Greedy algorithms – part 2, and Huffman code

Two main properties:

1. Greedy choice property: At each decision point, make the choice that is best at the moment. We typically show that if we make a greedy choice, only one property remains (unlike dynamic programming, where we need to solve multiple subproblems to make a choice)

2. Optimal substructure: This was also a hallmark of dynamic programming. In greedy algorithms, we can show that having made the greedy choice, then a combination of the optimal solution to the remaining subproblem and the greedy choice, gives an optimal solution to the original problem. (note: this is assuming that the greedy choice indeed leads to an optimal solution; not every greedy choice does so).

Greedy vs dynamic:

- both dynamic programming and greedy algorithms use optimal substructure

- but when we have a dynamic programming solution to a problem, greedy sometimes does or does not guarantee the optimal solution (when not optimal, can often prove by contradiction; find an example in which the greedy choice does not lead to an optimal solution)

- could be subtle differences between problems that fit into the two approaches

Example: two knapsack problems.

a. 0-1 knapsack problem

- ***n*** items worth ***vi*** dollars each, and weighing ***wi*** pounds each.

- a thief robbing a store (or someone packing for a picnic…) can carry at most ***w*** pounds in the knapsack and wants to take the most valuable items

- It's called 0-1, because each item can either be taken in whole, or not taken at all.

b. Fractional knapsack problem:

- same as above, except that thief (or picnic packer) can now take fractions of items.

Here the fractional knapsack problem (b) has a greedy strategy that is optimal but the 0-1 problem (a) does not!

We show the figure in the book, and then give a brief explanation.

Figure:

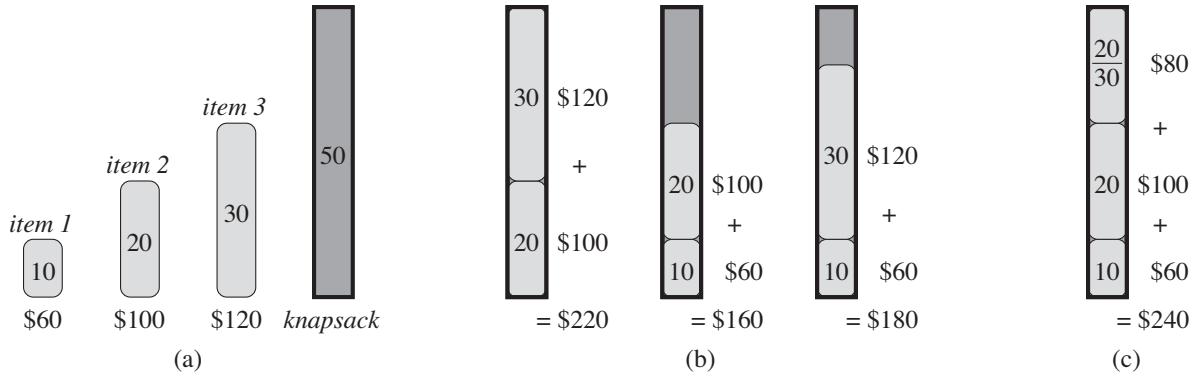

Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though 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.

Explanation:

a. 0-1: Consider taking items in a greedy manner based on the highest value per pound. In the 0-1 knapsack problem, this would lead to a contradiction, since then would take 10 the pound item first with value per pound equal to 6 (non optimal).

To get the optimal solution for the 0-1 problem, we must compare the solution or subproblem that includes the 10 pound item, with the solution or subproblem that excludes it. There are many overlapping subproblems we must compare.

b. Fractional: The fractional does have an optimal greedy solution of filling the highest value per pound first until the knapsack is full. This works indeed because we can fill fractions of items, and so proceed until knapsack is entirely full. See panel (c) in the figure.

Note that: both problems have optimal substructure:

a. 0-1: Consider the most valuable load that weighs at most w pounds. If we remove item ***j***, the remaining load must be the most valuable load weighing at most ***W-wj*** pounds, that the thief can take from the n-1 original items (excluding ***j***).

b. Fractional: If we remove weight ***w*** from item ***j*** (a fraction of the weight of ***j***), then the remaining load must be the most valuable weighing at most ***W – w*** that the thief can take from the ***n-1*** original items, plus ***wj – w*** pounds from item ***j***

Also note: These are well known problems you still hear about in conferences today; could be for many optimization problems…

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:

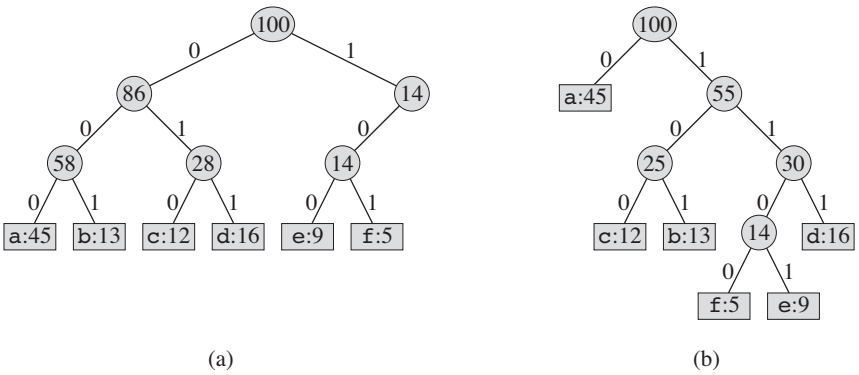

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . .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

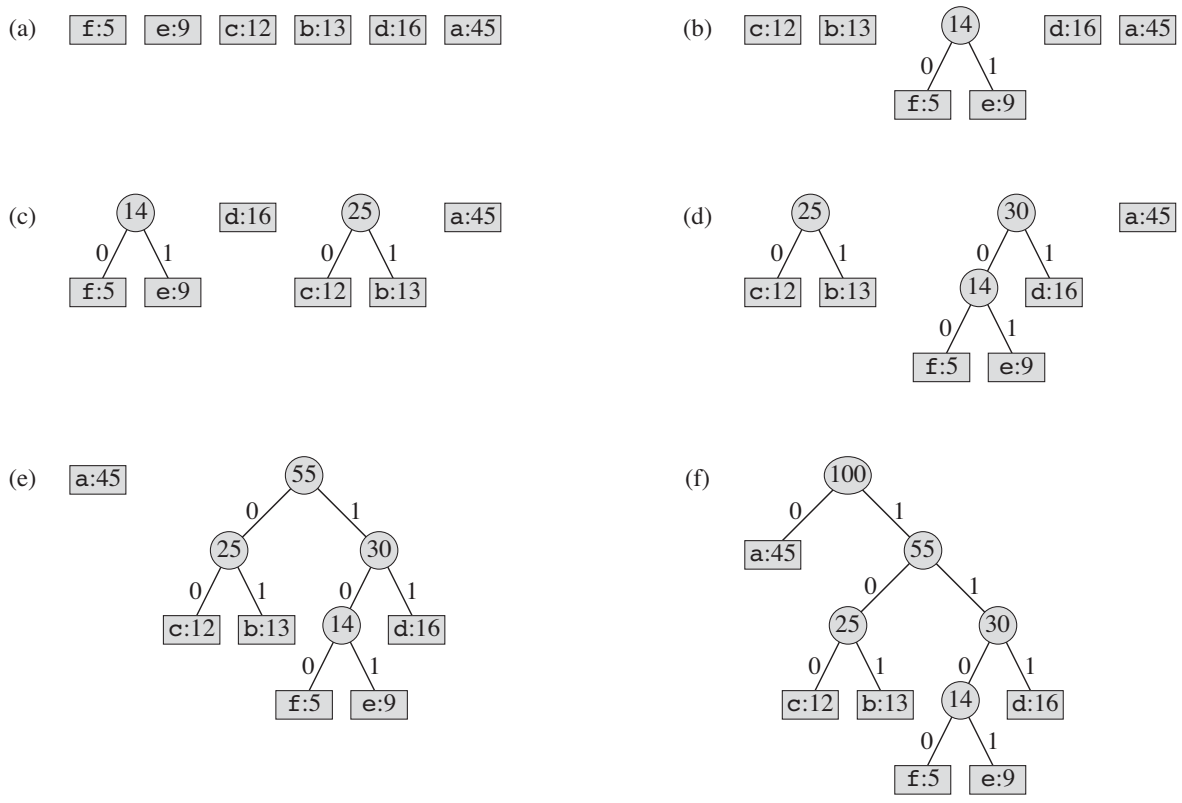

Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency.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