

Huffman Coding / Huffman Encoding / Huffman Codes

- Huffman codes are a widely used and very effective technique for compressing data (reducing size of the data or message); savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Variable length encoding technique

Why to encode and compress the size??

- To reduce cost or size of message while storing in a file or sending over a network.
- Cost of message is measured in terms of bits.

Cost of message without encoding:

Message $\rightarrow$ BCCABBDDECCBBAEDDCC

There are 20 letters in the message.
length = 20

↳ Computers use ASCII codes for characters.

↳ ASCII codes are 8 bits.

Character	ASCII	Binary
A	65	0100 0001
B	66	0100 0010
C	67	0100 0011
D	68	0100 0100
E	69	0100 0101

we have these characters in the message

↳ There are 20 characters in the message

↳ Each character takes 8-bits.

Cost (or) Size of message = $8 \times 20 = 160$ bits.

Size of the message without encoding

Fixed Length Encoding: Same number of bits in the codewords of characters

- ↳ Can we use our own code for characters instead of ASCII codes? - Yes.
- ↳ We are not using all characters in the character set.
- ↳ We use only 5 characters A, B, C, D, E in our message.
- ↳ Do we need 8-bit representation to represent only 5 characters? - NO.

8-bit ASCII codes for 128 characters
we have 5 characters in our message.

for 5 characters,

If we use 1-bit - $\left. \begin{matrix} 0 \\ 1 \end{matrix} \right\}$ can represent only 2 characters.

2 bits $\left. \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \right\}$ can represent only 4 characters.

As we have 5 characters, we can use 3-bits.

000, 001, 010, 011, 100, 101, 110, 111
↳ 8 combinations are possible. we can take any 5

Message $\rightarrow$ B C C A B B D D A E C C B B A E D D C C

Character	Count / Frequency	code
A	3	0001
B	5	001
C	6	010
D	4	011
E	2	100

Message $\rightarrow$ B C C A B B D D A E C C B B A E D D C C

001 010 011 000 001 010 011 000 100 010 011 000 001 010 011 000

↳ We have 20 characters and each takes 3-bits

↳ So, size of message = $20 \times 3 = 60$ bits.

Variable Length Encoding: Different number of bits in the codewords of characters

Huffman encoding $\rightarrow$ variable length encoding

$\downarrow$

Can still reduce the size.

- ↳ Codes are assigned based on the number of Occurrence.
- ↳ Idea is, if we assign small size codes for more appearing characters, then total size of entire message can be reduced.

Huffman Codes:

- We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.
- Suppose we have a 100,000-character data file that we wish to store compactly.
- We observe that the characters in the file occur with the frequencies given below:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

That is, only six different characters appear, and the character a occurs 45,000 times.

- Each character is represented by a unique binary string called **binary character code**.
- If we use a **fixed-length code**, we need 3 bits to represent six characters: a = 000, b = 001, . . . , f = 101. This method requires **300,000 bits** to code the entire file.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

- A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.

Figure 16.3 shows such a code;

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

- Here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.
- This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000$ bits to represent the file, a savings of approximately 25%.

Prefix codes

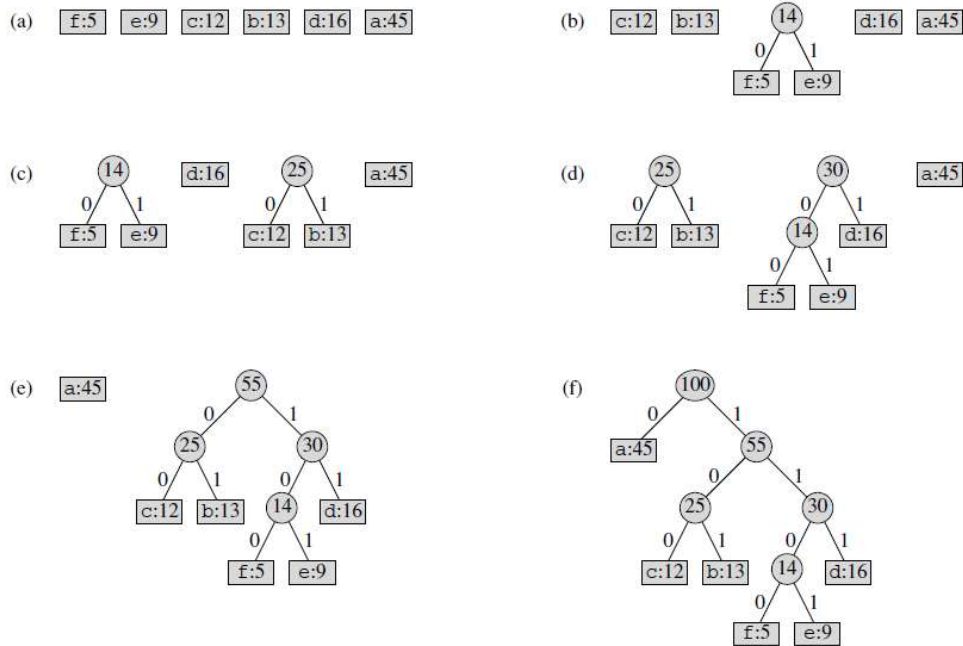
- We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called **prefix codes**.

- For example, with the variable-length prefix code of Figure 16.3, we code the 3-character file **abc** as **$0 \cdot 101 \cdot 100 = 0101100$** , where we use “ $\cdot$ ” to denote concatenation.
- Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.
- In our example, the string **001011101** **parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to aabe.**
- A **binary tree called optimal merge pattern tree** is used to generate or represent Huffman codes.
- **Leaves are the given characters.**
- We interpret the **binary codeword for a character as the path from the root to that character**, where **0** means “go to the left child” and **1** means “go to the right child.”

Constructing a Huffman code

- Huffman invented a greedy algorithm that constructs an optimal prefix code called a *Huffman code*.
- In the pseudocode, we assume that C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.
- It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ “merging” operations to create the final tree.
- A min-priority queue Q , keyed on f , is used to identify the two least-frequent objects to merge together.
- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

For our example, Huffman's algorithm proceeds as shown below:



- Since there are 6 letters in the alphabet, the initial queue size is $n = 6$, and 5 merge steps are required to build the tree.
- The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the path from the root to the letter.

HUFFMAN(C)

```

1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$ 
3  for  $i \leftarrow 1$  to  $n - 1$ 
4      do allocate a new node  $z$ 
5           $left[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$ 
6           $right[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8           $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$             $\triangleright$  Return the root of the tree.
```

- Line 2 initializes the min-priority queue Q with the characters in C .

- The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger.
- The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child.
- After $n - 1$ mergers, the one node left in the queue—the root of the code tree—is returned in line 9.

Time Complexity:

- The analysis of the running time of Huffman's algorithm assumes that Q is implemented as a binary min-heap.
- For a set C of n characters, the initialization of Q in line 2 can be performed in $O(n)$ time using the BUILD-MINHEAP procedure.
- The **for** loop in lines 3–8 is executed exactly $n-1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time.
- Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.