

Greedy Algorithm - II

Elements of Greedy Strategy

Instructor: Ashok Singh Sairam

Lecture Plan

- Steps in designing a greedy algorithm
 - Greedy vs. dynamic programming
- Knapsack problem
 - Subtleties of greedy algorithm
- Huffman code

Steps involved in activity-selection problem

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show greedy choice results in one subproblem
4. Prove greedy choice is safe
one of the optimal choices is the greedy choice
5. Develop a recursive algorithm
6. Convert recursive algorithm to iterative one

Designing a greedy algorithm

- The steps involved are typically
 1. Cast the optimization problem s.t. on making a choice we are left with one subproblem
 2. Prove that the greedy choice is always safe.
 3. Demonstrate that,
Optimal solution = greedy choice \cup Optimal solution of subproblem

Elements of a greedy algorithm

- Greedy choice property
 - A global optimal solution can be achieved by making a local optimal (greedy) choice
- Optimal substructure
 - An optimal solution to the problem within its optimal solution to subproblem.

Greedy vs. Dynamic Programming

Solving the bigger problem include

- one choice (greedy) vs multiple choices (DP)

Greedy vs. Dynamic Programming

Solving the bigger problem include

- one choice (greedy) vs multiple choices (DP)
- one subproblem Lots of overlapping subproblems

Greedy vs. Dynamic Programming

Solving the bigger problem include

- one choice (greedy) vs multiple choices (DP)
- one subproblem Lots of overlapping subproblems
- Top-down Bottom-up

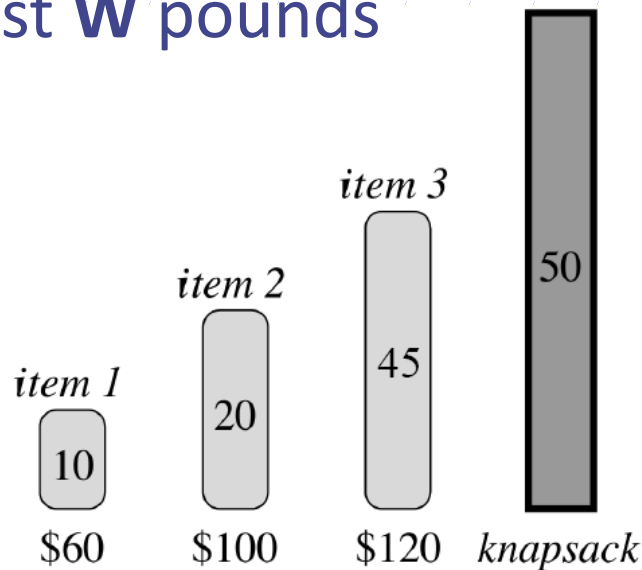
Greedy vs. Dynamic Programming

Solving the bigger problem include

- one choice (greedy) vs multiple choices (DP)
- one subproblem Lots of overlapping subproblems
- Top-down Bottom-up
- Both have optimal substructure

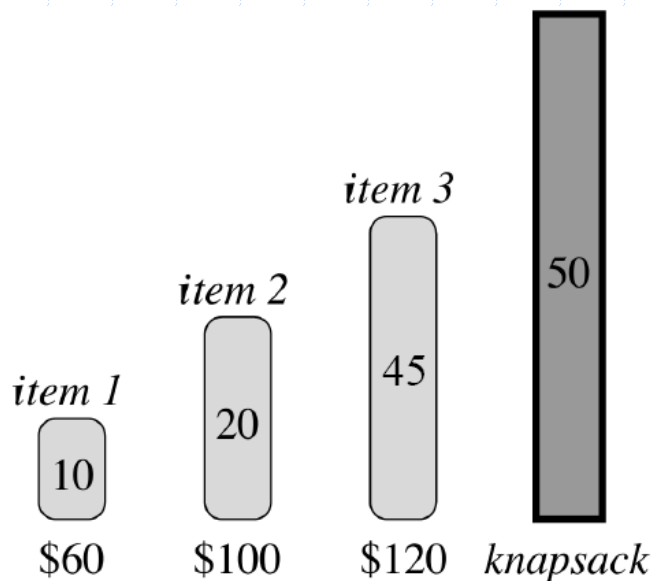
knapsack problem

- Thief robbing a store finds n items
 - Item i is worth v_i dollars and weighs w_i pounds
 - Take maximum valuable load but he can carry at most W pounds



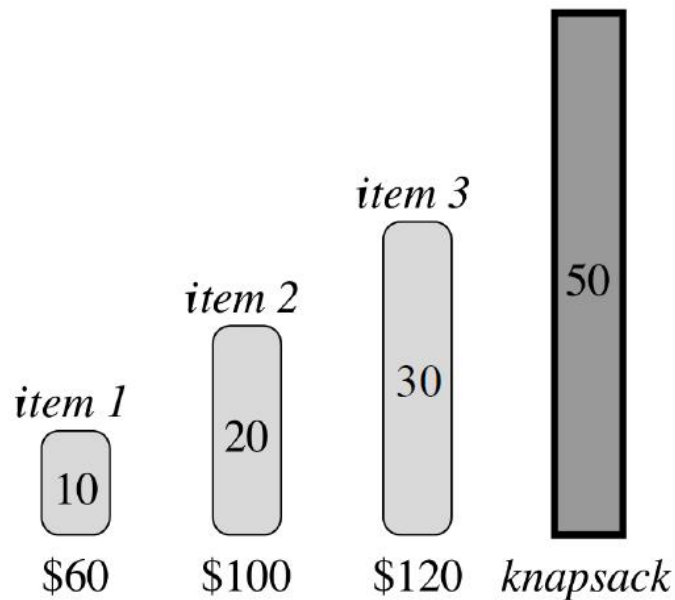
0-1 knapsack problem

- 0-1 Knapsack: Each item is either included or not
 - Take an item or leave it



Fractional knapsack problem

- Fractional Knapsack: Same setup but part item can be included



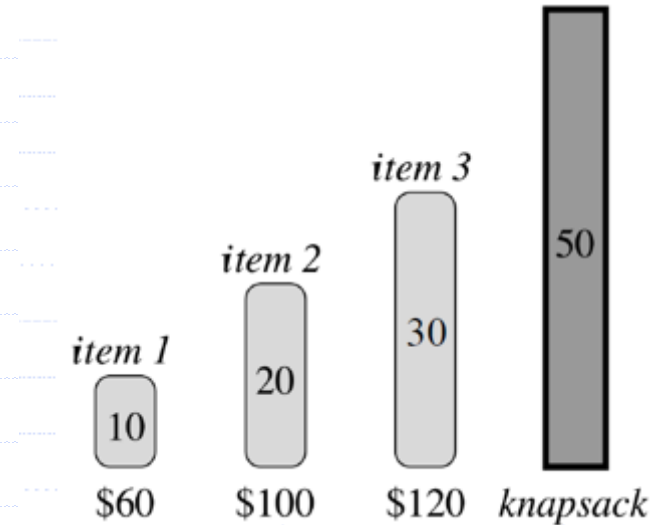
Optimal substructure

- Both knapsack problems exhibit optimal substructure
- 0-1 knapsack: Weight of most valuable load is W
 - If we remove **item j** , the remaining load is most valuable weighing $W - w_j$
- Fractional knapsack: Remove weight w of one item j from the optimal load
 - Remaining load is most valuable load weighing $W - w$ of $n - 1$ original plus $w_j - w$ pounds of item j

Fractional knapsack problem: Greedy choices

- Greedy choices
 - Take the most valuable item first \Rightarrow Does not lead to optimal solution
 - Take as much as possible of item with greatest value/pound \Rightarrow Leads to optimal solution

Greedy choice works !!!

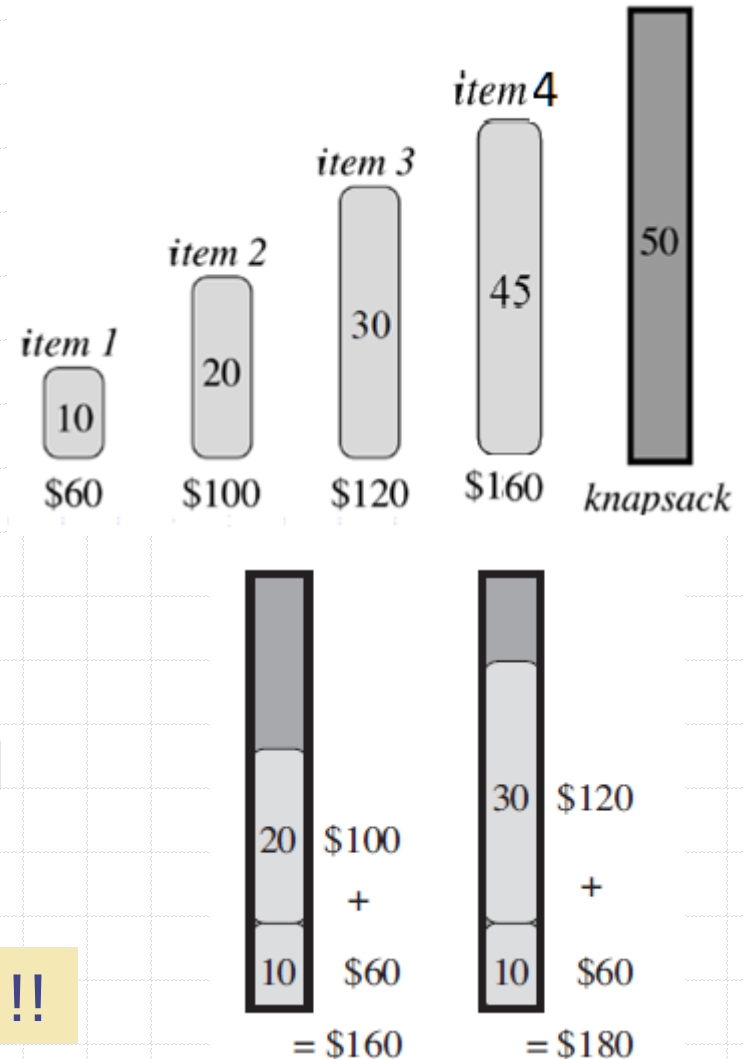


$$\begin{array}{r} 20 \\ 30 \\ \hline 20 \\ 10 \\ \hline \end{array} \begin{array}{l} \$80 \\ + \\ \$100 \\ + \\ \$60 \\ \hline = \$240 \end{array}$$

0-1 knapsack problem: Greedy choices

- Greedy choices
 - Take the most valuable item first \Rightarrow Does not lead to optimal solution
- Take as much as possible of item with greatest value/pound \Rightarrow **Does not work**

Greedy choice does not work !!!



Data Compression

- Typical saving in space from compressing data ranges from 20 to 90%
- Data: Sequence of characters
 - Ex: 10K characters, frequency of characters

	a	b	c	d	e	f
Frequency (in hundred)	45	13	12	16	9	5

- Goal: Store data compactly

Fixed-length vs. variable-length codeword

- Fixed-length code: All characters have same length
 $(45 + 13 + 12 + 16 + 9 + 5) * 3 * 100 = 30,000$ bits
- Variable-length code: Give frequent characters short code and infrequent ones long code
- $(45 * 1 + 13 * 3 + 12 * 3 + 16 * 3 + 9 * 4 + 5 * 4) * 100 = 22,400$ bits
 - Saving of 25%

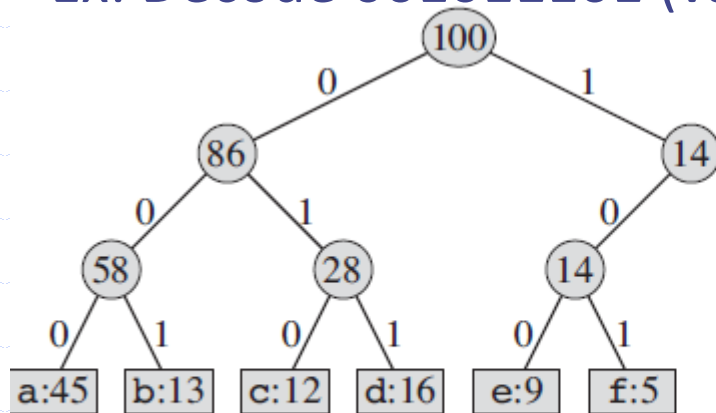
	a	b	c	d	e	f
Frequency (in hundred)	45	13	12	16	9	5
Fixed length codeword	000	001	010	011	100	101
Variable length codeword	0	101	100	111	1101	1100

Prefix codes

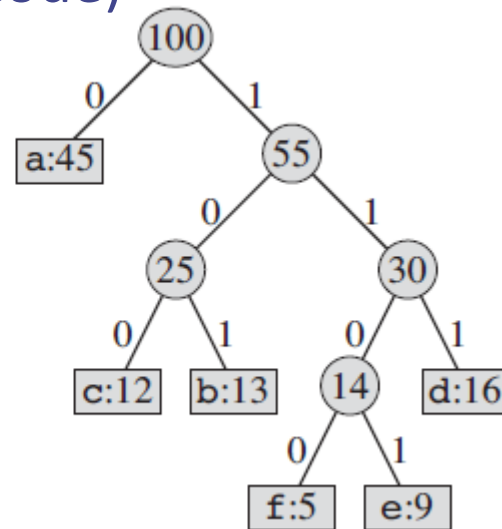
- Codes in which no codeword is a prefix of some other codeword
 - Unambiguous
 - Results in optimal data compression
 - Simple encoding and decoding.
- An optimal code for a file is always represented by a full binary tree.

Binary Tree Representation

- Simplifies decoding process
 - Leaves labelled with a character and its frequency
 - Internal nodes labelled with sum of frequencies
 - Ex: Decode 001011101 (variable code)



Fixed codeword



Variable codeword

Cost of Tree

- **T**: Binary tree corresponding to prefix code
- **C**: Alphabet of the data, example $C=\{a, b, c, d, e, f\}$
- **c.freq** ($c \in C$): Frequency of c
- **$d_T(c)$** : depth of c 's leaf in the tree
 - Also the length of codeword
- **B(T)**: No. of bits required to encode a file
 - Defined as cost of the tree

$$B(T) = \sum_{c \in C} c.\text{freq} \cdot d_T(c)$$

Huffman code

- Huffman code is an optimal prefix code
- Greedy algorithm is used to build the tree T
 - Builds the tree in a bottom up manner
 - Begins with $|C|$ leaves and performs $|C|-1$ merging operations
 - Identify two least-frequent subtrees and merges them

Huffman code algorithm

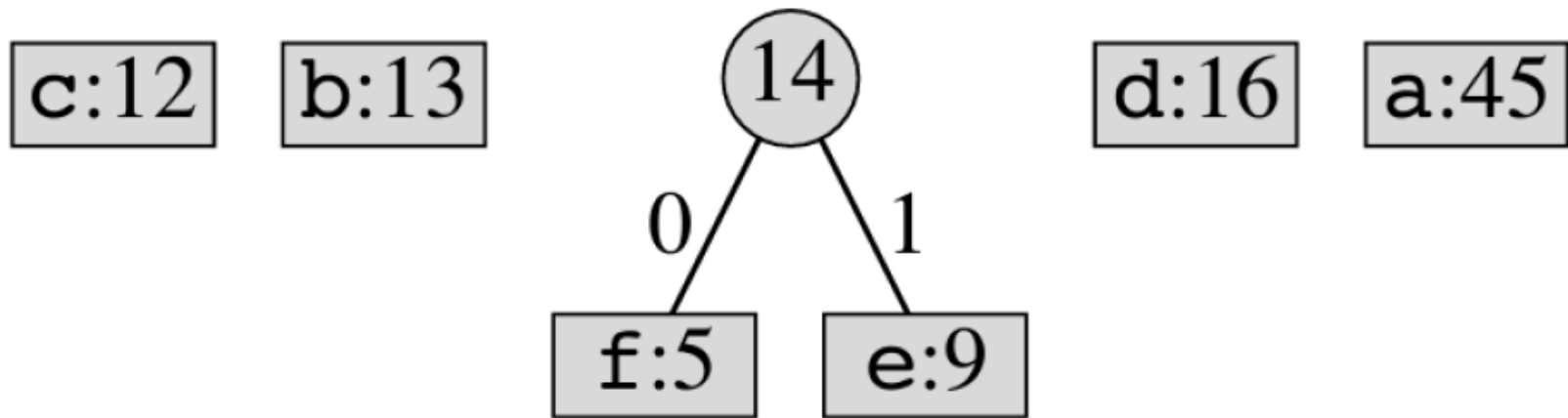
HUFFMAN(C)

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

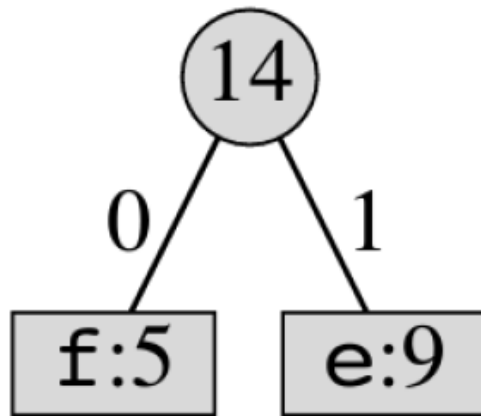
Huffman code: Tree construction

f:5 e:9 c:12 b:13 d:16 a:45

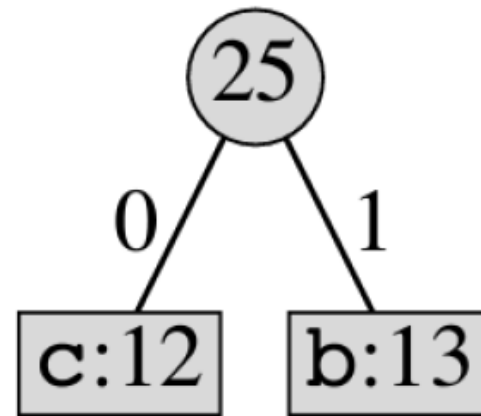
Huffman code: Tree construction



Huffman code: Tree construction

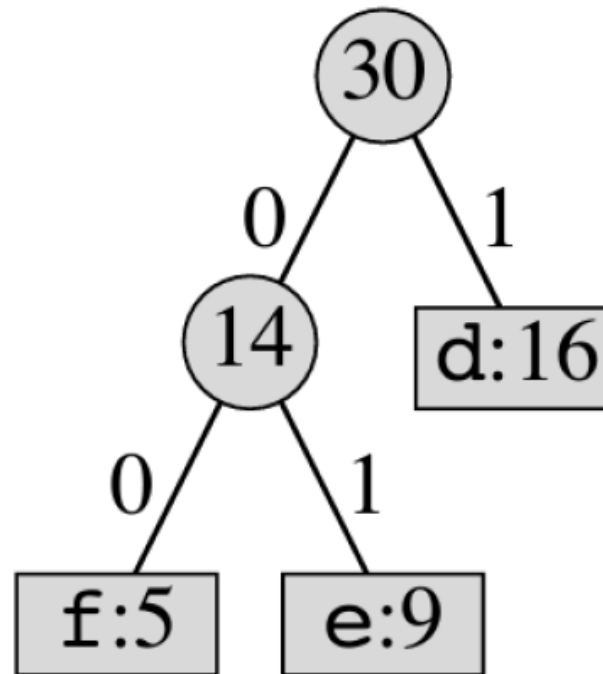
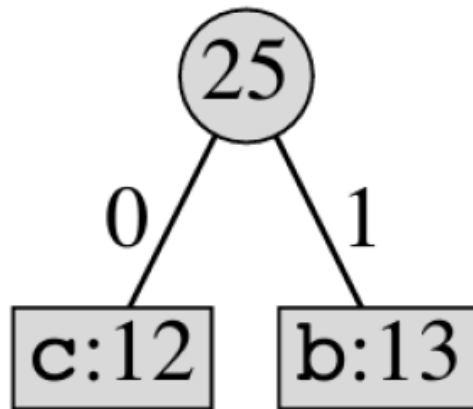


d:16



a:45

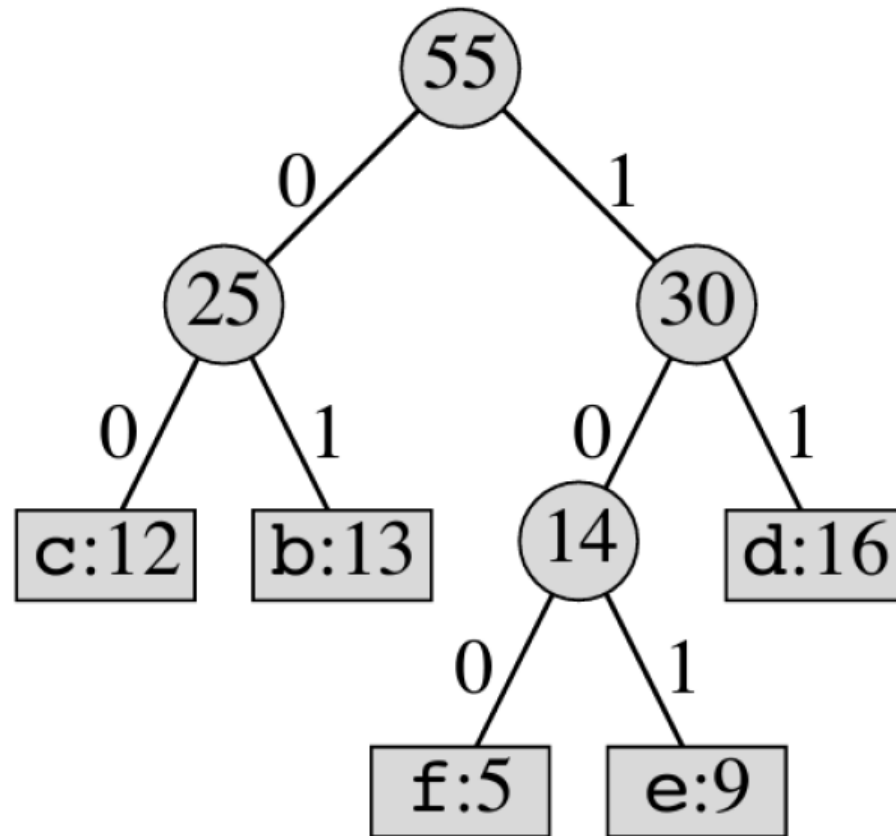
Huffman code: Tree construction



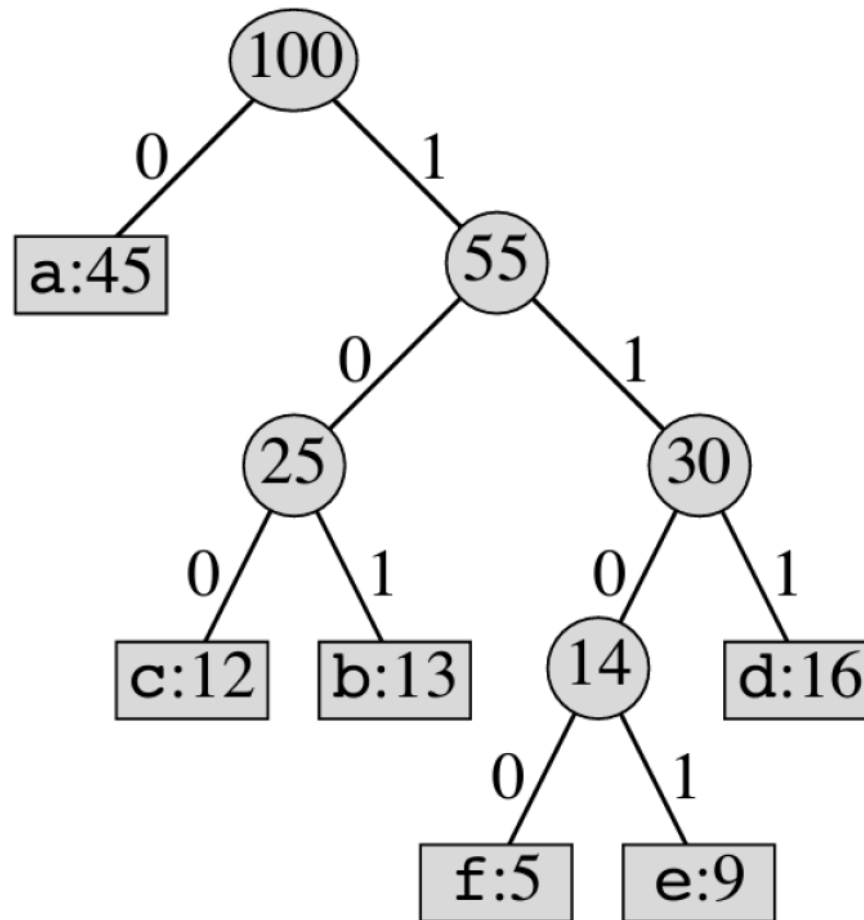
a:45

Huffman code: Tree construction

a:45



Huffman code: Tree construction



Huffman coding is greedy safe

- Step 1: Two smallest frequencies must be siblings
 - Deepest leaves correspond to longest words
 - Put the least frequent symbols at these positions
 - In any optimal tree, you can rearrange the symbols so that least two frequent symbols are at bottom without increasing cost
- Step 2: Reduce the problem
 - Make the two nodes a “super” node
 - Original problem with n symbols reduces to $n-1$

Huffman coding is greedy safe - contd

- Step 3: Greedy safe
 - At each step only 1 option, select two least frequent symbols and merge them
 - Because of step 1, the nodes can be made siblings in an optimal tree. Due to step 2, the smaller problem remains optimal

Exercise

- Prove that the fractional knapsack problem has the greedy choice property

Exercise

- Prove that the fractional knapsack problem has the greedy choice property
- Proof Sketch: Prove by contradiction.
- Assume the greedy soln (G) is not optimal (O, is the optimal soln)
- Assume that the last item added to knapsack is w of w_j
 - The solution is not optimal. There are two possibilities
- we can add more of w_j
 - But soln step say the knapsack must be full, a contradiction
- There is less of w_j in the optimal solution
 - There must be some item k in the knapsack, s.t.
 - $\frac{v_k}{w_k} < \frac{v_j}{w_j}$; Replace the item k with j . The value of knapsack will increase
 - Contradiction

