Greedy algorithms – part 3

Huffman code

A useful application for greedy algorithms is for compression—storing images or words with least amount of bits.

1. Example of coding letters (inefficiently)-

A -> 00 (“code word”)

B -> 01

C -> 10

D -> 11

AABABACA is coded by:

0000010001001000

This is wasteful; some characters might appear more often than others, but all are represented with two bits.

2. More efficient: if some characters appear more frequently, then we can code them with shorter length in bits. Let’s say A appears more frequently and then B.

A -> 0 (frequent)

B -> 10

C -> 110

D -> 111 (less frequent)

AABABACA is coded by:

001001001100

We represented the same sequence with less bits = compression. This is a variable length code.

This is for instance relevant for the English language (“a” more frequent than “q”).

Prefix codes: We consider only codes in which no code word is a prefix for the other one (= a start for the other one).

[so we’re really only considering non prefix codes, although that’s the word that is used]

Prefix codes are useful because as we’ll see, it is easier to decode (go from 001001001100 to the characters).

Example from book:

	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 we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

Fixed: Coding entire file: 3 bits every character: 3 x 100000 = 300,000 bits

Variable: (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000 = 224,000 bits

Trees corresponding to the coding examples:

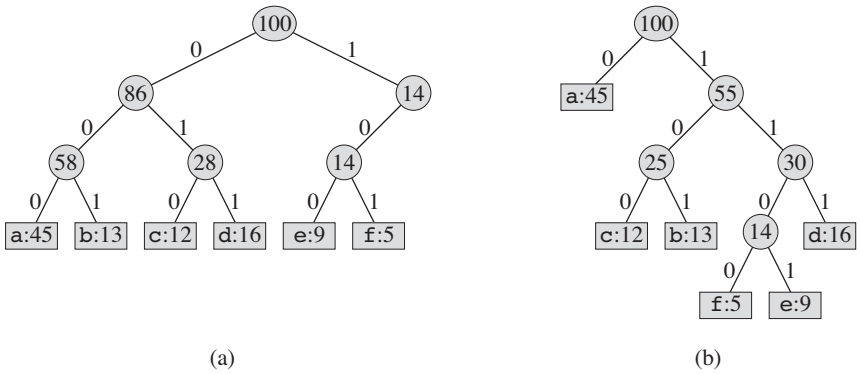

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . . . , f = 101. **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.

Some notes on the trees:

- Later: how to construct tree and the frequency nodes.

- We can easily use tree for decoding – keep going down until reach a leaf as you go through 101 (b) 0 (a) 1101 (f) (in the variable length).

- Variable length is a full binary tree (two children for every node until reach leaves). Fixed length is not.

Main greedy approach for constructing the Huffman tree: Begins with a set of leaves, and each time identifies the two least frequent objects to merge together.

When we merge the two objects, the result is now an object whose sum is the frequency of the merged objects.

An example for constructing a Huffman tree is given on the next page.

Example constructing Huffman code tree:

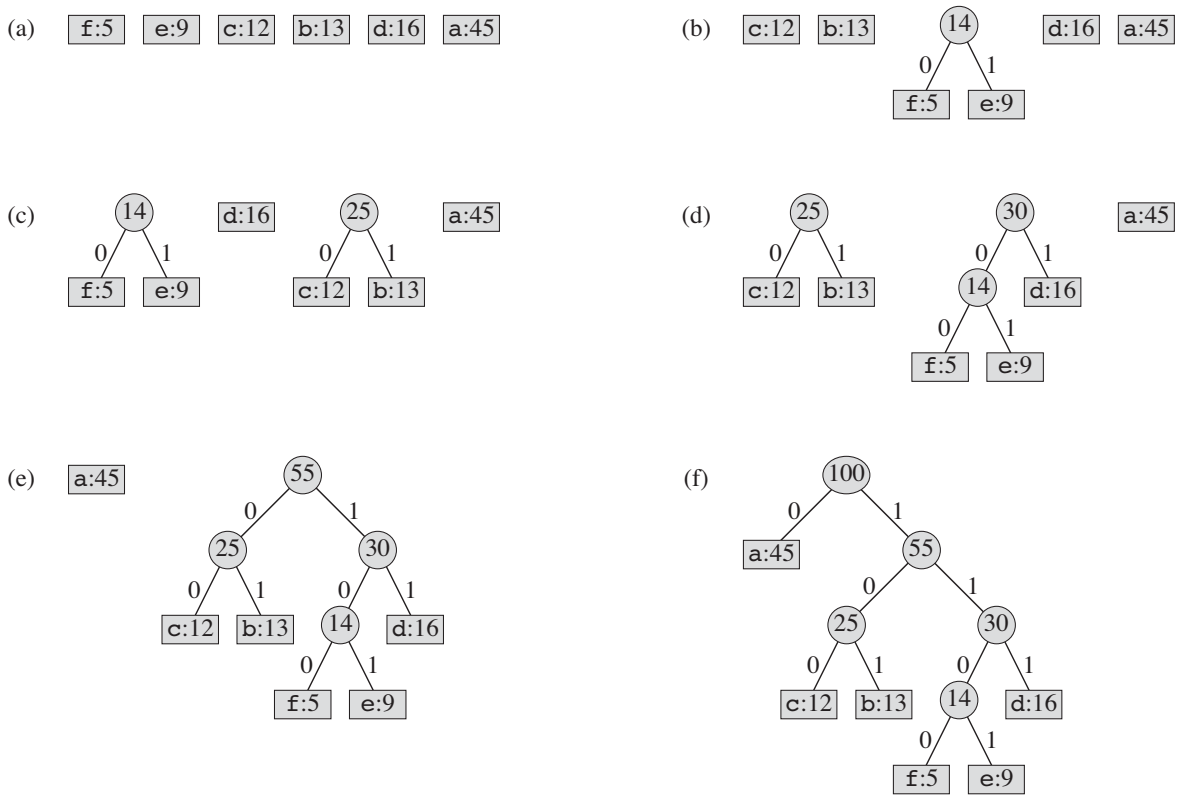

Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

Pseudo code:

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Main idea Huffman pseudo code: Repeatedly extracts two minimum frequencies from a priority queue Q (eg, a heap) and merges them as a new node in the queue. At the end, returns the one node left in the queue, which is the optimal tree.

Run time: Huffman code:

For loop runs n-1 times O(n)

Each extracting min requires O(log n)

Total: O(n log n)

(the heap initialization also requires O(n), which we didn't count above, but does not change the overall run time)