THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

COMP2013
Data Structures and Algorithms

Lecture 11
Greedy Algorithms

Dr. Jieming Shi

Dynamic Programming vs. Greedy

- DP
  - Exhaust all subproblems to optimally solve a problem
  - A subproblem is solved once and reused
- Greedy
  - Make the choice looking best at the moment
    - *Locally* optimal choice without considering all subproblems,
    - → in hope that this will lead to globally optimal solution
  - May or may not yield optimal solutions
    - Only if you can prove it leads to optimal

Greedy algorithms for

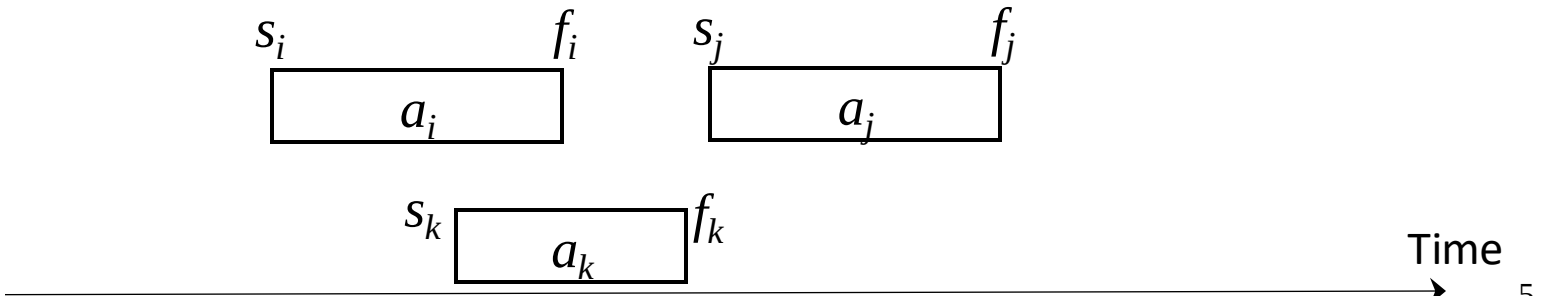
- **Activity selection problem**
- Huffman codes

Activity selection problem

- Given a set of competing activities that require *exclusive* use of a common resource (e.g., a conference room)
  - The room servers only one activity at a time
- Goal: select a maximum-size set of mutually compatible activities
  - *Compatible*: the time duration between any selected activities do not overlap
  - *Max*: support max number of activities

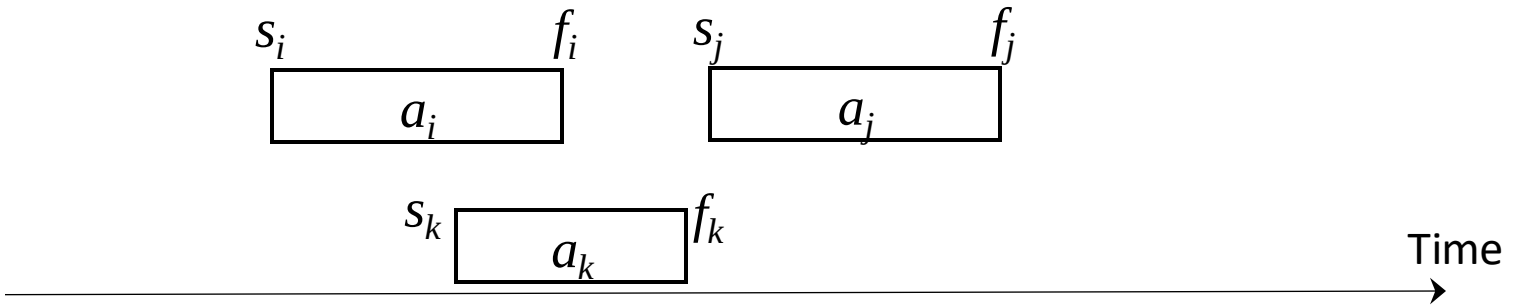
Activity selection problem

- Given a set of competing activities $S = \{a_1, a_2, \ldots, a_n\}$
- Each a_i has a *start* time s_i and *finish* time f_i
  - $[s_i, f_i)$
- a_i and a_j are *compatible* if
  - $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - $s_i \geq f_j$ or $s_j \geq f_i$
- From S, find a *max-size* subset of mutually compatible activities



Activity selection problem

- Given a set of competing activities $S = \{a_1, a_2, \ldots, a_n\}$
- Each a_i has a *start* time s_i and *finish* time f_i
  - $[s_i, f_i)$
- Assume that the activities in S are sorted in monotonically increasing order of finish time (If not, sort in this way first)
  - $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$



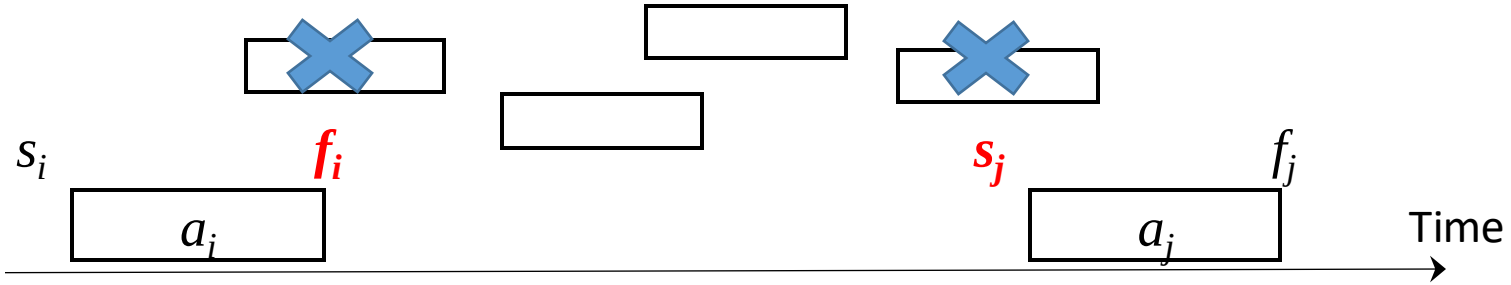
Activity selection problem

- Given a set of competing activities $S = \{a_1, a_2, \ldots, a_n\}$
- Each a_i has a *start* time s_i and *finish* time f_i
  - $[s_i, f_i)$
- Assume that the activities in S are sorted in monotonically increasing order of finish time (If not, sort in this way first)
  - $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Optimal substructure of the activity-selection problem

- Now, formulate subproblems and build the relationship between (sub)problems
- Problem size:
  - the number of activities in S
- S_{ij}: the set of activities starting after a_i finishes and finishing before a_j starts
  - These activities are compatible to and in between a_i and a_j



- $S_{1,9}$?
  - $= \{a_4\}$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Optimal substructure of the activity-selection problem

- S_{ij}: the set of activities starting after a_i finishes and finishing before a_j starts
  - We break it into subproblems
  - **Then find if the optimal solution of a problem can be built from the optimal solutions of its subproblems**

Optimal substructure of the activity-selection problem

- Suppose A_{ij} is an optimal solution of S_{ij} (A_{ij} is a *maximum* set with compatible activities from S_{ij})
  - A_{ij} includes an activity a_k
  - Divide S_{ij} into S_{ik}, $\{a_k\}$, S_{kj} (two subproblems)
  - $A_{ik} = A_{ij} \cap S_{ik}$
    - Is A_{ik} *an optimal solution of* S_{ik} *?*
    - Yes. Prove by *contradiction:*
      - If A_{ik} is not optimal, then there exists an optimal solution A'_{ik} of S_{ik} with $|A'_{ik}| > |A_{ik}|$,
      - and then in A_{ij}, you can replace A_{ik} by A'_{ik} to get a larger A'_{ij} with $|A'_{ij}| > |A_{ij}|$,
        - contradicting the assumption that A_{ij} is an optimal solution of S_{ij}

Optimal substructure of the activity-selection problem

- Suppose A_{ij} is an optimal solution of S_{ij} (A_{ij} is a *maximum* set with compatible activities from S_{ij})
  - A_{ij} includes an activity a_k
  - Divide S_{ij} into S_{ik}, $\{a_k\}$, S_{kj} (two subproblems)
  - $A_{ik} = A_{ij} \cap S_{ik}$
    - A_{ik} is an optimal solution of S_{ik}
  - $A_{kj} = A_{ij} \cap S_{kj}$
    - A_{kj} is an optimal solution of S_{kj}
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

- *Optimal substructure:*
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$

Optimal substructure of the activity-selection problem

- Optimal substructure:
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$
- Let $c[i, j]$ denote $|A_{ij}|$
  - $c[i, j] = c[i, k] + c[k, j] + 1$
- Consider all possible a_k:

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

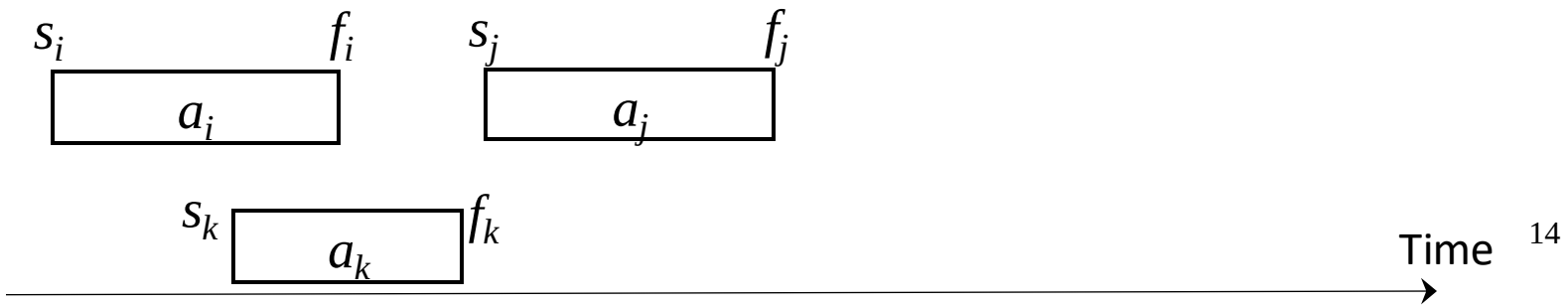
Recursive or DP solutions after optimal substructure

- Exhaust all subproblems
  - Recursion
  - Top-down or bottom-up DP

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

Faster greedy algorithm for activity selection problem

- Make a greedy choice, without having to solve all subproblems
- Choose the activity in *S* with the *earliest* finish time (why?)
  - Leave the resource available for as many of the activities that follow it as possible
  - For the activity selection problem, we can prove this greedy method can get an optimal solution
  - This is just one greedy way, but not the only way



Greedy algorithm

- $S = \{a_1, a_2, \ldots, a_n\}$, assume that the activities in S are sorted in monotonically increasing order of finish time
  - $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$

- First greedy choice: a_1
- Then only one remaining subproblem to solve:
  - Select the *earliest-finish* activity among all the activities *compatible* with a_1 (i.e., the activities start after a_1 finishes)
- Repeat the greedy procedure until no activity remains

Greedy algorithm

- $S = \{a_1, a_2, \ldots, a_n\}$
- $S_k = \{a_i \in S : s_i \geq f_k\}$
  - Activities starting after a_k finishes -> compatible with a_k
- Steps:
  - Greedy choice in S to get a_1 with earliest finish time
  - Get S_1 and greedy choice in S_1
    - to get an activity with the earliest finish time
      - (which may not be a_2, as a_2 may not be in S_1)
  - …
  - Get S_k and greedy choice in S_k
    - to get an activity with the earliest finish time
  - …
  - Until no more choice exists

Greedy algorithm

- The actual implementation does not need to materialize S_k
- Just scan each activity in S
- Running example:
  - n=11
    - Greedily put a_1 into A
  - $f_1 = 4$
  - $m = 2,3$: not compatible
  - $m = 4$: compatible and earliest-finish-time activity in S_1
    - Greedily put a_4 into A
  - $f_4 = 7$
  - $m = 5,6,7$: not compatible
  - $m = 8$: compatible and earliest-finish-time activity in S_4
    - Greedily put a_8 into A
  - $f_8 = 11$
  - $m = 9,10$: not compatible
  - $m = 11$: compatible and earliest-finish-time activity in S_8
    - Greedily put a_{11} into A

m	1	2	3	4	5	6	7	8	9	10	11
s_m	1	3	0	5	3	5	6	7	8	2	12
f_m	4	5	6	7	9	9	10	11	12	14	16

What is the running time?

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

Does this greedy approach get an optimal solution?

Is the greedy algorithm optimal?

- For *any* subproblem (set) S_k, is the earliest-finish-time activity a_m in S_k in some optimal solution of S_k?
  - Yes.
- Prove:
  - Let A_k be an optimal solution of S_k ($A_k \subseteq S_k$)
  - Let a_j be the earliest-finish-time activity in A_k
  - If $a_j = a_m$, done.
  - If $a_j \neq a_m$, $f_m \leq f_j$,
    - we can construct $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ contains *compatible* activities (since a_m finishes earlier than a_j),
    - A'_k, which includes a_m, is an optimal solution of S_k since $|A'_k| = |A_k|$
- In a nutshell, we are always choosing activities in optimal solutions, for the problem of activity selection

Key elements in greedy algorithm

- Optimal substructure
  - If an optimal solution to the problem contains within its optimal solutions to subproblems
- Greedy-choice property
  - You can assemble a globally optimal solution by making *locally* optimal (greedy) choices
    - without considering results from other subproblems
  - Needs proof for greedy algorithms

Greedy algorithms for

- Activity selection problem
- **Huffman codes**

Encoding

- Everything in a computer is encoded as a sequence of binary code in 0 and 1 bits
- E.g., Fixed-length code for each character
  - 010000101100
    - cafe
    - Space cost: 3*4 bits = 12 bits

a	b	c	d	e	f
000	001	010	011	100	101

Encoding

- An encoding scheme should not cost too much space to represent data
- *Huffman codes* compress data well

Huffman codes

- A file containing 100,000 characters, with 6 *distinct* characters (the alphabet C, $|C|=6$) appear in the file, each with *frequency* in the table
- Fixed-length codes:
  - 3*100,000 = 300,000 bits to encode the entire file
- *Variable-length* codes:
  - (45*1+13*3+12*3+16*3+9*4+5*4)*1000 = 224,000 bits
    - **Space saved**, to encode the same file

	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

Huffman codes

- Variable-length codes
- Idea: give *frequent* characters *short* codewords and infrequent characters long codewords

	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

Huffman codes

- Variable-length codes
- Prefix-free codes
  - in which *no* codeword is also a *prefix* of some other codeword (to avoid ambiguity when decoding)
  - Counter-example:
    - a: 0; b: 01; c: 1
    - The code of a is a prefix of b
    - What is the meaning of 001?
      - “ab” or “aac”?
  - Prefix-free codes avoid the above ambiguous issue

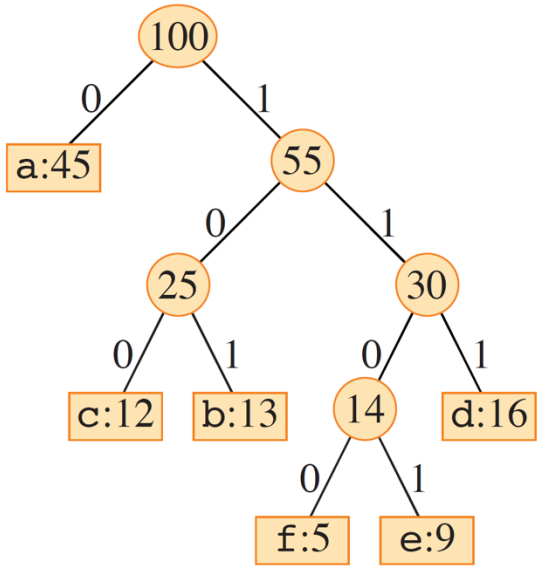
Huffman codes

- Variable-length codes
- Prefix-free codes
- Based on the last row of Huffman codes below, what is the meaning of 100011001101? (Decoding process)
  - cafe: 100 0 1100 1101
- How to do the decoding?

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

Huffman codes – a binary tree representation

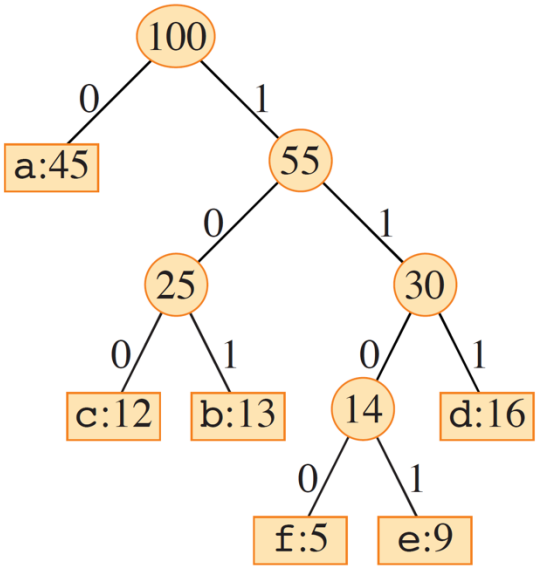
- A binary tree to represent the codes
  - All characters are on leaf nodes; intermediate nodes are not characters
    - The number of leaves is the number of alphabet
  - Number in a node is the total frequency of all characters under the tree rooted at the node
  - An edge represents one bit, either 0 or 1
  - A codeword of a character is on the path from the root to a *leaf* (0 go left, 1 go right)
    - A path from root to intermediate node does not represent a codeword
- Why these codewords in the binary tree are prefix-free?
  - A leaf is unique to a codeword



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

Huffman codes – a binary tree representation

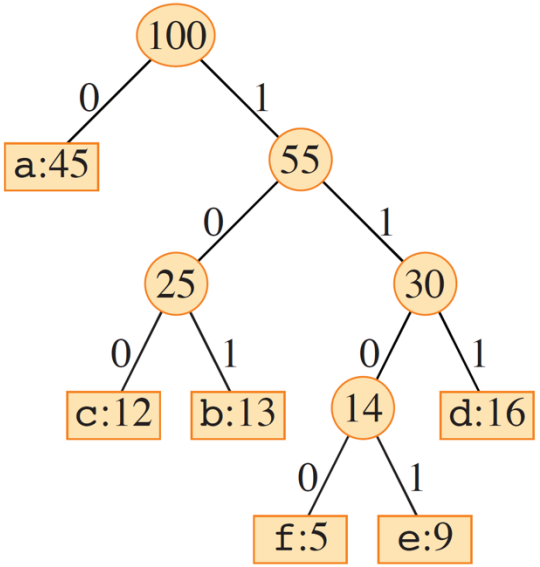
- How to decode 100011001101?
  - Starting from the root, every time, read a bit and decide to go left or right, one level down
  - When reaching a leaf, decode the corresponding character
  - Then repeat above again from the root to decode next character
  - until all bits are processed



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

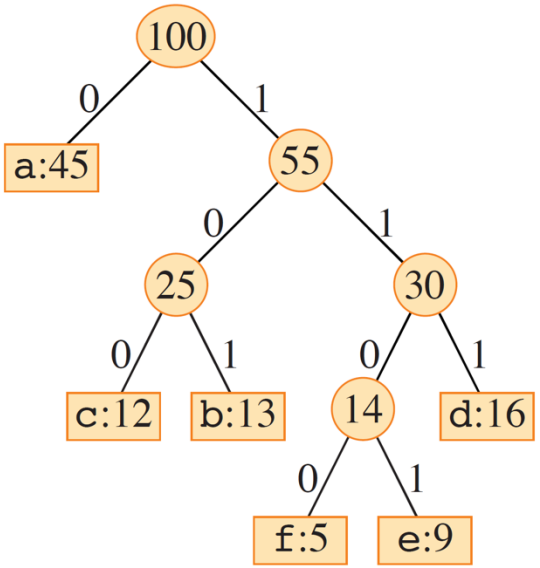
How to get the Huffman codes – How to build this tree?

- The alphabet is $C=\{a,b,c,d,e,f\}$
- Goal: Huffman codes save space to encode a file
  - $B(T)$: the number of bits required to encode the file by using the codes in the tree T:
  - $d_T(c)$: depth from root to leaf c
    - The number of bits in the code for c
  - $c.freq$: frequency of c in the file
  - The space cost of c is $c.freq \times d_T(c)$
  - The total space cost of T to encode the file is
    - $B(T) = \sum_{c \in C} c.freq \times d_T(c)$



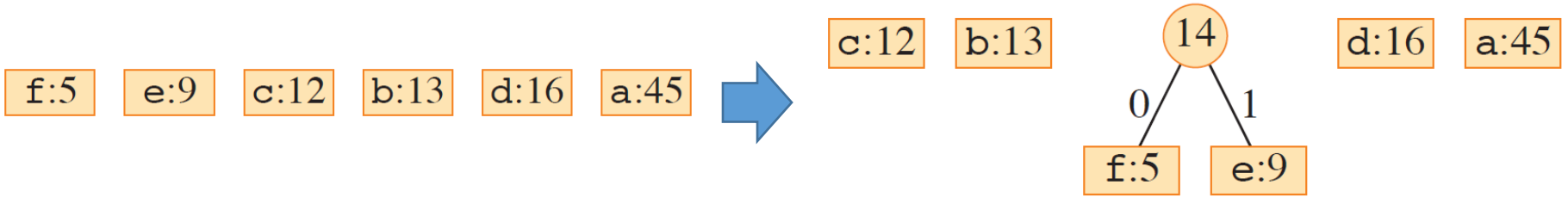
Huffman codes (an optimization problem)

- Given *a file* with *alphabet* C and character *frequencies*, how to construct a tree T with **minimum cost** $B(T)$ for alphabet C to *encode the file?*
  - $B(T) = \sum_{c \in C} c.freq \times d_T(c)$



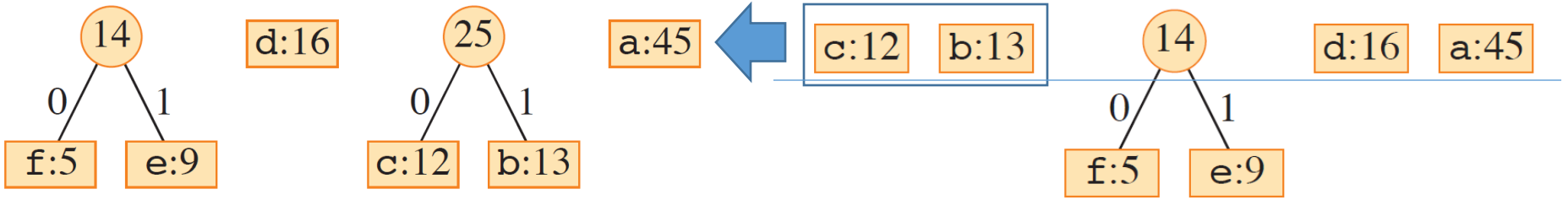
Construct Huffman codes by greedy algorithm

- Idea: give *frequent* characters *short* codewords and *infrequent* characters *long* codewords
  - A long codeword means a *long path* in binary tree *T* for Huffman codes
- Greedy:
  - Build *T* in a bottom-up way and handle least-frequent objects first
  - Identify the two *least-frequent* objects (nodes) to merge to form the parent object (node) of the two objects
    - the parent has frequency that is the sum of the frequencies of the two objects merged



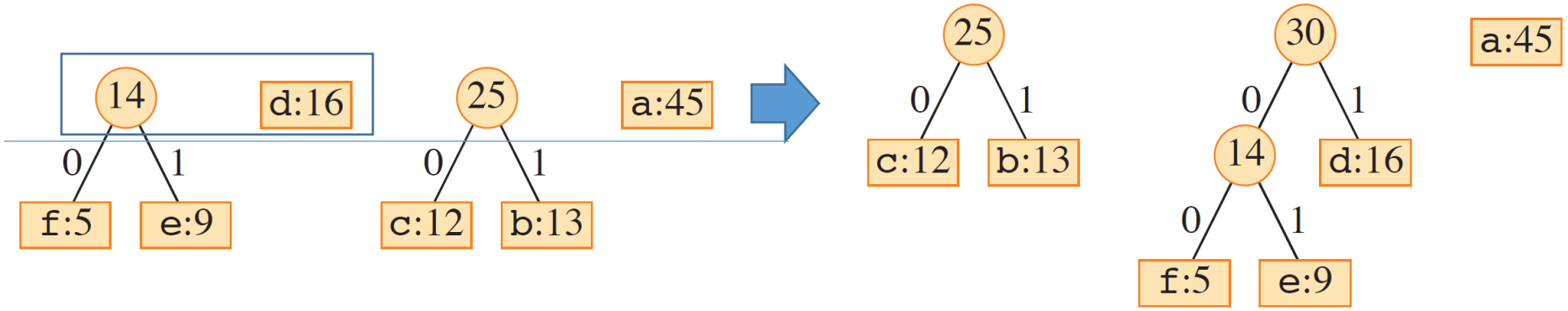
Construct Huffman codes by greedy algorithm

- Greedy:
  - Identify the two *least-frequent* objects (nodes) to merge to form the parent object (node) of the two objects
    - the parent has frequency that is the sum of the frequencies of the two objects merged



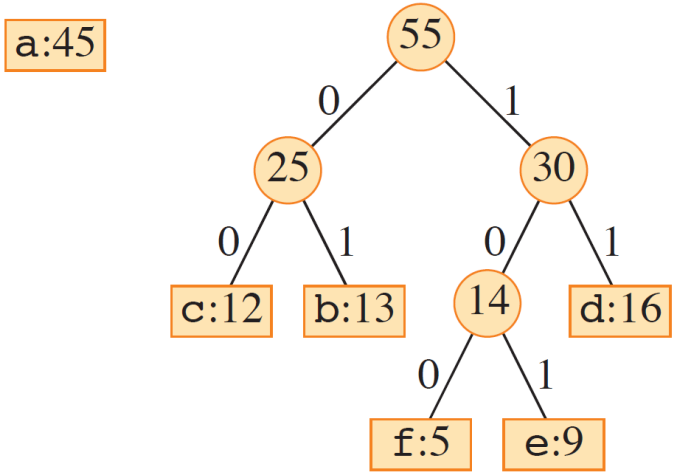
Construct Huffman codes by greedy algorithm

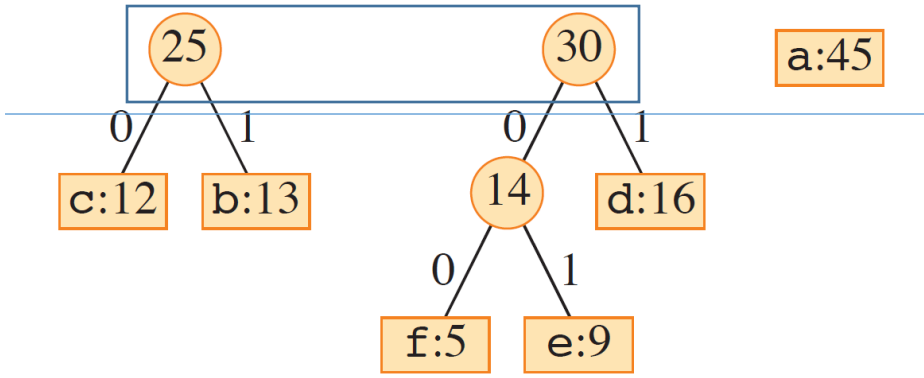
- Greedy:
  - Identify the two *least-frequent* objects (nodes) to merge to form the parent object (node) of the two objects
    - the parent has frequency that is the sum of the frequencies of the two objects merged



Construct Huffman codes by greedy algorithm

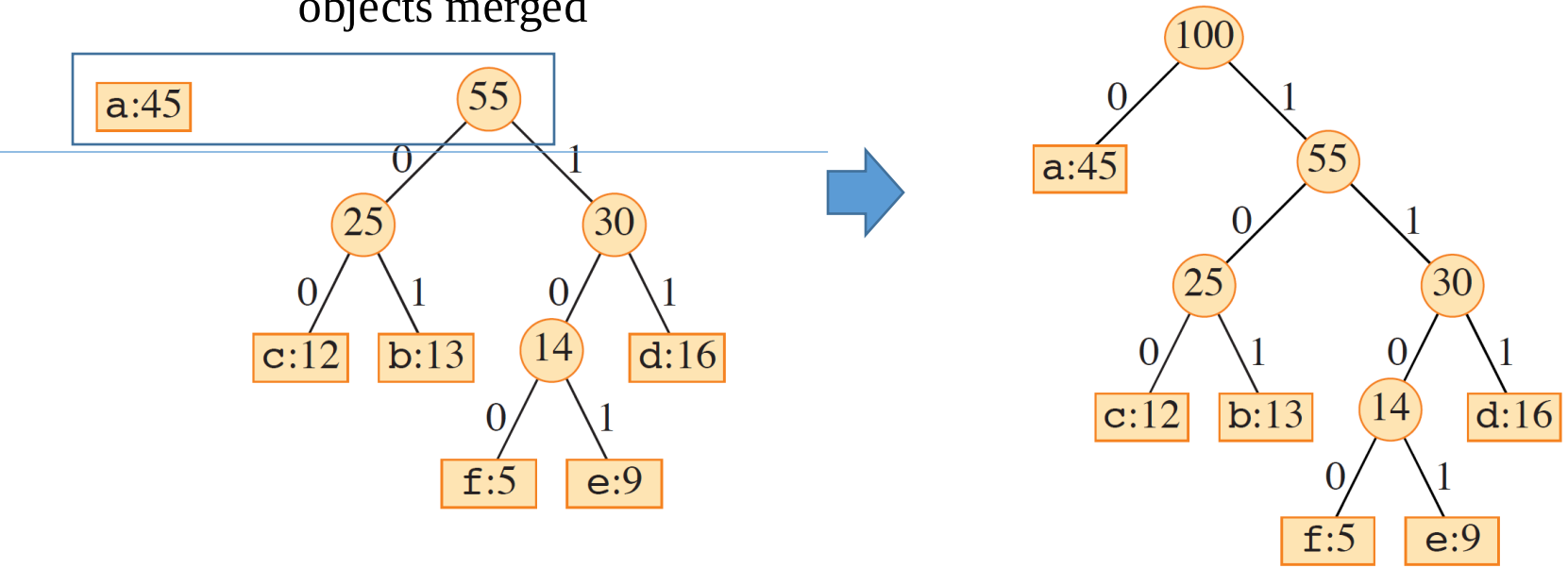
- Greedy:
  - Identify the two *least-frequent* objects (nodes) to merge to form the parent object (node) of the two objects
    - the parent has frequency that is the sum of the frequencies of the two objects merged





Construct Huffman codes by greedy algorithm

- Greedy:
  - Identify the two *least-frequent* objects (nodes) to merge to form the parent object (node) of the two objects
    - the parent has frequency that is the sum of the frequencies of the two objects merged



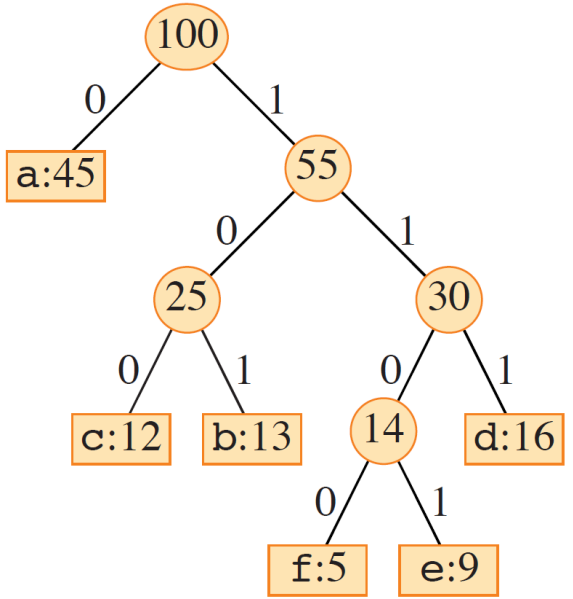
Huffman code – pseudo code

- The greedy approach to construct the tree produces an optimal prefix-free code
- What data structure do you need?
  - Leverage a min-priority queue Q: lower frequency has higher priority

```
HUFFMAN(C)
1   n = |C|                                  Total time complexity: O(nlogn)
2   Q = C
3   for i = 1 to n − 1
4       allocate a new node z
5       x = EXTRACT-MIN(Q)                   O(logn) in a priority queue
6       y = EXTRACT-MIN(Q)
7       z.left = x
8       z.right = y
9       z.freq = x.freq + y.freq
10      INSERT(Q, z)                         O(logn) in a priority queue
11  return EXTRACT-MIN(Q)                    // the root of the tree is the only node left
```

Is this greedy way to construct Huffman code optimal?

- Is the cost *B(T)* of the greedily constructed tree *T* minimized?
- 1. prove that the optimal prefix-free codes have the *greedy-choice property*
- 2. prove that the optimal prefix-free codes have the *optimal-substructure property*



Greedy Algorithms

- In this lecture, we focus on two problems that can be solved by greedy algorithms to get optimal solutions
- *There are other variants*
  - E.g., greedy algorithms leading to non-optimal but approximate solutions

Key elements in greedy algorithm

- Greedy-choice property
  - You can assemble a globally optimal solution by making *locally* optimal (greedy) choices
    - without considering results from other subproblems
- Optimal substructure
  - If an optimal solution to the problem contains within its optimal solutions to subproblems
    - Needs proof for greedy algorithms

Thank you