13.4 Text Compression and the Greedy Method

In this section, we will consider the important task of ***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 wish to reduce bandwidth for digital communications, so as to minimize the time needed to transmit our text. Likewise, text compression is useful for storing large documents more efficiently, so as to allow 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 ASCII, use fixed-length binary strings to encode characters (with 7 or 8 bits in the traditional or extended ASCII systems, respectively). The Unicode system was originally proposed as a 16-bit fixed-length representation, although common encodings reduce the space usage by allowing common groups of characters, such as those from the ASCII system, with fewer bits. 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. Furthermore, the Huffman code uses a variable-length encoding specifically optimized for a given string X over any alphabet. 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.

To encode the string X, we convert each character in X to a 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 be a prefix of another code-word in our encoding. Such a code is called a ***prefix code***, and it simplifies the decoding of Y to retrieve X. (See Figure 13.12.) 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 written 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 edge in T represents a bit in a code-word, with an edge to a left child representing a "0" and an edge to a right child representing a "1". Each leaf v is associated with a specific character, and the code-word for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v. (See Figure 13.12.) Each leaf 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 leaves 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

(b)

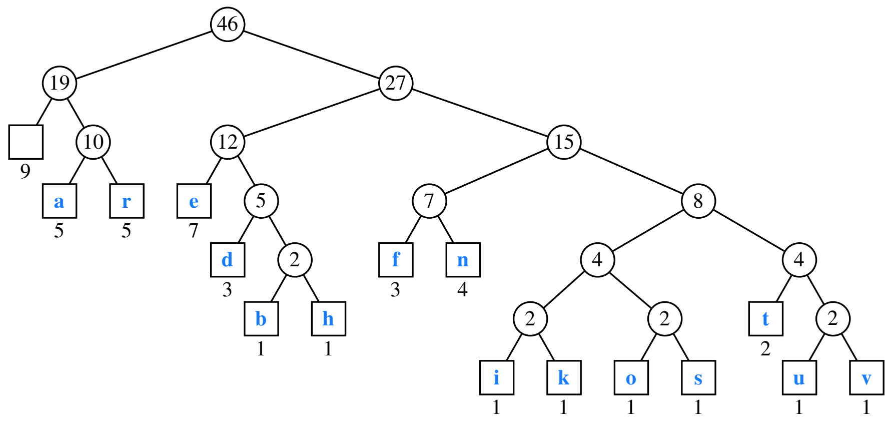

Figure 13.12: An illustration of an example Huffman code for the input string $X =$ `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the leaf where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for `"r"` is 011, and the code for `"h"` is 10111.

13.4.1 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 13.5.)

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 will be 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 here, 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 13.7: *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
      Entry $e_1 = Q$.removeMin() with $e_1$ having key $f_1$ and value $T_1$.
      Entry $e_2 = Q$.removeMin() with $e_2$ having key $f_2$ and value $T_2$.
      Create a new binary tree $T$ with left subtree $T_1$ and right subtree $T_2$.
      Insert $T$ into $Q$ with key $f_1 + f_2$.
   Entry $e = Q$.removeMin() with $e$ having tree $T$ as its value.
   return tree $T$
```

Code Fragment 13.5: Huffman coding algorithm.

13.4.2 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.