

# Chapter16. 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 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**. (i.e. **hope**: a locally optimal choice will lead to a globally optimal solution)
    - not always, but in many situations, it works.
- Everyday example: driving, shopping...

# Greedy Strategy

- At each decision point, do what looks best “locally”
- Choice does not depend on evaluating potential future choices or solving subproblems
- Top-down algorithmic structure
  - With each step, reduce problem to a smaller problem
- Greedy Choice Property:
  - **“locally best” = “globally best”**

# Some Greedy Algorithms

- Fractional knapsack algorithm
- Huffman codes
- Kruskal's Minimum Spanning Tree (MST) algorithm
- Prim's Minimum Spanning Tree (MST) algorithm
- Dijkstra's Single Source Shortest Path(SSSP) algorithm
- .....

# Real World Examples

- Activity selection problem:  
get your money's worth out of a festival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
- Tour planning
- Customer satisfaction planning
- Conference scheduling problem
- .....

# Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:  
At each step, take the largest possible bill or coin Example: To make \$6.39, you can choose:
  - a \$5 bill
  - a \$1 bill, to make \$6
  - a 25¢ coin, to make \$6.25
  - A 10¢ coin, to make \$6.35
  - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution

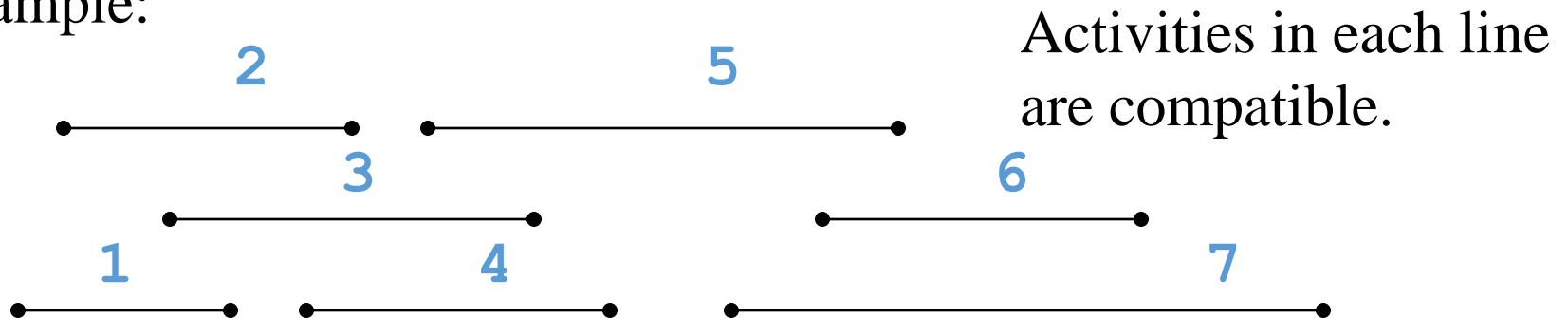
# A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

# Activity-selection Problem

- Input: Set  $S$  of  $n$  activities,  $a_1, a_2, \dots, a_n$ .
  - $s_i$  = start time of activity  $i$ .
  - $f_i$  = finish time of activity  $i$ .
- Output: Subset  $A$  of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.  
(if  $[s_i, f_i) \cap [s_j, f_j)$  is null)

Example:





# Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \dots \leq f_n$ .
- Suppose an optimal solution includes activity  $a_k$ .
  - This generates two subproblems.
  - Selecting from  $a_1, \dots, a_{k-1}$ , activities compatible with one another, and that finish before  $a_k$  starts (compatible with  $a_k$ ).
  - Selecting from  $a_{k+1}, \dots, a_n$ , activities compatible with one another, and that start after  $a_k$  finishes.
  - The solutions to the two subproblems must be optimal.

# Recursive Solution

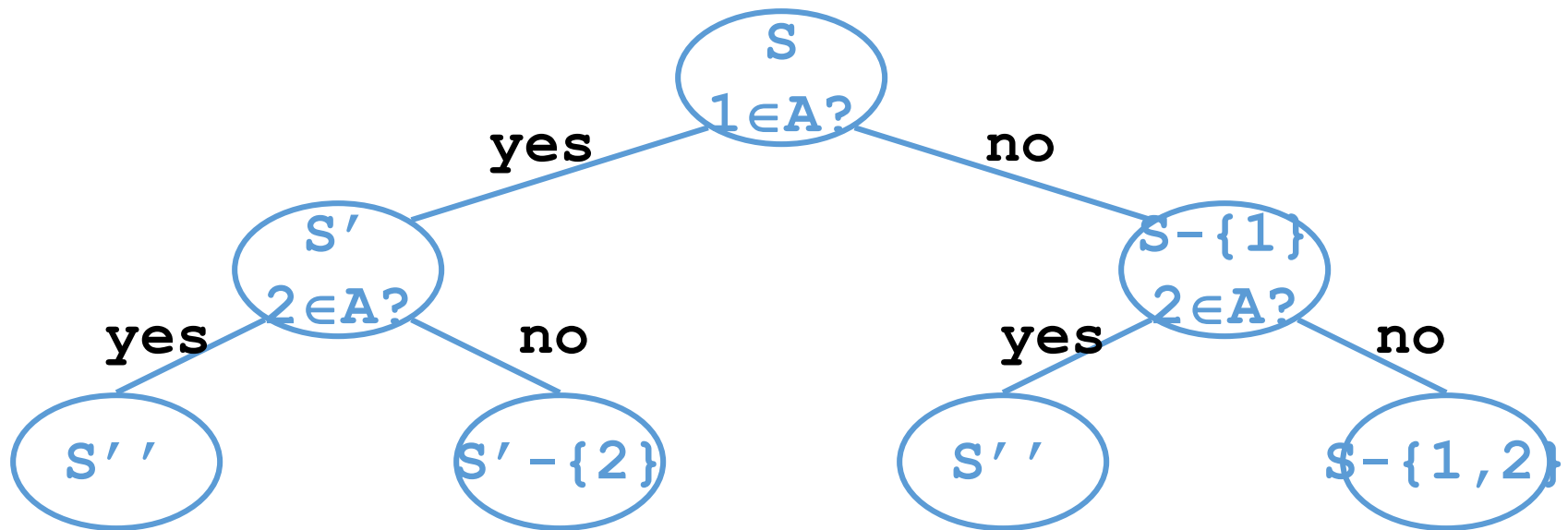
- Let  $S_{ij}$  = subset of activities in  $S$  that start after  $a_i$  finishes and finish before  $a_j$  starts.
- Subproblems: Selecting maximum number of mutually compatible activities from  $S_{ij}$ .
- Let  $c[i, j]$  = size of maximum-size subset of mutually compatible activities in  $S_{ij}$ .

**Recursive Solution:**

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

# Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



# Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

## Greedy-Activity-Selection( $s, f$ )

1.  $n := \text{length}[s]$
2.  $A := \{a_1\}$
3.  $j := 1$
4. for  $k:=2$  to  $n$  do
5.   if  $s_k \geq f_j$  // compatible activity
6.   then  $A := A \cup \{a_k\}$
7.        $j := k$
8. Return  $A$

- Example:  
11 activities sorted by finish time: (1, 4), (3, 5), (0, 6), (5,7), (3, 8), (5, 9),(6, 10), (8, 11), (8, 12), (2, 13), (12, 14)
- Solution:  
(1,4),(5,7),(8,11),(12,14)
- Is this algorithm correct?
  - Output is set of non-overlapping activities, but is it the largest possible?

- Theorem: The above algorithm picks a maximal set of activities(i.e. Greedy works for Activity-Selector.)  
Given activities  $A = \{A_1, A_2, \dots, A_n\}$  ordered by finish time, there is an optimal solution containing  $A_1$ .
- Proof: Suppose  $S$  is optimal solution( $S \subset A$ )  
If  $A_1 \in S$ , we are done  
If  $A_1 \notin S$ : Let first activity in  $S$  be  $A_k$   
Make new solution  $S - \{A_k\} \cup \{A_1\}$  by removing  $A_k$  and using  $A_1$  instead. This is valid solution ( $f_1 < f_k$ ) of maximal size ( $|S - \{A_k\} \cup \{A_1\}| = |S|$ )  
\* what means this?

# Greedy Algorithms vs. Dynamic Programming

- Both techniques use optimal substructure (optimal solution contains optimal solution for subproblems within it).
- In dynamic programming, solution depends on solution to subproblems.(i.e., compute the optimal solutions for each possible choice and then compute the optimal way to combine things together.
- In greedy algorithm we choose what looks like best solution at any given moment and recurse (choice does not depend on solution to subproblems).  
Note: Shortsightedness: Always go for seemingly next best thing, optimizing the present without regard for the future, and never change past choices.



# Greedy Algorithms vs. Dynamic Programming

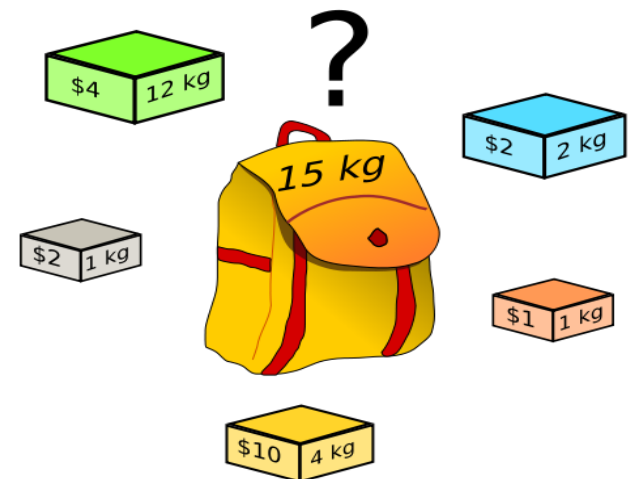
- Any problem that can be solved by a greedy algorithm can be solved by dynamic programming, but not the other way around.
- It is often hard to figure out when being greedy works!
- How do we know if being greedy works?  
Try dynamic programming first and understand the choices.  
Try to find out if there is a locally best choice, i.e. a choice that looks better than the others (without computing recursive solutions to subproblems).  
Now try to prove that it works correctly.

# 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?

# Knapsack problem

- Given a knapsack with maximum capacity  $W$ , and a set  $S$  consisting of  $n$  items
- Each item  $i$  has some weight  $w_i$  and benefit value  $b_i$  (all  $w_i$ ,  $b_i$  and  $W$  are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

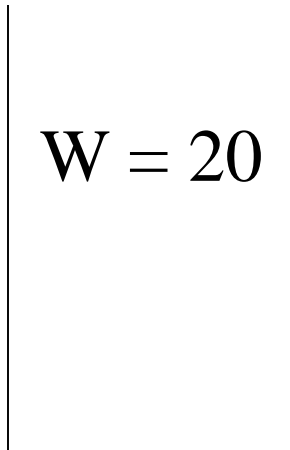






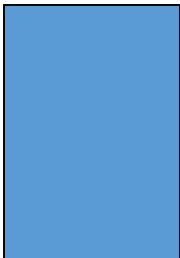
# Knapsack problem

- More formally, the 0-1 knapsack problem:
  - The thief must choose among  $n$  items, where the  $i$ th item worth  $v_i$  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” : each item must be taken or left in entirety
- A variation, the fractional knapsack problem:
  - Thief can take fractions of items

# 0-1 Knapsack problem: a picture

This is a knapsack  
Max weight:  $W = 20$



Items	Weight $W_i$	Benefit value $b_i$
	2	3
	3	4
	4	5
	5	8
	9	10

# Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the “*Fractional Knapsack Problem*”, where we can take fractions of items.

# 0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are  $n$  items, there are  $2^n$  possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to  $W$
- Running time will be  $O(2^n)$
- Can we do better?
  - an algorithm based on dynamic programming(?)
  - We need to carefully identify the subproblems

# Defining a Subproblem

If items are labeled  $1...n$ , then a subproblem would be to find an optimal solution for

$$S_k = \{items\ labeled\ 1, 2, .. k\}$$

- This is a valid subproblem definition.
- The question is: Can we describe the final solution ( $S_n$ ) in terms of subproblems ( $S_k$ )?
- Unfortunately, we can't do that. Why?



# Defining a Subproblem

$w_1=2$	$w_2=4$	$w_3=5$	$w_4=3$	
$b_1=3$	$b_2=5$	$b_3=8$	$b_4=4$	

Max weight:  $W = 20$

**For  $S_4$ :**

Total weight: 14;  
total benefit: 20

$w_1=2$	$w_2=4$	$w_3=5$	$w_5=9$
$b_1=3$	$b_2=5$	$b_3=8$	$b_5=10$

**For  $S_5$ :**

Total weight: 20  
total benefit: 26

	Weight	Benefit
Item $w_i$	$w_i$	$b_i$
#		
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

$S_4$  is indicated by a bracket around items 1, 2, 3, and 4.

$S_5$  is indicated by a bracket around items 1, 2, 3, 4, and 5.

**Solution for  $S_4$  is not part of the solution for  $S_5$ !**

# Defining a Subproblem

- As we have seen, *the solution for  $S_4$  is not part of the solution for  $S_5$*
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter:  $w$ , which will represent the exact weight for each subset of item  $s$
- The subproblem then will be to compute  $B[k, w]$

# Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{o.w} \end{cases}$$

- It means, that the best subset of  $S_k$  that has total weight  $w$  is one of the two:
  - 1) the best subset of  $S_{k-1}$  that has total weight  $w$ , **or**
  - 2) the best subset of  $S_{k-1}$  that has total weight  $w-w_k$  plus the item  $k$

# Recursive Formula

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{o.w} \end{cases}$$

- The best subset of  $S_k$  that has the total weight  $w$ , either contains item  $k$  or not.
- First case:  $w_k > w$ . Item  $k$  can't be part of the solution, since if it was, the total weight would be  $> w$ , which is unacceptable
- Second case:  $w_k \leq w$ . Then the item  $k$  can be in the solution, and we choose the case with greater value

# The Knapsack Problem

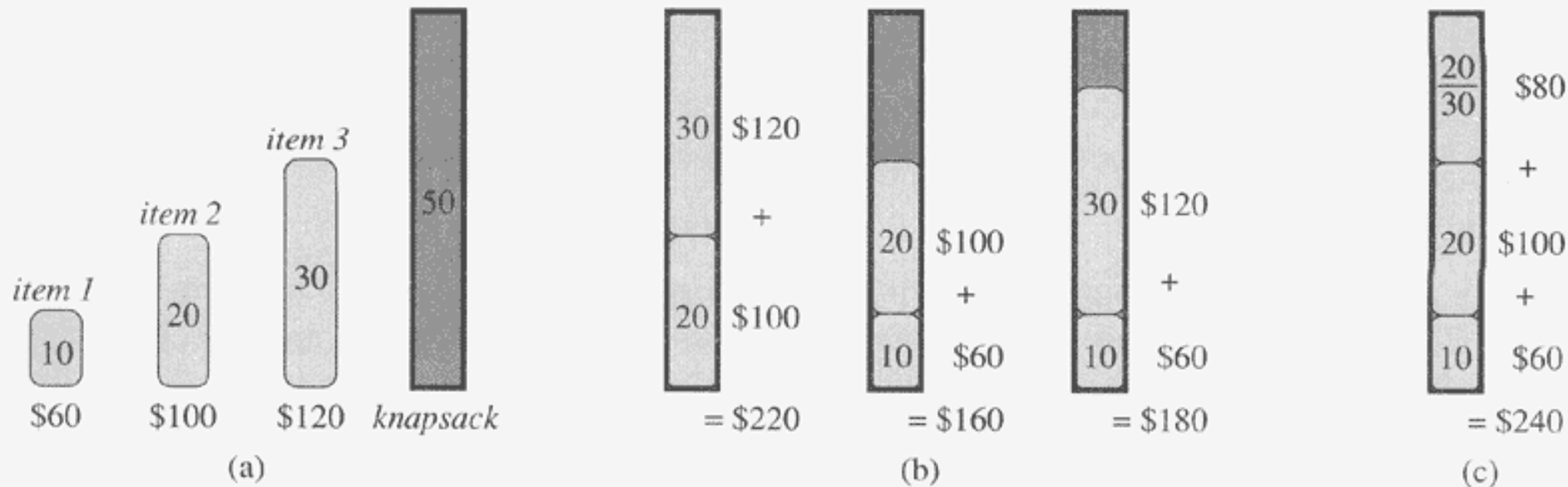
## And Optimal Substructure

- 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?*
  - Ans: 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
- 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*

# Example



**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.

# The Knapsack Problem: Greedy Vs. Dynamic

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



# 0-1 Knapsack Problem

for  $w = 0$  to  $W$

$B[0,w] = 0$

for  $i = 0$  to  $n$

$B[i,0] = 0$

for  $w = 0$  to  $W$

if  $w_i \leq w$  // item  $i$  can be part of the solution

if  $b_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = b_i + B[i-1,w-w_i]$

else

$B[i,w] = B[i-1,w]$

else  $B[i,w] = B[i-1,w]$  //  $w_i > w$

# Running time

for w = 0 to W

B[0,w] = 0

$O(W)$

for i = 0 to n

B[i,0] = 0

Repeat  $n$  times

for w = 0 to W

< the rest of the code >  $O(W)$

.....

What is the running time of this algorithm?

$O(n*W)$

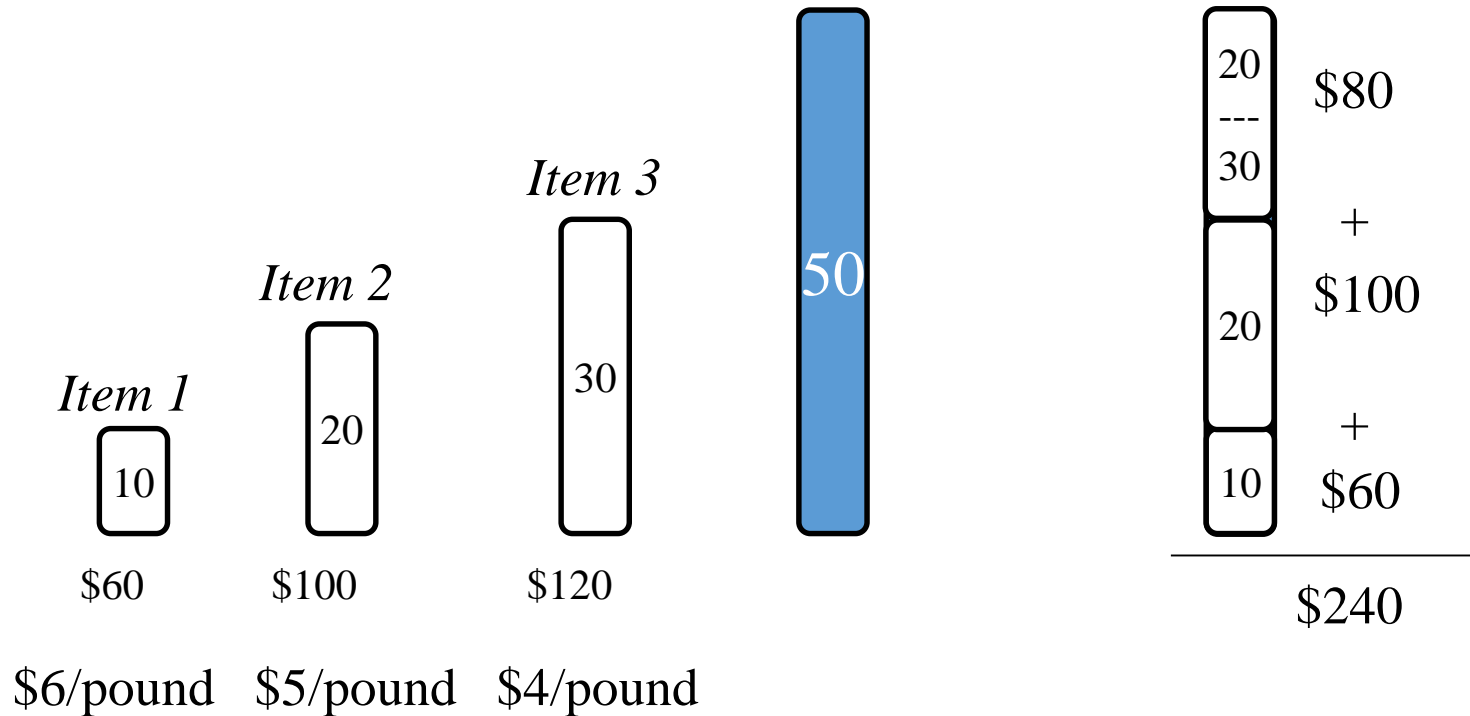
Remember that the brute-force algorithm  
takes  $O(2^n)$ !

# 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  $0 \leq x_i \leq 1$ ,  $i = 1, 2, \dots, n$   
 $\sum w_i x_i \leq W$  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

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

# 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  $\Theta(n \log n)$

# 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 codeword 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	C: Alphabet
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	

**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

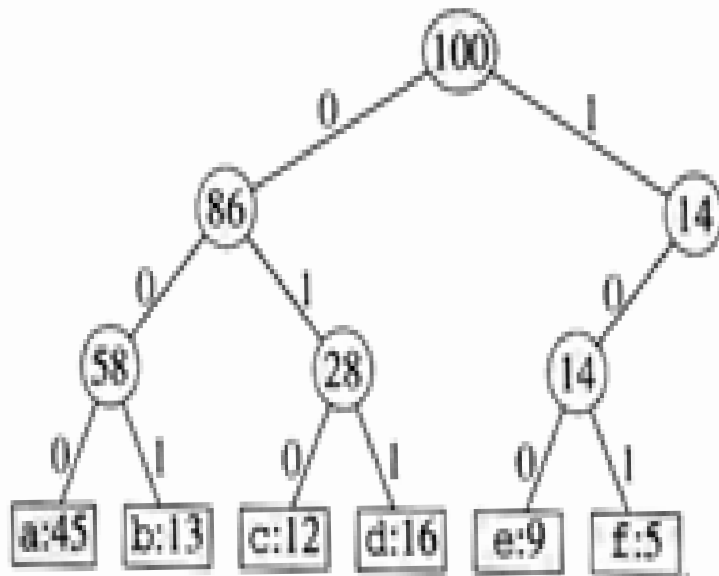
- Assume we are given a data file that contains only 6 symbols, namely a, b, c, d, e, f  
With the following frequency table:

	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

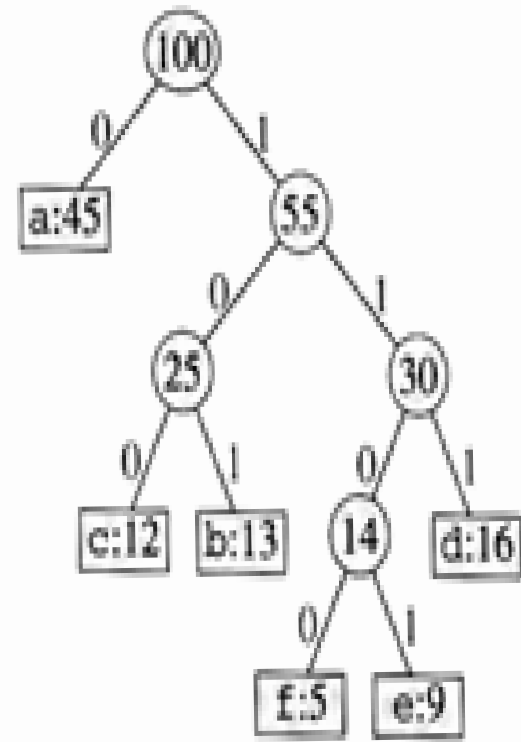
- Find a variable length prefix-free encoding scheme that compresses this data file as much as possible?

# Huffman Code Problem

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



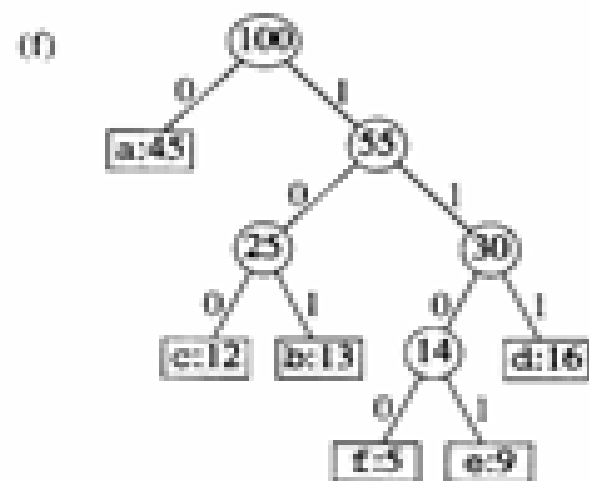
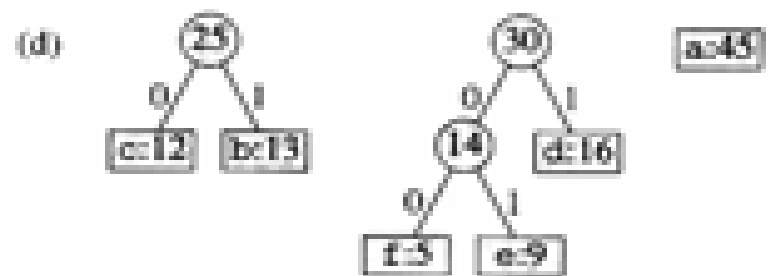
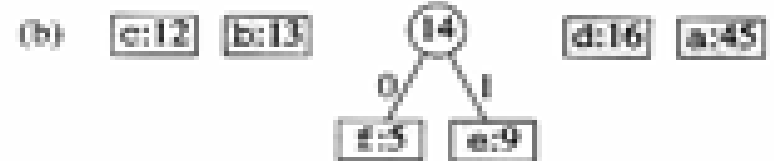
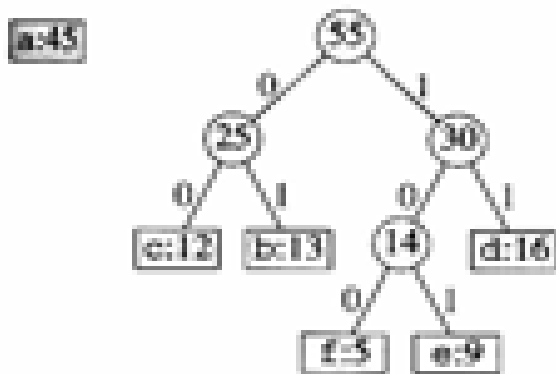
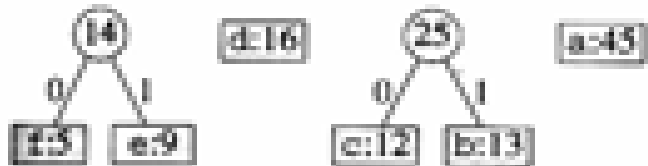
(a)



(b)

# Example

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



# Constructing A Huffman Code

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

Total computation time =  $O(n \log n)$

# 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  $c \in C$  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  $O(\log n)$ , the loop contributes  $O(n \log n)$  to the running time. Thus, the total running time of HUFFMAN on a set of  $n$  characters is  $O(n \log n)$ .

# Prefix Code

- Prefix(-free) code: no codeword is also a prefix of some other codewords (Unambiguous)
  - 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  $\rightarrow$  0 · 101 · 100 = 0101100
    - Decoding: 001011101 = 0 · 0 · 101 · 1101  $\rightarrow$  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:

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

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

	Divide & Conquer	Dynamic Programming	Greedy Algorithm
View problem as collection of subproblems	✓	✓	✓
“Recursive” nature	✓	✓	✓
Independent subproblems	✓	overlapping	typically sequential dependence
Number of subproblems	depends on partitioning factors	typically small	
Preprocessing			✓ typically sort
Characteristic running time	typically log function of n	depends on number and difficulty of subproblems	often dominated by $n \log n$ sort
Primarily for optimization problems		✓	✓
Optimal substructure: <i>optimal solution to problem contains within it optimal solutions to subproblems</i>		✓	✓
Greedy choice property: <i>locally optimal produces globally optimal</i>			✓
Heuristic version useful for bounding optimal value			✓