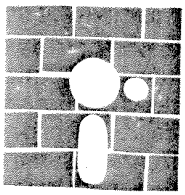
Exhibit C

DATA COMPRESSION WITH HUFFMAN CODING

BY JONATHAN AMSTERDAM

A close look at an elegant way to compress information



Am I the only one, or have you also noticed that there's never enough room on a disk? No matter how big a floppy is—200K, 400K, or even 800K bytes—it's almost too easy to stuff it to the gills. The same goes for hard disks. Sure, it takes a while to fill up 20 megabytes. But eventually, things get so tight you couldn't fit your own name into the space left.

Using data-compression techniques, you can shorten files by compressing the information they contain. But data compression can do more than just save disk space. It can also cut down on the time needed to transmit large files between computers, especially if the transmission is done over slow links like telephone lines. If you compress the file before sending it and uncompress it on the receiving end, you can reduce the total time for the transmission. The technique can work interactively, too. If you are using your computer as a terminal to communicate with a host computer via a modem, the host can send compressed commands and data that your computer uncompresses before displaying. The result can be apparent communication speeds that greatly exceed the actual transmission rate of the hookup. Such a system could make remote full-screen editing pleasant, even over 1200-bps lines.

This month, I will discuss an elegant data-compression algorithm called Huffman coding. Invented by David Huffman in 1952, it's easy to implement and widely used. In a sense I'll make precise later, Huffman coding is the "best" way to compress data in general.

THE PROBLEM DEFINED

For the sake of concreteness, I will discuss Huffman coding in the context of compressing ASCII text files. The program I will construct takes as input a text file, that is, a sequence of 1-byte characters. Hopefully, the output will be a shorter file. A separate uncompressing program will turn the compressed file back into the original one when you so desire.

How is it possible to reduce the size of a file without losing some of the information it contains? The answer involves constructing a code for each character of the file. Note that ASCII, as its full name—American Standard Code for Information Interchange—suggests, is itself a character code. ASCII assigns a unique 7-bit pattern to each character. Since all the codes have

(continued)

Jonathan Amsterdam is a graduate student at the Massachusetts Institute of Technology Artificial Intelligence Laboratory. He can be reached at 1643 Cambridge St. #34, Cambridge, MA 02138.

the same length, ASCII is a fixed-length encoding scheme.

The idea behind Huffman coding (and, by the way, Morse code as well) is that variable-length codes can achieve a higher data density than fixed-length codes if the characters differ in frequency of occurrence. For instance, in a file of English text, the space character will probably be by far the most common character, accounting for perhaps one-sixth of the file's contents. The letter "e" will likely finish second. Letters like "x" and "z," on the other hand, will be infrequent, if present at all. By using an English-letter frequency table, you can construct a coding scheme that assigns short codes to the frequent characters and long codes to the infrequent ones and use it to encode English files. The size of a file will be reduced if it conforms well to the frequency table. But you can do better by constructing a different code for each file, using the actual number of occurrences of characters in that file as the frequency table.

For those of you who remain skeptical about the ability of this method to shorten files, let's take as an example the sentence you are now reading. The frequency counts for the characters in the sentence, along with the coding schemes produced by the Huffman algorithm, appear in table 1. The sentence contains 147 characters, including spaces and punctuation. If the sentence were stored as a text file, each character would occupy 1 byte, so the sentence would take up 147×8 or 1176 bits. Since 27 distinct characters are found in the sentence, the best fixed-length encoding you could hope for would use 5 bits per character; the sentence would thus occupy 735 bits. The Huffman-coded version occupies only 612 bits, a 17 percent improvement over the best fixed-length code and a 48 percent savings over the ASCII storage method. In practice, I have found Huffman coding to reduce file sizes by about 30 percent.

Variable-length codes can compress information, but they have their drawbacks. For one thing, they are difficult to manipulate inside a computer, which prefers fixed-size objects; hence, a fixed-length code like ASCII is superior when space or transmission time isn't an issue. Variable-length codes are also sensitive to mangled bits. A single incorrect bit in a variable-length encoding can throw off the rest of the message. In a fixed-length code, an incorrect bit will affect only one character.

A third problem with variable-length codes crops up when you try to decode an encoded file. Say your coding scheme assigns the code 0 to the character "a," 1 to "b," and 01 to "c." The string "ab" is encoded as 01. But when decoding this string, you can't know whether the original string was "ab" or "c."

To avoid ambiguity, it is sufficient that the code possess the prefix property. In a code with this property, no character code occurs as a prefix—an initial sequence—of any other code. The code in the previous paragraph does not have the prefix property because the code for "a," 0, is a prefix of the code for "c," 01. The code in table 1 does have the prefix property, as do all codes generated by the Huffman algorithm. The prefix property makes decoding easy. The decoder can read one bit after another until the sequence of bits read so far corresponds to a character code. It then outputs the character and begins reading again.

Table 1: *The frequency distribution and Huffman encoding for the sentence "For those of you who remain skeptical about the ability of this method to shorten files, let's take as an example the sentence you are now reading."*

Character	Frequency	Code
space	26	101
e	17	010
t	13	1000
o	12	1001
a	10	1101
s	8	0111
i	7	0000
h	7	0001
n	7	0010
l	5	11001
r	5	11000
f	3	01101
y	3	00111
u	3	00110
m	3	111011
w	2	111111
k	2	111110
p	2	111100
b	2	111101
d	2	011001
c	2	011000
x	1	1110101
g	1	1110100
F	1	1110010
' (apostrophe)	1	1110011
, (comma)	1	1110001
. (period)	1	1110000

AN OVERVIEW

Let's now take a close look at the Huffman algorithm, which uses a frequency table to construct a variable-length code with the prefix property. The secret to the algorithm is that versatile and elegant data structure, the binary tree. It so happens that a correspondence exists between codes with the prefix property and binary trees where every node has either two children or none. The nodes with no children—the leaves of the tree—are labeled with characters. Each left branch of the tree is labeled with a 0,

and each right branch with a 1. Let us call such a tree a code tree. Figure 1 shows a simple code tree.

To get the code for a particular character, just trace the path from the root of the tree to the leaf labeled with that character. Every time you turn left, add a 0 to the character's code; every time you turn right, add a 1. In the tree shown, the character code for "d" is 011. The form of the tree embodies the prefix property, because for one code to be a prefix of another, one character would have to be on a path between the root and some other character. But this is impossible, because the characters occur only at the leaves of the tree.

To construct such a tree from the frequency table, the Huffman algorithm begins by finding the two characters with the lowest nonzero frequencies. (If a character has a frequency of zero, it can be ignored. If more than one character has the minimum frequency, it doesn't matter which is chosen.) It combines these two characters into a tree by creating a new node and making the characters its children. The tree is assigned a frequency that is the sum of the frequencies of its children. The algorithm again picks the two lowest-frequency values that occur, this time including the newly constructed tree in its search. Again, it pairs the two lowest values into a tree. It continues this process until only one tree remains; that tree is the Huffman code tree for the frequency table, and the character codes can be read off it directly. Figure 2 shows how the algorithm constructs the tree in figure 1. Figure 3 provides a more formal description of the algorithm.

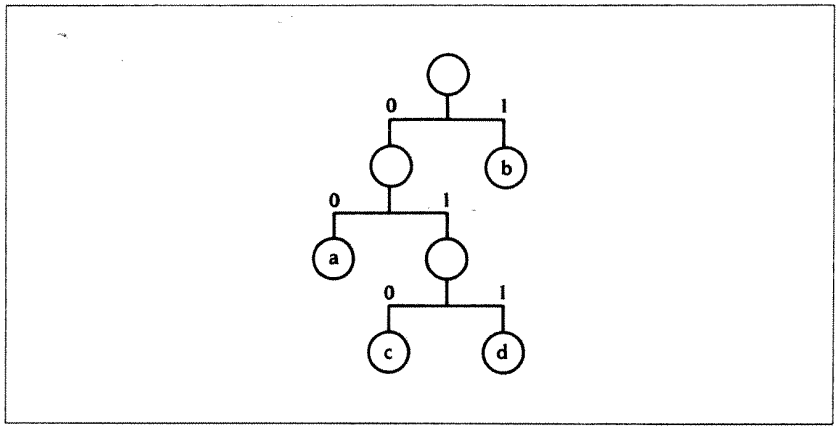


Figure 1: A *simple code tree.*

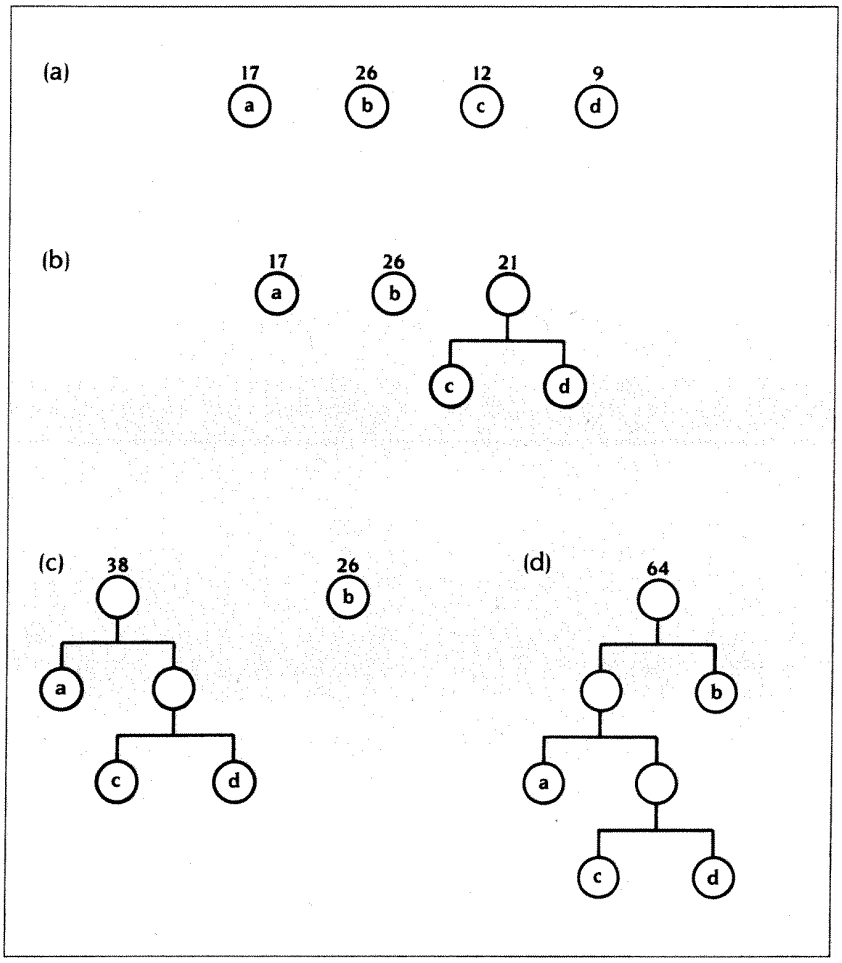


Figure 2: *How the Huffman algorithm constructs the tree in figure 1: (a) It begins with a list of characters and their frequencies; (b) It combines the characters with the lowest frequencies, "c" and "d," into a subtree and puts the subtree in the list; (c) Now the character "a" and the newly constructed subtree have the lowest frequencies, so they are combined; (d) Finally, the remaining character is combined with the subtree to form the completed code tree.*

HUFFMAN IS "THE BEST"

It should be clear that the Huffman algorithm constructs a binary tree that can be used to encode characters, but it is far from obvious that it constructs the best such tree. It does, but before proving it, I should define what I mean by "best."

Since the goal of the algorithm is to minimize the space occupied by a file, it's clear that the best encoding of the file is the one that will take up the

(continued)

least space, that is, the fewest bits. How does that translate to code trees? Let's assume that the frequency table used to construct the tree consisted of the number of times each character occurred in the file rather than, say, the percentages of the characters' occurrence in English. (The Huffman algorithm will work either way, but it is guaranteed to be best only with the former interpretation of the frequency table.) Given a code tree, it's easy to calculate exactly how many bits the corresponding file will occupy. Observe that the length of the path from the root to a character—the number of branches you have to traverse to get to the character—is the same as the number of bits in that character's code. Since the frequency of a character is the number of times it occurs in the file, multiplying the frequency of the character by the number of bits in its code yields the total number of bits the character will occupy in the encoded file.

Performing the same calculation for each character in the tree and adding the results together yields the number of bits in the entire encoded file. This value is called the weighted path length of the tree because it is computed by multiplying the length of each root-to-leaf path in the tree by the path's weight, which is the frequency of the character at the leaf. For instance, the weighted path length of the tree in figure 2d is $17\times2 + 12\times3 + 9\times3 + 26\times1 = 123$. The weighted path length of a code tree is what you should minimize. I am claiming that, for any distribution of frequencies, the Huffman algorithm constructs the tree that has the smallest weighted path length of all possible code trees constructed from that frequency distribution.

Some Observations

Three interesting facts about code trees bear on the proof of the above claim. The first I call the generality observation: You can construct any code tree by repeatedly taking two subtrees and combining them into a larger tree. Therefore, the important aspect of the Huffman algorithm is not the way it builds up the tree from subtrees, but the fact that it always chooses the smallest subtrees to combine at each step.

The second observation is this: If you increase the path length of a subtree by 1, you add the frequency of the subtree to the weighted path length of the entire tree. I call this the lowering observation because increasing a subtree's path length is just like lowering the level at which the subtree occurs in the overall code tree. For instance, say I have a tree whose weighted path length is 30. A particular subtree has frequency 5, and the path from the root to that subtree is three branches long. If I increase the path to four branches while holding everything else constant, the tree's weighted path length becomes 35.

This is easy to see when the subtree is a leaf. If I increase the path length of a leaf by 1, it's as if I added 1 bit to the character code for the character at that leaf. If the character occurs n times, I have in effect added n bits to the length of the file. But that's the same as saying that I have increased the tree's weighted path length by n, since I showed before that the weighted path length is equal to the number of bits in the file.

For the case of a nonleaf subtree, recall that the subtree's frequency is the sum of the frequencies of its children, which are in turn the sum of the frequencies of their children, and so on down until the leaves are reached. So the frequency of a subtree is just the sum of the frequencies of all the leaves under that subtree. Now, if I increase the path length of the subtree by 1, I have also increased the path length of each of its leaves by 1. By the argument in the previous paragraph, I have increased the weighted path length of the tree by the sum of the frequencies of the subtree's leaves. But this is just the frequency of the subtree itself.

The third and most important observation I call the swap observation. Consider any two subtrees of a code tree. If the higher subtree—the one with the shorter path length—has a smaller frequency than the lower subtree, you can swap the two subtrees and thereby decrease the weighted path length of the entire tree. Figure 4 provides an example. I think the easiest way to convince yourself of the truth of this observation is to imagine lowering the higher subtree one level at a time until it reaches the level of the lower subtree. By the lowering observation, each time the subtree descends a level, the entire tree's weighted path length increases by the frequency of the subtree. Now imagine raising the lower subtree up in the tree until it reaches the level of the higher subtree. By the same reasoning used for the lowering observation, you can see that each level this subtree ascends will decrease the tree's overall weighted path length. Now which is greater, the amount of increase or of decrease? Both subtrees move the same number of levels because they are being swapped, but the subtree responsible for the decrease is the ascending subtree, which has a higher frequency. So the net effect must be that the overall weighted path length is decreased.

```
For each character with a nonzero frequency,
   add the character to the list of subtrees.
While the list of subtrees contains more than one subtree,
   remove from the list the two subtrees with the smallest frequencies;
   make them the children of a new subtree whose frequency is the sum of
      their frequencies;
   add the new subtree to the list.
The remaining subtree is the Huffman code tree.
```

Figure 3: *The Huffman algorithm takes a table of characters and their frequency counts as input and produces a code tree as output.*

The Proof

I will now show that no code tree using a given frequency distribution

(continued)

can have a lower weighted path length than the tree produced by the Huffman algorithm using the same frequency distribution. Other trees may have the same weighted path length, but none can be better.

By the generality observation, any code tree can be constructed by building it up out of smaller subtrees. To be more concrete about this, say that you are building a code tree that diverges from the one that would be built by the Huffman algorithm. At the point of divergence, the frequencies of the remaining subtrees are 9, 12, 17, 26, and 32. The Huffman algorithm would combine the 9 subtree and the 12 subtree (see figure 5a), but to make the divergent subtree, you combine the subtrees 12 and 26 and then 9 with 17.

In this divergent tree, the lowest-frequency subtree and the one it is combined with (subtrees 9 and 17 in this example) must be at the same level. But where does the subtree of second-lowest frequency (subtree 12) turn up? There are three possibilities: at a lower level than 9 (figure 5b), a higher level than 9 (figure 5c), or the same level (figure 5d).

If 12 is lower than 9, the swap observation says that we can swap the two subtrees to get a better tree. If 12 is higher than 9, it must also be higher than 17, so we can swap 12 and 17 to again get a better tree. Any tree that diverges from the Huffman tree in one of these two ways can't have the lowest weighted path length.

Now to the third case. If 12 is on the same level as 9, it is also on the same level as 17. Swapping 12 and 17 won't change the weighted path length of the tree. But it will make 9 and 12 children of the same node, which is just where the Huffman algorithm would have put them. So although the tree in figure 5d diverged from the Huffman algorithm at this point, there is another tree with the same weighted path length that doesn't diverge from it here.

Abandon the tree in figure 5d and begin the argument again with the tree in figure 5e. The subtrees to consider are 17, 21, 26, and 32 (9 and 12 subtrees make a subtree of frequency 21). Since 21 is higher on the tree than both 17 and 26, it can be swapped with 26 to get a better code tree. Hence, the tree in figure 5e is worse than the Huffman tree. In general, one of two things will happen to a new tree. Either it will diverge from the Huffman tree as in case 1 or case 2 above, showing itself to be worse than the Huffman tree, or it won't diverge at all, in which case it just is the Huffman tree.

That concludes the proof. Let me sum up. The idea is that a tree can diverge from the tree constructed by the algorithm in only three ways. If it diverges in one of the first two ways, it can't be minimal. Only if it diverges in the third way can it be minimal—but then the Huffman algorithm will produce a different tree with the same weighted path length, hence also minimal. It follows that no code tree can be better than the Huffman tree.

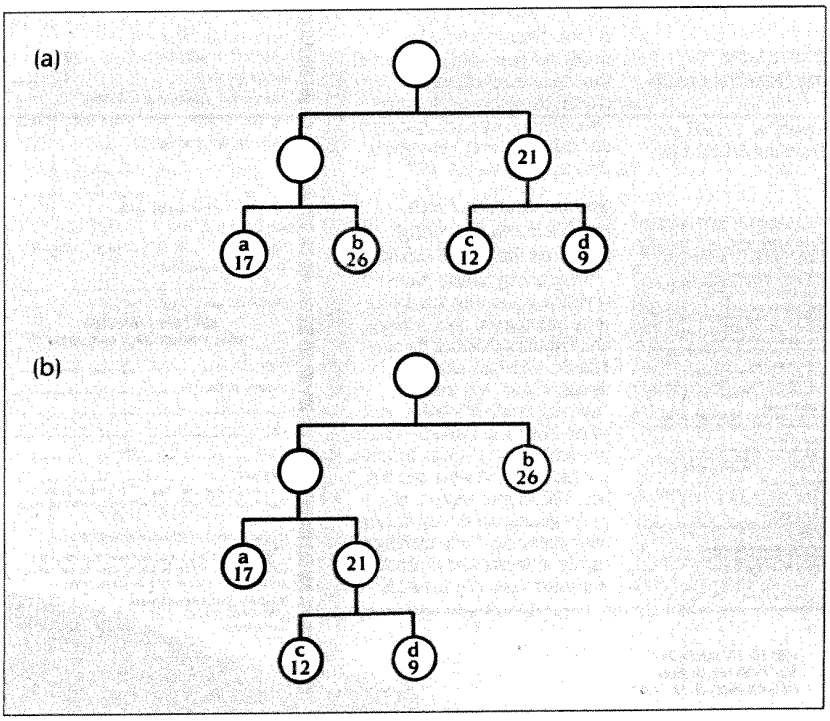


Figure 4: *An illustration of the swap observation. Before the swap, the red subtree is higher in the code tree than the blue subtree. The weighted path length of the tree before the swap is 128; after the swap, it's 123.*

Is "Best" Always Best?

I've only shown that, given a frequency distribution, the Huffman algorithm produces the shortest character code with the prefix property over that distribution. Huffman coding is optimal only in this narrow sense. In many cases, a Huffman code based on characters will fare worse than some other scheme. For instance, consider the sentence "John, where Bill had had 'had,' had had 'had had.'" An encoding in which "had" is 1, "Bill" is 01, "where" is 001, and "John" is 000 does much better than a Huffman code based on characters. In fact, that encoding is what you would get with the Huffman algorithm if you applied it to the words of the sentence. In many cases, the Huffman algorithm will do poorly no matter what. For example, pictures are often represented digitally as bit maps. Since many pictures consist of large regions that contain all 0s or 1s, a preferable compression technique is run-length encoding, in which long sequences of identical bits (or characters) are represented by a single character and a count indicating the number of consecutive occurrences of that character.

Despite its drawbacks, Huffman coding applied to the characters of a

(continued)

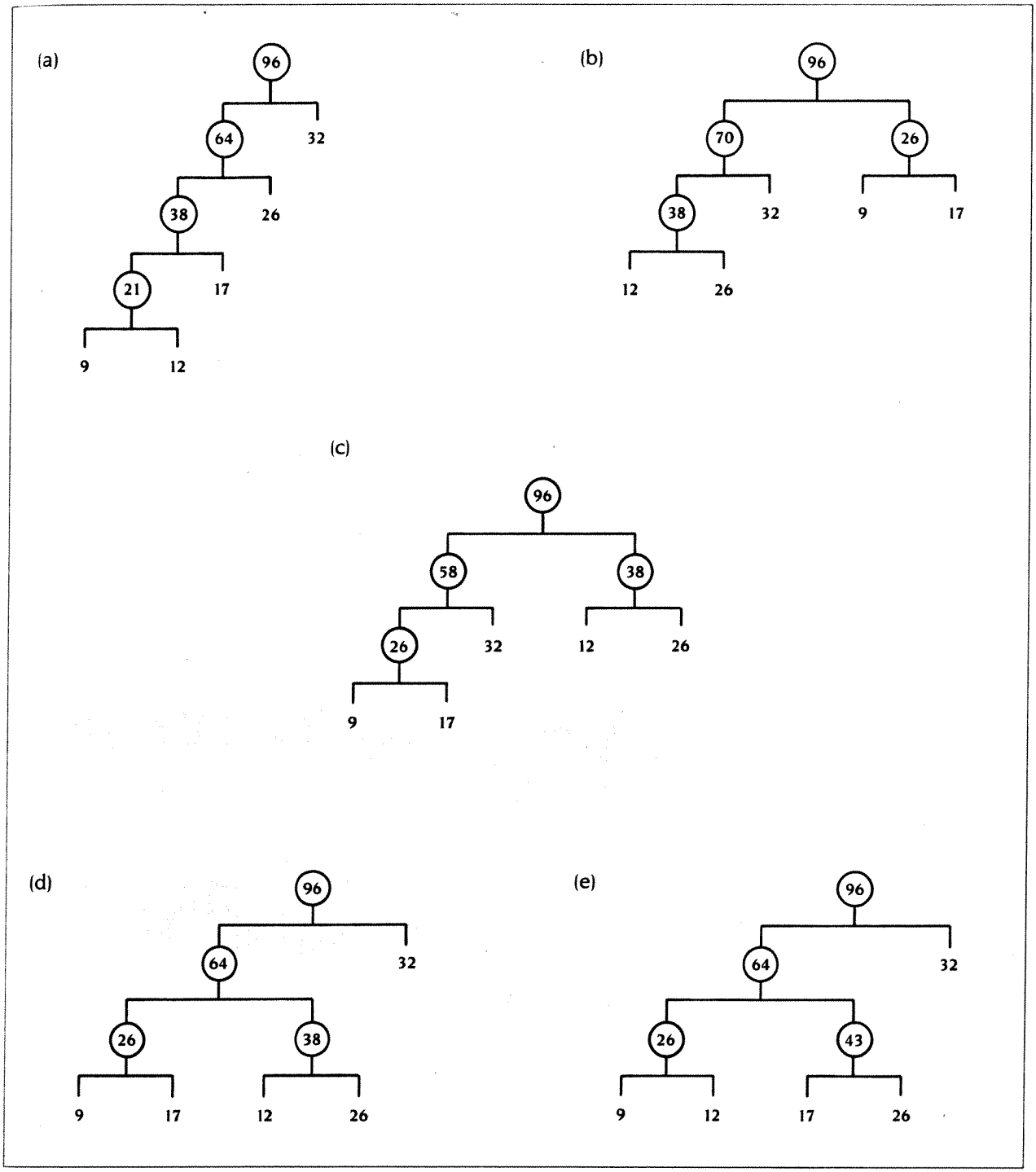


Figure 5: *Given the subtrees 9, 12, 17, 26, and 32, examine some trees that diverge from the tree produced by the Huffman algorithm. For now, look at only the frequencies of these subtrees and ignore the characters these frequencies correspond to: (a) The tree the Huffman algorithm would build from these subtrees; (b) The subtree with the second-lowest frequency (12) is lower in the tree than the subtree with the lowest frequency; (c) The subtree with the second-lowest frequency is higher in the tree than the subtree with the lowest frequency; (d) The second-lowest frequency is at the same level as the lowest frequency; (e) Swapping 12 and 17 gives a tree that diverges from the tree that would be produced by the Huffman algorithm at a later point than that of figure 5d and yet still has the same weighted path length of the tree of figure 5d.*

file is usually your best bet, especially for text files like program sources and written documents.

HUFFMAN IMPLEMENTATION

My implementation of the Huffman algorithm uses a data structure I call a node to construct the code tree. A node is just a Modula-2 or Pascal record that contains five fields: a character; a frequency count; two children, which are pointers to other nodes; and a parent, which is also a pointer to a node. The children pointers are used to trace down the tree when decoding, and the parent pointers are used to trace paths from the leaves to the root for encoding.

My program begins by creating a node for each character in the frequency table. The character is assigned to the character field of the node, and its frequency is assigned to the node's frequency field. Since this node is a leaf, both its children are assigned the value NIL (in Pascal and Modula-2, NIL is a pointer value that points to nothing). The program also constructs an array of the leaves indexed by character to serve as an index into the tree for use in encoding.

The program then places all the leaf nodes in a list and does the following: When the list contains more than one node, it removes the two smallest, combines them, and puts the resulting node back in the list. To combine two nodes A and B, the program constructs a new node C, sets its children fields to point to A and B, sets the parent fields of A and B to point to C, and assigns the sum of the frequencies of A and B to C's frequency field. (C's character field is never used, so it need not be set.) When only one node is left in the list, that node is the root of the Huffman tree.

I haven't specified how the list of nodes is represented or how the smallest nodes are chosen. My program uses an array for the list, and I choose the smallest node by stepping through each node in the array and comparing it against the smallest found so far. The procedure is similar to Selection Sort and results in a Huffman algorithm with the same time complexity: $O(n^2)$, where n is the number of characters in the frequency table (see my "An Analysis of Sorts" on page 104 of the September 1985 BYTE for an explanation of Selection Sort and "Big O" notation). Although it's possible to do better ($O(n \log n)$, in fact) using a more sophisticated data structure, it's probably not worth the trouble, for two reasons.

First, if you are encoding characters, that is, bytes, there cannot be more than 256 distinct characters in the file since a byte can represent only 256 distinct values. Usually, there will be many fewer. For such small values of n, it is not clear that the theoretically faster but more complex algorithm will be quicker in practice. Another reason in favor of the simpler algorithm is that, when using Huffman coding to compress a file, most of the time taken by the compression program is spent reading from and writing to the disk. For large files, the time taken to construct the tree pales in comparison.

FILE-COMPRESSION PROGRAMS

Some additional apparatus is needed before you can use the Huffman algorithm to compress files. You must write an encoding program and a decoding program. The encoding program takes a file as input and produces an encoded and (hopefully) shorter file as output. The decoding program takes an encoded file and restores it to its original state. Two problems must be solved before you write these programs: saving the code tree and performing bit-oriented I/O.

If the encoding program uses the input file to calculate the frequency distribution, the Huffman algorithm produces a minimal tree. But this method has a drawback that makes it unsuitable for short files. Since the algorithm generates a different code for each file, it is necessary to store the code tree along with the encoded file so the decoding program can do its job. If the file is too short, the number of bits saved in compression is less than the number of bits it takes to store the tree. You can solve this problem in a couple of ways. One way is to combine many short files into a longer one, which is then compressed. Another is to forgo the minimal encoding by using a common frequency distribution to encode many files. If the frequency distributions of the files are close enough to the common one, this method will save bits.

Since I want my program to work well with a wide range of files, I've chosen to compute the frequency table from the file. But this leaves me with the problem of outputting the code tree along with the encoded file. The tree should be stored in a way that allows the decoding program to reconstruct it easily, and, of course, it should take up as little space as possible.

To output the code tree, my program starts with the root node and does the following: If the node is a leaf, it outputs a 0 followed by 8 bits that represent the character stored at that leaf. If the node is not a leaf, it outputs a 1 and recursively outputs the left and right children of the node. The algorithm and its counterpart for input are shown in figure 6. Since every code tree with n leaves has $n-1$ nonleaf nodes (a fact I leave to you to verify), the space occupied by the code tree is $(8+1)n + n-1 = 10n - 1$ bits. Note that it isn't necessary to save the frequencies of the nodes; the structure of the tree and the characters appearing at the leaves suffice for decoding.

The second hurdle to be overcome is that the Huffman algorithm produces an encoding in terms of bits, but all programming languages and file systems deal with data in byte-size chunks. Routines for doing bit-by-bit input and output are required.

My solution uses a single-byte buffer to accumulate bits to be output. A counter, call it n, is initialized to zero. Each time the bit output routine is called, the nth bit of the byte is set or cleared and the counter incremented. When the byte is full, it is output to the file and the counter is reset. Input works analogously: An entire byte is read in at once and its bits doled out one at a time. The algorithms for bit-oriented I/O are shown in figure 7.

A subtle but important complication arises in bit-oriented I/O: It is im-

(continued)

possible to determine exactly where the end of a file of bits occurs. Since file systems keep track only of bytes, a program reading a file does not know how many of the bits in the file's final byte were actually written when the file was created. To get around this problem, my encoding program stores the number of characters of the file with the file's encoding. The decoding program then does not have to worry about where the end of the file is, because all it has to do is decode the indicated number of characters.

Now that all the pieces are in place, I will describe the encoding and decoding programs. The encoding program begins by reading the input file and constructing the frequency table, which is just an array of integers indexed by character. It also notes the length of the file, in characters. It then passes the frequency table to the Huffman algorithm, which constructs the code tree. The program then opens the output file, outputs the number of characters in the input file (as a 16-bit number) and the Huffman code tree (whose output format I described above), and proceeds to reread the input file and encode it. A character is encoded by looking it up in the index to the Huffman tree, which provides a pointer to the leaf containing the character. The parent links are then traced until the root of the tree is reached, then the path is retraced from top to bottom with the appropriate bits being output: 0 for left branches, 1 for right.

The decoding program begins by reading in the number of characters and the code tree from the file to be decoded. It then sets a pointer to point to the root of the code tree and reads a bit from the file. If the bit is a 1, it takes the right branch of the tree; if a 0, the left. The program reads bits and traverses the tree until a leaf is reached, then outputs the character at that leaf. It then starts again at the top of the tree to decode the next character. It decodes as many characters as indicated by the number stored at the beginning of the file.

The encoding and decoding programs consist of five modules: one encapsulates all the procedures directly related to the Huffman algorithm, two others provide CharStream and BitStream data types for performing character- and bit-oriented I/O, and the remaining two are the main modules for the two programs. Separating the Huffman algorithm and the I/O procedures into separate modules makes it possible to use them for other programs without having to copy, edit, or recompile code. [*Editor's note: The Modula-2 source code for the encoding and decoding programs is available on* BYTEnet Listings *at* (617) 861-9764. *These programs are also available on disk. See page* 469 *for details.*] ■

```
Algorithm writeTree takes a tree node as input. (To output a tree, call
    writeTree with the root of the tree.)
If the node is a leaf,
    output a 0 bit followed by the 8-bit code for the character at the leaf.
Otherwise,
    output a 1 bit;
    call outputTree with the node's left child;
    call outputTree with the node's right child.


Algorithm readTree returns a tree.
Read a bit from the file.
If it is a 1,
    call readTree to get the left child, L;
    call readTree to get the right child, R;
    construct a new node N with children L and R;
    set the parent of L and R to be N;
    return N.
If the bit is a 0,
    read the next 8 bits and convert them into a character, C;
    construct a new mode, storing C in its character field;
    set both the node's children to NIL;
    return the node.
```

Figure 6: *Algorithms for writing and reading the code tree to a file.*

```
Algorithm readBit returns a bit (1 or 0). To initialize a file for reading,
  set curBit to 7. At end of file, readBit will keep returning the last bit of the file.
If curBit = 7,
  If not end of file,
    read a byte from the file and put it in curByte;
  set curBit to 0.
Otherwise, increment curBit.
In both cases, return the value of the curBit'th bit of curByte.


Algorithm writeBit takes a bit as argument. To initialize a file for
  writing, set curBit to 0.
Set the curBit'th bit of curByte to the bit given as argument;
If curBit = 7,
  write curByte to the file;
  set curBit to 0.
Otherwise,
  increment curBit.
```

Figure 7: *Algorithms for bit-oriented* I/O. *The routines assume bits numbered from* 0 *to* 7. *A file can be opened for reading or writing, but not both at once. Both routines use the global variables curBit and curByte. The expression "the curBit'th bit of curByte" means the bit of curByte whose number is the current value of curBit.*