Analysis of Algorithms

Greedy Algorithms,
Huffman Code

Fractional Knapsack

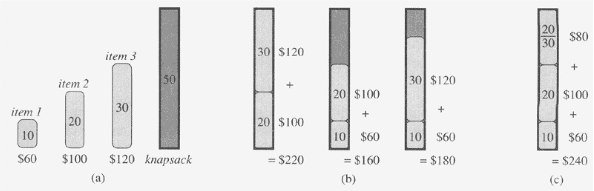
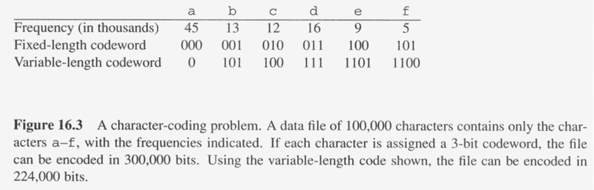


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Huffman Code

- A data file with100K characters, which we want to store or transmit compactly.
- Only 6 different characters in the file, with their frequencies shown below.



	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.

Huffman Code

- Design binary codes for the characters to achieve maximum compression
- Using fixed length code, we need 3 bits to represent six characters as shown in example.
- Storing the 100K characters requires 300K bits using this code.
- Can we do better?

Huffman Codes

- We can improve on this using variable length codes
- Motivation: use shorter codes for more frequent letters, and longer codes for infrequent letters as shown in example.
- Using code 2, the file requires (1*45 +3*13+ 3*12 + 3*16 + 4*9 + 4*5) K bits, which is 224K bits.
- Improvement is 25% over fixed length codes. In general, variable length codes can give 20% - 90% savings.

Variable Length Codes

- In fixed length coding, decoding is trivial. Not so with variable length codes.
- Suppose 0 and 000 are codes for x and y what should decoder do upon receiving 00000?
- We could put special marker codes but that reduce efficiency.
- Instead we consider prefix codes: no codeword is a prefix of another codeword. (perhaps prefix free code would be a better name)

Variable Length Codes

- So 0 and 000 will not be prefix codes, but 0,101,100,111,1101,1100 are prefix code.
- To encode, just concatenate the codes for each letter of the file; to decode, extract the first valid codeword, and repeat.
- Example: Code for ‘abc’ is 0101100.
- ‘001011101’ uniquely decodes to ‘aabe’.

Tree Representation

- Decoding best represented by a binary tree, with letter as leaves.
- Code for a letter is the sequence of bits between root and that leaf

Tree Representation

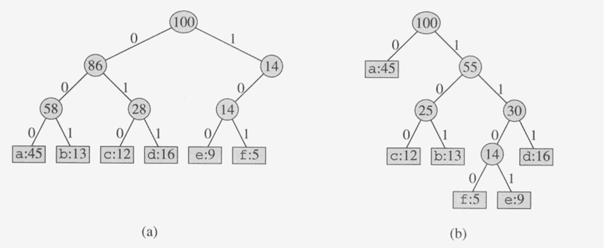


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, …,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

Tree Representation

- An optimal tree must be full: each internal node has two children. Otherwise we can improve the code.
- The fixed length code above is not optimal.

Measuring optimality

- Let C be the alphabet. Let f(x) be the frequency of a letter x ∈ C.
- Let T be the tree for a prefix code; let $d_T(x)$ be the depth of x in T.
- The number of bits needed to encode our file using this code is

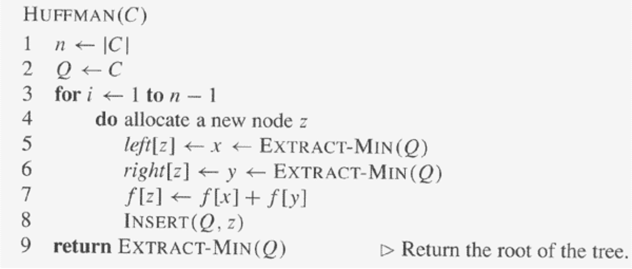
$$B(T) = \sum_{x \in C} f(x) d_T(x)$$

- We want T that minimizes B(T)

Huffman's Algorithm

- Initially, each letter represented by a single node tree. The weight of the tree is the letter's frequency.
- Huffman repeatedly chooses the two smallest trees (by weight), and merges them.
- The new tree's weight is the sum of the two children's weights
- If there are n letters in the alphabet, there are n-1 merges.

Pseudocode



```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n - 1$
    do allocate a new node $z$
        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
        $f[z] \leftarrow f[x] + f[y]$
        INSERT($Q, z$)
return EXTRACT-MIN($Q$)          ▷ Return the root of the tree.
```

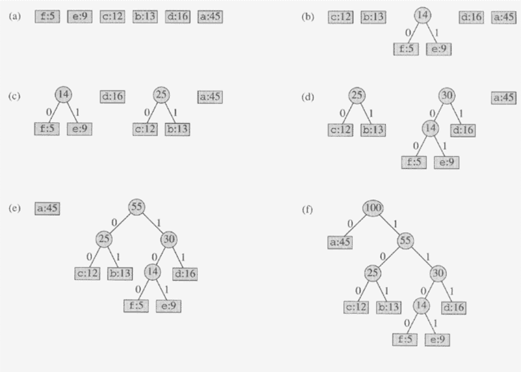


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its 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.

Complexity

- Time complexity is O(n logn). Initial sorting plus 'n' heap operations.