CST 405
Algorithm Analysis & Design

Al Lake
Oregon Institute of Technology

Chapter 16

Greedy Algorithms

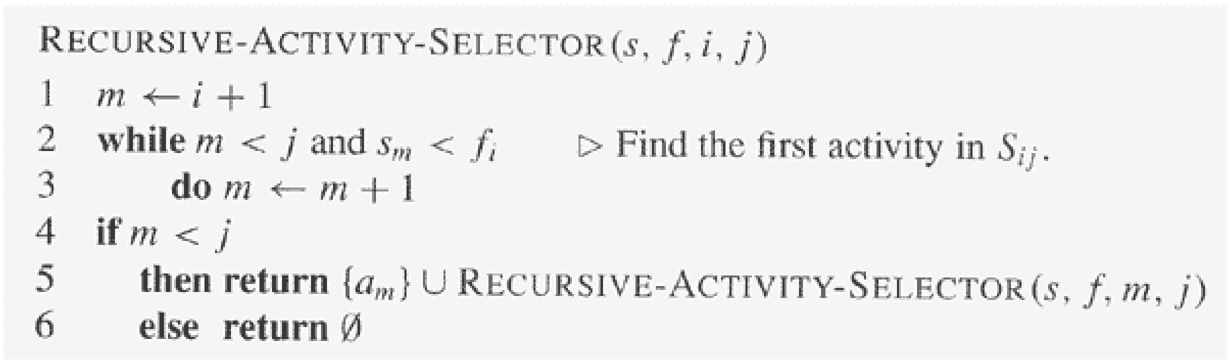
Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment.
- Greedy algorithms do not always yield the optimal solution.

Activity Selection Problem

- The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum size set of mutually compatible activities.

Recursive Greedy Algorithm



```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, i, j)$
$m \leftarrow i + 1$
while $m < j$ and $s_m < f_i$        ▷ Find the first activity in $S_{ij}$.
        do $m \leftarrow m + 1$
if $m < j$
    then return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, j)$
    else  return $\emptyset$
```

Greedy Strategy

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices
- For each decision point in the algorithm, the choice that seems best at the moment is chosen.

Greedy Steps

- Steps:
  - Determine the optimal substructure
  - Develop a recursive solution
  - Prove that at any stage of the recursion, one of the optimal choices is the greedy choice
  - Show that all but one of the subproblems induced by having made the greedy choice are empty
  - Develop a recursive algorithm that implements the greedy strategy
  - Convert the recursive algorithm to an iterative algorithm

Greedy Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- We make the choice that looks best in the current problem, without considering the results from subproblems.

Optimal Substructure

- A key property in determining the applicability of dynamic programming and greedy algorithms.
- A problem exhibits optimal substructure if all of the subproblems contain optimal solutions.

0-1 Knapsack Problem

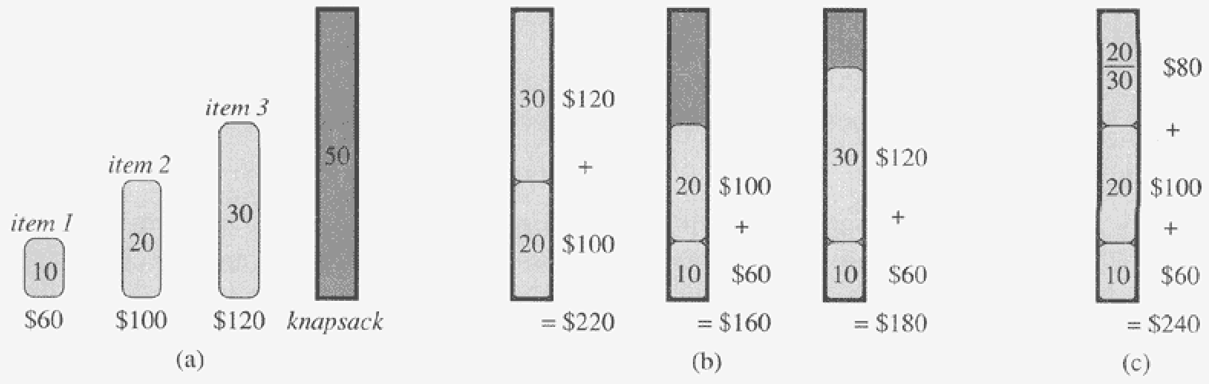
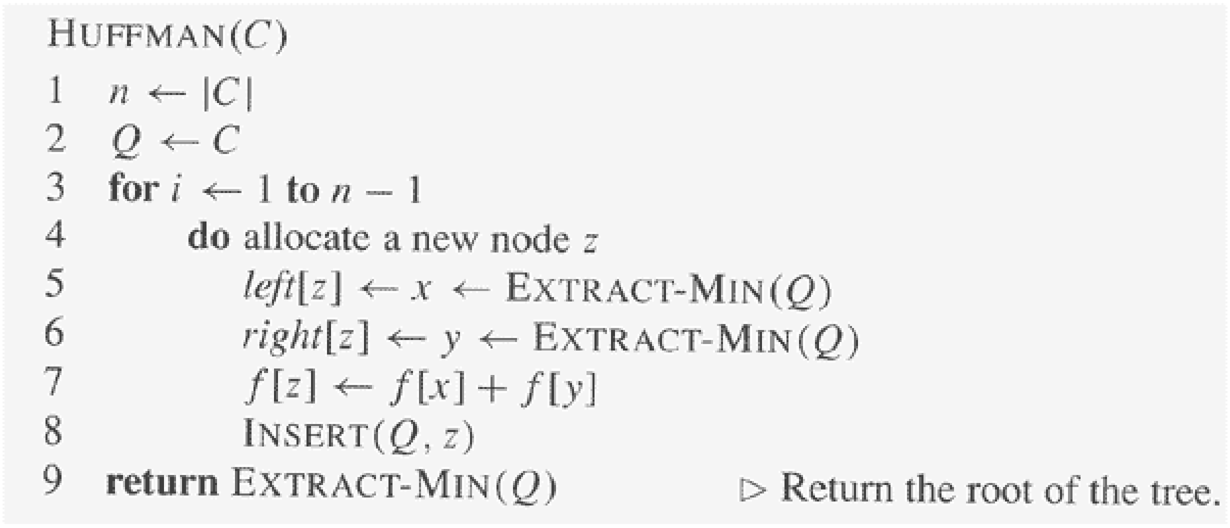


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Huffman Codes

- Huffman codes are used to compress data files.
- The problem is designing a binary character code where each character is represented by a unique binary string.
- Fixed-length code
  - 3 bits to represent 6 characters
  - a = 000, b = 001,…, f = 101
- Variable-length code
  - Convert frequent sets of characters into short codewords and infrequent characters into long codewords

Huffman



```
HUFFMAN($C$)
n ← |C|
Q ← C
for i ← 1 to n − 1
        do allocate a new node z
           left[z] ← x ← EXTRACT-MIN(Q)
           right[z] ← y ← EXTRACT-MIN(Q)
           f[z] ← f[x] + f[y]
           INSERT(Q, z)
return EXTRACT-MIN(Q)                 ▷ Return the root of the tree.
```

Huffman Algorithm

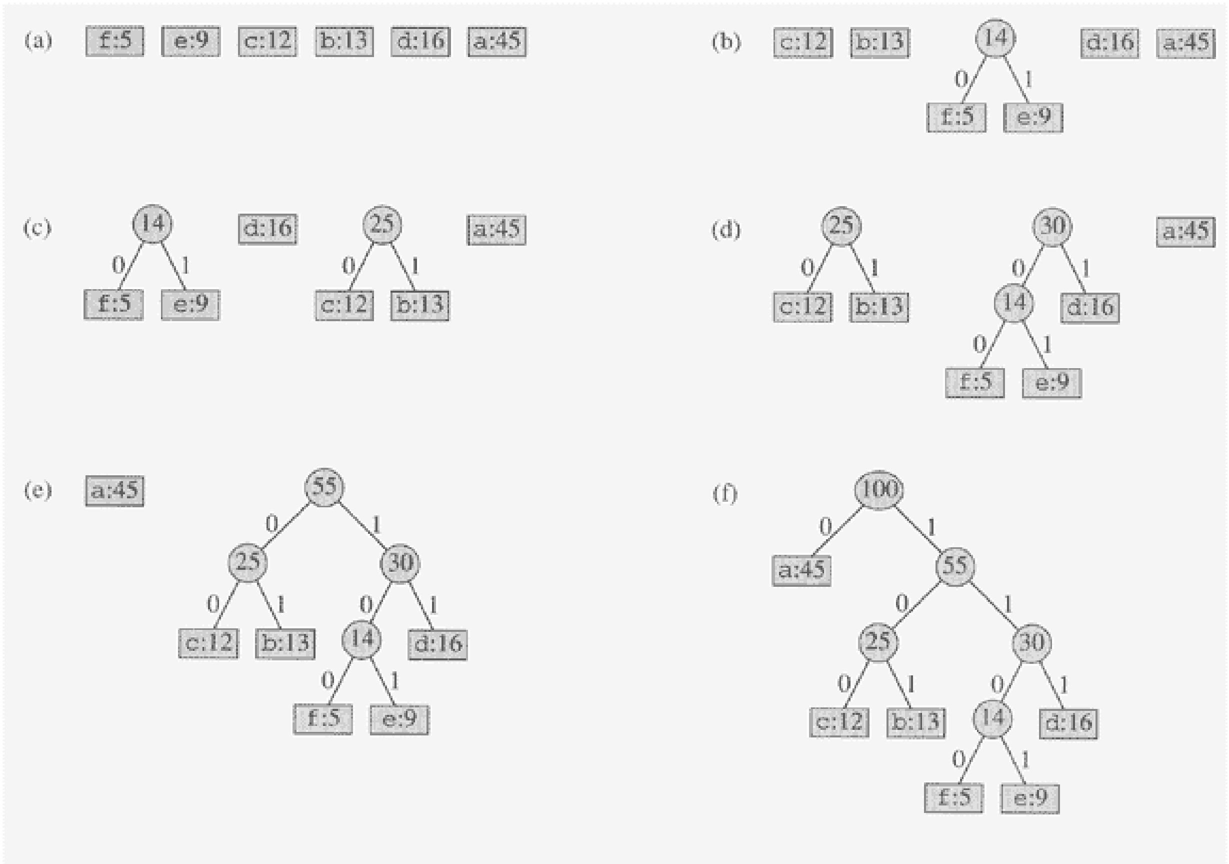


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency.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