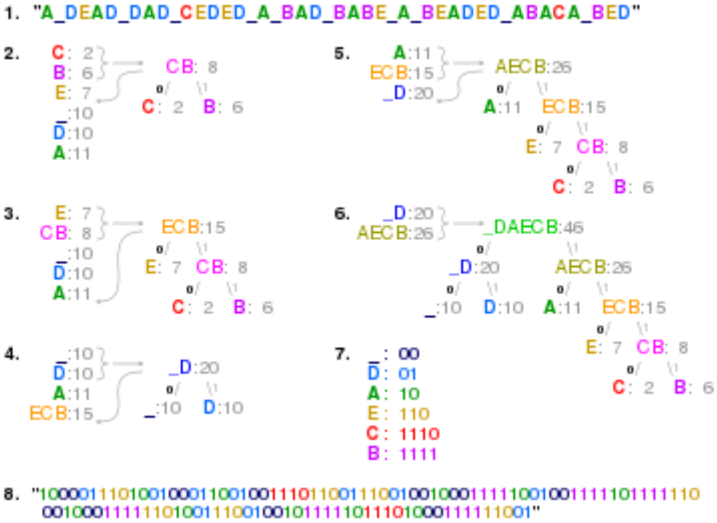
The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   1. Remove the two nodes of highest priority (lowest probability) from the queue

    2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log *n*) time per insertion, and a tree with *n* leaves has 2*n*−1 nodes, this algorithm operates in O(*n* log *n*) time, where *n* is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(*n*)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:



Visualisation of the use of Huffman coding to encode the message "A_DEAD_DAD_CEDED_A_BAD_BABE_A_BEADED_ABACA_BED". In steps 2 to 6, the letters are sorted by increasing frequency, and the least frequent two at each step are combined and reinserted into the list, and a partial tree is constructed. The final tree in step 6 is traversed to generate the dictionary in step 7. Step 8 uses it to encode the message.

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
    1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
    2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
    3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

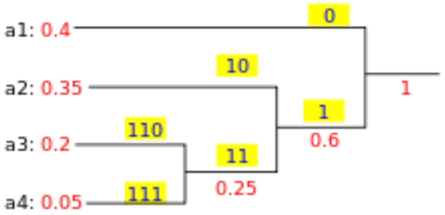
Once the Huffman tree has been generated, it is traversed to generate a dictionary which maps the symbols to binary codes as follows:

1. Start with current node set to the root.
2. If node is not a leaf node, label the edge to the left child as *0* and the edge to the right child as *1*. Repeat the process at both the left child and the right child.

The final encoding of any symbol is then read by a concatenation of the labels on the edges along the path from the root node to the symbol.

In many cases, time complexity is not very important in the choice of algorithm here, since *n* here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when *n* grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.



A source generates 4 different symbols $\{a_1, a_2, a_3, a_4\}$ with probability $\{0.4; 0.35; 0.2; 0.05\}$. A binary tree is generated from left to right taking the two least probable symbols and putting them together to form another equivalent symbol having a probability that equals the sum of the two symbols. The process is repeated until there is just one symbol. The tree can then be read backwards, from right to left, assigning different bits to different branches. The final Huffman code is:

Symbol	Code
a1	0
a2	10
a3	110
a4	111

The standard way to represent a signal made of 4 symbols is by using 2 bits/symbol, but the entropy of the source is 1.74 bits/symbol. If this Huffman code is used to represent the signal, then the average length is lowered to 1.85 bits/symbol; it is still far from the theoretical limit because the probabilities of the symbols are different from negative powers of two.

Decompression

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using canonical encoding, the compression model can be precisely reconstructed with just $B \cdot 2^B$ bits of information (where B is the number of bits per symbol). Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).

Main properties

The probabilities used can be generic ones for the application domain that are based on average experience, or they can be the actual frequencies found in the text being compressed. This requires that a frequency table must be stored with the compressed text. See the Decompression section above for more information about the various techniques employed for this purpose.

Optimality

Huffman's original algorithm is optimal for a symbol-by-symbol coding with a known input probability distribution, i.e., separately encoding unrelated symbols in such a data stream. However, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown. Also, if symbols are not independent and identically distributed, a single code may be insufficient for optimality. Other methods such as arithmetic coding often have better compression capability.

Although both aforementioned methods can combine an arbitrary number of symbols for more efficient coding and generally adapt to the actual input statistics, arithmetic coding does so without significantly increasing its computational or algorithmic complexities (though the simplest version is slower and more complex than Huffman coding). Such flexibility is especially useful when input probabilities are not precisely known or vary significantly within the stream. However, Huffman coding is usually faster and arithmetic coding was historically a subject of some concern over patent issues. Thus many technologies have historically avoided arithmetic coding in favor of Huffman and other prefix coding techniques. As of mid-2010, the most commonly used techniques for this alternative to Huffman coding have passed into the public domain as the early patents have expired.

For a set of symbols with a uniform probability distribution and a number of members which is a power of two, Huffman coding is equivalent to simple binary block encoding, e.g., ASCII coding. This reflects the fact that compression is not possible with such an input, no matter what the compression method, i.e., doing nothing to the data is the optimal thing to do.

Huffman coding is optimal among all methods in any case where each input symbol is a known independent and identically distributed random variable having a probability that is dyadic. Prefix codes, and thus Huffman coding in particular, tend to have inefficiency on small alphabets, where probabilities often fall between these optimal (dyadic) points. The worst case for Huffman coding can happen when the probability of the most likely symbol far exceeds $2^{-1} = 0.5$, making the upper limit of inefficiency unbounded.

There are two related approaches for getting around this particular inefficiency while still using Huffman coding. Combining a fixed number of symbols together ("blocking") often increases (and never decreases) compression. As the size of the block approaches infinity, Huffman coding theoretically approaches the entropy limit, i.e., optimal compression.[6] However, blocking arbitrarily large groups of symbols is impractical, as the complexity of a Huffman code is linear in the number of possibilities to be encoded, a number that is exponential in the size of a block. This limits the amount of blocking that is done in practice.

A practical alternative, in widespread use, is run-length encoding. This technique adds one step in advance of entropy coding, specifically counting (runs) of repeated symbols, which are then encoded. For the simple case of Bernoulli processes, Golomb coding is optimal among prefix codes for coding run length, a fact proved via the techniques of Huffman coding.[7] A similar approach is taken by fax machines using modified Huffman coding. However, run-length coding is not as adaptable to as many input types as other compression technologies.

Variations

Many variations of Huffman coding exist,[8] some of which use a Huffman-like algorithm, and others of which find optimal prefix codes (while, for example, putting different restrictions on the output). Note that, in the latter case, the method need not be Huffman-like, and, indeed, need not even be polynomial time.

n-ary Huffman coding

The ***n*-ary Huffman** algorithm uses the {0, 1,..., *n* − 1} alphabet to encode message and build an *n*-ary tree. This approach was considered by Huffman in his original paper. The same algorithm applies as for binary ($n = 2$) codes, except that the *n* least probable symbols are taken together, instead of just the 2 least probable. Note that for *n* greater than 2, not all sets of source words can properly form an *n*-ary tree for Huffman coding. In these cases, additional 0-probability place holders must be added. This is because the tree must form an *n* to 1 contractor; for binary coding, this is a 2 to 1 contractor, and any sized set can form such a contractor. If the number of source words is congruent to 1 modulo *n*−1, then the set of source words will form a proper Huffman tree.

Adaptive Huffman coding

A variation called **adaptive Huffman coding** involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates. It is used rarely in practice, since the cost of updating the tree makes it slower than optimized adaptive arithmetic coding, which is more flexible and has better compression.

Huffman template algorithm

Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only that the weights form a totally ordered commutative monoid, meaning a way to order weights and to add them. The **Huffman template algorithm** enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition). Such algorithms can solve other minimization problems, such as minimizing $\max_i \left[w_i + \mathrm{length}\left(c_i\right)\right]$, a problem first applied to circuit design.

Length-limited Huffman coding/minimum variance Huffman coding

Length-limited Huffman coding is a variant where the goal is still to achieve a minimum weighted path length, but there is an additional restriction that the length of each codeword must be less than a given constant. The package-merge algorithm solves this problem with a simple greedy approach very similar to that used by Huffman's algorithm. Its time complexity is $O(nL)$, where L is the maximum length of a codeword. No algorithm is known to solve this problem in $O(n)$ or $O(n \log n)$ time, unlike the presorted and unsorted conventional Huffman problems, respectively.

Huffman coding with unequal letter costs

In the standard Huffman coding problem, it is assumed that each symbol in the set that the code words are constructed from has an equal cost to transmit: a code word whose length is *N* digits will always have a cost of *N*, no matter how many of those digits are 0s, how many are 1s, etc. When working under this assumption, minimizing the total cost of the message and minimizing the total number of digits are the same thing.

Huffman coding with unequal letter costs is the generalization without this assumption: the letters of the encoding alphabet may have non-uniform lengths, due to characteristics of the transmission medium. An example is the encoding alphabet of Morse code, where a 'dash' takes longer to send than a 'dot', and therefore the cost of a dash in transmission time is higher. The goal is still to minimize the weighted average codeword length, but it is no longer sufficient just to minimize the number of symbols used by the message. No algorithm is known to solve this in the same manner or with the same efficiency as conventional Huffman coding, though it has been solved by Karp (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1057615&newsearch=true&queryText=Minimum-redundancy%20coding%20for%20the%20discrete%20noiseless%20channel) whose solution has been refined for the case of integer costs by Golin (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=705558&queryText=dynamic%20programming%20golin%20constructing%20optimal%20prefix-free&newsearch=true).

Optimal alphabetic binary trees (Hu–Tucker coding)

In the standard Huffman coding problem, it is assumed that any codeword can correspond to any input symbol. In the alphabetic version, the alphabetic order of inputs and outputs must be identical. Thus, for example, $A = \{a, b, c\}$ could not be assigned code $H(A, C) = \{00, 1, 01\}$, but instead should be assigned either $H(A, C) = \{00, 01, 1\}$ or $H(A, C) = \{0, 10, 11\}$. This is also known as the **Hu–Tucker** problem, after T. C. Hu and Alan Tucker, the authors of the paper presenting the first $O(n \log n)$-time solution to this optimal binary alphabetic problem,[9] which has some similarities to Huffman algorithm, but is not a variation of this algorithm. A later method, the Garsia–Wachs algorithm of Adriano Garsia and Michelle L. Wachs (1977), uses simpler logic to perform the same comparisons in the same total time bound. These optimal alphabetic binary trees are often used as binary search trees.[10]

The canonical Huffman code

If weights corresponding to the alphabetically ordered inputs are in numerical order, the Huffman code has the same lengths as the optimal alphabetic code, which can be found from calculating these lengths, rendering Hu–Tucker coding unnecessary. The code resulting from numerically (re-)ordered input is sometimes called the *canonical Huffman code* and is often the code used in practice, due to ease of encoding/decoding. The technique for finding this code is sometimes called **Huffman–Shannon–Fano coding**, since it is optimal like Huffman coding, but alphabetic in weight probability, like Shannon–Fano coding. The Huffman–Shannon–Fano code corresponding to the example is $\{000, 001, 01, 10, 11\}$, which, having the same codeword lengths as the original solution, is also optimal. But in *canonical Huffman code*, the result is $\{110, 111, 00, 01, 10\}$.

Applications

Arithmetic coding and Huffman coding produce equivalent results — achieving entropy — when every symbol has a probability of the form 1/2^{k}. In other circumstances, arithmetic coding can offer better compression than Huffman coding because — intuitively — its "code words" can have effectively non-integer bit lengths, whereas code words in prefix codes such as Huffman codes can only have an integer number of bits. Therefore, a code word of length *k* only optimally matches a symbol of probability 1/2^{k} and other probabilities are not represented optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol. This difference is especially striking for small alphabet sizes.

Prefix codes nevertheless remain in wide use because of their simplicity, high speed, and lack of patent coverage. They are often used as a "back-end" to other compression methods. Deflate (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes; these are often called "Huffman codes" even though most applications use pre-defined variable-length codes rather than codes designed using Huffman's algorithm.

References

1. Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes" (http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf) (PDF). *Proceedings of the IRE*. **40** (9): 1098–1101. doi:10.1109/JRPROC.1952.273898 (https://doi.org/10.1109%2FJRPROC.1952.273898).
2. Van Leeuwen, Jan (1976). "On the construction of Huffman trees" (http://www.staff.science.uu.nl/~leeuw112/huffman.pdf) (PDF). *ICALP*: 382–410. Retrieved 2014-02-20.
3. Ze-Nian Li; Mark S. Drew; Jiangchuan Liu (2014-04-09). *Fundamentals of Multimedia* (https://books.google.com/books?id=R6vBBAAAQBAJ). Springer Science & Business Media. ISBN 978-3-319-05290-8.
4. J. Duda, K. Tahboub, N. J. Gadil, E. J. Delp, *The use of asymmetric numeral systems as an accurate replacement for Huffman coding* (https://ieeexplore.ieee.org/document/7170048), Picture Coding Symposium, 2015.
5. Huffman, Ken (1991). "Profile: David A. Huffman: Encoding the "Neatness" of Ones and Zeroes" (http://www.huffmancoding.com/my-uncle/scientific-american). *Scientific American*: 54–58.
6. Gribov, Alexander (2017-04-10). "Optimal Compression of a Polyline with Segments and Arcs". arXiv:1604.07476 (https://arxiv.org/abs/1604.07476) [cs.CG (https://arxiv.org/archive/cs.CG)].
7. Gallager, R.G.; van Voorhis, D.C. (1975). "Optimal source codes for geometrically distributed integer alphabets". *IEEE Transactions on Information Theory*. **21** (2): 228–230. doi:10.1109/TIT.1975.1055357 (https://doi.org/10.1109%2FTIT.1975.1055357).
8. Abrahams, J. (1997-06-11). "Code and parse trees for lossless source encoding". Written at Arlington, VA, USA. *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. Division of Mathematics, Computer & Information Sciences, Office of Naval Research (ONR). Salerno: IEEE. pp. 145–171. CiteSeerX 10.1.1.589.4726 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.589.4726). doi:10.1109/SEQUEN.1997.666911 (https://doi.org/10.1109%2FSEQUEN.1997.666911). ISBN 0-8186-8132-2. S2CID 124587565 (https://api.semanticscholar.org/CorpusID:124587565).
9. Hu, T. C.; Tucker, A. C. (1971). "Optimal Computer Search Trees and Variable-Length Alphabetical Codes". *SIAM Journal on Applied Mathematics*. **21** (4): 514. doi:10.1137/0121057 (https://doi.org/10.1137%2F0121057). JSTOR 2099603 (https://www.jstor.org/stable/2099603).

Discovery of Huffman codes

Inna Pivkina[*]

1 Introduction

The story of the invention of Huffman codes is described in an article by Gary Stix in the September 1991 issue of *Scientific American,* pp. 54, 58 ([4]). The following is excerpted from the article.

PROFILE: DAVID A. HUFFMAN
Encoding the "Neatness" of Ones and Zeroes

Large networks of IBM computers use it. So do high-definition television, modems and a popular electronic device that takes the brain work out of programming a videocassette recorder. All these digital wonders rely on the results of a 40-year-old term paper by a modest Massachusetts Institute of Technology graduate student—a data compression scheme known as Huffman encoding.

In 1951 David A. Huffman and his classmates in an electrical engineering graduate course on information theory were given the choice of a term paper or a final exam. For the term paper, Huffman's professor, Robert M. Fano, had assigned what at first appeared to be a simple problem. Students were asked to find the most efficient method of representing numbers, letters or other symbols using a binary code. Besides being a nimble intellectual exercise, finding such a code would enable information to be compressed for transmission over a computer network or for storage in a computer's memory.

Huffman worked on the problem for months, developing a number of approaches, but none that he could prove to be the most efficient. Finally, he despaired of ever reaching a solution and decided to start studying for the final. Just as he was throwing his notes in the garbage, the solution came to him. "It was the most singular moment of my life," Huffman says. "There was the absolute lightning of sudden realization."

That epiphany added Huffman to the legion of largely anonymous engineers whose innovative thinking forms the technical underpinnings for the accoutrements of modern living - in his case, from facsimile machines to modems and a myriad of other devices. "Huffman code is one of the fundamental ideas that people in computer science and data communications are using all the time," says Donald E. Knuth of Stanford University, who is the author of the multivolume series *The Art of Computer Programming.*

Huffman says he might never have tried his hand at the problem - much less solved it at the age of 25 - if he had known that Fano, his professor, and Claude E. Shannon, the creator of information theory, had struggled with it. "It was my luck to be there at the right time and also not have my professor discourage me by telling me that other good people had struggled with this problem," he says.

[*] Department of Computer Science; New Mexico State University; Las Cruces, NM 88003; ipivkina@cs.nmsu.edu.

2 Shannon-Fano Coding 1944

In this section we will study work of Shannon and Fano, in particular, definition of a unit of information, how to determine the amount of information, and a recording procedure (Shannon-Fano coding) they developed for improving the efficiency of transmission by reducing the number of digits required to transmit a message. Shannon-Fano coding was developed independently by Shannon and Fano in 1944. A greedy strategy can be used to produce a Shannon-Fano coding. The strategy uses a top-down approach and does not necessarily produce the optimal code.

Let us start with the notion of a unit of information. It is defined in section I from Fano [1].

I Definition of the Unit of Information

In order to define, in an appropriate and useful manner, a unit of information, we must first consider in some detail the nature of those processes in our experience which are generally recognized as conveying information. A very simple example of such processes is a yes-or-no answer to some specific question. A slightly more involved process is the indication of one object in a group of N objects, and, in general, the selection of one choice from a group of N specific choices. The word "specific" is underlined because such a qualification appears to be essential to these information-conveying processes. It means that the receiver is conscious of all possible choices, as is, of course, the transmitter (that is, the individual or the machine which is supplying the information). For instance, saying "yes" or "no" to a person who has not asked a question obviously does not convey any information. Similarly, the reception of a code number which is supposed to represent a particular message does not convey any information unless there is available a code book containing all the messages with the corresponding code numbers.

Considering next more complex processes, such as writing or speaking, we observe that these processes consist of orderly sequences of selections from a number of specific choices, namely, the letters of the alphabet or the corresponding sounds. Furthermore, there are indications that the signals transmitted by the nervous system are of a discrete rather than of a continuous nature, and might also be considered as sequences of selections. If this were the case, all information received through the senses could be analyzed in terms of selections. The above discussion indicates that the operation of selection forms the bases of a number of processes recognized as conveying information, and that it is likely to be of fundamental importance in all such processes. We may expect, therefore, that a unit of information, defined in terms of a selection, will provide a useful basis for a quantitative study of communication systems.

Considering more closely this operation of selection, we observe that different informational value is naturally attached to the selection of the same choice, depending on how likely the receiver considered the selection of that particular choice to be. For example, we would say that little information is given by the selection of a choice which the receiver was almost sure would be selected. It seems appropriate, therefore, in order to avoid difficulty at this early stage, to use in our definition the particular case of equally likely choices - that is, the case in which the receiver has no reason to expect that one choice will be selected rather than any other. In addition, our natural concept of information indicates that the information conveyed by a selection increases with the number of choices from which the selection is made, although the exact functional relation between these two quantities is not immediately clear.

On the basis of the above considerations, it seems reasonable to define as the unit of information the simplest possible selection, namely the selection between two equally likely choices, called, hereafter, the "elementary selection". For completeness, we must add to this definition the postulate, consistent with our intuition, that N independent selections of this type constitute N units of information. By independent selections we mean, of course, selections which do not affect one another. We shall adopt for this unit the convenient name of "bit" (from "binary digit"), suggested by Shannon. We shall also refer to a selection between two choices (not necessarily equally likely) as a "binary selection", and to a selection from N choices, as an N-order selection. When the choices are, a priori, equally likely, we shall refer to the selection as an "equally likely selection".

Exercise 2.1. Fano writes about specific questions and specific choices. Explain in your own words what he means by specific. Give an example of a specific question or choice. Explain why it is specific. Give an example of a question or choice which is not specific. Explain why it is not specific.

Exercise 2.2. Give an example of the simplest possible selection (or the "elementary selection"). Give an example of a selection which is not the simplest possible one.

Exercise 2.3. How much information does one flip of a coin convey? How much information is conveyed in N flips of a coin?

Once the unit of information is defined, it can be used to measure information conveyed by selections and messages. Fano talks about selection from several equally likely choices in section II of his work.

II Selection from N Equally Likely Choices

Consider now the selection of one among a number, N, of equally likely choices. In order to determine the amount of information corresponding to such a selection, we must reduce this more complex operation to a series of independent elementary operations. The required number of these elementary selections will be, by definition, the measure in bits of the information given by such an N-order selection.

Let us assume for the moment that N is a power of two. In addition (just to make the operation of selection more physical), let us think of the N choices as N objects arranged in a row, as indicated in Figure 1.

These N objects are first divided in two equal groups, so that the object to be selected is just as likely to be in one group as in the other. Then the indication of the group containing the desired object is equivalent to one elementary selection, and, therefore, to one bit. The next step consists of dividing each group into two equal subgroups, so that the object to be selected is again just as likely to be in either subgroup. Then one additional elementary selection, that is a total of two elementary selections, will suffice to indicate the desired subgroup (of the possible four subgroups). This process of successive subdivisions and corresponding elementary selections is carried out until the desired object is isolated from the others. Two subdivisions are required for $N = 4$, three from $N = 8$, and, in general, a number of subdivisions equal to $\log_2 N$, in the case of an N-order selection.

The same process can be carried out in a purely mathematical form by assigning order numbers from 0 to $N - 1$ to the N choices. The numbers are then expressed in the binary system, as shown in Figure 1, the number of binary digits (0 or 1) required being equal to $\log_2 N$. These digits represent

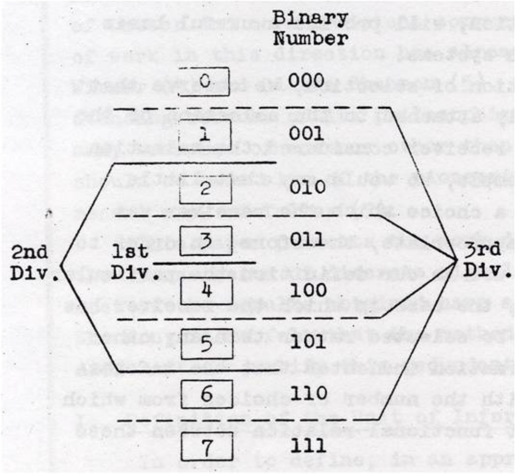


Fig. 1 Selection procedure for equally likely choices.

an equal number of elementary selections and, moreover, correspond in order to the successive divisions mentioned above. In conclusion, an N-order, equally likely selection conveys an amount of information

$$H_N = \log_2 N \tag{1}$$

The above result is strictly correct only if N is a power of two, in which case H_N is an integer. If N is not a power of two, then the number of elementary selections required to specify the desired choice will be equal to the logarithm of either the next lower or the next higher power of two, depending on the particular choice selected. Consider, for instance, the case of $N = 3$. The three choices, expressed as binary numbers, are then

$$00; 01; 10.$$

If the binary digits are read in order from left to right, it is clear that the first two numbers require two binary selections - that is, two digits, while the third number requires only the first digit, 1, in order to be distinguished from the other two. In other words, the number of elementary selections required when N is not a power of two is equal to either one of the two integers closest to $\log_2 N$. It follows that the corresponding amount of information must lie between these two limits, although the significance of a non-integral value of H is not clear as this point. It will be shown in the next section that Eq.(1) is still correct when N is not a power of two, provided H_N is considered as an average value over a large number of selections.

Exercise 2.4. Assume that we select one letter from the English alphabet at random. How much information does this selection convey?

Exercise 2.5. Assume that we have the following equally likely choices expressed as binary numbers: 000, 001, 010, 100, 101, and 110. If the binary digits are read in order from left to right, how many selections does each of the numbers require? Will your answer change if the digits are read from right to left?

Now that we know the amount of information conveyed by one selection, how can we determine the amount of information contained in a sequence of selections, or in a message? It is explained in the following quote from section III from Fano [1].

III Messages and Average Amount of Information

We have determined in the preceding section the amount of information conveyed by a single selection from N equally likely choices. In general, however, we have to deal with not one but long series of such selections, which we call messages. This is the case, for instance, in the transmission of written intelligence. Another example is provided by the communication system known as pulse-code modulation, in which audio waves are sampled at equal time intervals and then each sample is quantized, that is approximated by the closest of a number N of amplitude levels.

Let us consider, then, a message consisting of a sequence of n successive N-order selections. We shall assume, at first, that these selections are independent and equally likely. In this simpler case, all the different sequences which can be formed equal in number to

$$S = N^n, \tag{2}$$

are equally likely to occur. For instance, in the case of $N = 2$ (the two choices being represented by numbers 0 and 1) and $n = 3$, the possible sequences would be 000, 001, 010, 100, 011, 101, 110, 111. The total number of these sequences is $S = 8$ and the probability of each sequence is 1/8. In general, therefore, the ensemble of the possible sequences may be considered as forming a set of S equally likely choices, with the result that the selection of any particular sequence yields an amount of information

$$H_S = \log_2 S = n \log_2 N. \tag{3}$$

In words, n independent equally likely selections give n times as much information as a single selection of the same type. This result is certainly not surprising, since it is just a generalization of the postulate, stated in Section II, which forms an integral part of the definition of information.

It is often more convenient, in dealing with long messages, to use a quantity representing the average amount of information per N-order selection, rather than the total information corresponding to the whole message. We define this quantity in the most general case as the total information conveyed by a very long message divided by the number of selections in the message, and we shall indicate it with the symbol H_N, where N is the order of each selection. It is clear that when all the selections in the message are equally likely and independent and, in addition, N is a power of two, the quantity H_N is just equal to the information actually given by each selection, that is

$$H_N = \frac{1}{n} \log_2 S = \log_2 N. \tag{4}$$

We shall show now that this equation is correct also when N is not a power of two, in which case H_N has to be actually an average value taken over a sufficiently long sequence of selections.

The number S of different and equally likely sequences which can be formed with n independent and equally likely selections is still given by Eq.(2), even when N is not a power of two. On the contrary, the number of elementary selections required to specify any one particular sequence must be written now in the form

$$B_S = \log_2 S + d, \tag{5}$$

where d is a number, smaller in magnitude than unity, which makes B_S an integer and which depends on the particular sequence selected. The average amount of information per N-order selection is then, by definition,

$$H_N = \lim_{n\to\infty} \frac{1}{n}(\log_2 S + d). \tag{6}$$

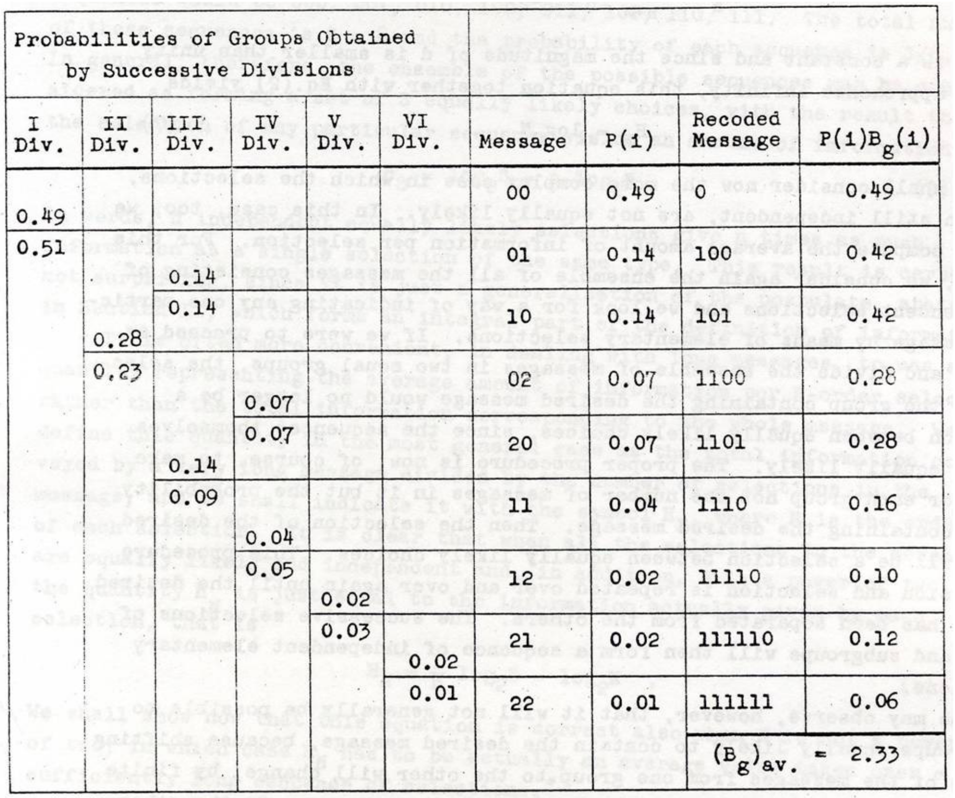
Since N is a constant and since the magnitude of d is smaller than unity while n approaches infinity, this equation together with Eq.(2) yields

$$H_N = \log_2 N. \tag{7}$$

We shall consider now the more complex case in which the selections, although still independent, are not equally likely. In this case, too, we wish to compute the average amount of information per selection. For this purpose, we consider again the ensemble of all the messages consisting of independent selections and we look for a way of indicating any one particular message by means of elementary selections. If we were to proceed as before, and divide the ensemble of messages in two equal groups, the selection of the group containing the desired message would no longer be a selection between equally likely choices, since the sequences themselves are not equally likely. The proper procedure is now, of course, to make equal for each group not the number of messages in it but the probability of its containing the desired message. Then the selection of the desired group will be a selection between equally likely choices. This procedure of division and selection is repeated over and over again until the desired message has been separated from the others. The successive selections of groups and subgroups will then form a sequence of independent elementary selections.

One may observe, however, that it will not generally be possible to form groups equally likely to contain the desired message, because shifting any one of the messages from one group to the other will change, by finite amounts, the probabilities corresponding to the two groups. On the other hand, if the length of the messages is increased indefinitely, the accuracy with which the probabilities of the two groups can be made equal becomes better and better since the probability of each individual message approaches zero. Even so, when the resulting subgroups include only a few messages after a large number of divisions, it may become impossible to keep the probabilities of such subgroups as closely equal as desired unless we proceed from the beginning in an appropriate manner as indicated below. The messages are first arranged in order of their probabilities, which can be easily computed if the probabilities of the choices are known. The divisions in groups and subgroups are then made successively without changing the order of the messages, as illustrated in Figure 2. In this manner, the smaller subgroups will contain messages with equal or almost equal probabilities, so that further subdivisions can be performed satisfactory. It is clear that when the above procedure is followed, the number of binary selections required to separate any message from the others varies from message to message. Messages with a high probability of being selected require less binary selections than those with lower probabilities. This fact is in agreement with the intuitive notion that the selection of a little-probable message conveys more information than the selection of a more-probable one. Certainly, the occurrence of an event which we know a priori to have a 99 per cent probability is hardly surprising or, in our terminology, yields very little information, while the occurrence of an event which has a probability of only 1 per cent yields considerably more information.

In the above, Figure 2 illustrates a method which can be used to code messages. Assume that we are given an ensemble of messages and probabilities of all the messages in the ensemble. First, we sort the messages in decreasing order of their probabilities. Then, we divide the sequence of messages into two consecutive groups in such a way that the probability of a message being in the first group is as nearly equal to the probability of a message being in the second group as possible. After the 1st division, we have two groups each of which consists of one or more messages. If a part consists of more than one message, we divide it into two parts again in the same manner - probabilities of the two parts should be as nearly equal as possible. We continue divisions until



Probabilities of Groups Obtained by Successive Divisions									
I Div.	II Div.	III Div.	IV Div.	V Div.	VI Div.	Message	P(i)	Recoded Message	$P(i)B_g(i)$
0.49						00	0.49	0	0.49
0.51		0.14				01	0.14	100	0.42
	0.28	0.14				10	0.14	101	0.42
	0.23		0.07			02	0.07	1100	0.28
		0.14	0.07			20	0.07	1101	0.28
		0.09	0.04			11	0.04	1110	0.16
			0.05	0.02		12	0.02	11110	0.10
				0.03	0.02	21	0.02	111110	0.12
					0.01	22	0.01	111111	0.06
								$(B_g)_{av.}$ =	2.33

Fig. 2

every part contains one message. Each division of a group of messages contributes one binary digit to codes of the messages in the group. When divided into two subgroups, messages in one subgroup have 0 added to their codes; messages in the other subgroup have 1 added to their codes. This method of encoding messages was developed independently by Shannon [3]. We will call it Shannon-Fano coding. The following quote from [3] gives one more description of Shannon-Fano coding: "...arrange the messages ... in order of decreasing probability. Divide this series into two groups of as nearly equal probability as possible. If the message is in the first group its binary digit will be 0, otherwise 1. The groups are similarly divided into subsets of nearly equal probability and the particular subset determines the second binary digit. This process is continued until each subset contains only one message."

Let us see how Shannon-Fano coding is done in the example from Figure 2. We start with messages 00, 01, 10, 02, 20, 11, 12, 21, and 22; their probabilities are listed in column "P(i)". After the 1st division, the two groups of messages are {00} and {01, 10, 02, 20, 11, 12, 21, 22} with probabilities 0.49 and 0.51, respectively. Message 00 is assigned code 0; messages in the second group have their codes begin with 1. Since the second group consists of more than one message it is divided again in two groups, {01, 10} and {02, 20, 11, 12, 21, 22}, with probabilities 0.28 and 0.23, respectively. Messages 01 and 10 have 0 added as 2nd binary digit in their codes; their codes begin with "10". Messages 02, 20, 11, 12, 21, and 22 have 1 added as 2nd binary digit in their codes. Divisions continue until each group contains one message only. Codes obtained for each message are listed in column "Recoded message".

Successive divisions of messages and obtained message codes can be represented as a code tree as shown in Figure 3. Each leaf is labeled with a message and its probability. Each internal node is labeled with the sum of probabilities of the leaves in its subtree. The root is labeled with 1 -

the sum of probabilities of all the messages. Each edge from a node to its child is labeled with a digit (0 or 1) which appears in codes of leaves of the subtree rooted at the child. The code of a message can be obtained by reading labels of edges on the path from the root of the tree to the leaf containing the message. For instance, path from the root of the tree in Figure 3 to the leaf containing message "20" consists of edges labeled 1, 1, 0, and 1, which indicates that the code of message "20" is "1101".

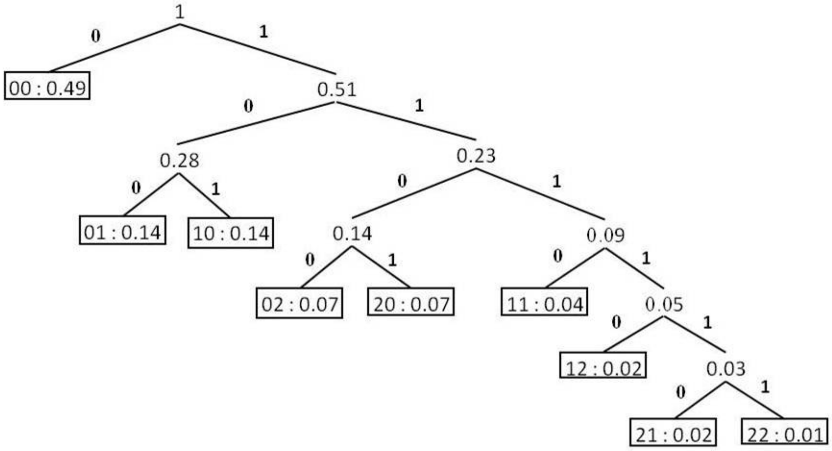


Fig. 3 Tree corresponding to the coding scheme in Figure 2.

Shannon-Fano coding is an example of a greedy algorithm. Greedy algorithms are algorithms that at every step make a choice that looks best at the moment. In Shannon-Fano coding, at every step (division) a choice (devision of messages into two consecutive groups) is made that looks best at the moment. Best in a sense that the probabilities of the two parts should be as nearly equal as possible. Shannon-Fano coding uses top-down approach, dividing groups of messages into subgroups. The code tree is built top-down, from the root (that corresponds to all the messages) to the leaves (each leaf corresponds to a single message). At each step, a group of messages is divided into two subgroups and two children corresponding to the subgroups are added to the node that corresponds to the original group.

Exercise 2.6. Assume that we have messages A, B, C, D, E, F, G. The probability of A is 0.12, the probability of B is 0.34, the probability of C is 0.05, the probability of D is 0.22, the probability of E is 0.15, the probability of F is 0.02, and the probability of G is 0.10. Use Shannon-Fano coding to produce recoded messages for each of A, B, C, D, E, F, G. Represent obtained coding scheme as a tree.

Exercise 2.7. Assume that there are n messages. For each message i, we know the probability of the message, P(i). Write a pseudocode of the procedure described by Fano (Shannon-Fano coding). What is the running time of the procedure?

Exercise 2.8. Implement Fano's method. Given a set of messages with their probabilities, your program should output codes for each message.

Sometimes, instead of probabilities we know frequencies of messages - how often a message appears in a certain text. In this case, we can compute probabilities by assuming that they are proportional to corresponding frequencies. In other words, the probability of a message is set to be the frequency of the message divided by the sum of frequencies of all messages. However, in the

Shannon-Fano coding procedure we do not need to compute probabilities, we can use frequencies instead.

Exercise 2.9. Argue that if you use frequencies instead of probabilities in Shannon-Fano coding, you get the same codes for messages as you would get when using probabilities.

Exercise 2.10. Use Fano's procedure to produce encodings for the six symbols with their frequencies shown in Figure 4. Draw a code tree that corresponds to the encoding you obtained.

Symbol	Frequency
A	56
B	40
C	33
D	75
E	6
F	24

Fig. 4

Shannon-Fano coding method does not always produce the best (optimal) encoding - encoding that reduces the number of digits required to transmit a message the most. Huffman approach is better as it always finds an optimal encoding.

3 Huffman encoding: I

Let us look at the work of Huffman. The Introduction from Huffman's paper "A Method for the Construction of Minimum-Redundancy Codes" ([2]) is quoted below. It defines the "optimum code" and presents basic restrictions that a code must satisfy to be optimum.

Introduction

One important method of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one symbol. If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) of a message is directly proportional to the number of symbols associated with it. In this paper, the symbol or sequence of symbols associated with a given message will be called the "message code." The entire number of messages which might be transmitted will be called the "message ensemble." The mutual agreement between the transmitter and the receiver about the meaning of the code for each message of the ensemble will be called the "ensemble code."

Probably the most familiar ensemble code was stated in the phrase "one if by land and two if by sea", and the message codes were "one" and "two."

In order to formalize the requirements of an ensemble code, the coding symbols will be represented by numbers. Thus, if there are D different types of symbols to be used in coding, they will be represented by the digits 0, 1, 2, .., $(D-1)$. For example, a ternary code will be constructed using the three digits 0,1, and 2 as coding symbols.

The number of messages in the ensemble will be called N. Let $P(i)$ be the probability of the ith message. Then

$$\sum_{i=1}^{N} P(i) = 1 \tag{1}$$

The length of a message, $L(i)$, is the number of coding digits assigned to it. Therefore, the average message length is:

$$L_{av} = \sum_{i=1}^{N} P(i)L(i) \tag{2}$$

The term "redundancy" has been defined by Shannon as a property of codes. A "minimum-redundancy code" will be defined here as an ensemble code which, for a message ensemble consisting of a finite number of members, N, and for a given number of coding digits, D, yields the lowest possible average message length. In order to avoid the use of the lengthy term "minimum-redundancy," this term will be replaced here by "optimum." It will be understood then that, in this paper, "optimum code" means "minimum-redundancy code."

The following basic restrictions will be imposed on an ensemble code:

(a) No two messages will consist of identical arrangements of coding digits.

(b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known.

Restriction (b) necessitates that no message be coded in such a way that its code appears, digit for digit, as the first part of any message code of greater length. Thus, 01, 102, 111, and 202 are valid message codes for an ensemble of four members. For instance, a sequence of these messages 1111022020101111102 can be broken up into the individual messages 111-102-202-01-01-111-102. All the receiver needs to know is the ensemble code. However, if the ensemble has individual message codes including 11, 111, 102, and 02, then when a message sequence starts with the digits 11, it is not immediately certain whether the message 11 has been received or whether it is only the first two digits of the message 111. Moreover, even if the sequence turns out to be 11102, it is still not certain whether 111-02 or 11-102 was transmitted. In this example, change of one of the two message codes 111 or 11 indicated.

C.E. Shannon and R. M. Fano have developed ensemble coding procedures for the purpose of proving that the average number of binary digits required per message approaches from above the average amount of information per message. Their coding procedures are not optimum, but approach the optimum behavior when N approaches infinity. Some work has been done by Kraft toward deriving a coding method which gives an average code length as close to possible to the ideal when the ensemble contains a finite number of members. However, up to the present time, no definite procedure has been suggested for the construction of such a code to the knowledge of the author. It is the purpose of this paper to derive such a procedure.

Exercise 3.1. In the Introduction Huffman defined a notion of "message ensemble". What is the message ensemble in the example of Figure 2? What is the average message length of the encoding in Figure 2?

Exercise 3.2. What is the average message length of the encoding in Exercise 2.10?

Exercise 3.3. Does the coding in Figure 2 satisfy restrictions (a) and (b) from the Introduction of Huffman's paper? Will an encoding produced by Shannon-Fano method always satisfy restrictions (a) and (b), or not?

Exercise 3.4. Recall that Huffman defines a code to be "optimum" if it yields the lowest possible average message length. Let a message ensemble consists of three messages: A, B, and C. Assume that only two symbols, 0 and 1, can be used in coding. Give an example of a code for the ensemble which is not optimum. Prove that it is not optimum. You may choose any values for probabilities of the messages. (Hint: In order to prove that the code is not optimum, give another code that has a smaller average message length.)

4 Huffman encoding: II

In the following excerpts from sections Derived Coding Requirements and Optimum Binary Code from Huffman's paper ([2]) Huffman derives more requirements that an optimum code must satisfy, proposes a procedure to find an optimum code and proves that it is correct (that the code found by the procedure is always optimum).

Derived Coding Requirements

For an optimum code, the length of a given message code can never be less than the length of a more probable message code. If this requirement were not met, then a reduction in average message length could be obtained by interchanging the codes for the two messages in question in such a way that the shorter code becomes associated with the more probable message. Also, if there are several messages with the same probability, then it is possible that the codes for these messages may differ in length. However, the codes for these messages may be interchanged in any way without affecting the average code length for the message ensemble. Therefore, it may be assumed that the messages in the ensemble have been ordered in a fashion such that

$$P(1) \geq P(2) \geq \cdots \geq P(N-1) \geq P(N) \tag{3}$$

And that, in addition, for an optimum code, the condition

$$L(1) \leq L(2) \leq \cdots \leq L(N-1) \leq L(N) \tag{4}$$

holds. This requirement is assumed to be satisfied throughout the following discussion.

It might be imagined that an ensemble code could assign q more digits to the Nth message than to the $(N-1)$th message. However, the first $L(N-1)$ digits of the Nth message must not be used as the code for any other message. Thus the additional q digits would serve no useful purpose and would unnecessarily increase L_{av}. Therefore, for an optimal code it is necessary that $L(N)$ be equal to $L(N-1)$.

The kth prefix of a message code will be defined as the first k digits of that message code. Basic restriction (b) could then be restated as: No message shall be coded in such a way that its code is a prefix of any other message, or that any of its prefixes are used elsewhere as a message code.

Imagine an optimum code in which no two of the messages coded with length $L(N)$ have identical prefixes of order $L(N)-1$. Since an optimum code has been assumed, then none of these messages of length $L(N)$ can have codes or prefixes of any order which correspond to other codes. It would then be possible to drop the last digit of all of this group of messages and thereby reduce the value of L_{av}. Therefore, in an optimum code, it is necessary that at least two (and no more than D) of the codes with length $L(N)$ have identical prefixes of order $L(N)-1$.

One additional requirement can be made for an optimum code. Assume that there exists a combination of the D different types of coding digits which is less than $L(N)$ digits in length and which is not used as a message code or which is not a prefix of a message code. Then this combination of digits could be used to replace the code for the Nth message with a consequent reduction of L_{av}. Therefore, all possible sequences of $L(N)-1$ digits must be used either as message codes, or must have one of their prefixes used as message codes.

The derived restrictions for an optimum code are summarized in condensed form below and considered in addition to restrictions (a) and (b) given in the first part of this paper:

(c) $L(1) \leq L(2) \leq \cdots \leq L(N-1) = L(N)$. (5)

(d) At least two and not more than D of the messages with code length $L(N)$ have codes which are alike except for their final digits.

(e) Each possible sequence of $L(N)-1$ digits must be used either as a message code or must have one of its prefixes used as a message code.

Optimum Binary Code

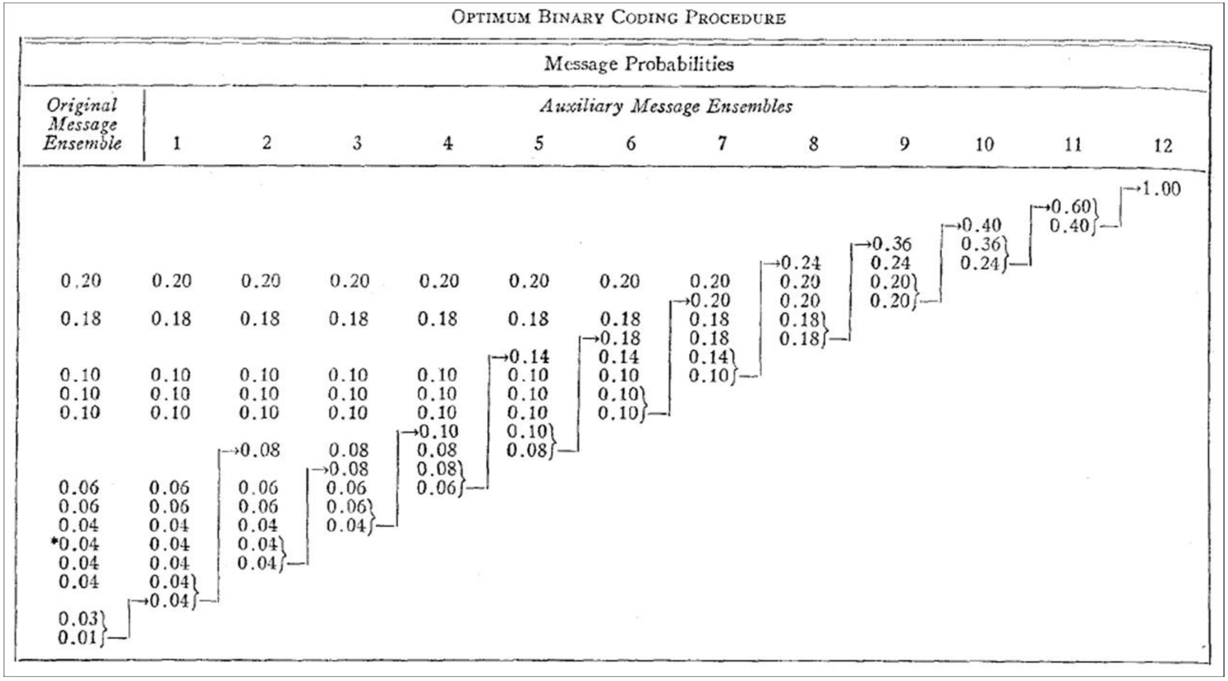
For ease of development of the optimum coding procedure, let us now restrict ourselves to the problem of binary coding. Later this procedure will be extended to the general case of D digits.

Restriction (c) makes it necessary that the two least probable messages have codes of equal length. Restriction (d) places the requirement that, for D equal to two, there be only two of the messages with coded length $L(N)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the Nth and the $(N-1)$st messages since at this point it is not known whether or not other codes of length $L(N)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $L(N)-1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages will be called the first auxiliary message ensemble.

This newly created ensemble contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. The codes for each of the two least probable messages in this new ensemble are required to be identical except in their final digits; 0 and 1 are assigned as these digits, one for each of the two messages. Each new auxiliary ensemble contains one less message than the preceding ensemble. Each auxiliary ensemble represents the original ensemble with full use made of the accumulated necessary coding requirements.

The procedure is applied again and again until the number of members in the most recently formed auxiliary message ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite message with probability unity, and the coding is complete. Now let us examine Table I. The left-hand column

Table I (Huffman)



OPTIMUM BINARY CODING PROCEDURE

Message Probabilities

Original Message Ensemble	*Auxiliary Message Ensembles* 1	2	3	4	5	6	7	8	9	10	11	12
												→1.00
											→0.60}	
										→0.40	0.40}	
									→0.36	0.36}		
								→0.24	0.24	0.24}		
0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20}			
							→0.20	0.20	0.20}			
0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18}				
						→0.18	0.18	0.18}				
					→0.14	0.14	0.14}					
0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10}					
0.10	0.10	0.10	0.10	0.10	0.10	0.10}						
0.10	0.10	0.10	0.10	0.10	0.10	0.10}						
				→0.10	0.10}							
		→0.08	0.08	0.08	0.08}							
			→0.08	0.08}								
0.06	0.06	0.06	0.06	0.06}								
0.06	0.06	0.06	0.06}									
0.04	0.04	0.04	0.04}									
*0.04	0.04	0.04}										
0.04	0.04	0.04}										
0.04	0.04}											
	→0.04}											
0.03}												
0.01}												

contains the ordered message probabilities of the ensemble to be coded. N is equal to 13. Since each combination of two messages (indicated by a bracket) is accompanied by the assigning of a new digit to each, then the total number of digits which should be assigned to each original message is the same as the number of combinations indicated for that message. For example, the message marked *, or a composite of which it is a part, is combined with others five times, and therefore should be assigned a code length of five digits.

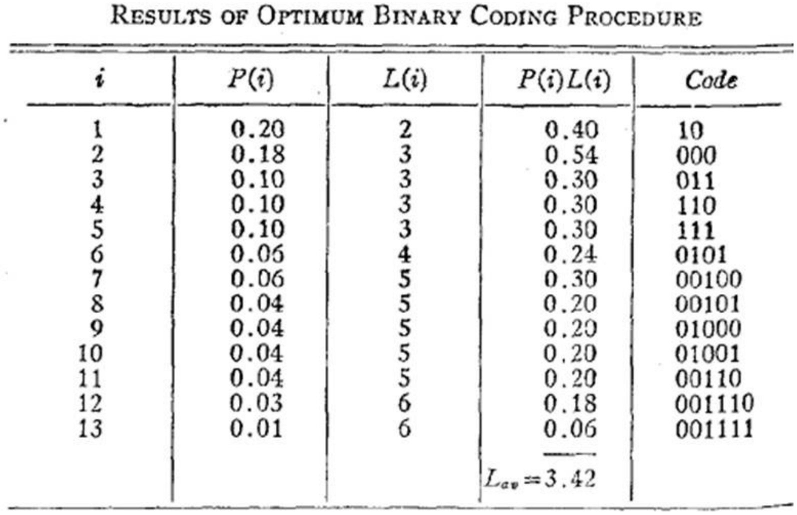
When there is no alternative in choosing the two least probable messages, then it is clear that the requirements, established as necessary, are also sufficient for deriving an optimum code. There may arise situations in which a choice may be made between two or more groupings of least likely messages. Such a case arises, for example, in the fourth auxiliary ensemble of Table I. Either of the messages of probability 0.08 could have been combined with that of probability 0.06. However, it is possible to rearrange codes in any manner among equally likely messages without affecting the average code length, and so a choice of either of the alternatives could have been made. Therefore, the procedure given is always sufficient to establish an optimum binary code.

The lengths of all the encoded messages derived from Table I are given in Table II.

Having now determined proper lengths of code for each message, the problem of specifying the actual digits remains. Many alternatives exist. Since the combining of messages into their composites is similar to the successive confluences of trickles, rivulets, brooks, and creeks into a final large river, the procedure thus far described might be considered analogous to the placing of signs by a water-borne insect at each of these junctions as he journeys downstream. It should be remembered that the code which we desire is that one which the insect must remember in order to work his way back up stream. Since the placing of the signs need not follow the same rule, such as "zero-right-returning", at each junction, it can be seen that there are at least 2^{12} different ways of assigning code digits for our example.

The code in Table II was obtained by using the digit 0 for the upper message and the digit 1 for the lower message of any bracket. It is important to note in Table I that coding restriction (e) is

Table II (Huffman)



RESULTS OF OPTIMUM BINARY CODING PROCEDURE

i	$P(i)$	$L(i)$	$P(i)L(i)$	Code
1	0.20	2	0.40	10
2	0.18	3	0.54	000
3	0.10	3	0.30	011
4	0.10	3	0.30	110
5	0.10	3	0.30	111
6	0.06	4	0.24	0101
7	0.06	5	0.30	00100
8	0.04	5	0.20	00101
9	0.04	5	0.20	01000
10	0.04	5	0.20	01001
11	0.04	5	0.20	00110
12	0.03	6	0.18	001110
13	0.01	6	0.06	001111
			$L_{av}=3.42$	

automatically met as long as two messages (and not one) are placed in each bracket.

Huffman algorithm is another example of a greedy algorithm as it looks for two least probable messages at every step. Huffman algorithm uses bottom-up approach. At every step it combines two messages into a single composite message. The code tree is built bottom-up, from the leaves to the root. At each step, two nodes that correspond to two least probable messages are made children of a new node that corresponds to the composite message.

Both, Shannon-Fano and Huffman algorithms are greedy algorithms that try to solve the same problem. One uses top-down approach while the other one uses bottom-up approach. However, the greedy strategy of Shannon-Fano does not always produce an optimal solution. Huffman approach, on the other hand, always finds an optimal solution.

Exercise 4.1. Draw a code tree that corresponds to Tables I and II in Huffman's paper.

Exercise 4.2. Use the Huffman algorithm for messages and probabilities from Figure 2 (Fano). What codes do you get for the messages? Is the encoding better or worse than the one obtained by Fano?

Exercise 4.3. Use Huffman algorithm for symbols and frequencies from Figure 4. What code tree do you get? What are codes for the symbols? Is the obtained encoding better or worse than the one produced earlier by Shannon-Fano approach?

Exercise 4.4. Assume that message ensemble consists of n messages: 1, 2, $\ldots$, n. Frequency of message i is equal to Fibonacci number F_i ($1 \leq i \leq n$). Recall that $F_1 = 1$, $F_2 = 1$, $F_k = F_{k-1} + F_{k-2}$. What will a code tree produced by Huffman algorithm look like?

Exercise 4.5. Use Shannon-Fano approach for messages and probabilities from Table II (Huffman). What codes do you get for the messages? Is the encoding better or worse than the one obtained by Huffman?

Exercise 4.6. Give your own example when Shannon-Fano procedure produces an optimal code. You need to specify what the messages are, what their probabilities (or frequencies) are, compute Shannon-Fano coding, and compare it with optimal code obtained by using Huffman procedure.

Exercise 4.7. Give your own example when Shannon-Fano procedure does not produce an optimal code. You need to specify what the messages are, what their probabilities (or frequencies) are, compute Shannon-Fano coding, and compare it with the optimal code obtained by using Huffman procedure.

Exercise 4.8. Given a message ensemble and the ensemble code, how can you show that the code is optimal? Is verification of a code being optimal easier or not than finding an optimal code?

Exercise 4.9. You are given a code tree for a message ensemble. If you swap two leaves in the tree and update labels of internal nodes so that they are equal to the sum of probabilities of their descendant leaves, you get a different code for the ensemble. Consider the following conjecture: if every code obtained by swapping two leaves in the code tree is no better (is the same or worse) than the code of the original tree, then the code of the original tree is an optimum code. Prove the conjecture or give a counterexample.

Exercise 4.10. Write a program that would produce a Huffman encoding given messages and their frequencies.

Exercise 4.11. In the procedure described by Fano, the divisions in groups and subgroups are made without changing the order of the messages. Changing the order of messages may allow us to divide the group into subgroups with less difference between probabilities of subgroups. Assume that at every step of the Fano procedure we get the best possible split. Will this modified procedure yield an optimum code?

5 Huffman encoding: III

Huffman describes a generalization of his method to the case when three or more digits are used for coding. It is done in Generalization of the Method section from Huffman's paper ([2]).

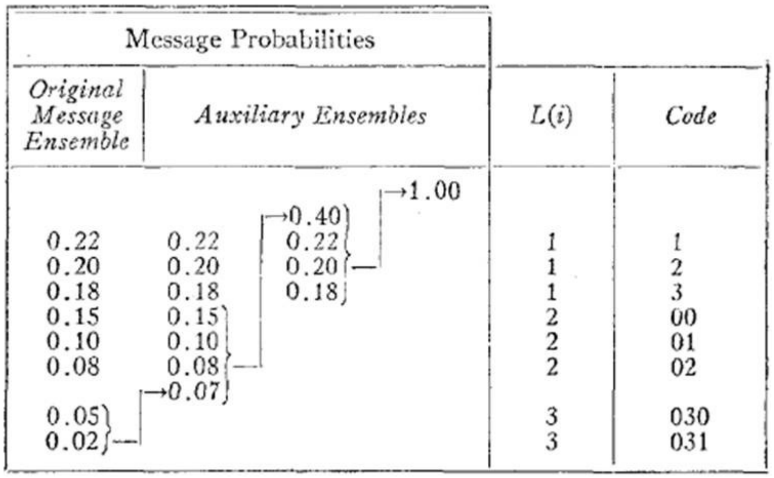
Generalization of the Method

Optimum coding of an ensemble of messages using three or more types of digits is similar to the binary coding procedure. A table of auxiliary message ensembles similar to Table I will be used. Brackets indicating messages combined to form composite messages will be used in the same way as was done in Table I. However, in order to satisfy restriction (e), it will be required that all these brackets, with the possible exception of one combining the least probable messages of the original ensemble, always combine a number of messages equal to D.

It will be noted that the terminating auxiliary ensemble always has one unity probability message. Each preceding ensemble is increased in number by $D-1$ until the first auxiliary ensemble is reached. Therefore, if N_1 is the number of messages in the first auxiliary ensemble, then $(N_1-1)/(D-1)$ must be an integer. However $N_1 = N - n_0 + 1$, where n_0 is the number of the least probable messages combined in a bracket in the original ensemble. Therefore, n_0 (which, of course, is at least two and no more than D) must be of such a value that $(N-n_0)/(D-1)$ is an integer.

Table III (Huffman)

Optimum Coding Procedure for $D=4$



Message Probabilities					
Original Message Ensemble	*Auxiliary Ensembles*			*L(i)*	*Code*
			→1.00		
		→0.40			
0.22	0.22	0.22		1	1
0.20	0.20	0.20		1	2
0.18	0.18	0.18		1	3
0.15	0.15			2	00
0.10	0.10			2	01
0.08	0.08			2	02
	→0.07				
0.05				3	030
0.02				3	031

In Table III an example is considered using an ensemble of eight messages which is to be coded with four digits; n_0 is found to be 2. The code listed in the table is obtained by assigning the four digits 0, 1, 2, and 3, in order, to each of the brackets.

Exercise 5.1. Implement Huffman generalized method to produce encoding which uses a given number of digits.

References

[1] R.M. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., 1949.

[2] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[3] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[4] Gary Stix. Profile: David A. Huffman. *Scientific American*, 265(3):54, 58, September 1991.

Notes to the Instructor

The story of the invention of Huffman codes is a great story that demonstrates that students can do better than professors. David Huffman was a student in an electrical engineering course in 1951. His professor, Robert Fano, offered students a choice of taking a final exam or writing a term paper. Huffman did not want to take the final so he started working on the term paper. The topic of the paper was to find the most efficient (optimal) code. What professor Fano did not tell his students was the fact that it was an open problem and that he was working on the problem himself. Huffman spent a lot of time on the problem and was ready to give up when the solution suddenly came to him. The code he discovered was optimal, that is, it had the lowest possible average message length. The method that Fano had developed for this problem did not always produce an optimal code. Therefore, Huffman did better than his professor. Later Huffman said that likely he would not have even attempted the problem if he had known that his professor was struggling with it.

The project uses excerpts from Fano's work ([1]) and from Huffman's paper ([2]) where they present their encodings. Both Fano and Huffman used greedy strategies to find the codes. However, Fano's greedy algorithm would not always produce an optimal code while Huffman's greedy algorithm would always find an optimal solution. The purpose of the project is for students to learn greedy algorithms, prefix-free codes, Huffman encoding, binary tree representations of codes, and the basics of information theory (unit and amount of information). The project demonstrates that greedy strategy could be applied in different ways to the same problem, sometimes producing an optimal solution and sometimes not.

The project is designed for a junior level Data Structures and Algorithms course. It can be used when covering the topic of greedy algorithms. The project uses quotes from Fano's and Huffman's papers ([1], [2]) to look into greedy algorithms, prefix-free codes, Huffman encoding, binary tree representations of codes, unit and amount of information.

The project is divided into several sections. Each section, except for Introduction, contains a reading assignment (read selected quotes from the original sources and some additional explanations) and a list of exercises on the material from the reading. Exercises are of different types, some requiring simple understanding of a definition or an algorithm, and some requiring proofs. Exercises 2.8, 4.10, and 5.1 are programming exercises and require good programming skills.

The material naturally splits in two parts. The first part (sections 1 and 2) is based on Fano's paper ([1]). It talks about measuring information and Fano encoding. Students may need substantial guidance with this part. The second part (sections 3, 4, and 5) is based on Huffman's paper ([2]). Sections 3 and 4 discuss Huffman encoding and compare it to Fano's. Section 5 can be skipped as it describes a generalization of Huffman's method to the case when three or more digits are used for coding. Each of the two parts can be used as a one week homework assignment which can be done individually or in small groups. For instance, the first assignment may ask students to read sections 1 and 2 and do exercises 2.1-2.10. The second assignment may ask students to read sections 3 and 4 and do exercises 3.1-3.4 and 4.1-4.11 (section 5 is skipped).