Greedy Algorithms

- Huffman Coding
- Knapsack Problem

Data Encoding for Data Compression

- Normal Messaging
- Fixed Length Encoding
- Variable Length Encoding

Normal Messaging

BCCABBDDAECCBBAEDDCC

- Length 20
- Use ASCII Codes-8 bits per character
- Total-160 bits required to encode

Fixed Length Encoding

BCCABBDDAECCBBAEDDCC

3-bit fixed length code representation

Bits required: 60 bits

Reference Table : 8*5 + 3*5=55

Total Bits required:115 bits

	Frequency codeword	Fixed-length codeword
‘a’	3	000
‘b’	5	001
‘c’	6	010
‘d’	4	011
‘e’	2	100
	20	

Variable Length Coding

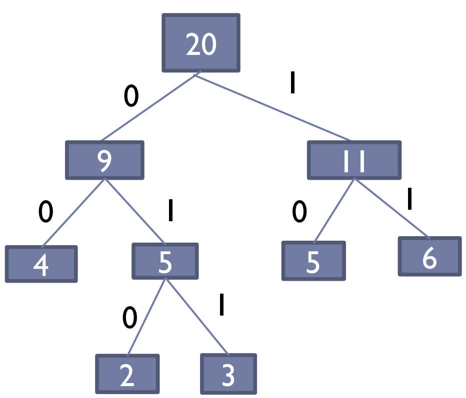
BCCABBDDAECCBBAEDDCC

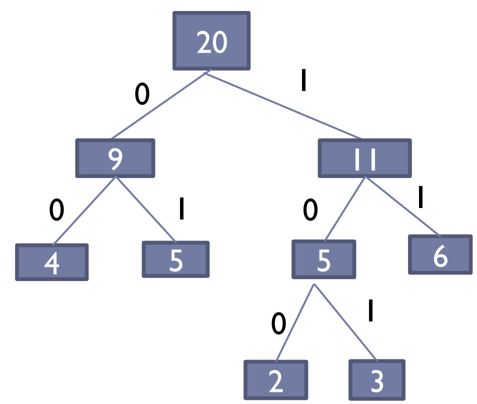
Bits required: 45 bits, Reference Table : 8*5 + 12=52, Total Bits required:97 bits

	Frequency	Variable-length codeword
'a'	3	011
'b'	5	10
'c'	6	11
'd'	4	00
'e'	2	010

OR

	Frequency	Variable-length codeword
'a'	3	101
'b'	5	01
'c'	6	11
'd'	4	00
'e'	2	100





Huffman Codes

Huffman Codes

- For compressing data (sequence of characters)
- Widely used
- Very efficient (saving 20-90%)
- Use a table to keep frequencies of occurrence of characters.
- Output binary string.

"Today's weather is nice"

"001 0110 0 0 100 1000 1110"

Huffman Codes

Example:

A file of 100,000 characters.
Containing only 'a' to 'f'

	Frequency	Fixed-length codeword	Variable-length codeword
'a'	45000	000	0
'b'	13000	001	101
'c'	12000	010	100
'd'	16000	011	111
'e'	9000	100	1101
'f'	5000	101	1100

eg. "abc" = "000001010"

300,000 bits

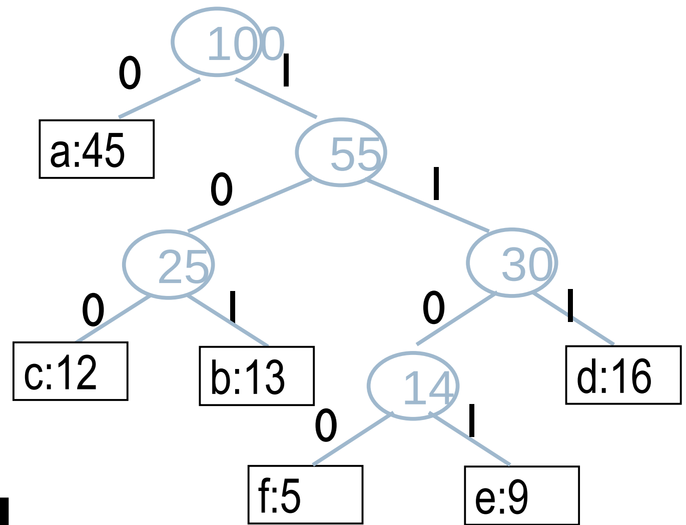
eg. "abc" = "0101100"

1*45000 + 3*13000 + 3*12000 +
3*16000 + 4*9000 + 4*5000
= 224,000 bits

Huffman Codes

A file of 100,000 characters.

	Frequency (in thousands)	Variable-length codeword
‘a’	45	0
‘b’	13	101
‘c’	12	100
‘d’	16	111
‘e’	9	1101
‘f’	5	1100



To find an optimal code for a file:

File size must be smallest.

=> Can be represented by a full binary tree.

=> Usually less frequent characters are at bottom

Let C be the alphabet (eg. C={‘a’,’b’,’c’,’d’,’e’,’f’})

For each character c, no. of bits to encode all c’s occurrences = $freq_c * depth_c$

File size $B(T) = \Sigma_{c \in C} freq_c * depth_c$

A full binary tree

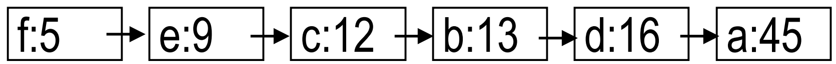
every nonleaf node has 2 children

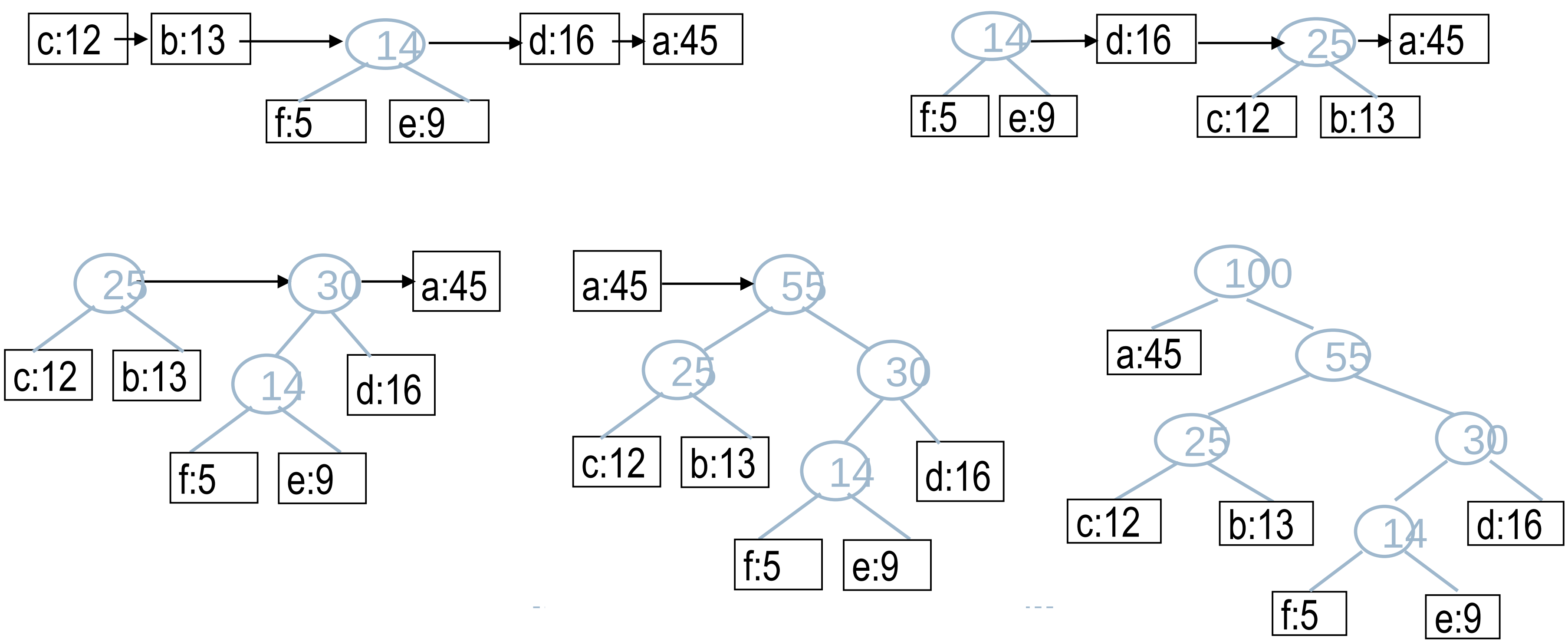
Huffman Codes

How do we find the optimal prefix code?

Huffman code (1952) was invented to solve it.
A Greedy Approach.

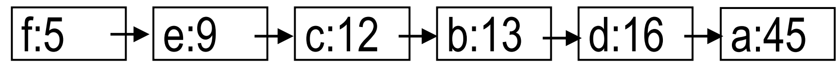
Q:A min-priority queue

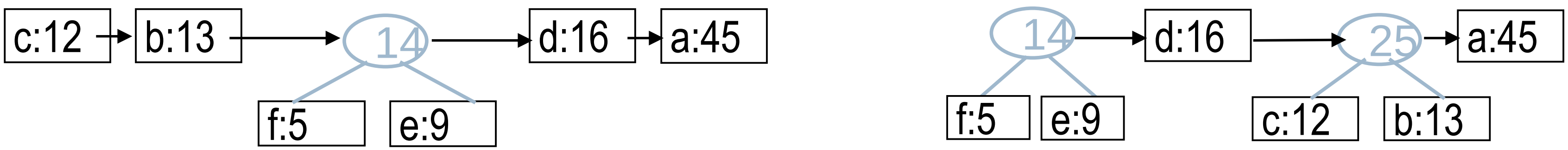




Huffman Codes

Q:A min-priority queue





....

```
HUFFMAN(C)
Build Q from C
For i = 1 to |C|-1
        Allocate a new node z
        z.left = x = EXTRACT_MIN(Q)
        z.right = y = EXTRACT_MIN(Q)
        z.freq = x.freq + y.freq
        Insert z into Q in correct position.
Return EXTRACT_MIN(Q)
```

If Q is implemented as a binary min-heap,
"Build Q from C" is O(nlogn)
"EXTRACT_MIN(Q)" is O(lg n)
"Insert z into Q" is O(lg n)
Huffman(C) is O(n lg n)

Problem-2

Construct a Huffman tree and the Huffman code for the following characters

Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Review:
The Knapsack Problem

- The famous *knapsack problem*:

"A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?"

Review: The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth p_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Fractional Knapsack Problem

- In fractional knapsack problem, where we are given a set S of n Objects, s.t., each item *O* has a *positive* profit p_i and a *positive* weight w_i, and we wish to find the maximum-benefit subset that *doesn't exceed* a given weight *W*.
- We are also allowed to take *arbitrary fractions* of each item.

Fractional Knapsack Problem

Object	1	2	3	4	5	6	7
Profit p_i	10	5	15	7	6	18	3
Weight w_i	2	3	5	7	1	4	1

Fractional Knapsack Problem-solved

Max wt Knapsack can hold– W=15

Object	1	2	3	4	5	6	7
Profit p_i	10	5	15	7	6	18	3
Weight w_i	2	3	5	7	1	4	1
p_i / w_i	5	1.6	3	1	6	4.5	3
Sol^n Vector X	1	2/3	1	0	1	1	1
	2	2	5	0	1	4	1

Fractional Knapsack Problem

- I.e., we can take an amount x_i of each item i such that

$$0 \leq x_i \leq w_i \text{ for each } i \in S \quad \text{and} \quad \sum_{i \in S} x_i \leq W.$$

- The *total benefit* of the items taken is determined by the *objective function*

$$\sum_{i \in S} p_i \, (x_i / w_i)$$

Fractional Knapsack Problem

Algorithm FractionalKnapsack(S, W):

Input: Set S of items, such that each item $i \in S$ has a positive **Profit p_i** and a positive weight w_i; positive maximum total weight W

Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**

  $x_i \leftarrow 0$

  $v_i \leftarrow$ p_i / w_i {***value index*** of item i}

$w \leftarrow 0$ {total weight}

while $w < W$ **do**

  remove from S an item i with highest value index {greedy choice}

  $a \leftarrow \min\{w_i, W - w\}$ {more than $W - w$ causes a weight overflow}

  $x_i \leftarrow a$

  $w \leftarrow w + a$

Fractional Knapsack Problem

- In the solution we use a heap-based *PQ* to store the items of S, where the *key* of each item is its *value index*
- With *PQ*, each greedy choice, which removes an item with the greatest value index, takes $O(\log n)$ time
- The *fractional knapsack algorithm* can be implemented in time $O(n \log n)$.

Problem-2

Consider that the capacity of the knapsack $W = \mathbf{60}$ and the list of provided items are shown in the following table.

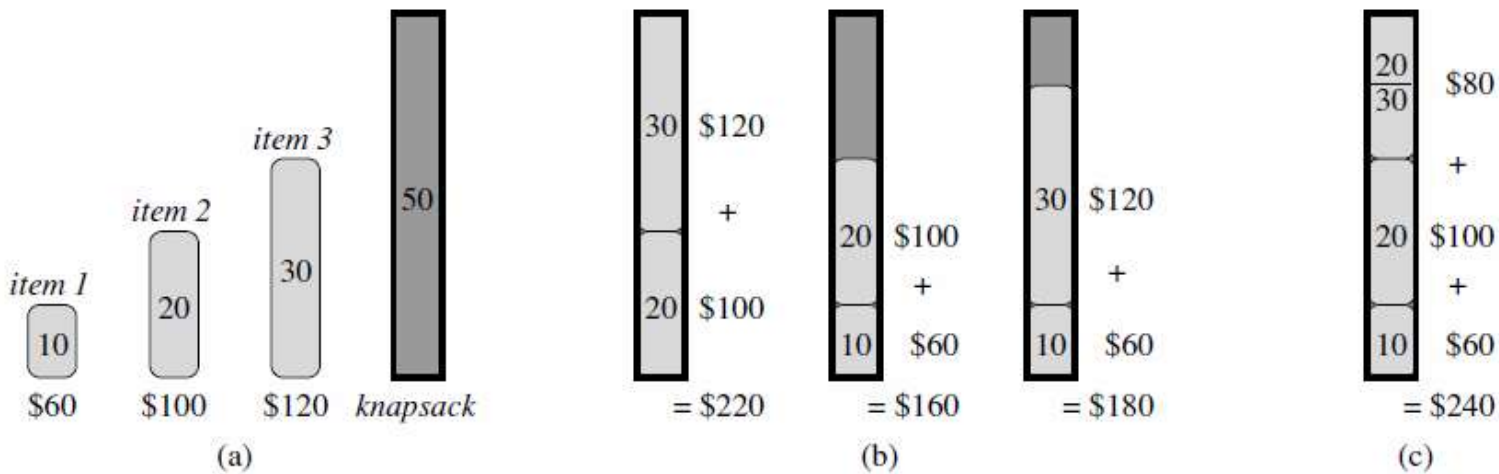
Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Review: The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail*



item 1
10
$60
item 2
20
$100
item 3
30
$120
50
knapsack
(a)
30
$120
+
20
$100
= $220
20
$100
+
10
$60
= $160
30
$120
+
10
$60
= $180
(b)
20
30
$80
+
20
$100
+
10
$60
= $240
(c)

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - however, it can be solved with dynamic programming