

	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

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

There are many ways to represent such a file of information. We consider the problem of designing a *binary character code* (or *code* for short) wherein each character is represented by a unique binary string. If we use a *fixed-length code*, we need 3 bits to represent six characters: $a = 000, b = 001, \dots, f = 101$. This method requires 300,000 bits to code the entire file.

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

$$(45 \cdot 1 + 12 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 5) = 224$$

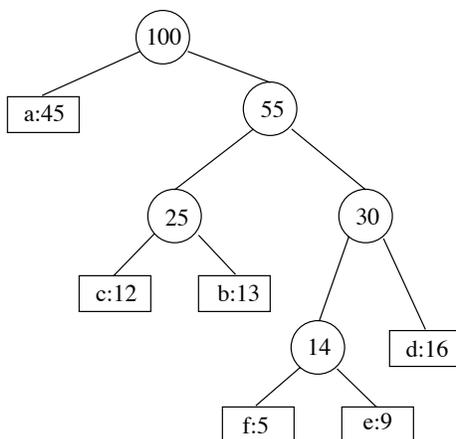
that is, 224,000 bits, and a savings of approximately 25%.

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called *prefix codes*. It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code, so there is no loss of generality in restricting attention to prefix codes.

Encoding is always simple for any prefix code; we just concatenate the codewords representing each character of the file. For example, with the variable-length code given above we represent 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. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as aabe.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. 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”. The following is the binary tree corresponding to an optimal prefix code for our running example:



Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree.

Exercise 1.10 Give the binary tree for the fixed-length code of our running example.

Exercise 1.11 Show that an optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children. Is the fixed-length code optimal?

Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see exercise 1.1).

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet C , let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c 's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c . The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

which we define as the *cost* of the tree T .

1.3.1 Constructing a Huffman code

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.

```

n ← |C|
Q ← C
for i = 1..(n - 1)
    allocate a new node z
    left[z] ← x = extract - min(Q)
    right[z] ← y = extract - min(Q)
    f(z) = f(x) + f(y)
    insert z in Q
end for

```

The idea is that the loop extracts two nodes x, y from Q which are of lowest frequency, and replaces them with a new node z , which is the result of the merge. The objects of Q are trees, initially consisting of $|C|$ many single nodes, and at the end of a single tree.

1.3.2 Correctness of Huffman's algorithm

Lemma 1.6 Let C be an alphabet in which each character c has frequency $f(c)$. Let x and y be any two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

PROOF: The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can do this, then their codewords will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T . Without loss of generality, we assume that $f(a) \leq f(b)$ and $f(x) \leq f(y)$. Since the latter are the two lowest frequencies and the former are any two frequencies, it follows that $f(x) \leq f(a)$ and $f(y) \leq f(b)$.

Let T' be T with a, x exchanged, and let T'' be T' with b, y exchanged. Then, the difference of costs of T', T'' is

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x)d_{T'}(x) - f(a)d_{T'}(a) & (*) \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) - f(a)d_T(x) & (**) \\ &= (f(a) - f(x))(d_T(a) - d_T(x)) \geq 0 & (***) \end{aligned}$$

Similarly, $B(T') - B(T'') \geq 0$. Therefore, $B(T'') \leq B(T)$, and since T is optimal, it follows that $B(T) = B(T'')$, T'' is also optimal. \square

Exercise 1.12 Make sure you can justify each of the steps $(*)$, $(**)$, $(***)$ in the above proof. Also, carry out the steps of the proof $B(T') - B(T'') \geq 0$.

Lemma 1.7 Let x, y be two characters in C of minimum frequency. Let $C' = C - \{x, y\} \cup \{z\}$, where $f(z) = f(x) + f(y)$. Let T' be any tree representing an optimal prefix code for C' . Then the tree T , obtained from T' by replacing z with an internal node with children x, y is an optimal prefix code for C .

Exercise 1.13 Prove lemma 1.7. (**Hint:** First show that $B(T) = B(T') + f(x) + f(y)$, or, equivalently $B(T') = B(T) - f(x) - f(y)$. Now prove the lemma by contradiction: suppose that T is not an optimal prefix code for C . Then there exists a T'' such that $B(T'') < B(T)$. By lemma 1.6 we can assume T'' has x, y as siblings . . .)

Theorem 1.2 Huffman's algorithm produces an optimal prefix tree.

Exercise 1.14 Show that this theorem follows from lemmas 1.6 and 1.7. Give a suitable definition of "promising" and show that it is a loop invariant, and conclude that the algorithm produces an optimal code.

Exercise 1.15 Suppose that we have an alphabet C , $|C| = n$, where $f[c_i] = f_i$, in other words, the frequency of the i -th symbol, c_i , is the i -th Fibonacci number. How would an optimal Huffman code tree look for such an alphabet? Recall that $f_1 = f_2 = 1$ and $f_{i+2} = f_{i+1} + f_i$.

2 Stable Marriage

The *Stable Marriage Problem*² was introduced by Gale and Shapley in 1962 and is related to the problem of college admissions. They gave an algorithm for solving the finite problem, which was later discovered to have been used in the matching of graduate medical students with hospitals since 1952. Other variants of the problem have been studied in computer science, economics, game theory and operations research.

An instance of the stable marriage problem of size n consists of two disjoint finite sets of equal size $B = \{b_1, b_2, \dots, b_n\}$ (the set of *boys*) and $G = \{g_1, g_2, \dots, g_n\}$ (the set of *girls*). In addition, each boy b_i has a ranking or a linear ordering $<_i$ of G which reflects his preference for the girls that he wants to marry. That is, if $g_j <_i g_k$, then b_i would prefer to marry g_j over g_k . Similarly each girl g_j has a ranking or linear ordering $<^j$ of B which reflects her preference in the boys she would like to marry.

A *matching* (or *marriage*) M is a 1-1 correspondence between B and G . We say that b and g are *partners* in M if they are matched in M and write $p_M(b) = g$ and also $p_M(g) = b$. A matching M is *unstable* if there is a pair (b, g) from $B \times G$ such that b and g are not partners in M but b prefers g to $p_M(b)$ and g prefers b to $p_M(g)$. Such a pair (b, g) is said to *block* the matching M and is called a *blocking pair* for M . A matching M is *stable* if there is no blocking pair for M .

The result of Gale and Shapley is that any marriage problem has a solution. In fact, they give an algorithm which produces a solution in n stages and takes $\leq o(n^3)$ steps.

2.1 The Gale-Shapley Algorithm

The matching M is produced in stages M_s so that b_t always has a partner at stage $s \geq t$ and $p_{M_t}(b_t) <_t p_{M_{t+1}}(b_t) <_t \dots$. On the other hand, for each $g \in G$, if g has a partner at stage t , then g will have a partner at each stage $s \geq t$ and $p_{M_t}(g) >^t p_{M_{t+1}}(g) >^t \dots$. Thus, as s increases, the partners of b_t become less preferable and the partners of g become more preferable.

Here is the algorithm:

Stage 1. At stage 1, b_1 chooses the first girl g in his preference list and we set $M_1 = \{(b_1, g)\}$.

Stage ($s + 1$). At the end of stage s , assume that we have produced a matching

$$M_s = \{(b_1, g_{i(1,s)}), \dots, (b_s, g_{i(s,s)})\}.$$

We will say that partners in M_s are “engaged”. The idea is that at stage $s + 1$, b_{s+1} will try to get a partner by “proposing” to the girls in G in his order of preference. When b_{s+1} proposes to a girl g_j , g_j accepts his proposal if either g_j is not currently engaged or is currently engaged to a boy b such that $b_{s+1} <^j b$. In the case where g_j prefers b_{s+1} over her current partner b , then g_j breaks off an engagement with b and b then has to search for a new partner. To be more precise, we begin stage $s + 1$ by letting $M = M_s$ and letting $b^* = b_{s+1}$. Then we apply the following routine. We have b^* propose to the girls in order of his preference until once accepts. Here g will accept the proposal as long as she is either not engaged or prefers b^* to her current partner $p_M(g)$. Then we add (b^*, g) to M and proceed according to one of the following two cases:

²This section is based on §2 in the article *Proof-Theoretic Strength of the Stable Marriage Theorem and Other Problems*, by Douglas Cenzer and Jeffrey B. Remmel.

1. If g was not engaged, then we terminate the procedure and let $M_{s+1} = M \cup \{(b^*, g)\}$.
2. If g was engaged to b , then we set $M = (M - \{(b, g)\}) \cup \{(b^*, g)\}$ and $b^* = b$ and we continue.

Exercise 2.1 Show that there is exactly one girl that was not engaged at stage s but is engaged at stage $(s + 1)$ and that, for each girl g_j that is engaged in M_s , g_j will be engaged in M_{s+1} and that $p_{M_{s+1}}(g_j) <^j p_{M_s}(g_j)$.

The consequence of exercise 2.1 is that, for any girl g_j , once she becomes engaged, she will remain engaged and her partners will only gain in preference as the stages proceed.

Exercise 2.2 Show that each b need only propose at most once to each g , which gives an upper bound of $(s + 1)^2$ steps in the procedure.

Exercise 2.3 Suppose that $|B| = |G| = n$. Show that at the end of stage n , M_n will be a stable marriage.

Exercise 2.4 We say that a matching (b, g) is *feasible* if there exists a stable matching in which b, g are partners. We say that a matching is *boy-optimal* if every boy is paired with his highest ranked feasible partner. We say that a matching is *boy-pesimal* if every boy is paired with his lowest ranking feasible partner. Similarly, we define *girl-optimal/pesimal*. Show that our version of the algorithm produces a boy-optimal & girl-pesimal stable matching.