ขั้นตอนวิธีแบบละโมบ
Greedy Algorithms

การอบรมคอมพิวเตอร์โอลิมปิก สอวน. ค่ายที่ 2 ปีการศึกษา 2563

อ.ดร.ฐาปนา บุญชู

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี

มหาวิทยาลัยธรรมศาสตร์

Greedy Algorithms

Greedy Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do.
- A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a **globally** optimal solution.
- Greedy algorithms ***do not*** always yield optimal solutions, but for many problems they do.

An activity-selection problem

- The problem is to schedule several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually *compatible* activities.
- Suppose we have a set $S = \{a_1, a_2, \dots . a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, 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 $0 \leq s_i \leq f_i \leq \infty$
- If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.
- Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

An activity-selection problem

- In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible 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

- Dynamic Programming to solve?
  - We consider several choices when determining which subproblems to use in an optimal solution.
  - We shall then observe that we need to consider only one choice—the greedy choice—and that when we make the greedy choice, only one subproblem remains.

The optimal substructure of the activity-selection problem (Dynamic Programming)

- 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.
- Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij} and suppose further that such a maximum set is A_{ij}, which includes some activity a_k.
- By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} and S_{kj}
- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$.
- Thus, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- So, the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $\left|A_{ij}\right| = |A_{ik}| + \left|A_{kj}\right| + 1$ activities.

The optimal substructure of the activity-selection problem (Dynamic Programming)

- Solving this problem by dynamic programming.
- If we denote the size of an optimal solution for the set S_{ij} by $c[i,j]$, according the optimal sub structure we would have the following recurrence:

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

- In case that if we did not know that an optimal solution for the set S_{ij} includes activity a_k we would have to examine all activities in S_{ij} to find which one to choose:

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

Making the greedy choice

- What if we could choose an activity to add to our optimal solution without *having to first solve all the subproblems*?
  - That could save us from having to consider all the choices inherent in recurrence.
- In fact, for the activity-selection problem, we need consider only one choice: **the greedy choice**.
  - Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible.
  - Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible

Making the greedy choice

- Since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1.
- If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that start **after a_1 finishes**.
- Furthermore, we have already established that the activity-selection problem exhibits optimal substructure.
- Let $S_k = \{a_1 \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes.
- If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.

Making the greedy choice

- One big question remains: is our intuition correct? Is the greedy choice—in which we choose the first activity to finish—always part of some optimal solution?
- Please consider the following theorem:

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Making the greedy choice

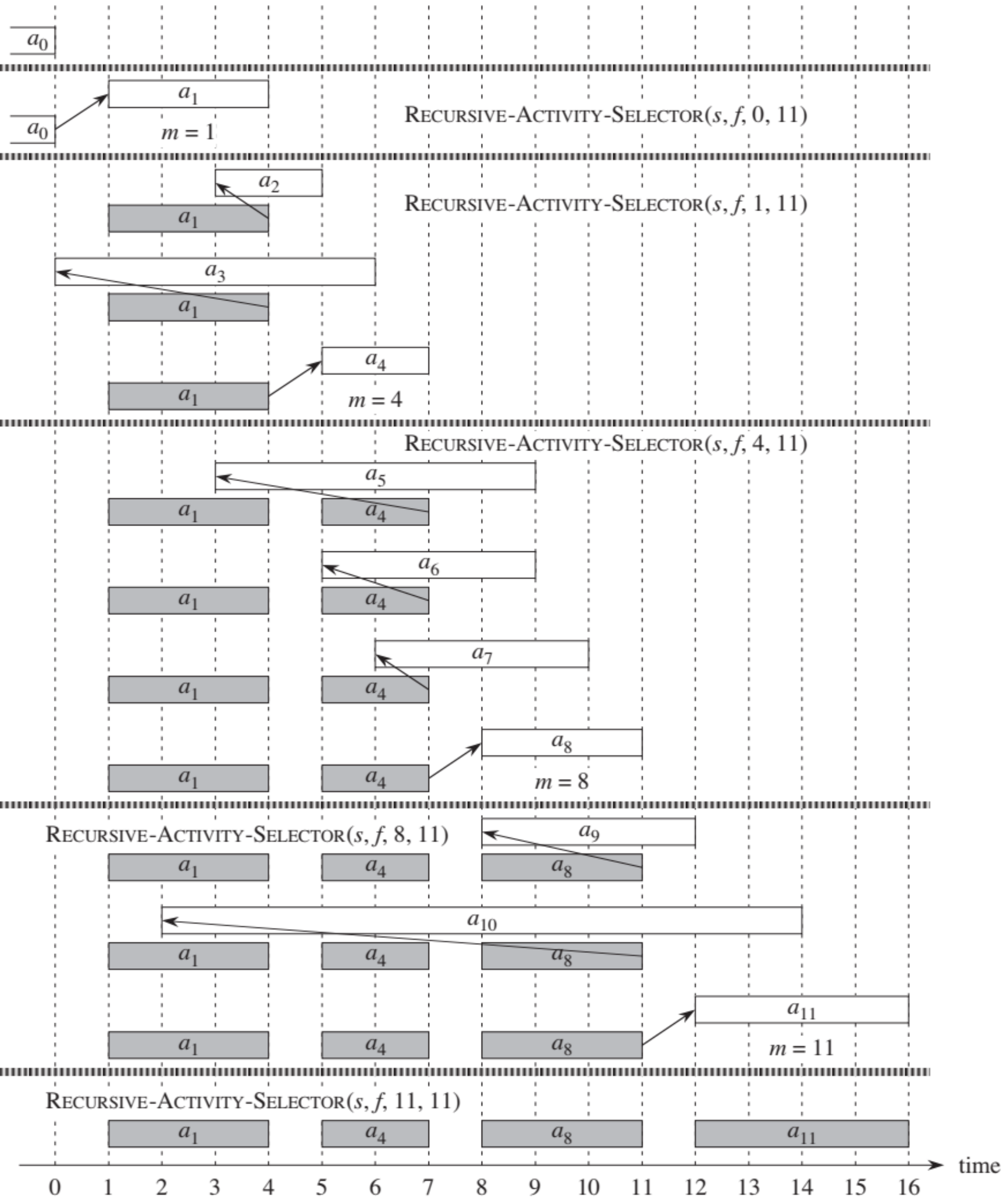
- An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm.
- Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.
- Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

- The procedure **RECURSIVEACTIVITY-SELECTOR** takes the start and finish times of the activities, represented as arrays s and f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem.
- It returns a maximum-size set of mutually compatible activities in S_k.
- We assume that the n input activities are already ordered by *monotonically increasing finish time*. If not, we can sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily.

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

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



An iterative greedy algorithm

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

Time complexity

- Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

6 steps

- 1. Determine the optimal substructure of the problem.
- 2. Develop a recursive solution. (Formulate the recurrence)
- 3. Show that if we make the greedy choice, then only one subproblem remains.
- 4. Prove that it is always safe to make the greedy choice.
- 5. Develop a recursive algorithm that implements the greedy strategy.
- 6. Convert the recursive algorithm to an iterative algorithm.

Exercise

- ให้นักเรียนเขียนโปรแกรมเพื่อแก้ปัญหาการเลือกทำกิจกรรม (Activity Selection Problem) โดยใช้ Greedy method.
- ให้ลองทดสอบกับ Inputs ต่อไปนี้

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

i	1	2	3	4	5	6
s_i	5	1	3	0	5	8
f_i	9	2	4	6	7	9

Exercise

- ให้นักเรียนเขียนโปรแกรมเพื่อแก้ปัญหาตำรวจจับโจร
  - กำหนดให้ Array มาให้เพื่อแสดงตำแหน่งโจร (T) และตำแหน่งของตำรวจ (P) เช่น
    - Positions [] = {'P', 'T', 'P', 'T', 'T', 'P’} และกำหนดค่าความสามารถของการจับโจรของตำรวจ (K) โดยค่า K คือระยะห่างที่ตำรวจจะจับโจรได้
  - **<u>โจทย์</u>**ให้นักเรียนเขียนโปรแกรมเพื่อช่วยหาว่าจะมีตำรวจกี่คนที่สามารถจับโจรได้
  - ตัวอย่าง [1] Input: {'P', 'T', 'P', 'T', 'T', ‘P’}, K = 3

    Output: 3

    [2] Input: {'P', 'T', 'T', 'P', ‘T’}, K=1

    Output: 1
  - Greedy Choice: ให้พิจารณาตำแหน่งของตำรวจและโจรแบบ Relative เรื่อยๆ หาก index ใคร น้อยกว่า ให้เพิ่ม index ของคนนั้น ในกรณีที่ตำรวจจับโจรได้ให้ขยับ index ของทั้งคู่

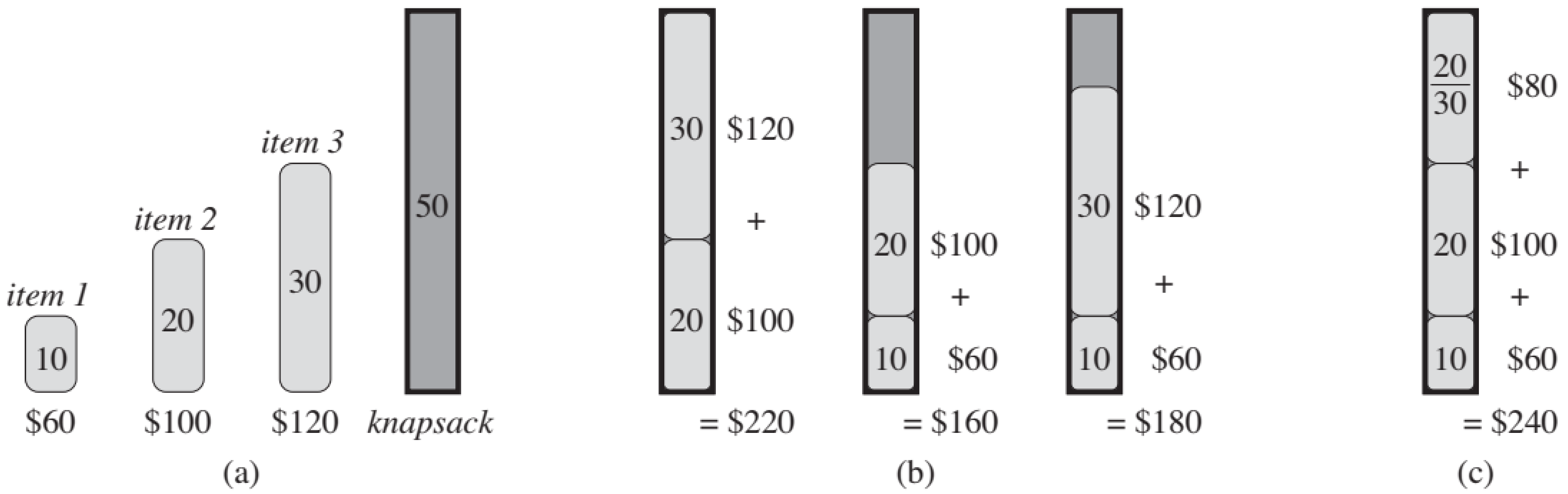
0-1 knapsack problem

0-1 knapsack problem

- A thief robbing a store finds n items.
- The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.
- The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W.

Which items should he take?

*In the **fractional knapsack** problem, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.*



An example showing that 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

- Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.
- Suppose we have a 100,000-character data file that we wish to store compactly.

	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

- Here, we consider the problem of designing a binary character code (or code for short) in which 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 data file of 100,000 characters):
  - a = 000, b = 001, ..., f=101.
  - This method requires 300,000 bits to code the entire file. Can we do better?
- A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
  - It takes 224, 000 bits to represent the file, a savings of approximately 25%.

	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

Prefix 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

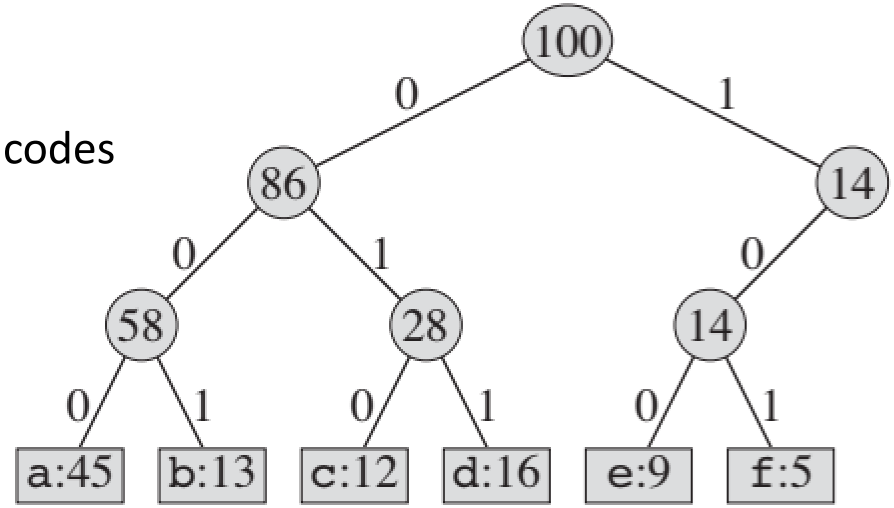
- **prefix codes**: codes in which no codeword is also a prefix of some other codeword.
- We code the 3-character file “**abc**” as 0•101•100, • denotes concatenation.
- Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.
- We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file.
- According to the table, the string **001011101** parses uniquely as 0•0•101•1101, which decodes to **aabe**.
- The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword.
  - A **binary tree** whose leaves are the given characters provides one such representation.

Prefix codes

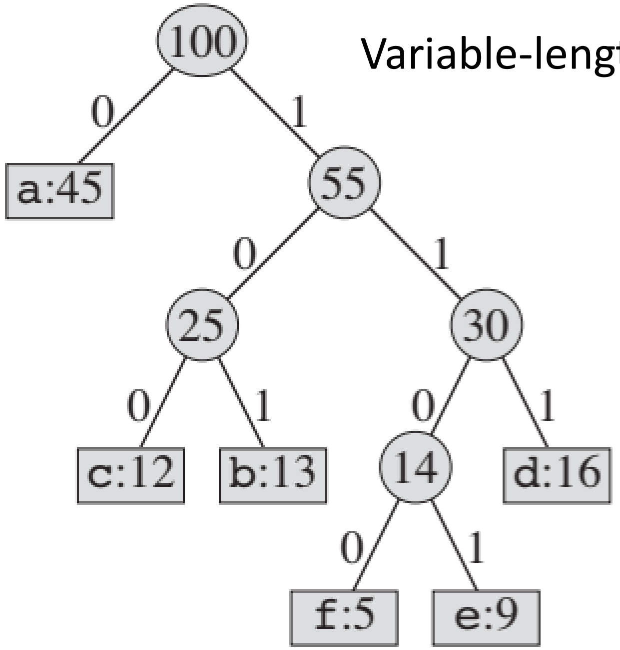
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- We interpret the 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."
- An **optimal code** for a file is always represented by a full binary tree, in which every nonleaf node has two children.

Fixed-length codes



Variable-length codes



Prefix 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

- We can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves.
  - One for each letter of the alphabet, and exactly $|C|-1$ internal nodes.
- Given a tree T corresponding to a prefix code, we can easily compute the *number of bits* required to encode a file.
- For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c.
- The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

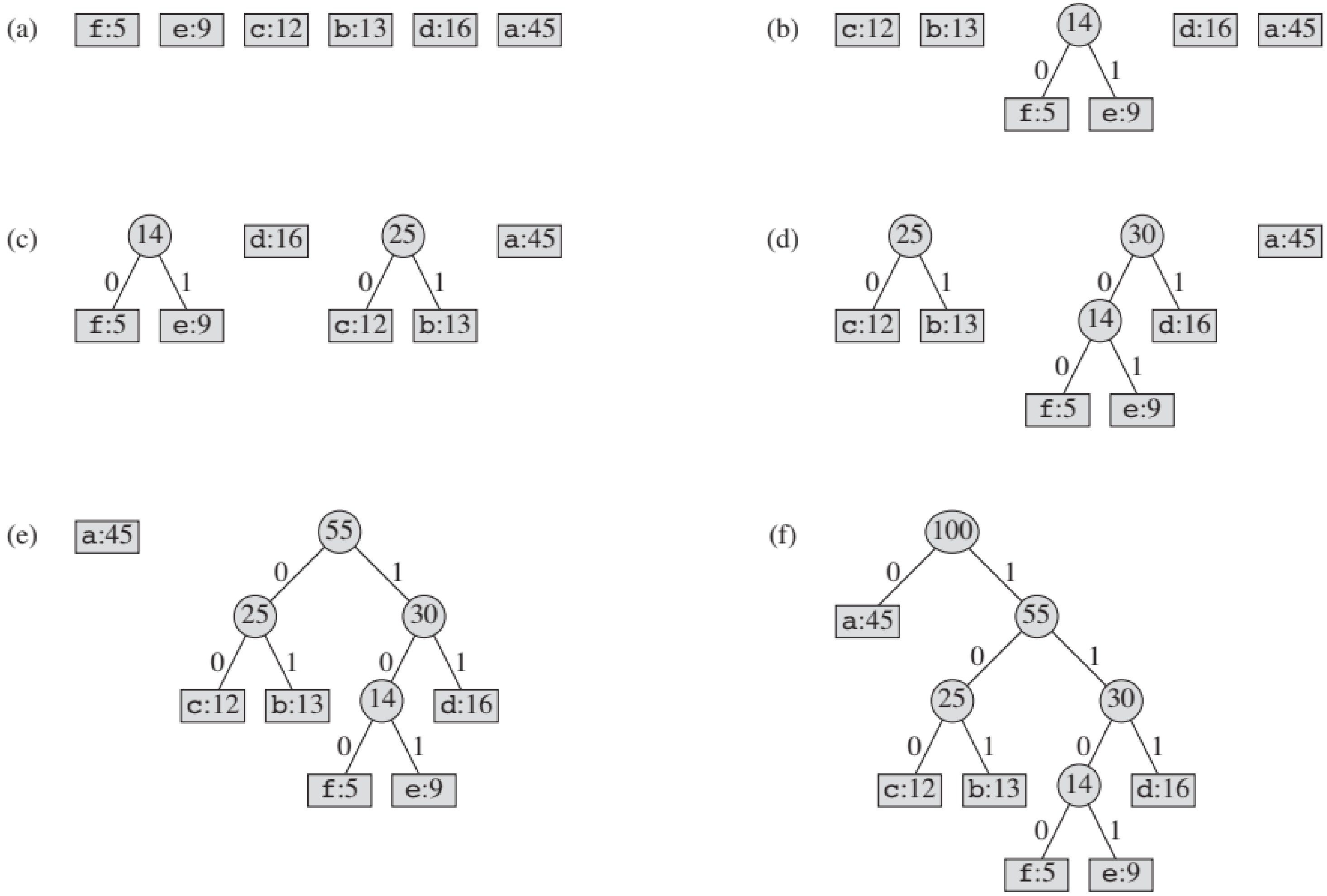
which we define as the cost of the tree T.

Constructing a Huffman code

- Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code.
- C is a set of n characters.

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Constructing a Huffman code



Complexity Analysis

- Q is implemented as a binary min-heap. For a set of C of n characters.
- Initializing Q takes $O(n)$
- In the loop starting at Line 3, it executes n-1 times, and each time requires $O(\lg n)$, so the loop contributes $O(n \lg n)$ to the running time.
- Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Exercise

- ให้นักเรียนเขียนโปรแกรมเพื่อนสร้าง Huffman code ของ Characters ต่อไปนี้

	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

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.