Greedy Algorithms

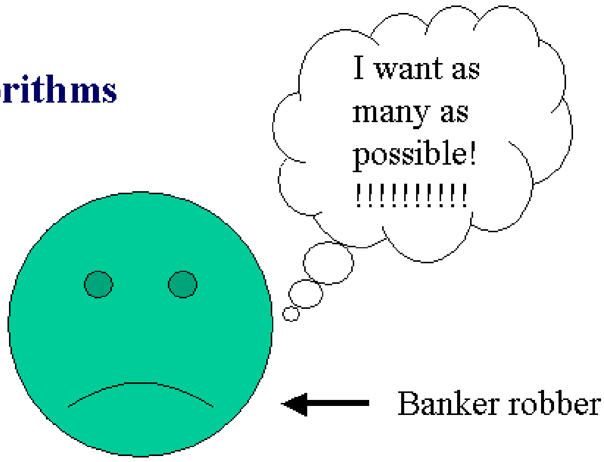
Overview

- Like dynamic programming, used to solve optimization problems.
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - 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 Strategy

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

Greedy algorithms

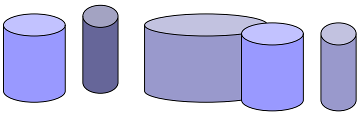


Makes a locally optimal choice

Does not always yield optimal solutions, but for many problems they do

The 0 - 1 knapsack problem

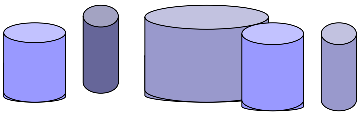
- A thief has a knapsack that holds at most W pounds.
- Item i : (v_i, w_i) (v = value, w = weight)
- Thief must choose items to maximize the value stolen and still fit into the knapsack.
- Each item must be taken or left (0 - 1).



5

Fractional knapsack problem

The setup is same as (0-1) Knapsack, but the thief can take fractions of items, rather than having to make a binary choice(0-1) for each item.



Contd…

- Both the 0 - 1 and fractional problems have the optimal substructure property
- Fractional: v_i / w_i is the value per pound.
- Clearly you take as much of the item with the greatest value per pound
- This continues until you fill the knapsack. Optimal (Greedy) algorithm takes *O (n log n) time*, as we must sort on vi / wi = di.

Example

- W = 50 lbs. (maximum knapsack capacity)

$w_1 = 10$	$v_1 = 60$	$d_1 = 6$
$w_2 = 20$	$v_2 = 100$	$d_2 = 5$
$w_3 = 30$	$v_3 = 120$	$d_3 = 4$

- were *d* is the value density

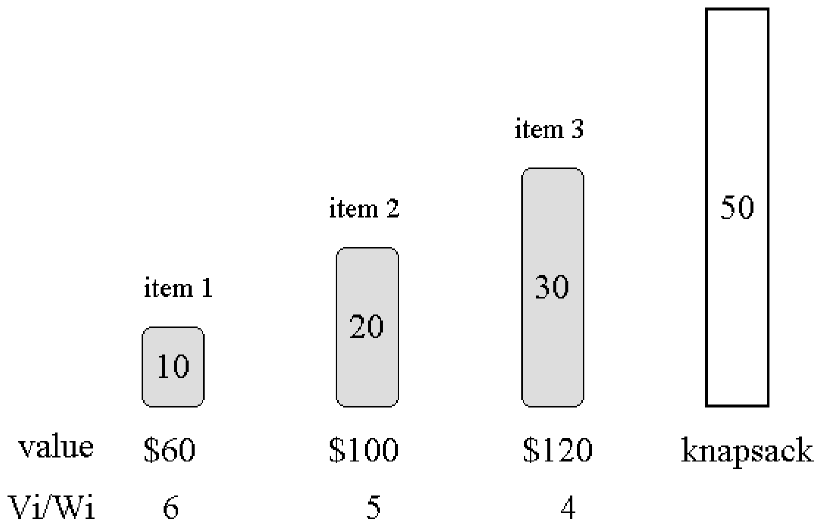
- **Greedy approach**: Take all of 1, and all of 2:
  v1+ v2 = 160

- Optimal solution is to take all of 2 and 3: *v2* + *v3*= *220*, other solution is to take all of 1 and 3 *v1*+ *v3* = *180.* All below 50 lbs.

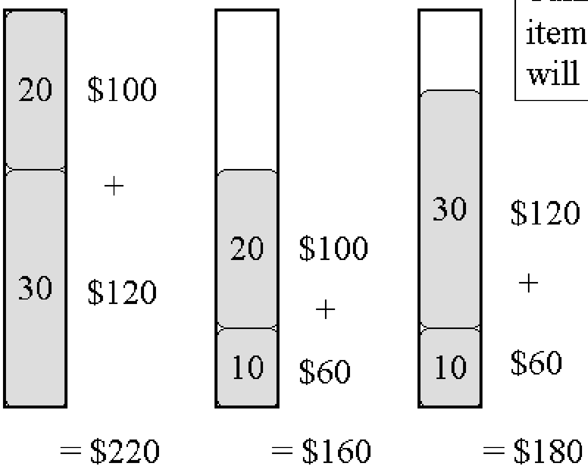
- When solving the 0 - 1 knapsack problem, empty space lowers the effective *d* of the load. Thus each time an item is chosen for inclusion we must consider both *i* included and excluded

- These are clearly overlapping sub-problems for different *i*'s and so best solved by DP!

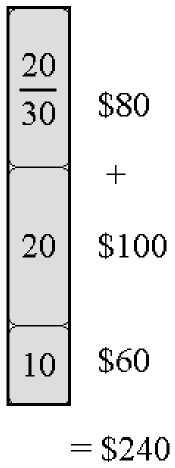
Greedy strategy doesn't work for the 0-1 knapsack problem.



Taking the most valuable items (item 1 and item2) will not work.



0-1 knapsack



Solvable by greedy algorithms

Fractional knapsack

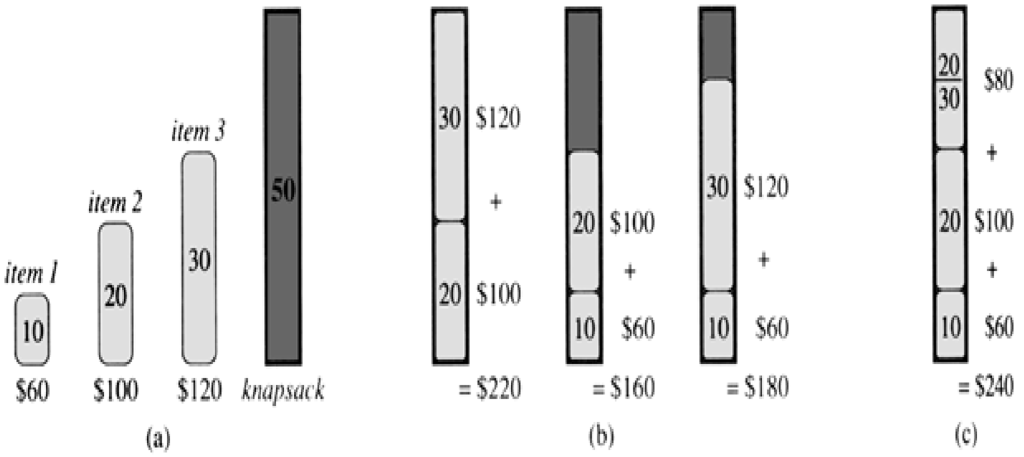


Figure 17.2 The greedy strategy does not work for the 0-1 knapsack problem. (**a**) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (**b**) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (**c**) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

An activity selection problem

- **Problem:**
  - Optimal scheduling of a resource among several competing activities (scheduling rehearsal times in a music studio)
  - Want to select as many mutually compatible activities as possible.

For a set of activities to be scheduled

1. Sort them by finish time S{1,2.….n}

2. Put 1 (the one with the earliest finish time) in the first place

3. Screen from 2,find the earliest one k compatible with 1, put k in the 2nd place

4. Screen from k+1, find the earliest one compatible with k, put in the 3rd place. So on………

Problem setup

- $S = \{1, 2, ..., n\}$ a set of n proposed activities.

- Each activity i has s_i a start time, and f_i a finish time $s_i \le f_i$.

- If activity i is selected, the resource is occupied in the interval $[s_i\ ,\ \ f_i)$. We say i and j are compatible activities if $[s_i\ ,\ \ f_i) \cap [s_j\ ,\ \ f_j) = \Phi$

Greedy-Activity-Selector (s, f)

- Assume that $f_1 \leq f_2 \leq \ldots \leq f_n$
- ***Greedy-Activity-Selector (s, f)***

  n ← length [s]
  A ← { 1 }
  j ← 1
  for i ← 2 to n
      do if $s_i \geq f_j$
          then A ← A U{ i }
              j ← i
  return A

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

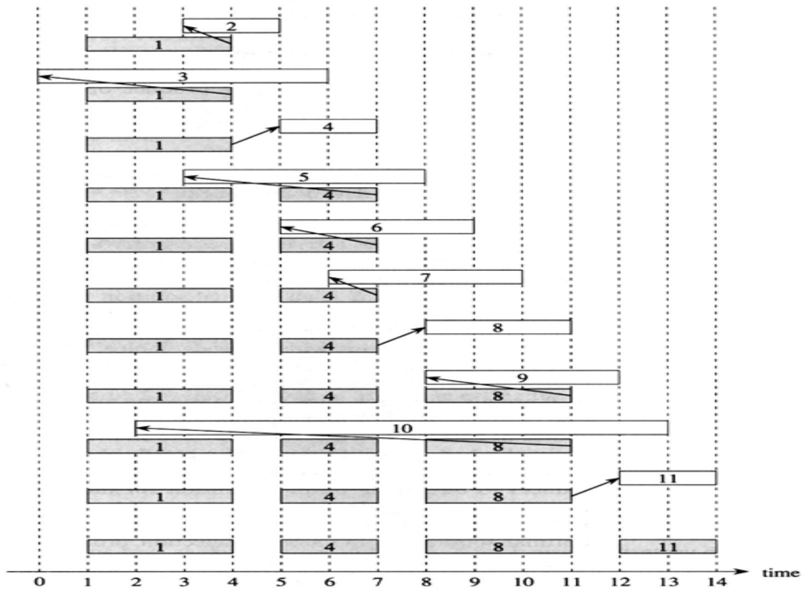


Figure 17.1 The operation of GREEDY-ACTIVITY-SELECTOR on 11 activities given at the left. Each row of the figure corresponds to an iteration of the **for** loop in lines 4–7. The activities that have been selected to be in set A are shaded, and activity i, shown in white, is being considered. If the starting time s_i of activity i occurs before the finishing time f_j of the most recently selected activity j (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is accepted and put into set A.

Example (Contd…)

- The activity picked is always the first that is compatible. Greedy algorithms do not always produce optimal solutions.

- Running time for sorting *n* activities by f_j $O(n \log n)$ and for Greed-Activity-Selector is $O(n)$

Theorem

Algorithm Greedy-Activity-Selector produces solutions of maximal size for the activities-selection problem.

Elements of the Greedy Strategy

- In general there is no way to tell if a greedy algorithm will solve a particular optimization problem or not.
- But there are two ingredients that are exhibited by most problems that lend themselves to a greedy strategy.
  - **Greedy-Choice Property**
  - **Optimal Substructure**

Greedy Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice

- This property is where Greedy algorithms differ from dynamic programming.

Greedy Vs Dynamic

- **Greedy Algorithm:**
  - Make whatever choice seems best at the moment. The choice depends on so far, not depend on any future choice.

- **Dynamic Programming**
  - The choice at each step depends on the solution to sub problems.

Optimal Substructure

- A problem exhibits optimal substructure, if an optimal solution to the problem contains within it, optimal solution to sub problems.
- A' = A - {1} (greedy choice) A' can be solved again with the greedy algorithm.
- This property is exploited by both greedy algorithm and dynamic programming.

Optimal Sub-structure of ASP

- Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the ASP over those activities in S, that are compatible with activity 1.
- If A is an optimal solution of the original problem, then A' = A – {1} is an optimal solution to the ASP S' = {i € S : $s_i \geq f_1$ }

Huffman Codes

- Effective technique for data compression
- Savings of 20-90% are typical
- Uses a table of frequencies of occurrences of characters to build up an optimal way of representing each character as a binary string

Examples of Different Binary Encodings

- Fixed length code
  - Each character in file represented by a different fixed length code
  - Length of encoded file depends only on the number of characters in the file
  - Example
    - 6 character alphabet (3 bit code), 25,000 character file takes 75,000 bits

27

Binary encodings continued

- If shorter binary strings are used for more frequent characters, a shorter encoding could be used

	A	B	C	D	E	F
Frequency (in thousands)	5	2	3	4	10	1
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	111	1001	101	110	0	1000

- Variable length encoding uses 58,000 bits

Encoding and Decoding

- Encoding
  - substitute code for the character
- Decoding
  - Fixed length: take x number of characters at a time and look up character corresponding to code
  - Variable length: must be able to determine when one code ends and another begins

29

Encoding

- Given a code (corresponding to some alphabet A') and a message it is easy to *encode* the message. Just replace the characters by the codewords.

Example: $\Gamma = \{a, b, c, d\}$

If the code is

$$C_1\{a = 00,\ b = 01,\ c = 10,\ d = 11\}.$$

then bad is encoded into 010011

If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into 1100111

Decoding

$$\begin{aligned} C_1 &= \{a = 00,\ b = 01,\ c = 10,\ d = 11\}. \\ C_2 &= \{a = 0,\ b = 110,\ c = 10,\ d = 111\}. \\ C_3 &= \{a = 1,\ b = 110,\ c = 10,\ d = 111\} \end{aligned}$$

Given an encoded message, *decoding* is the process of turning it back into the original message. A message is *uniquely decodable* if it can only be decoded in one way.

For example relative to C_1, 010011 is uniquely decodable to bad.

Relative to C_2 1100111 is uniquely decodable to bad.

But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

Prefix Codes

- Each code has a unique prefix.

  0 101 100
- Prefix constraint
  - The prefixes of an encoding of one character cannot be equal to a complete encoding of another character
- Decoding is never ambiguous
  - identify the first character
  - remove it from the file and repeat
  - Use binary tree to represent prefix codes for easy decoding

Important Fact:

- Every message encoded by a prefix tree code is uniquely decipherable. Since no codeword is a prefix of any other we can always find the first codeword in a message, peel it off and continue decoding.

$$01101100 = 01101100 = abba$$

Problem

- Given a text (a sequence of characters) find an encoding for the characters that satisfies the prefix constraint and that minimizes the number of bits need to encode the text.

Binary Tree Representation of Prefix Code

- 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
- Each leaf represents a character
- A left child represents the character 0 and a right child represents the character 1.
- The path from the root to the leaf represents the encoding for the leaf

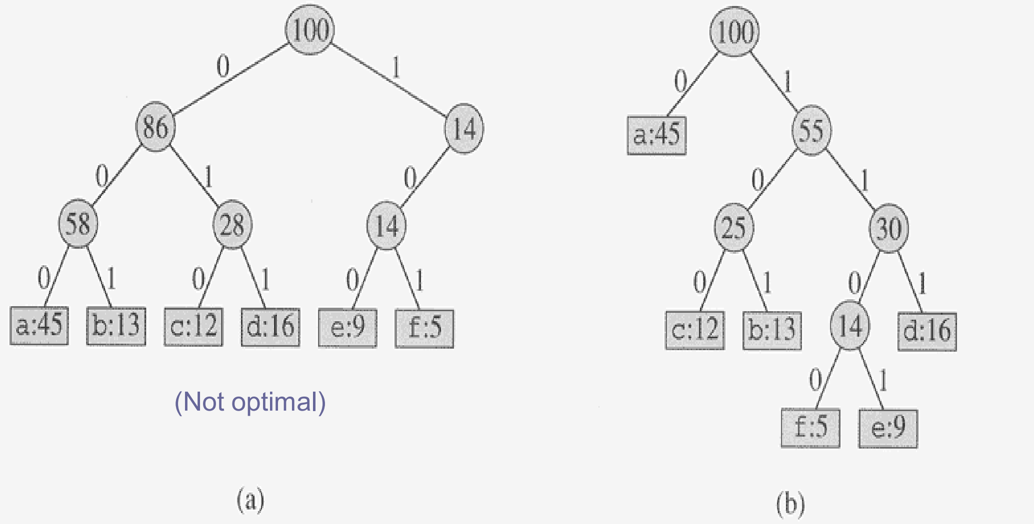


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, ...,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

Characteristics of Binary Tree

- Not a binary search tree
- The optimal code for a file is always represented by a full binary tree in which every non-leaf node has two children.
- If C is the alphabet, then a tree for the optimal prefix code has
  - |C| leaves
  - |C|-1 internal nodes

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)

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

Constructing a Huffman Code

- Greedy algorithm for constructing an optimal prefix code was invented by Huffman
- Codes constructed using the algorithm are called Huffman codes
- Bottom up algorithms
  - Start with a set of |C| leaves
  - perform a sequence of |C|-1 merging operations

Huffman Codes

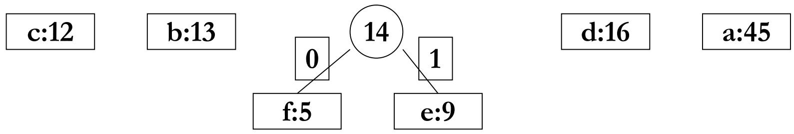
- A widely used compression algorithm
- Savings of 20% - 90%
- Uses the frequency of the characters
- Fixed codes versus variable-length codes:

Letter	freq	fixed code	variable
a	45	000	0
b	13	001	101
c	12	010	100
d	16	011	111
e	9	100	1101
f	5	101	1101

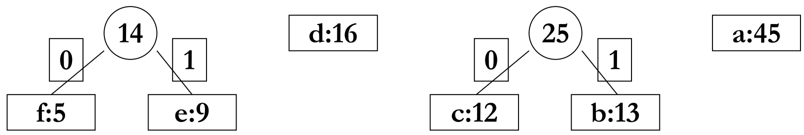
Constructing a Huffman Tree

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

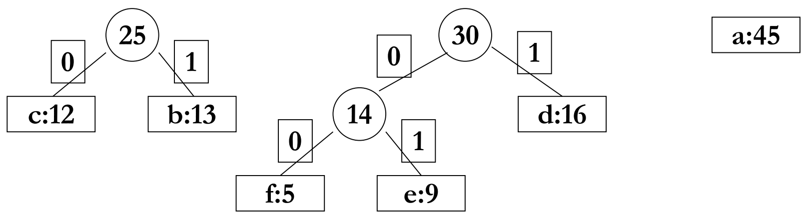
Constructing a Huffman Tree



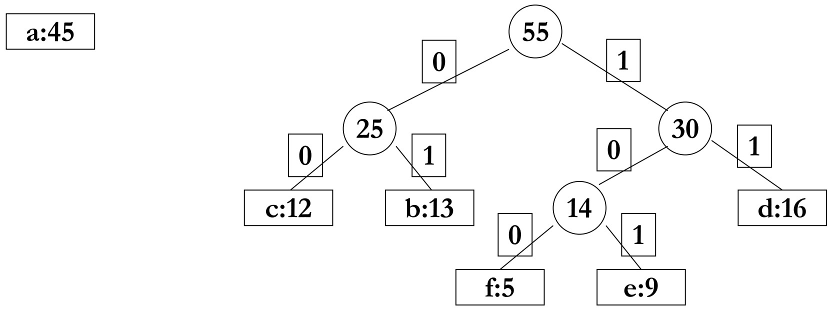
Constructing a Huffman Tree



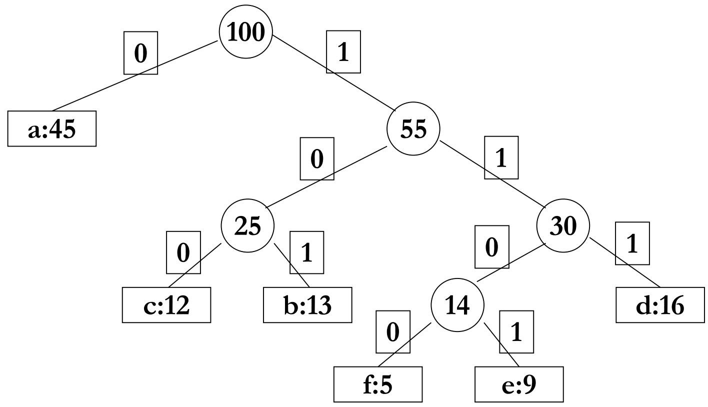
Constructing a Huffman Tree



Constructing a Huffman Tree



Constructing a Huffman Tree



HUFFMAN(C)

```
n ← |C|
Q ← C        ;  Characters are in a priority queue
for i ←1 to n-1
    do z ← ALLOCATE-NODE()
        x ← left[z] ← EXTRACT-MIN(Q)
        y ← right[z] ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q,z)
return EXTRACT-MIN(Q)
```

Running Time of Huffman's Algorithm

- Assume Q implemented as a binary heap
- Assume n characters in alphabet

48