Greedy Algorithm

What is a Greedy Algorithm?

- A greedy algorithm computes a solution to a problem in a step by step manner.
- In each step, the algorithm incrementally adds one component to the solution.
- It doesn't consider the larger problem as a whole. Once a decision is made, it is never reconsidered.
- Greedy algorithms are often used in ad hoc mobile networking to efficiently route packets with the fewest number of hops and the shortest delay possible. They are also used in machine learning, artificial intelligence (AI) and programming.

What is a Greedy Algorithm?

- It is usually an iterative algorithm, although recursion algorithms can also be greedy when each recursion call actually computes one further step.
- For example: A sorting algorithm-

compute **min(A[1….n])** and store it in A[1]

compute **min(A[2….n])** and store it in A[2]

and so on

Do All Problems Have Greedy Solution?

- The answer is '**NO**'. Only a few problems have greedy algorithms. For example : LCS problem, Merge sort, Quick sort are not greedy algorithms.

- Usually a greedy algorithm happens to be faster than other algorithms. However, there are many exceptions. Greedy algorithms may be slower than other algorithms (e.g. dynamic algorithm) of the same problem.

Greedy Algorithm

An Activity-Selection Problem

An Activity-Selection Problem

- Given a set S = {a_1, a_2, … ,a_n} of *n* proposed ***activities.*** These share the same resource which can serve only one activity at a time.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i ,

  where 0 $s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities.

An Activity-Selection Problem

- We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$$

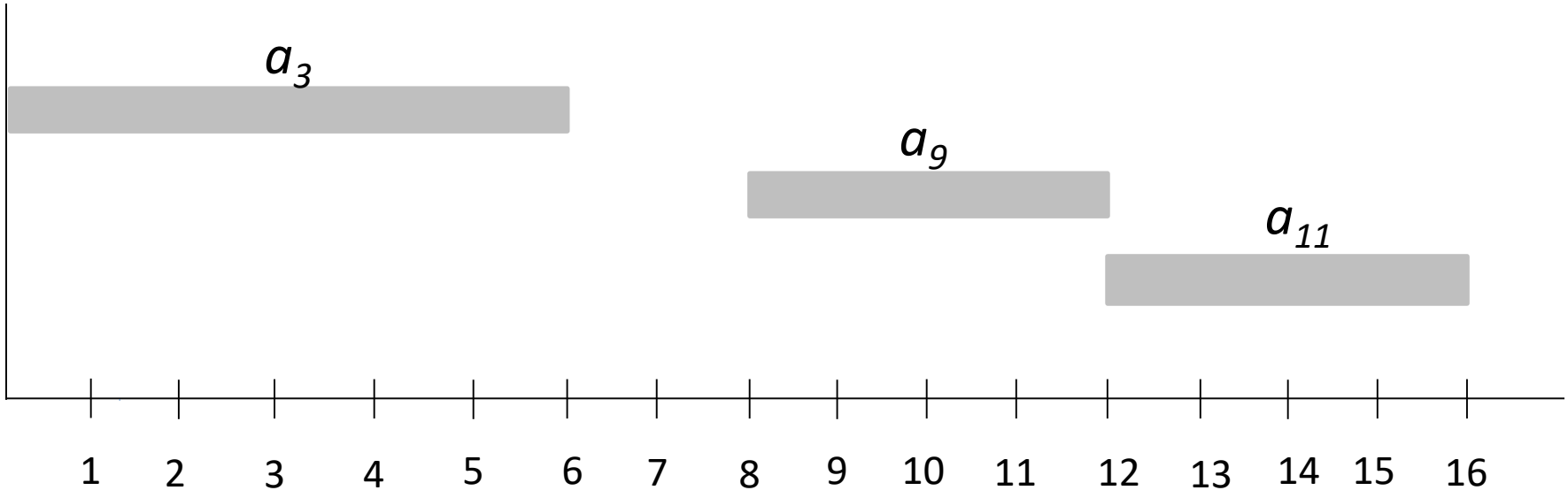
- Let us consider the following set S of activities:

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	9	9	10	11	12	14	16

An Activity-Selection Problem

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	9	9	10	11	12	14	16

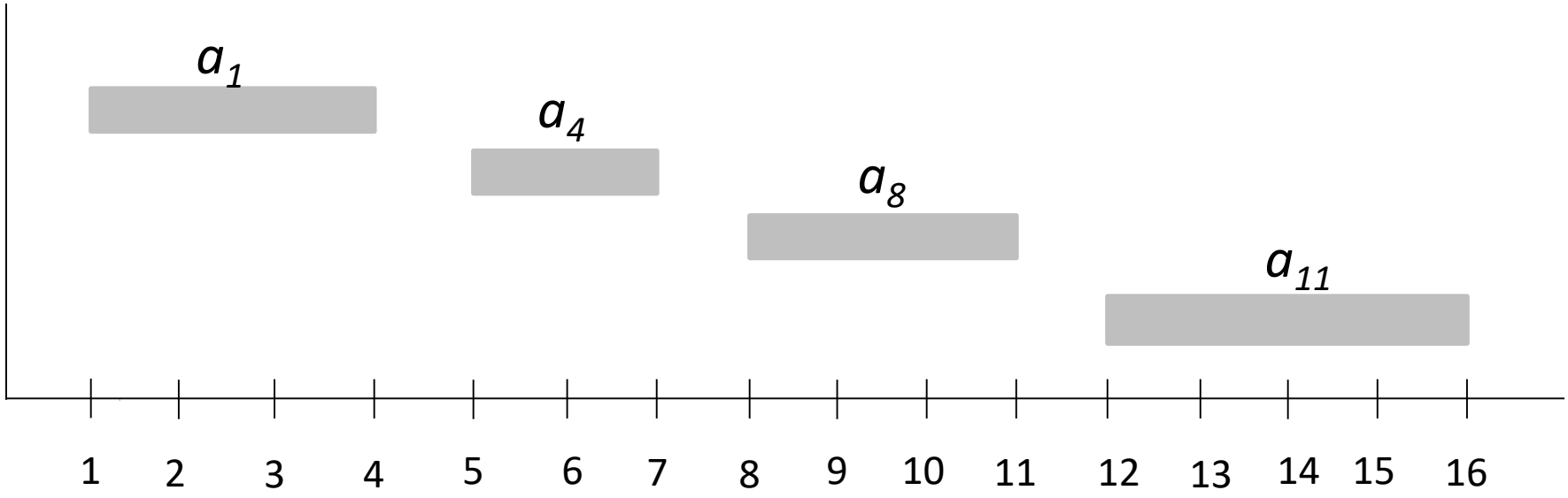
- For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.



An Activity-Selection Problem

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	9	9	10	11	12	14	16

- That is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. And it is one of the largest subsets.



Dynamic Programming Approach

- Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts.
- By including some activity a_k in an optimal solution, we are left with two subproblems:
  - finding mutually compatible activities in the set S_{ik}
  - finding mutually compatible activities in the set S_{kj}
- We denote the size of an optimal solution for the set S_{ij} by ***c[i, j]*** , then we would have the recurrence:

$$c[i, j] = c[i, k] + c[k, j] + 1$$

Dynamic Programming Approach

- As we do not know that an optimal solution for the set S_{ij} includes activity a_k , we have to examine all activities in S_{ij} to find which one to choose, so that

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

Recursive Algorithm

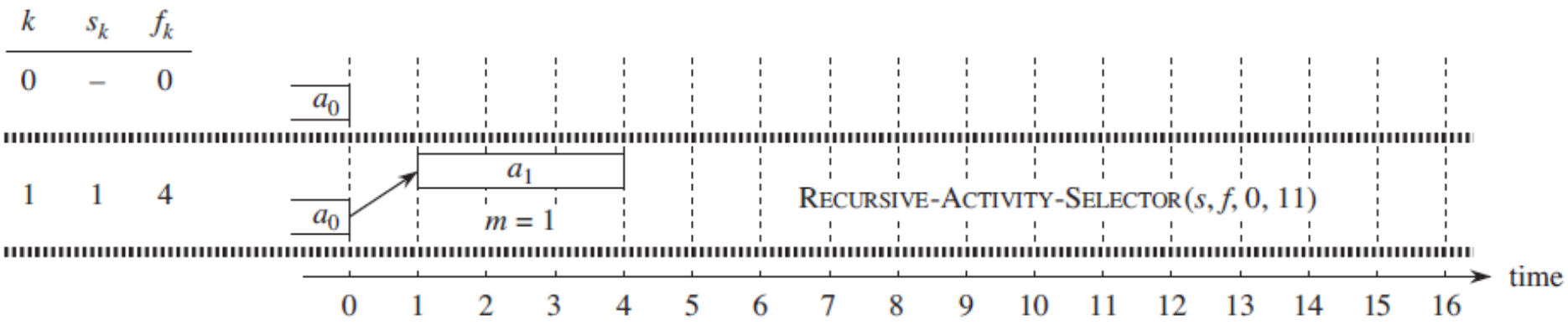
- Greedy choice: Select an activity with minimum f_i . Remove it and continue until no more activities left.
- Recursive algorithm for this approach:

```
RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3        m = m + 1
4  if m ≤ n
5        return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR (s, f, m, n)
6  else return ∅
```

- The loop examines a_{k+1}, a_{k+2}, … ,a_n , until it finds the first activity am that is compatible with a_k; such an activity has $s_m \geq f_k$

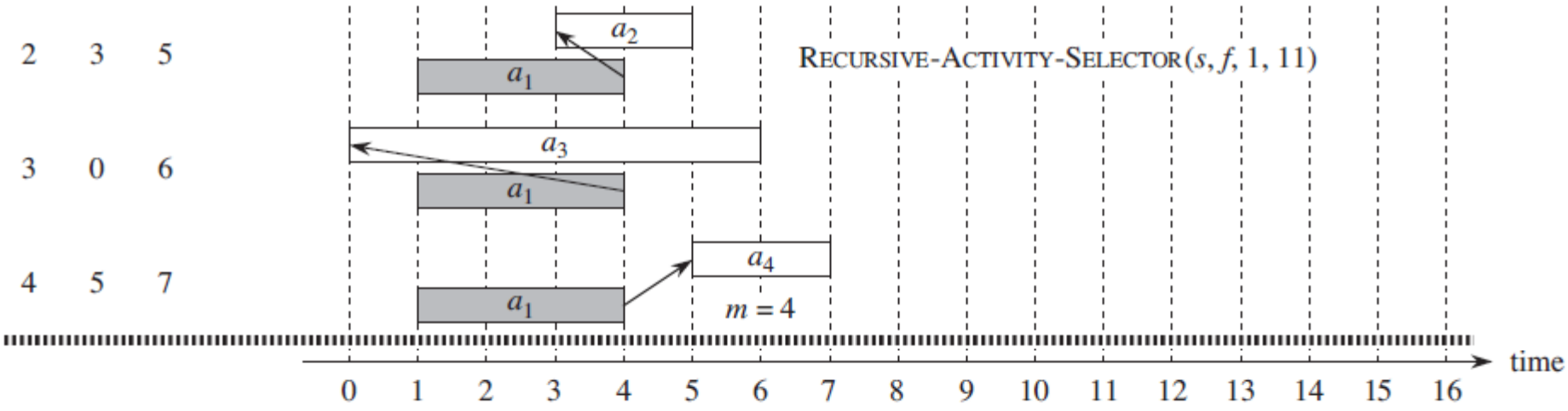
Simulation

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	9	9	10	11	12	14	16



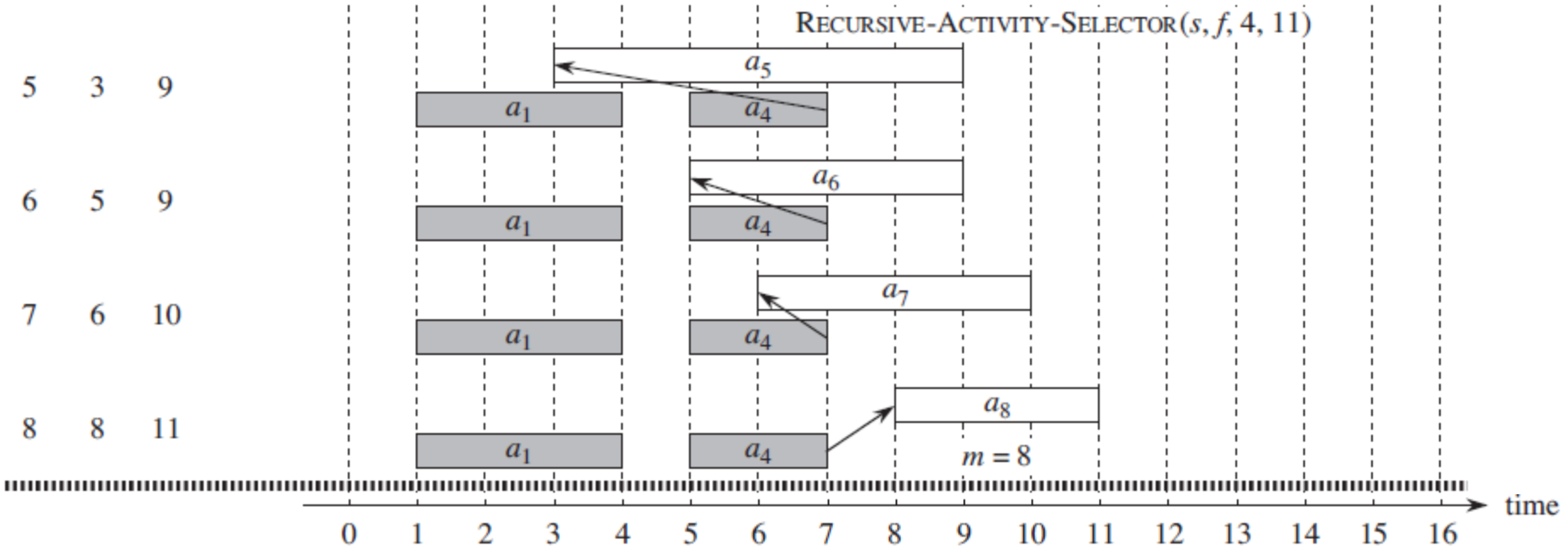
Simulation

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	9	9	10	11	12	14	16



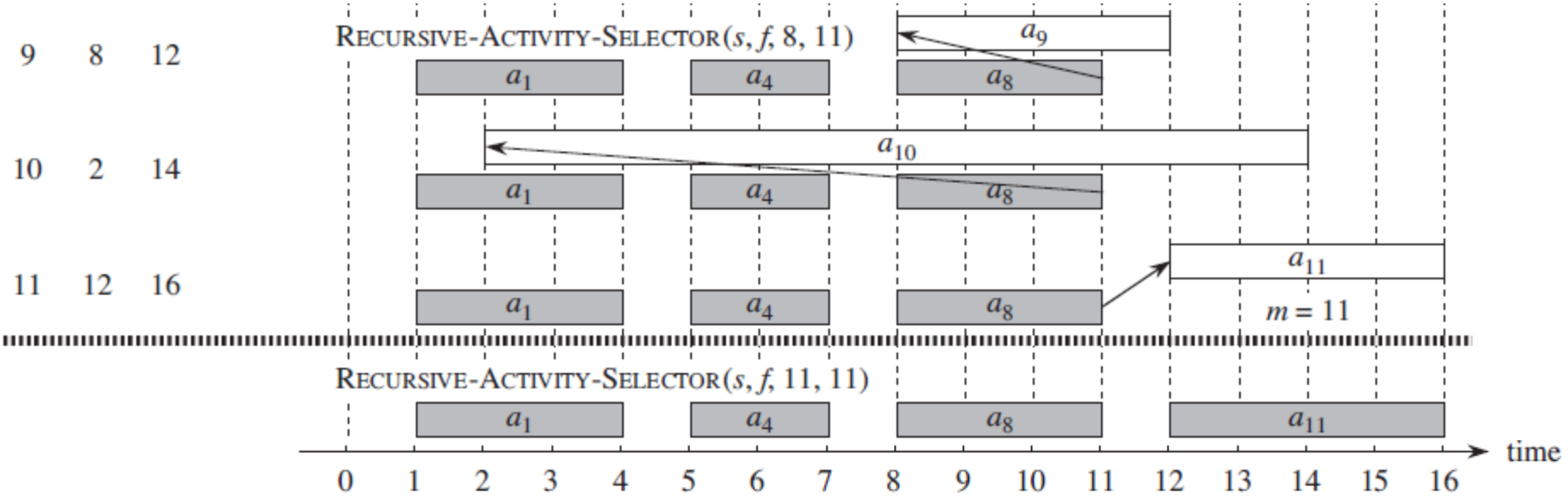
Simulation

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	9	9	10	11	12	14	16



Simulation

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	9	9	10	11	12	14	16



Greedy Algorithm

- In this algorithm the activities are sorted according to their finishing time, from the earliest to the latest.
- Then the activities are greedily selected by going down the list and by picking whatever activity that is compatible with the current selection.

- It collects selected activities into a set A and returns this set when it is done.

Greedy Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Greedy Algorithm

Huffman codes

Huffman codes

- Huffman codes compress data very effectively: savings of **20%** to **90%** are typical, depending on the characteristics of the data being compressed.
- Huffman's greedy algorithm uses a **table** giving how often each character occurs (i.e., its frequency).
- Suppose we have a **100,000-character data file** that we wish to store compactly.
- The characters in the file occur with the following frequencies:

	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

Fixed-length & Variable-length Codeword

- Each character is represented by a unique binary string, which we call a ***codeword***.
- If we use a ***fixed-length code***, we need 3 bits to represent 6 characters: a = 000, b = 001, . . . , f = 101.
  - This method requires 300,000 bits to code the entire file.
- A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
  - Representing ‘a’ as 1-bit string 0 and ‘f’ as 4-bit string 1100, the code requires

  (45 . 1 + 13 . 3 + 12 . 3 + 16 . 3 + 9 . 4 + 5 . 4) . 1,000 = 224,000 bits to represent the file.

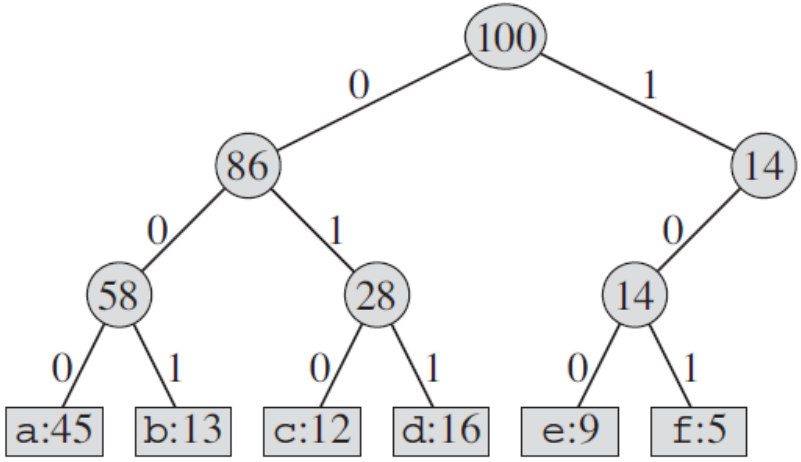
Prefix codes

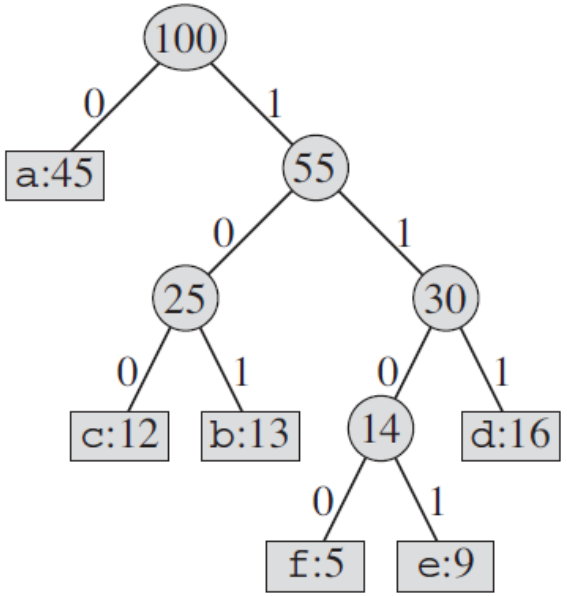
- No codeword is also a prefix of some other codeword. Such codes are called ***prefix codes***.
- Prefix codes are desirable because they simplify decoding.
- Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.
- For example: the string 001011101 parses uniquely as

  0 . 0 . 101 . 1101, which decodes to ‘aabe’.

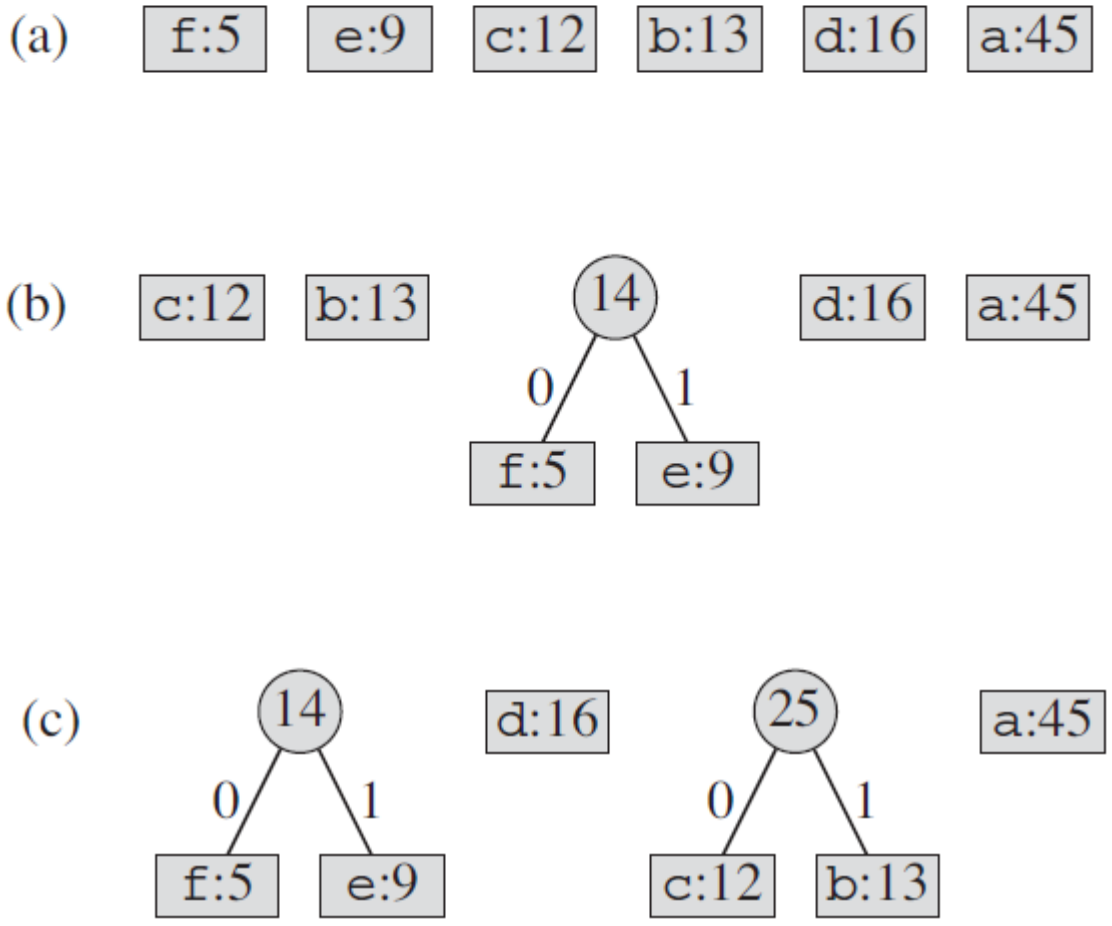
Optimal Solution

- An optimal code for a file is always represented by a ***full binary tree***, in which every nonleaf node has two children.
- Binary codeword for a character as the simple path from the root to that character, where
  - 0 means “go to the left child” and
  - 1 means “go to the right child.”

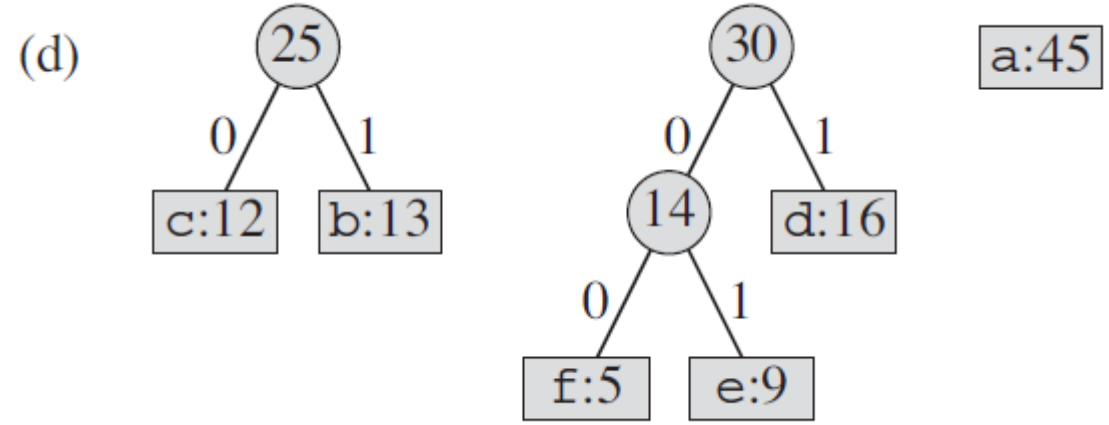


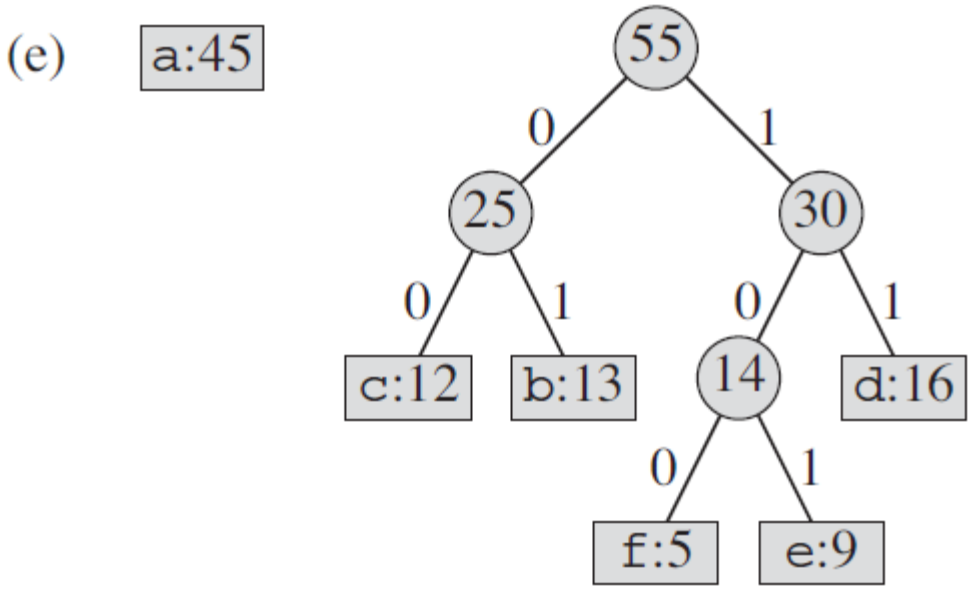


Constructing a Huffman Code

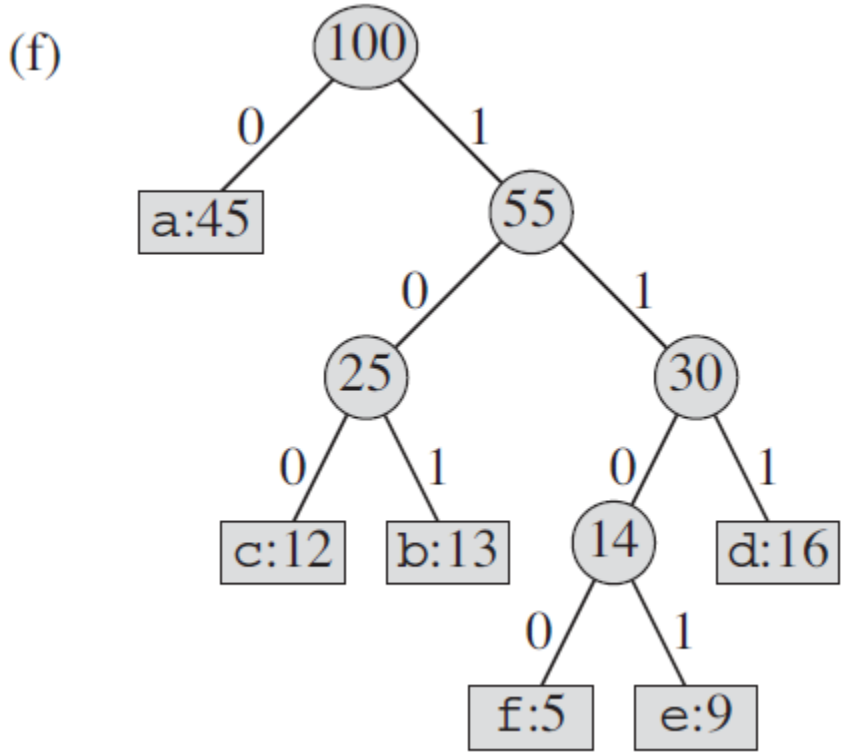


Constructing a Huffman Code





Constructing a Huffman Code



Constructing a Huffman Code

```
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 = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```