Greedy Algorithm

Greedy Algorithm

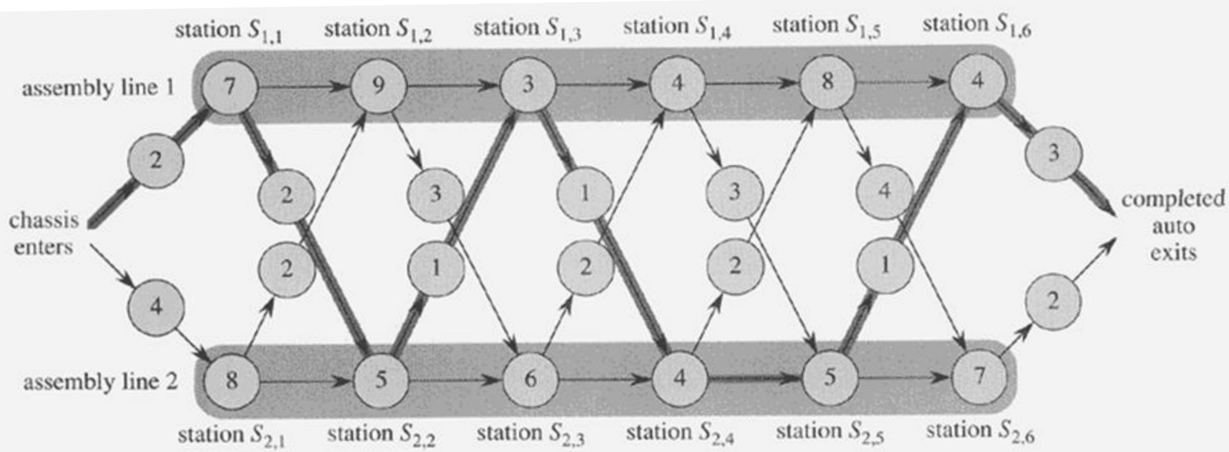
- Strategy
  - making the choice that looks best at the moment
- Simpler solution to optimization problems
  - 최적화 문제에 대한 단순하고 효과적인 solution 제공
  - Local optimal solution
  - Woks well for wide range of problems

(cf) dynamic programming

- Global optimal solution
- Works only for stage-state problems

Greedy vs. DP

(ex) Assembly scheduling problem



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

DP:

Greedy:

An Activity-Selection Problem

- Problem

Given:

Set of activities $S = \{ a_1, a_2, \dots , a_n \}$

s_i = ***start time*** of activity a_i

fi = ***finish time*** of activity a_i (단 $0 \le s_i < f_i < \infty$)

a_i and a_j are ***compatible*** ⇔ $[s_i , f_i]$ and $[s_j , f_j]$ do not overlap

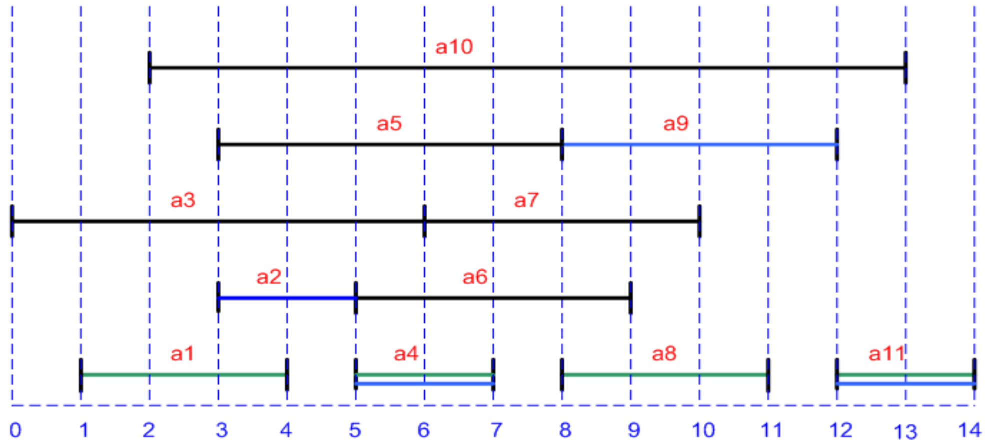
Find:

Set of compatible activities with maximum elements: A

An Activity-Selection Problem

(Example) 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



- Mutually compatible activities

  $\{a_3, a_9, a_{11}\}$, $\{a_1, a_6, a_{11}\}$, …… , $\{a_1, a_4, a_8, a_{11}\}$, $\{a_2, a_4, a_9, a_{11}\}$

An Activity-Selection Problem

- Greedy Solution
  - 현 시점의 compatible activities 중 최소의 finish time 을 갖는 activity 선택
  - Assumption

    activites already sorted by monotonically increasing finish time.
    (If not, then sort in $O(n \lg n)$ time.)
  - Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
  n ← length[s]
  A ← {a_1}
  i ← 1
  for m ← 2 to n
         do if s_m ≥ f_i
                 then A ← A ∪ {a_m}
                      i ← m
  return A
```

  - Running time: $\Theta(n)$

Knapsack Problem

- Definition
  - Given
    - Items 1, 2, …., n
    - i-th item: v_i dollars, w_i pounds
    - Max. pounds of knapsack: W
  - Store items to knapsack to maximize dollars
- Greedy strategy
  - Select item which has greatest value per pound

(ex) W = 50

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4

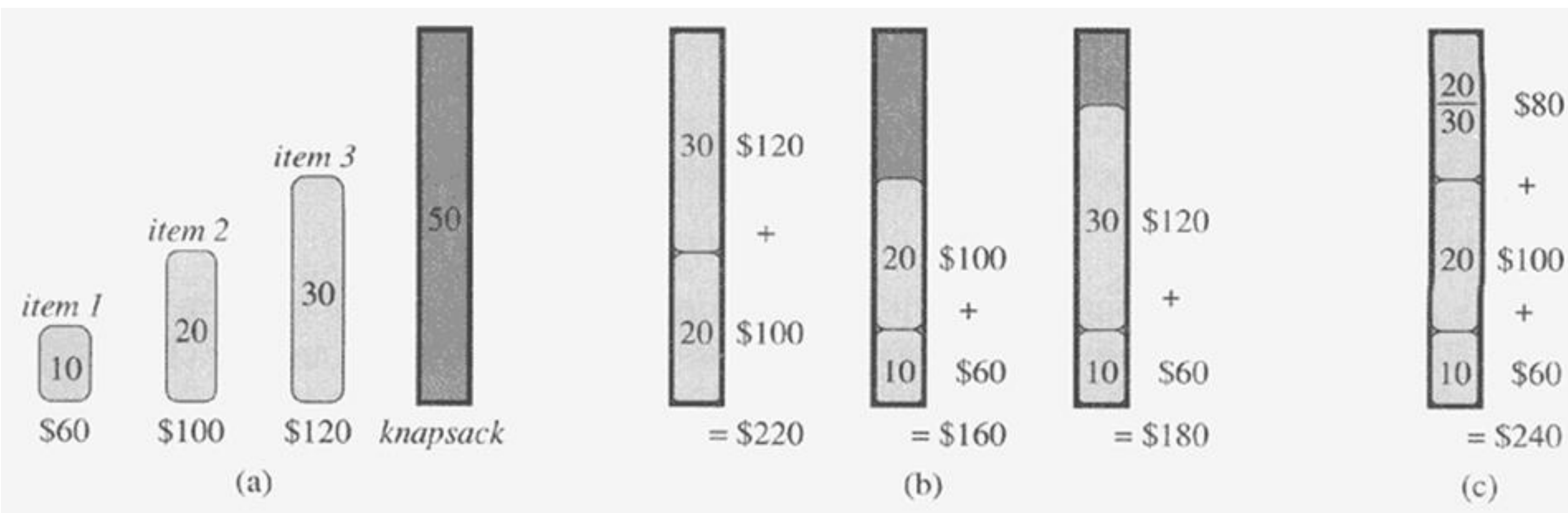
Knapsack Problem

(ex) 0-1 knapsack problem

- Allows only binary (0, 1) choice for each item
- Not solvable by greedy strategy

(ex) Fractional knapsack problem

- Allows fractional choice for each item
- Solvable by greedy strategy



0-1 knapsack   fractional knapsack

Huffman Codes

- Binary character code
  - fixed-length code
  - variable-length code

  (ex)

	a	b	c	d	e	f
Frequency(x1000)	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
Variable-length code	0	101	100	111	1101	1100

fixed-length code : 3bits * 100,000 characters = 300,000 bits

variable length code: (45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4)*1,000 = 224,000 bits

⇒ 25% savings

Huffman Codes

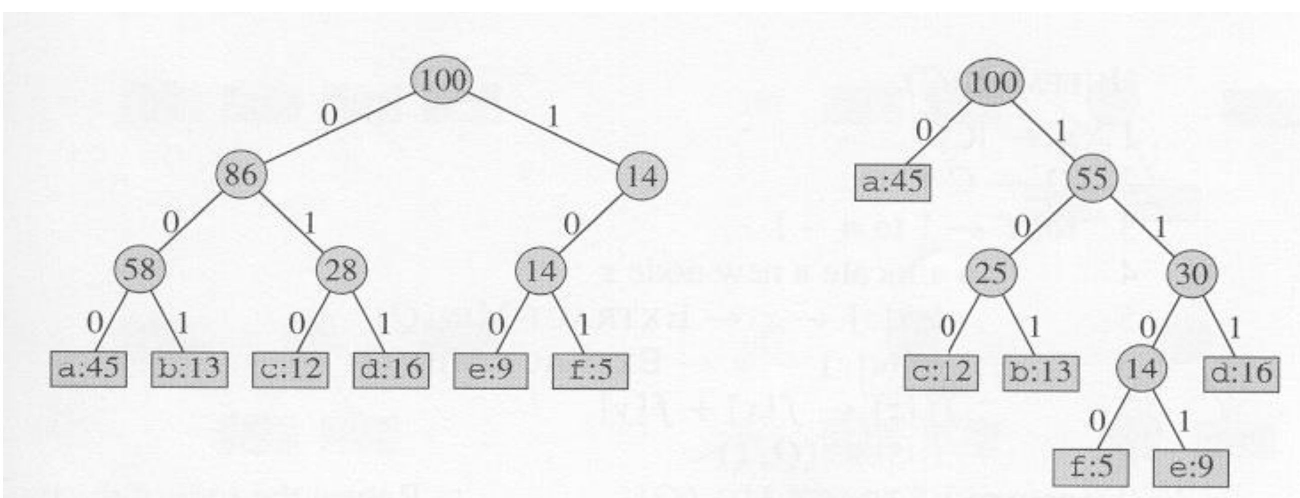
- Prefix Code
  - encoding 및 decoding 시 별도의 구두점(concatenation) 불필요

(ex)

encoding: abc => 0 101 100 (unique)

decoding: 001011101 = 0 0 101 1101 = aabe (unique !)

  - Optimal prefix code 는 ***full-binary tree*** 로 표현될 수 있다



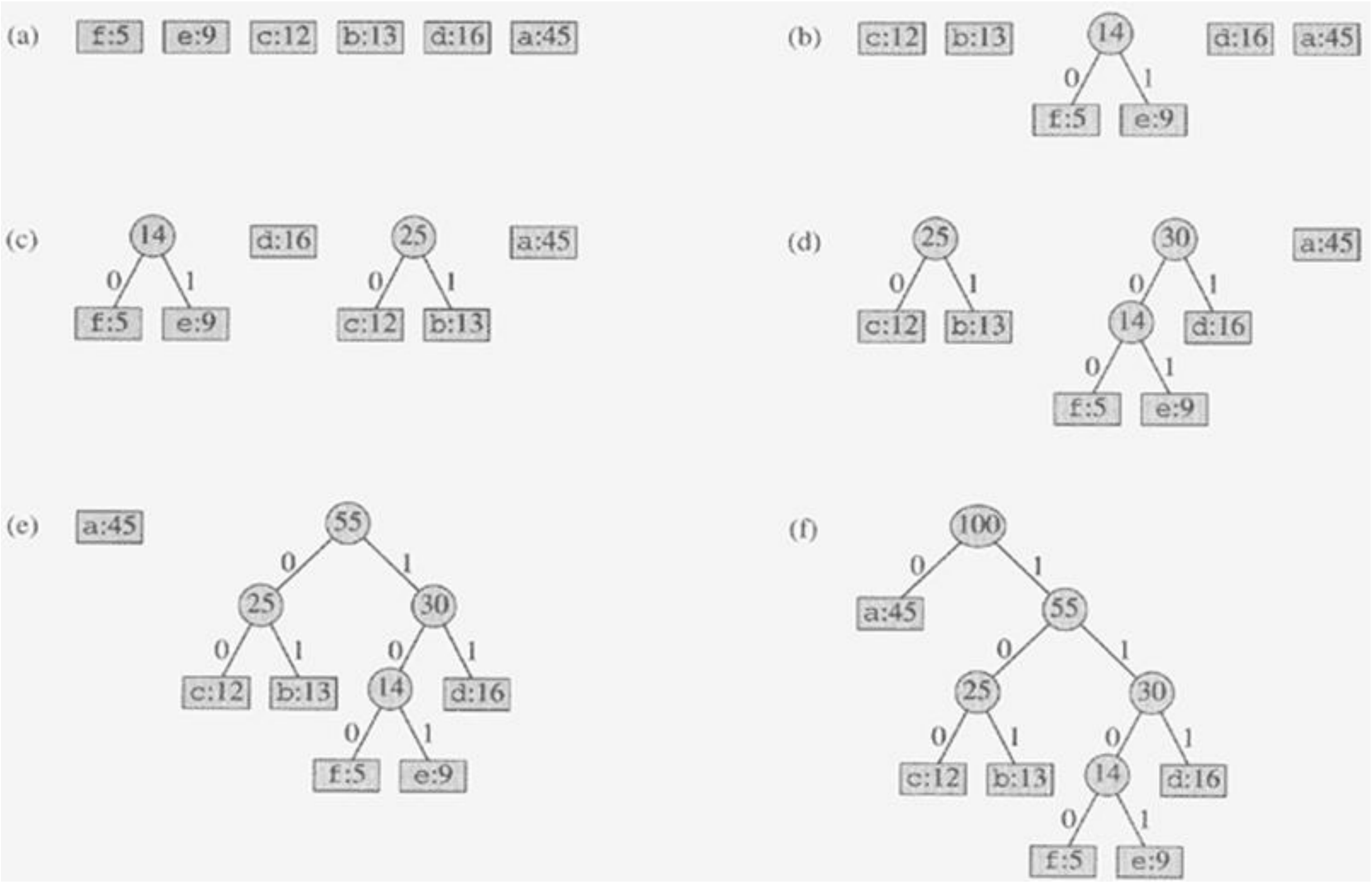
Huffman Codes

- Huffman Algorithm
  - Optimal prefix code 생성
  - Min-priority queue 사용
  - Greedy approach
- HUFFMAN(C)
  - C : set of characters
  - Q: min. priority queue
  - Binary tree: left-child, right-child representation
  - Running time: O (n lg n)

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do allocate a new node z
       left[z] ← x ← EXTRACT-MIN(Q)
       right[z] ← y ← EXTRACT-MIN(Q)
       f[z] ← f[x] + f[y]
       INSERT(Q, z)
return EXTRACT-MIN(Q)                ▷ Return the root of the tree.
```

Huffman Codes

- Operations



Huffman Codes

(Q)