Chapter 16
Greedy Algorithms

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, McGraw-Hill, 2001.

Chapter 16 Topics

- An example problem
- Elements of the greedy strategy
- Huffman codes
- Theoretical foundations of greedy methods

Greedy Algorithms

- Like Dynamic Programming, greedy algorithms are used for optimization.
- The basic concept is that when we have a choice to make, make the one that looks best right now.
- That is, make a locally optimal choice in hope of getting a globally optimal solution.
- Greedy algorithms won’t always yield an optimal solution, but sometimes, as in the activity selection problem, they do.

Common Situation for Greedy Problems

- a set (or a list) of candidates
- the set of candidates that have already been used
- a function that checks whether a particular set of candidates provides a solution to the problem
- a function to check feasibility
- a selection function that indicates the most promising candidate not yet used
- an objective function that gives the value of the solution

Basic Greedy Algorithm

```
function greedy (C: set) : set
S ← ∅
while not solution (S) and C ≠ ∅ do
    x ← an element of C maximizing select(x)
    C ← C - {x}
    if feasible (S ∪ {x})
        then S ← S ∪ {x}
    if solution (S)
        then return S
        else return "no solutions"
```

Change Making Problem

- Problem: make change for a customer using the smallest number of coins
  - candidates: finite set of coins (1¢, 5¢, 10¢, 25¢) with at least one coin of each type
  - solution: total value of amount needed
  - feasible set: set that does not exceed the amount to be paid
  - selection: highest valued coin available
  - objective function: # of coins used (minimize)

Change Making Problem

Can we solve this problem in a greedy way?

Yes:

1. Start with the *Amount* to be returned to the customer.
2. Divide by the largest *Denomination* coin.

   Give this *Number* of that denomination coin to the customer.

   Set *Amount* = A – (D * N).

   Set D = next lower Denomination.
3. Loop back to 2 and repeat until Amount = 0.

Change Making Problem

- Will the greedy algorithm return the optimum solution if our coins have the values 1¢, 6¢, and 10¢?
- No. Twelve cents in change will not be optimized.

Activity Selection Problem

Activity Selection Problem

- Problem: schedule a resource among several competing activities given the start and finish times of each activity
- Goal: Select a maximum-size set of mutually compatible activities

Formal Definition of Problem

- Let $S=\{a_1, a_2, . . . , a_n\}$ be the set of activities. One activity may want the same resource as some other activity. Only one activity can be done at a time.
- Each a_i has start time s_i and finish time f_i such that:

  $0 \leq s_i < f_i < \infty$
- An activity a_i takes place in the half-open interval $[s_i , f_i)$
- Activities *i* and *j* are *compatible* if $[s_i , f_i)$ and $[s_j , f_j)$ do not overlap, (i.e., $s_i \geq f_j$ or $s_j \geq f_i$)
- The activity selection problem is to select a maximum size set of mutually compatible activities

The Greedy Algorithm

Assumes that:

- Start and finish times are stored in arrays *s* and *f*
- Activities are sorted in order of increasing finish times:

  $f_1 \le f_2 \le . . . f_n$

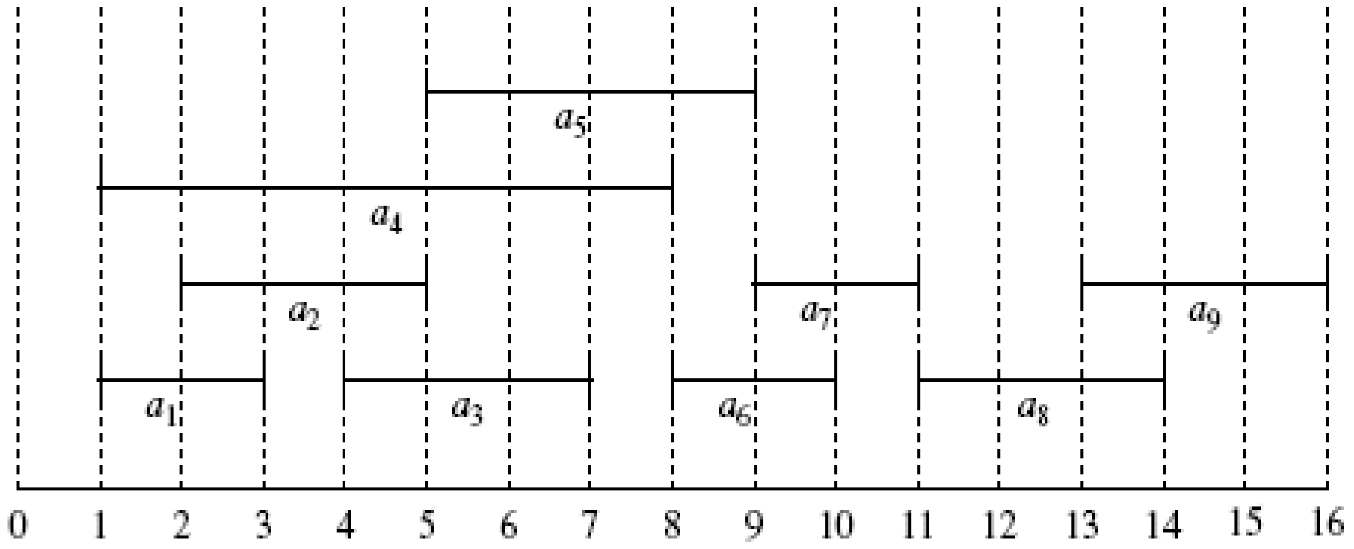
  (If not the case, we can sort them in O(n lg n) time.)

Sample Set of Activities

Sample set S (sorted by finish time):

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	4	5	7	8	9	10	11	14	16

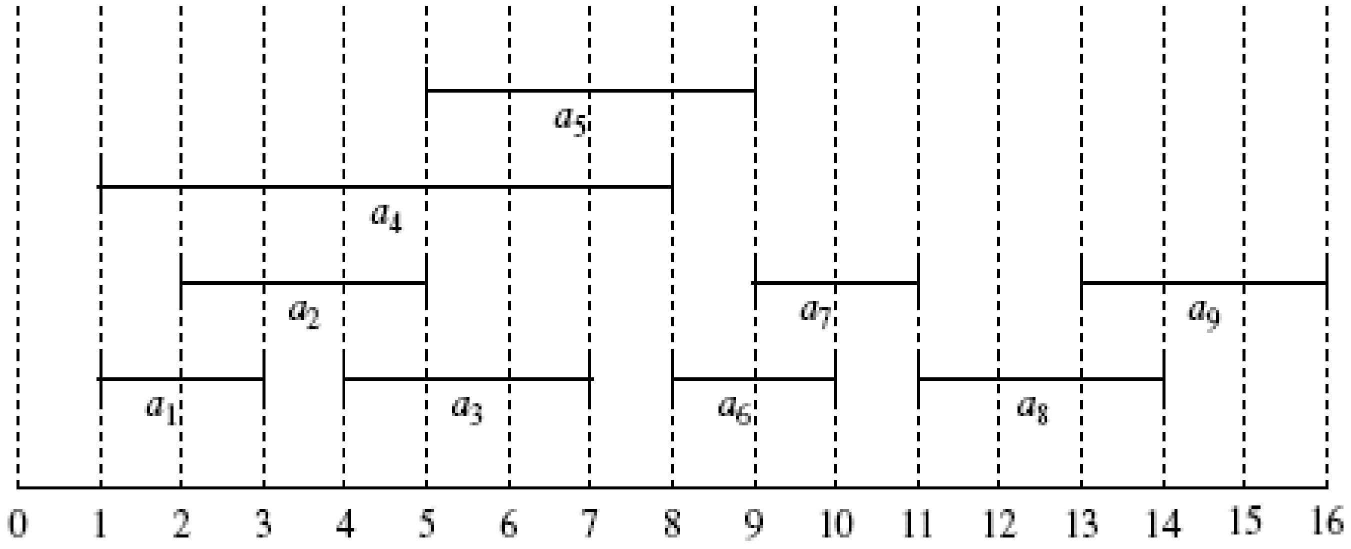
Sample Set of Activities



Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_2, a_5, a_7, a_9\}$.

Sample Set of Activities



• Simply picking the activity with the earliest finish time might be a good strategy, as it leaves the maximum-size unbroken block of time to schedule other activities.

• Can we prove that this is an optimum strategy?

Optimal substructure

In general, to solve a problem using the Greedy Method:

- The problem must exhibit the property of optimal substructure
- That is, an optimal solution to the problem must contain within it optimal solutions to subproblems.

Optimal Substructure of Activity-Selection Problem

- Define $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$.

  (subset of activities in S that can start after a_i finishes and finish before a_j starts)
- That is, activities in S_{ij} are compatible with:
  - All activities that finish by f_i, and
  - All activities that start no earlier than s_j.
- To avoid special cases, add fictitious activities a_0 and a_{n+1} and define $f_0 = 0$ and $s_{n+1} = \infty$.
- Assume activities are sorted in order of finish time:

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$$

- Thus space of subproblems is limited to selecting maximum-size subset from S_{ij} for $0 \leq i < j \leq n + 1$. (All other S_{ij} are empty.)

Optimal Substructure (continued)

Suppose that a solution to S_{ij} includes a_k

Then we have two subproblems:

- S_{ik} (start after a_i finishes, finish before a_k starts)
- S_{kj} (start after a_k finishes, finish before a_j starts)

Solution to S_{ij} is (solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (solution to S_{kj})

If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well.

Let A_{ij} = optimal solution to S_{ij}

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, assuming:

- S_{ij} is nonempty, and
- we know $a_{k.}$

Thus, solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} must be optimal as well.

Recursive Solution

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

Simplifying the Recursive Solution

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time:

$$f_m = \min \{f_k : a_k \in S_{ij}\}.$$

Then:

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Simplifying the Recursive Solution

Proof of Part 1:

Suppose A_{ij} is a maximum-size subset of mutually compatible activities of S_{ij}, and let us order the activities in A_{ij} in order of finish time. Let a_k be the first activity in A_{ij}.

Case 1: $a_k = a_m$

Then a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.

Simplifying the Recursive Solution

Case 2: $a_k \neq a_m$

Construct subset $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$.

The activities in A'_{ij} are disjoint since those in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, and $f_m \leq f_k$.

Noting that A'_{ij} has the same number of activities as A_{ij}, we see that A'_{ij} is a maximum-size subset of mutually compatible activities of S_{ij} that includes a_m.

Simplifying the Recursive Solution

Proof of Part 2:

Suppose S_{im} is nonempty; that is, there is some activity a_k such that $f_i \leq s_k < f_k \leq s_m < f_m$.

Then a_k is also in S_{ij} and has an earlier finish time than a_m.

This contradicts our choice of a_m.

Therefore S_{im} is empty.

Simplifying the Recursive Solution

The pattern to the subproblems:

- Original problem: $S = S_{0,\ n+1}$
- Choose a_{m_1} as activity in S with earliest finish time (i.e., $m_1 = 1$ since activities are sorted.)
- Next subproblem: $S_{m_1,\ n+1}$
- Choose a_{m_2} as activity in $S_{m_1,\ n+1}$ with the earliest finish time. (Notice that it is not necessarily true that $m_2 = 2$.)
- Next subproblem: $S_{m_2,\ n+1}$

The form of each subproblem is:

- $S_{m_i,\ n+1}$ for some activity a_{m_i}

Recursive Greedy Algorithm

- Assume that the start and finish times are stored in arrays *s* and *f*.
- Let *i* and *n* be indices of the subproblem to be solved.
- Assume activities are sorted in order of increasing finish times:

  $f_1 \leq f_2 \leq . . . f_n$

Recursive Greedy Algorithm

```
RECURSIVE-ACTIVITY-SELECTOR (s, f, i, n)
1  m ← i + 1
2  while m ≤ n and $s_m$ < $f_i$ do
                 ⊳ Find first activity in $S_{i,n+1}$
3    m ← m + 1
4  if m ≤ n
5    then return
     {$a_m$} ∪ RECURSIVE-ACTIVITY-SELECTOR (s,f,m,n)
6    else return ∅
```

Initial call:

RECURSIVE-ACTIVITY-SELECTOR (s, f, 0, n)

Running Time

What is the running time of this algorithm?

Assume the activities already have been sorted by finish times. We can do this in O(n lg n) time.

Every time we recursively call RECURSIVE-ACTIVITY-SELECTOR (s, f, i, n) we examine only 1 activity within the *while* loop test in line 2. Specifically, we will examine activity a_k during the last call made in which $i < k$.

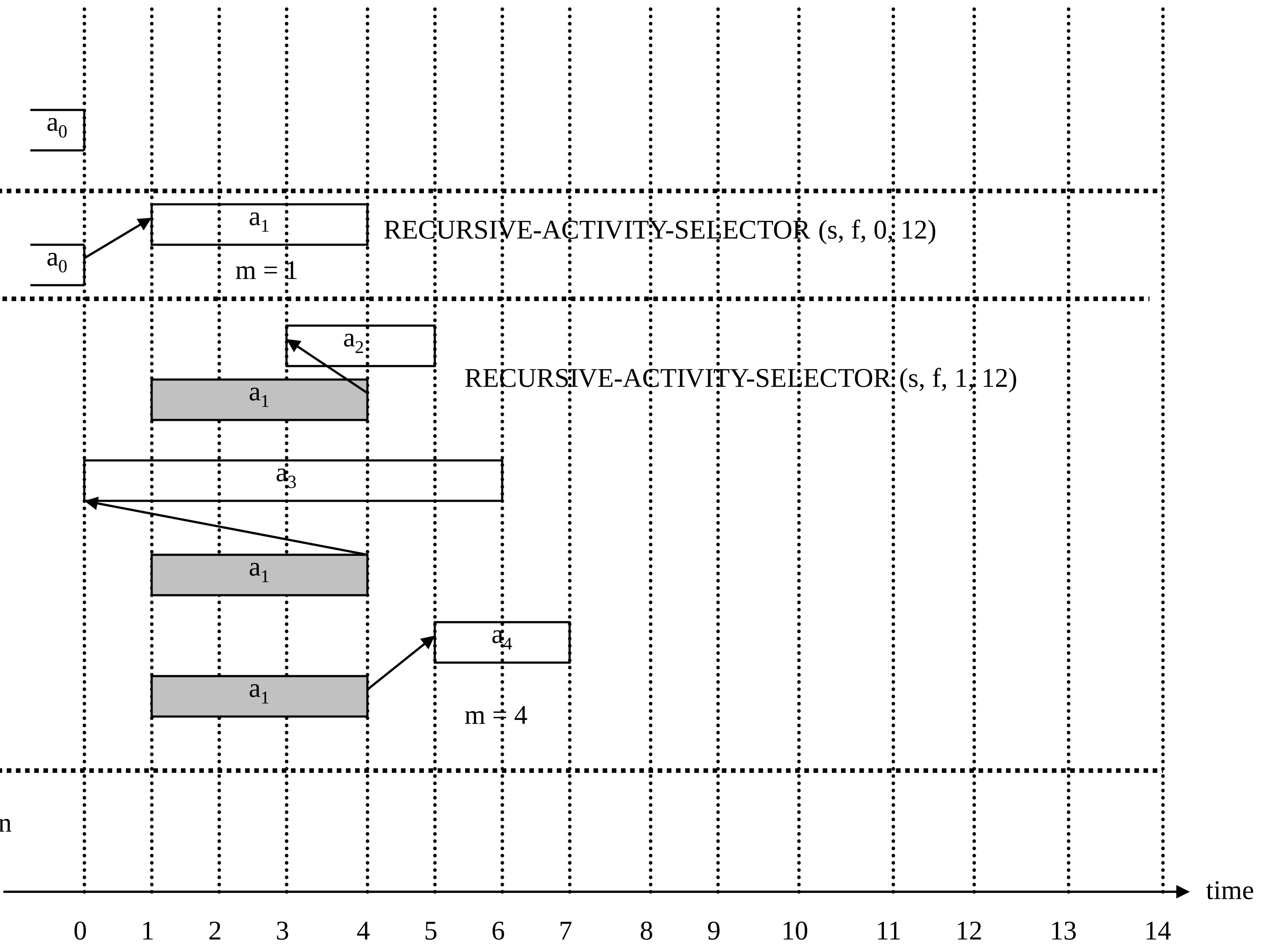
So the running time is $\Theta(n)$.

Sample Set of Activities

Sample set S (sorted by finish time):

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

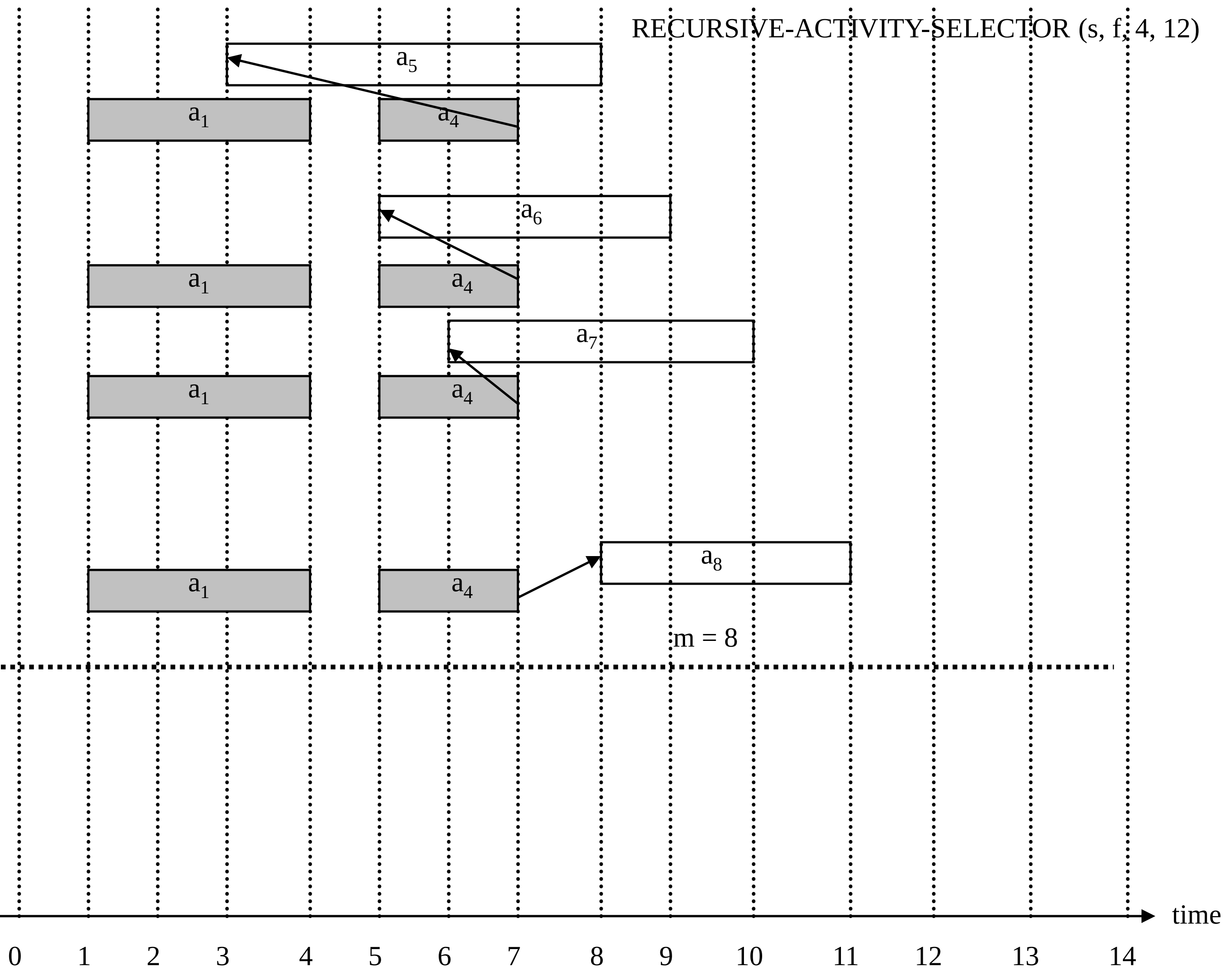
k	s_k	f_k
0	-	0
1	1	4
2	3	5
3	0	6
4	5	7



continued on next slide

RECURSIVE-ACTIVITY-SELECTOR (s, f, 4, 12)

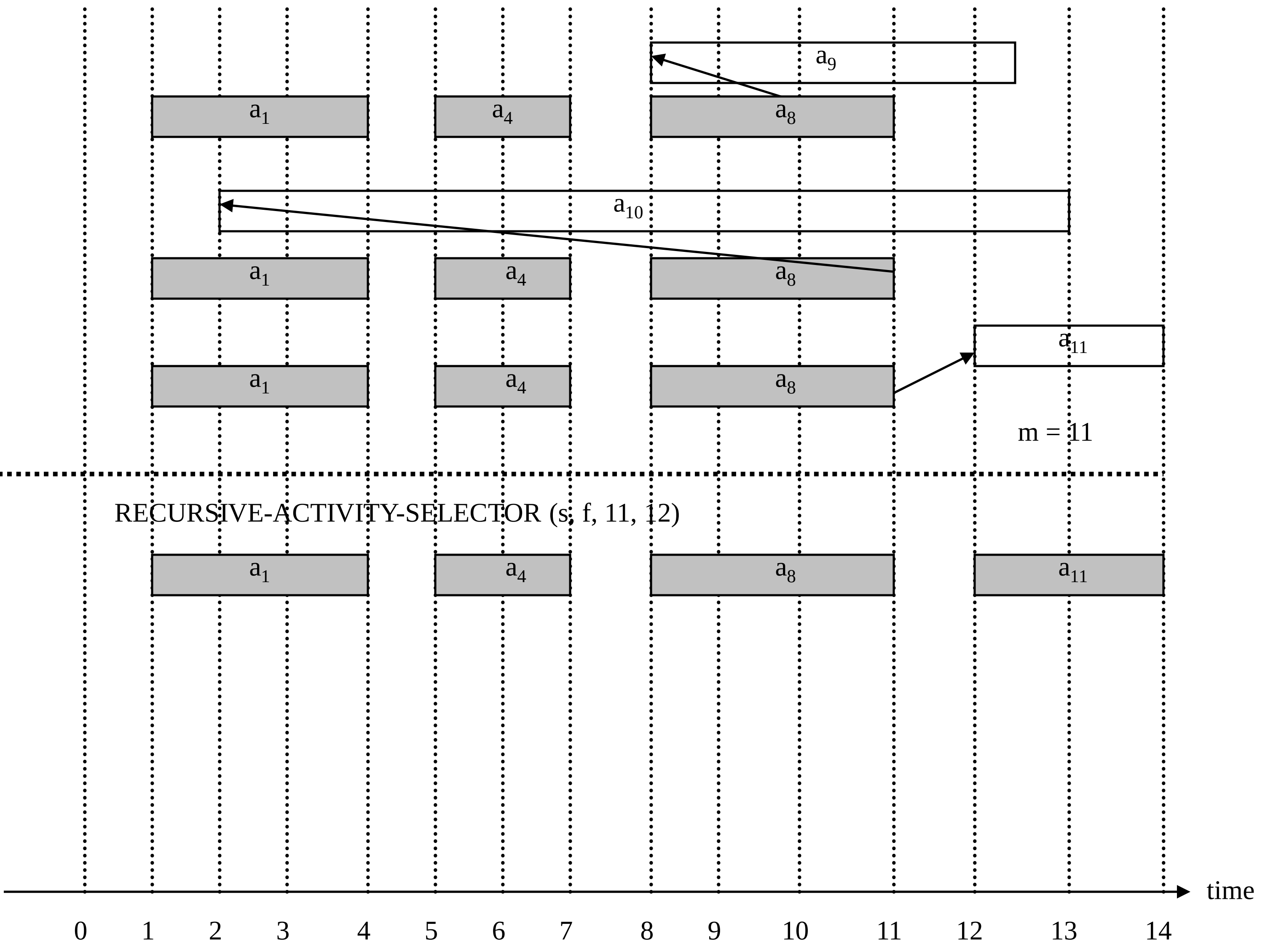
k	s_k	f_k
5	3	8
6	5	9
7	6	10
8	8	11



continued on
next slide

RECURSIVE-ACTIVITY-SELECTOR (s, f, 8, 12)

k	s_k	f_k
9	8	12
10	2	13
11	12	14
12	∞	-



Iterative Greedy-activity-selector

We can also implement this same process in iterative form instead of recursive.

Iterative Greedy-activity-selector

```
GREEDY-ACTIVITY-SELECTOR(s,f)
1   n ← length[s]
2   A ← {$a_1$}
3   j ← 1
4   for i ← 2 to n do
5       if $s_i \geq f_j$
6           then A ← A ∪ {$a_i$}
7               j ← i
8   return A
```

Analysis of the Algorithm

- The activity selected for consideration is always the one with the earliest finish
- Why does this work? Intuitively, it always leaves the maximum time possible to schedule more activities
- The greedy choice maximizes the amount of unscheduled time remaining

Elements of the Greedy Strategy

- A greedy strategy results in an optimal solution for some problems, but for other problems it does not.
- There is no general way to tell if the greedy strategy will result in an optimal solution
- Two ingredients are usually necessary
  - greedy-choice property
  - optimal substructure

Greedy-choice Property

- *Greedy-choice property:* A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Unlike dynamic programming, we solve the problem in a top down manner.
- Must prove that the greedy choices result in a globally optimal solution.

Optimal Substructure

- *Optimal substructure:* A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Dynamic programming also requires that a problem have the property of optimal substructure.

Choosing an algorithm

Choosing an algorithm

- Given a choice between using a greedy algorithm and a dynamic programming algorithm for the same problem, in general which would you choose?

Choosing an algorithm

- Advantages of top-down vs. bottom-up.

Choosing an algorithm

- Consider the *0-1 knapsack problem*: a thief robbing a store finds n items, where item i has a value of v_i dollars and a weight of w_i pounds . He wants to steal items with the highest total value, but his knapsack will carry only W pounds of items. W, v, and w are all integers.
- Will a greedy algorithm work for this problem?

The 0-1 Knapsack Problem

- Note that each item must either be taken or left behind; the thief cannot take a part of an item. Also, there is only one of each item.

The 0-1 Knapsack Problem

Possible greedy strategies:

1. Steal the items in order of decreasing value. Suppose we have 3 items:

A	25 lbs	$10
B	10 lbs	$9
C	10 lbs	$9

Our knapsack can hold *W* = 30 lbs. of items.

Will this greedy algorithm yield an optimal solution? No.

The 0-1 Knapsack Problem

Possible greedy strategies:

2. Steal the items in order of increasing weight. Suppose we have 3 items:

A	25 lbs	$10
B	12 lbs	$4
C	12 lbs	$4

Our knapsack can hold W = 30 lbs. of items.

Will this greedy algorithm yield an optimal solution? No.

The 0-1 Knapsack Problem

Possible greedy strategies:

3. Steal the items in order of decreasing profit per unit weight. Suppose we have 3 items:

A	5 lbs	$50
B	10 lbs	$60
C	20 lbs	$140

Our knapsack can hold W = 30 lbs. of items.

Will this greedy algorithm yield an optimal solution? No.

The Fractional Knapsack Problem

Now consider the *fractional knapsack problem*: a thief robbing a store finds a set S of n items, where item i has a value of v_i dollars and a weight of w_i pounds . He wants to steal items with the highest total value, but his knapsack will carry only W pounds of items. W, v, and w can be real numbers, not just integers; that is, the thief can steal fraction amounts of items.

The Fractional Knapsack Problem

Will a greedy algorithm work for this problem?

Yes. Steal the items in order of decreasing profit per unit weight. Put the most profitable item in your knapsack as, then the next most profitable, etc. If you can't get all of an item in, include as much of it as possible.

Choosing an algorithm

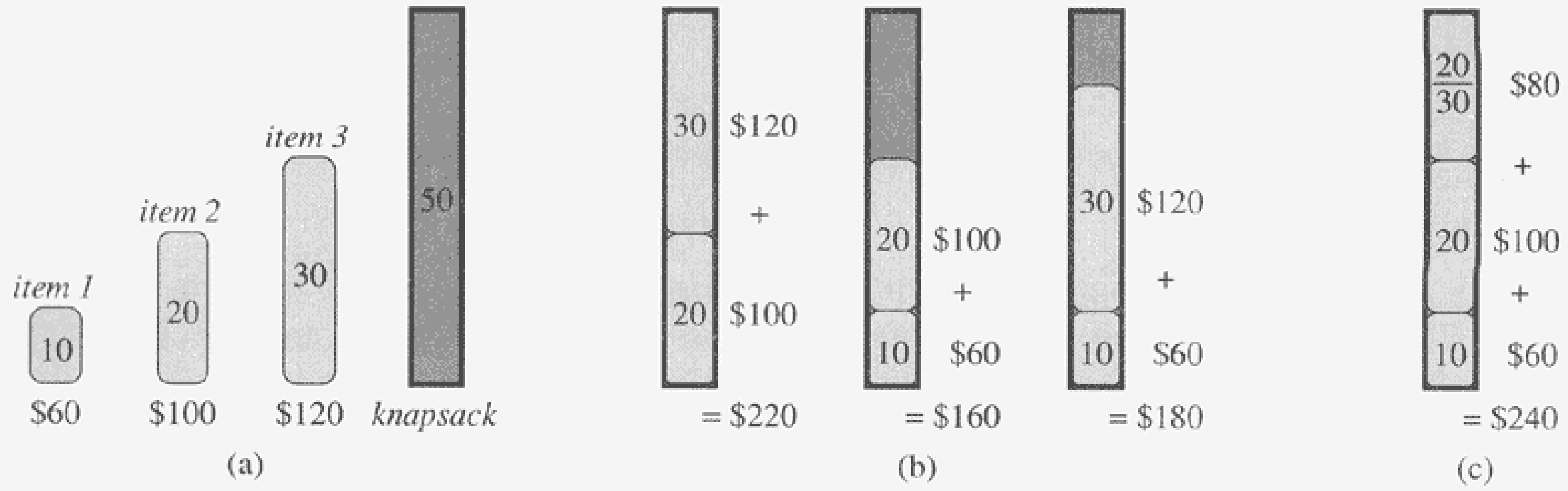


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest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

Greedy Fractional Knapsack Algorithm

Here is our problem definition:

Input: *S* is an array of *n* items, where item *i* has a value of v_i dollars and a weight of w_i pounds. *W* is the maximum weight the knapsack can hold.

Output: X is an array of *n* items, where item x_i is the amount of item *i* that is placed in the knapsack.

Greedy Fractional Knapsack Algorithm

```
GREEDY-FRACTIONAL-KNAPSACK (S, W)
for i ← 1 to n do
    v-ratio$_i$ ← $v_i$ / $w_i$
    $x_i$ ← 0
sort the items in S by v-ratio
$w_{total}$ ← 0
i ← 0
while $w_{total}$ < W do
    i ← i + 1
    a ← min($w_i$, W – $w_{total}$)
    $x_i$ ← a
    $w_{total}$ ← $w_{total}$ + a
```

The Fractional Knapsack Problem

What is the running time of the greedy algorithm solution to the fractional knapsack problem?

The running time is O(n lg n)

Why?

Comparing the Knapsack Problems

Both algorithms exhibit the optimal substructure property. Consider the most valuable load that weighs at most *W* pounds. If we remove item *j* from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ (*j selected*) or *W* (*j excluded*) that the thief can take from the $n - 1$ original items.

Comparing the Knapsack Problems

Why does the greedy algorithm approach work for the fractional knapsack problem but not for the 0-1 knapsack problem?

Because we may have wasted space in the 0-1 knapsack problem, but we don’t have to worry about that in the fractional knapsack problem.

Comparing the Knapsack Problems

This means that in the 0-1 knapsack problem, we can't just make our greedy choice and go on; we have to consider the possibility that this choice will result in wasted space, and compare the effect of the wasted space on the optimality of the solution. Thus, we must compare the greedy choice with other choices.

Huffman Codes

Huffman Codes

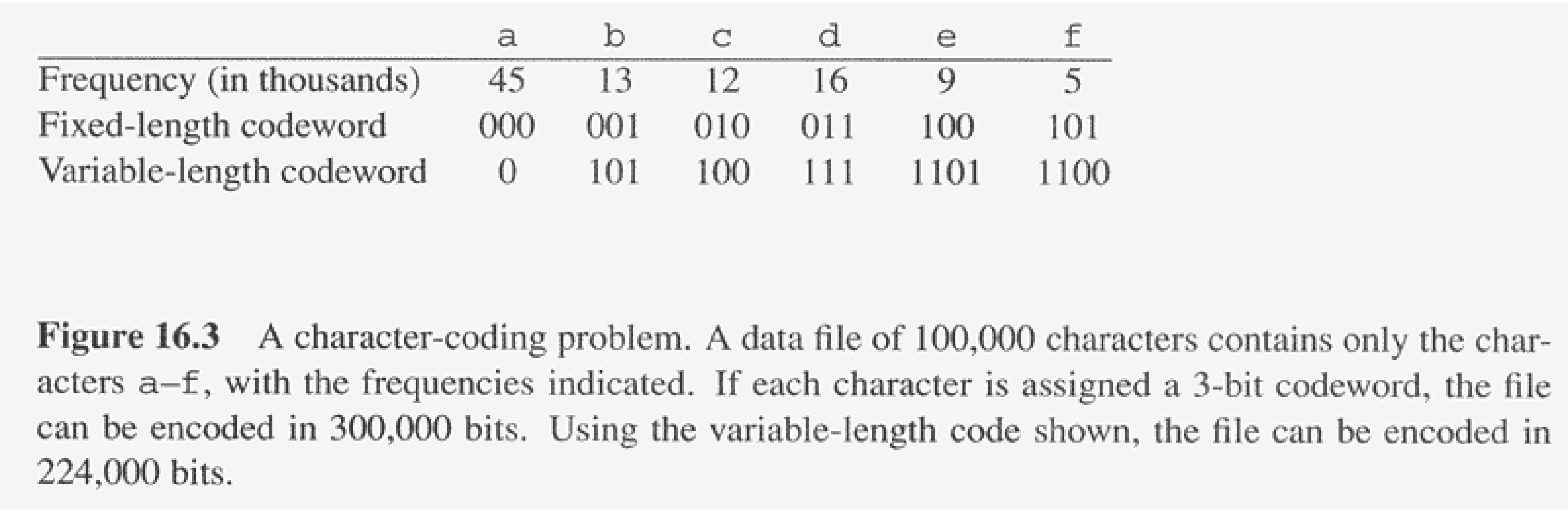
Consider the problem of data compression. Suppose that we have a 100,000-character file that contains only 6 different characters, a - f. Some characters occur more frequently than others.

Currently, the file is stored using a fixed-length code of 3 bits per character, where a = 000, b = 001, ..., f = 101. This requires 300,000 bits to store the file.

Huffman Codes

- Huffman coding uses a variable-length code to compress the file.
  - We can use a 0 to represent the most frequently-occurring letter in the file, which will save us two bits per occurrence.
- Huffman codes are prefix codes, which means that all bit patterns are unambiguous;
  - This requires that the bit-patterns for our other letters be 3 or 4 bits long.

Huffman Codes



	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

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.

Huffman Codes

Since we are using a prefix code, we can simply concatenate our codewords together to produce our bitstring.

For example, the string *abc* can be represented as 0101100.

This is unambiguous. Why?

Huffman Codes

Only one character can begin with 0; that is *a*. So *a* must be the first character in our string.

This leaves 101100. The next bit is a 1; five characters can begin with 1, so we look at the second bit. Two characters can begin with 10, so we look at the third bit. Only one character can begin with 101; that is *b*.

This leaves 100. Again, looking at all three bits, we see that this character must be *c*.

Huffman Codes

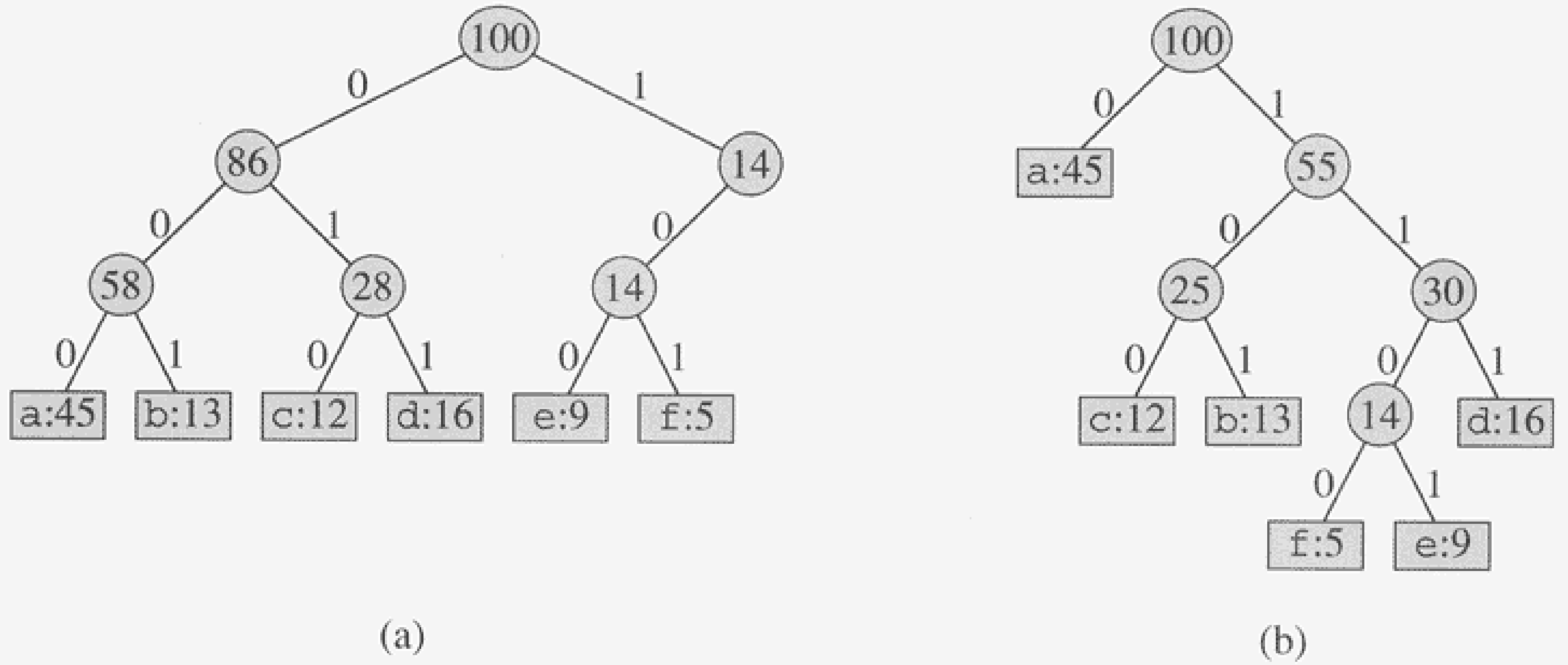


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . .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

Huffman Codes

- A greedy algorithm can be used to construct an optimal prefix code (called a Huffman code).
- This is done by building a tree corresponding to the optimal code.
- The tree is built from the bottom up, starting with a set of *n* leaves, where *n* is the number of characters in our code.
- The leaves are then merged into a tree.

Huffman Codes

• *C* is a set of *n* characters

• Each character *c* in *C* has a certain frequency of occurrence, *f*[*c*]

• The tree *T* is constructed starting with *n* leaves and performing *n -1* merging operations.

• A min-priority queue, keyed on *f*, is used to identify the two least-frequent nodes to merge together.

•The result of the merger is a new node whose frequency is the sum of the frequencies of the two nodes that were merged.

Huffman Codes

```
HUFFMAN (C)
n $\leftarrow$ |C|
Q $\leftarrow$ Build-Min-Heap(C)
for i $\leftarrow$ 1 to n – 1 do
    allocate a new node z
    left[z] $\leftarrow$ x $\leftarrow$ Extract-Min(Q)
    right[z] $\leftarrow$ y $\leftarrow$ Extract-Min(Q)
    f[z] $\leftarrow$ f[x] + f[y]
    Insert(Q, z)
return Extract-Min(Q)
```

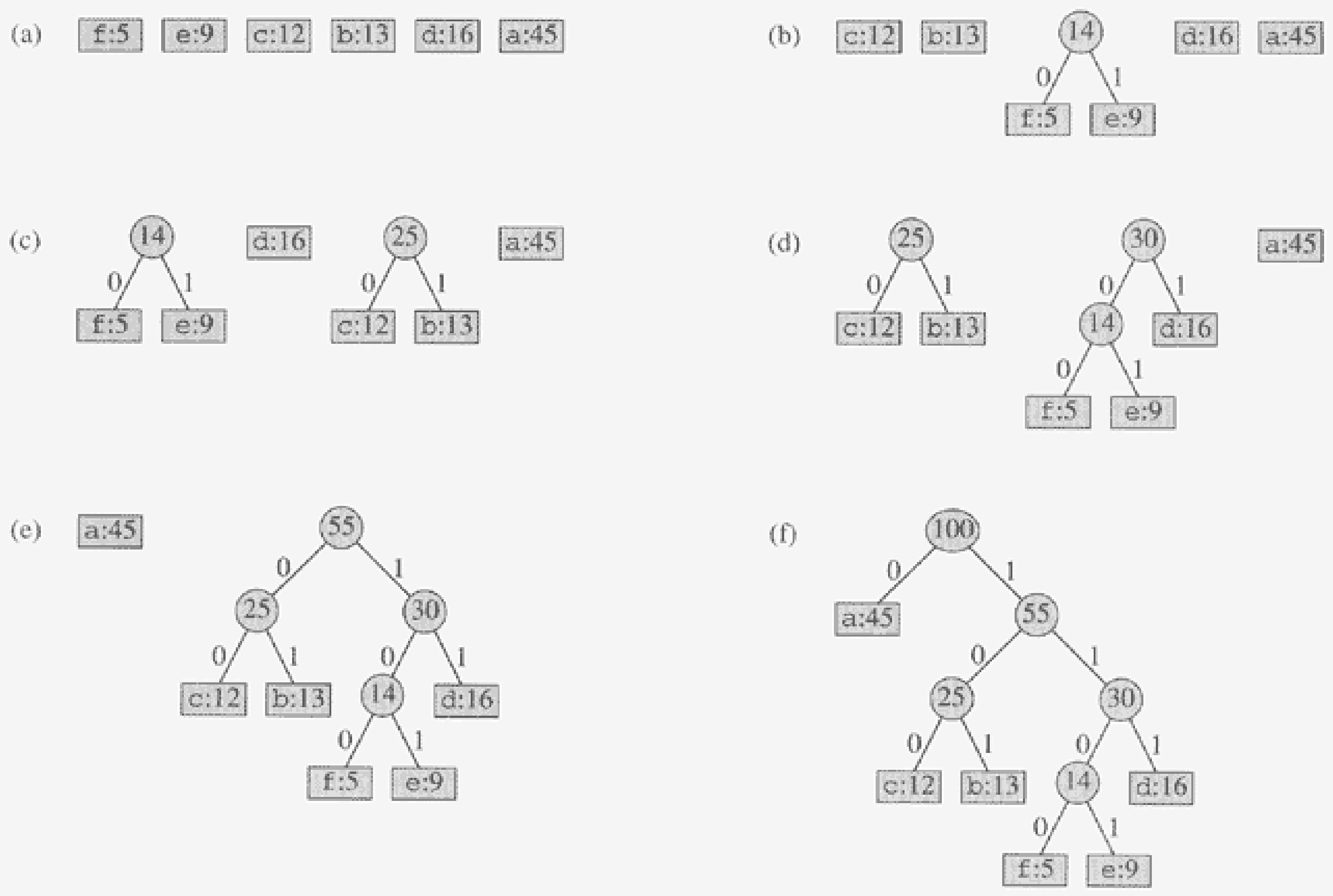


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency.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

Huffman Codes

What is the running time of the greedy Huffman code algorithm?

Remember that Build-Min-Heap (line 2) can be done in O(n) time.

The *for* loop in lines 3-8 is executed *n-1* times.

The Extract-Min and Insert queue operations within the *for* loop require O(lg n) each.

So the running time = O(n lg n)

Conclusion

A greedy algorithm works by always making the best choice at the moment - a “locally optimal” choice.

This means that we don’t have to consider any previous choices, or worry about any of the choices ahead; we just pick the one that seems best at the moment.

Conclusion

A greedy algorithm works well on a specific problem if the problem exhibits *optimal substructure* and has the *greedy-choice property*.

If a problem does not have the *greedy-choice property* the greedy algorithm is not guaranteed to produce optimal results, but usually enables you to find a solution faster than algorithms that guarantee an optimum solution.