Introduction to Algorithms

Greedy Algorithms

Dr.N.Naveenkumar
ASP/CSE,
MEC (Autonomous)

Greedy Algorithms

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems
- **Idea:** When we have a choice to make, make the one that looks best right now
  - Make a locally optimal choice in hope of getting a globally optimal solution
- Greedy algorithms don’t always yield an optimal solution
- Makes the choice that looks best at the moment in order to get optimal solution.

Fractional Knapsack Problem

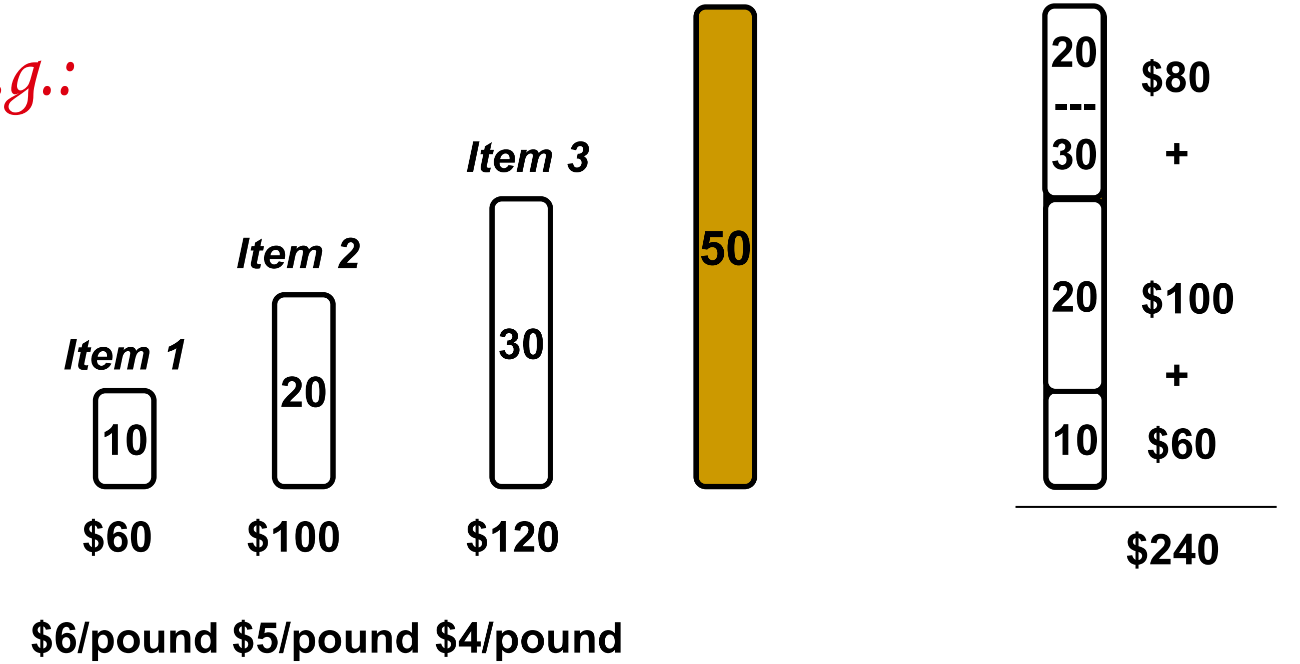
- Knapsack capacity: W
- There are n items: the i-th item has value v_i and weight w_i
- Goal:
  - find x_i such that for all $0 \le x_i \le 1$, $i = 1, 2, .., n$

    $\sum w_i x_i \le W$ and

    $\sum x_i v_i$ is maximum

Fractional Knapsack - Example

- *E.g.:*



Fractional Knapsack Problem

- Greedy strategy 1:
  - Pick the item with the maximum value
- *E.g.:*
  - $W = 1$
  - $w_1 = 100, \ v_1 = 2$
  - $w_2 = 1, v_2 = 1$
  - Taking from the item with the maximum value:

    Total value taken = $v_1/w_1 = 2/100$
  - Smaller than what the thief can take if choosing the other item

    Total value (choose item 2) = $v_2/w_2 = 1$

Fractional Knapsack Problem

Greedy strategy 2:

- Pick the item with the maximum value per pound v_i/w_i
- If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound
- It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Fractional Knapsack Problem

Alg.: Fractional-Knapsack (W, $v[n]$, $w[n]$)

1. While $w > 0$ and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_i w_i$

- w – the amount of space remaining in the knapsack ($w = W$)
- Running time: $\Theta(n)$ if items already ordered; else $\Theta(n\lg n)$

Huffman Code Problem

- Huffman's algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed.

Huffman Code Problem

- The more frequent a symbol occurs, the shorter should be the Huffman binary word representing it.

- The Huffman code is a prefix-free code.
  - No prefix of a code word is equal to another codeword.

Overview

- Huffman codes: compressing data (savings of 20% to 90%)
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string

	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

C: Alphabet

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.

Example

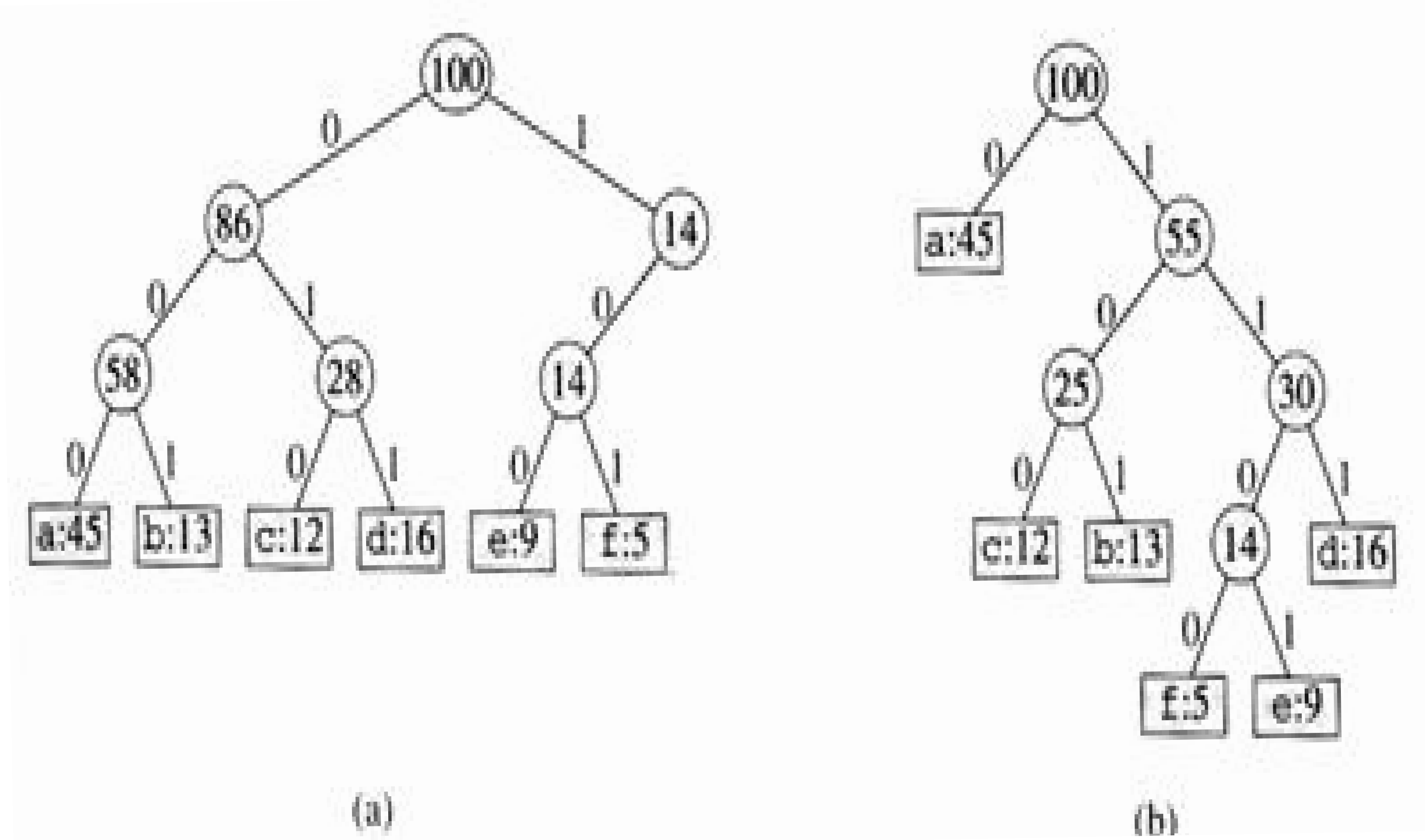
- Assume we are given a data file that contains only 6 symbols, namely a, b, c, d, e, f With the following frequency table:

	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

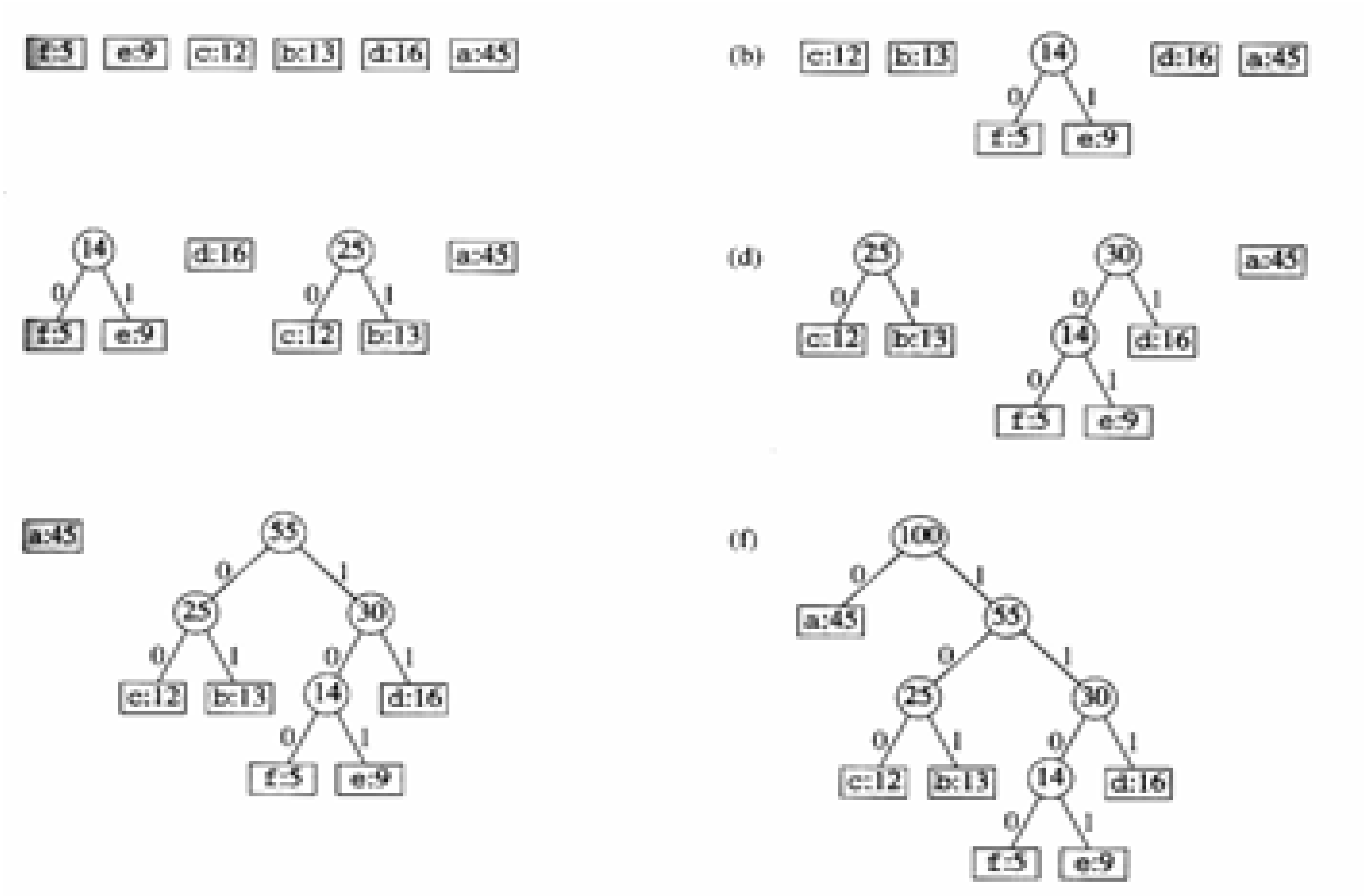
- Find a variable length prefix-free encoding scheme that compresses this data file as much as possible?

Huffman Code Problem

- Left tree represents a fixed length encoding scheme
- Right tree represents a Huffman encoding scheme



Example



Constructing A Huffman Code

```
HUFFMAN(C)  // C is a set of n characters
n ← |C|
Q ← C   // Q is implemented as a binary min-heap O(n)
for i ← 1 to n − 1
        do allocate a new node z
           left[z] ← x ← EXTRACT-MIN(Q)      O(lg n)
           right[z] ← y ← EXTRACT-MIN(Q)     O(lg n)
           f[z] ← f[x] + f[y]
           INSERT(Q, z)   O(lg n)
return EXTRACT-MIN(Q)        ▷ Return the root of the tree.
```

Total computation time = O(*n* lg *n*)

Cost of a Tree T

- For each character c in the alphabet C
  - let f(c) be the frequency of c in the file
  - let $d_T(c)$ be the depth of c in the tree
    - It is also the length of the codeword. Why?
- Let B(T) be the number of bits required to encode the file (called the cost of T)

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

Huffman Code Problem

In the pseudocode that follows:

- we assume that C is a set of n characters and that each character c €C is an object with a defined frequency f [c].
- The algorithm builds the tree T corresponding to the optimal code
- A min-priority queue Q, 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.

Running time of Huffman's algorithm

- The running time of Huffman's algorithm assumes that Q is implemented as a binary min-heap.
- For a set C of n characters, the initialization of Q in line 2 can be performed in O(n) time using the BUILD-MINHEAP
- The **for** loop in lines 3-8 is executed exactly n - 1 times, and since each heap operation requires time O(lg n), the loop contributes O(n lg n) to the running time. Thus, the total running time of HUFFMAN on a set of n characters is O(n lg n).

Prefix Code

- Prefix(-free) code: no codeword is also a prefix of some other codewords (Un-ambiguous)
  - An optimal data compression achievable by a character code can always be achieved with a prefix code
  - Simplify the encoding (compression) and decoding
    - Encoding: abc ➔ 0 . 101. 100 = 0101100
    - Decoding: 001011101 = 0. 0. 101. 1101 ➔ aabe
      - Use binary tree to represent prefix codes for easy decoding
- An optimal code is always represented by a full binary tree, in which every non-leaf node has two children
  - |C| leaves and |C|-1 internal nodes Cost:

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

Depth of c (length of the codeword)

Frequency of c

Huffman Code

- Reduce size of data by 20%-90% in general

- If no characters occur more frequently than others, then no advantage over ASCII

- Encoding:
  - Given the characters and their frequencies, perform the algorithm and generate a code. Write the characters using the code

- Decoding:
  - Given the Huffman tree, figure out what each character is (possible because of prefix property)

Application on Huffman code

- Both the .mp3 and .jpg file formats use Huffman coding at one stage of the compression

Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - We make a choice at each step
  - The choice depends on solutions to subproblems
  - Bottom up solution, from smaller to larger subproblems
- Greedy algorithm
  - Make the greedy choice and THEN
  - Solve the subproblem arising after the choice is made
  - The choice we make may depend on previous choices, but not on solutions to subproblems
  - Top down solution, problems decrease in size