Algorithms
Chapter 16
Greedy Algorithms

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **An activity-selection problem**
- Elements of the greedy strategy
- Huffman codes

Greedy Algorithms

- Similar to dynamic programming.
- Used for optimization problems.
- **Idea:** When we have a choice to make, make the one that looks best right now.
  - Make a locally optimal choice in hope of getting a globally optimal solution.
- Greedy algorithms don't always yield an optimal solution. But sometimes they do.
  - We'll see problems for which they do.
  - Also, we'll look at some general characteristics of when greedy algorithms give optimal solutions.

An activity-selection problem

- **Input:** A set $A = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities.
  - Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$.
- **Output:** A maximum set of compatible activities.
  - Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- For example: Consider the following set *A,* 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

- $\{a_3, a_9, a_{11}\}$ is a set of compatible activities.
- $\{a_1, a_4, a_8, a_{11}\}$ is a maximum set of compatible activities.

Greedy templates

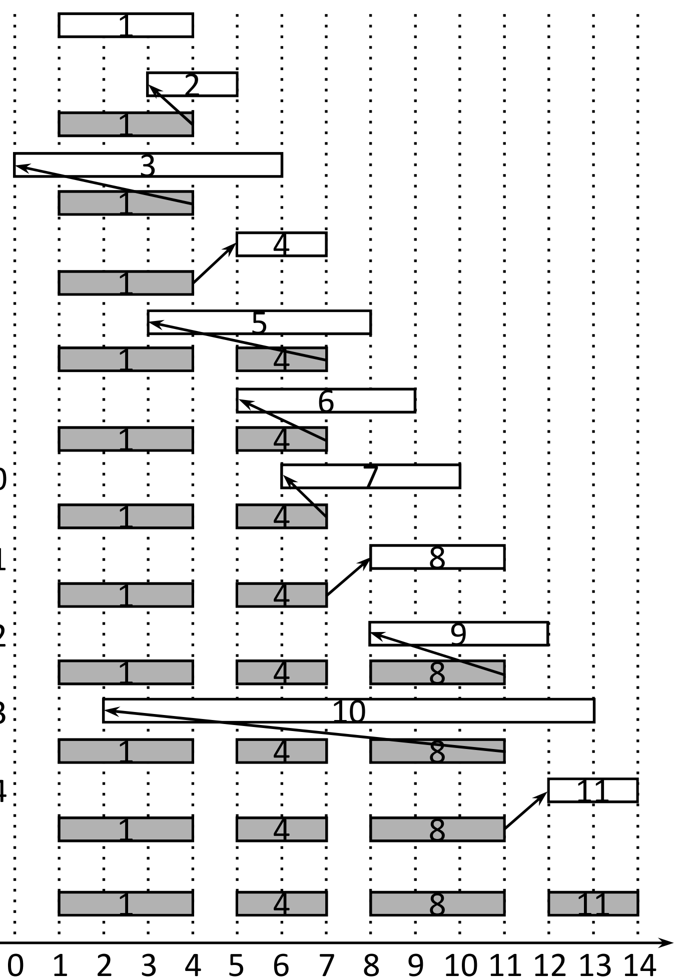
- **Earliest start time:**
  - Consider jobs in ascending order of s_i.
- **Earliest finish time:**
  - Consider jobs in ascending order of f_i.
- **Shortest interval:**
  - Consider jobs in ascending order of $f_i - s_i$.

GREEDY-ACTIVITY-SELECTOR pseudocode

```
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
```

- s: array of start times.
- f: array of finish times.
- The input is sorted by f_i.
- Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

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

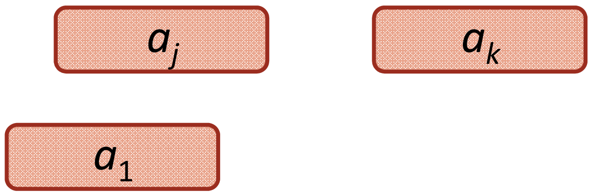


Correctness$_{1/3}$

- **Lemma 1** There exists an optimal activity selection contains a_1.

proof.

- Consider an optimal activity selection S.
- If $a_1 \in S$, then S is the desired selection.
- Otherwise, let a_j be the activity in S with the smallest finish time.
- Every $a_k \in S - \{a_j\}$ has $s_k \geq f_j$.
- So, $S - \{a_j\} \cup \{a_1\}$ is also a set of compatible activities.
- Thus, $S - \{a_j\} \cup \{a_1\}$ is an optimal selection.



Correctness$_{2/3}$

- **Theorem 2** Algorithm GREEDY-ACTIVITY-SELECTOR produces solutions of maximum size for the activity-selection problem.
- **proof.**
  - Induction on the number of $|A|$.
  - S = an activity selection by our algorithm.
  - T = an optimal activity selection of A containing a_1.
- **The basis:**
  - $|A| = 1$.
  - Clearly, $S = T = A$.

Correctness$_{3/3}$

- **Induction step:**
  - Suppose our selection algorithm works for all sets of activities with less than $|A|$ activities (strong induction).
  - $A' = \{a_i \in S \mid s_i \geq f_1\}$.
  - S' = our algorithm's selection for A'.
  - By inductive hypothesis, S' is an optimal selection of A'.
  - By greedy method, $S = \{a_1\} \cup S'$.
  - Let $T' = T - \{a_1\}$. Then $T' \subseteq A'$.
  - Therefore, $|T'| \leq |S'|$ by optimality of S'.
  - Hence, $|T| = |T'| + 1 \leq |S'| + 1 = |S|$.
  - Thus, S is an optimal selection of A.

Outline

- An activity-selection problem
- **Elements of the greedy strategy**
- Huffman codes

Elements of the greedy strategy

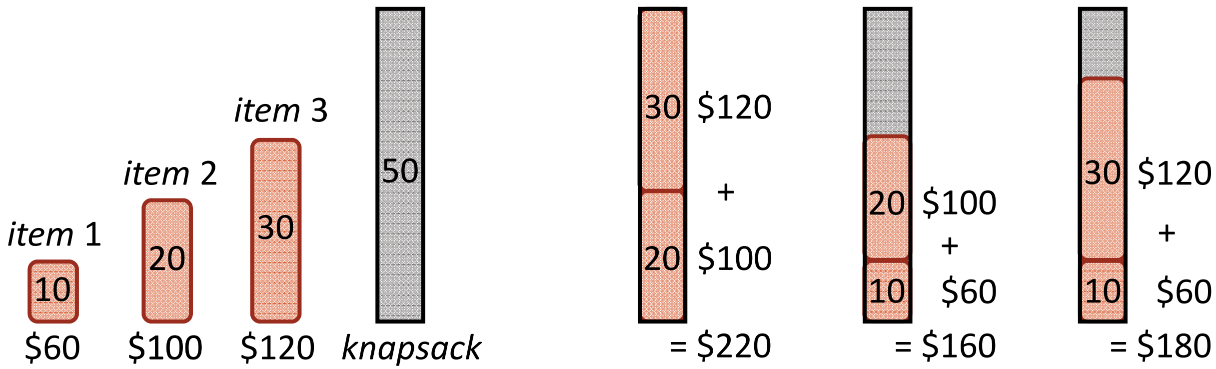
- **Greedy-choice property**
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - Typically show the greedy-choice property by what we did for activity selection.
    - Look at a globally optimal solution.
    - If it includes the greedy choice, done.
    - Else, modify it to include the greedy choice, yielding another solution that's just as good.
- **Optimal substructure**
  - An optimal solution to the problem contains within it optimal solutions to subproblems.

Greedy versus dynamic programming

- **Dynamic programming:**
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems.
  - Solve subproblems **first**.
  - Solve **bottom-up**.
- **Greedy:**
  - Make a choice at each step.
  - Make the choice **before** solving the subproblems.
  - Solve **top-down**.

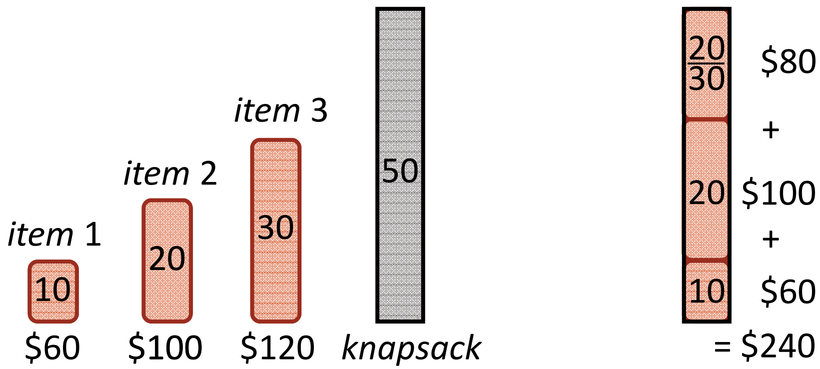
0-1 knapsack problem-- using DP

- **Input:** A set $A = \{a_1, a_2, \ldots, a_n\}$ of n items and a knapsack of capacity C.
  - Each item a_i is worth v_i dollars and weighs w_i pounds.
- **Output:** A subset of items whose total size is bounded by C and whose profit is maximized.
  - **Each item must either be taken or left behind.**
- For example:



Fractional knapsack problem-- using greedy

- **Input:** A set $A = \{a_1, a_2, \ldots, a_n\}$ of n items and a knapsack of capacity C.
  - Each item a_i is worth v_i dollars and weighs w_i pounds.
- **Output:** A subset of items whose total size is bounded by C and whose profit is maximized.
  - **The thief can take fractions of items.**
- For example:



FRACTIONAL-KNAPSACK pseudocode

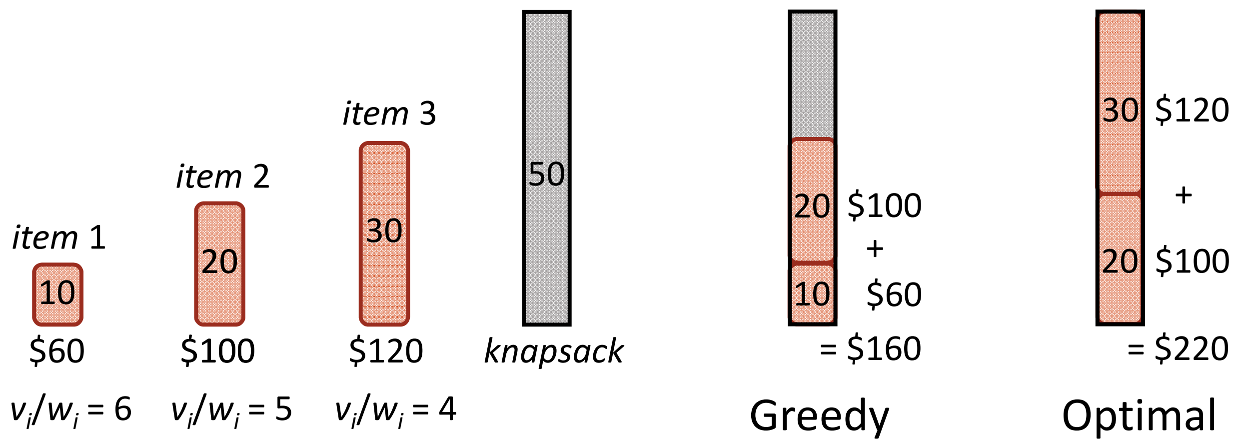
```
FRACTIONAL-KNAPSACK(v,w,C)
    load ← 0
    i ← 1
    while load < C and i ≤ n
        do if w_i ≤ C − load
            then take all of item i
            else take (C − load)/w_i of item i
        add what was taken to load
    i ← i + 1
```

- v: array of values.
- w: array of weights.
- C: capacity
- The input is sorted by v_i / w_i.

- Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Does greedy algorithm work for 0-1knapsack?

- Greedy doesn't work for the 0-1 knapsack problem.
- For example:



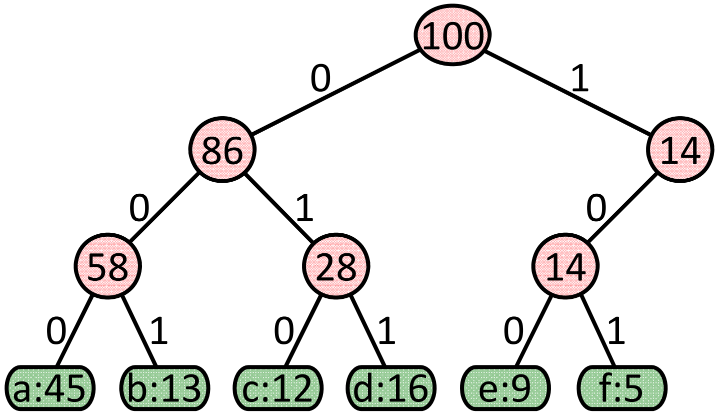
Outline

- An activity-selection problem
- Elements of the greedy strategy
- **Huffman codes**

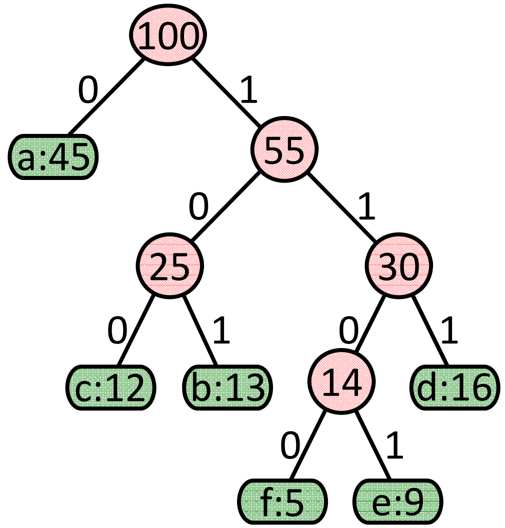
Huffman codes

- A very effective technique for compressing data.
- A **prefix code** in which no codeword is also a prefix of some other codeword.
- An optimal prefix binary code.
- **Huffman coding problem**
  - **Input:** A alphabet $C = \{c_1, c_2,\dots, c_n\}$ of n characters.
    - Each character c_i has a frequency f_i>0.
  - **Output:** A prefix binary code for C with minimum cost.
    - The code is represented by a full binary tree.
    - The leaves of the code tree represent the given characters.
    - $d_T(c)$ is the length of the codeword for character c.
    - The number of bits required to encode a file is $B(T) = \sum_{c \in C} f(c) d_T(c)$.

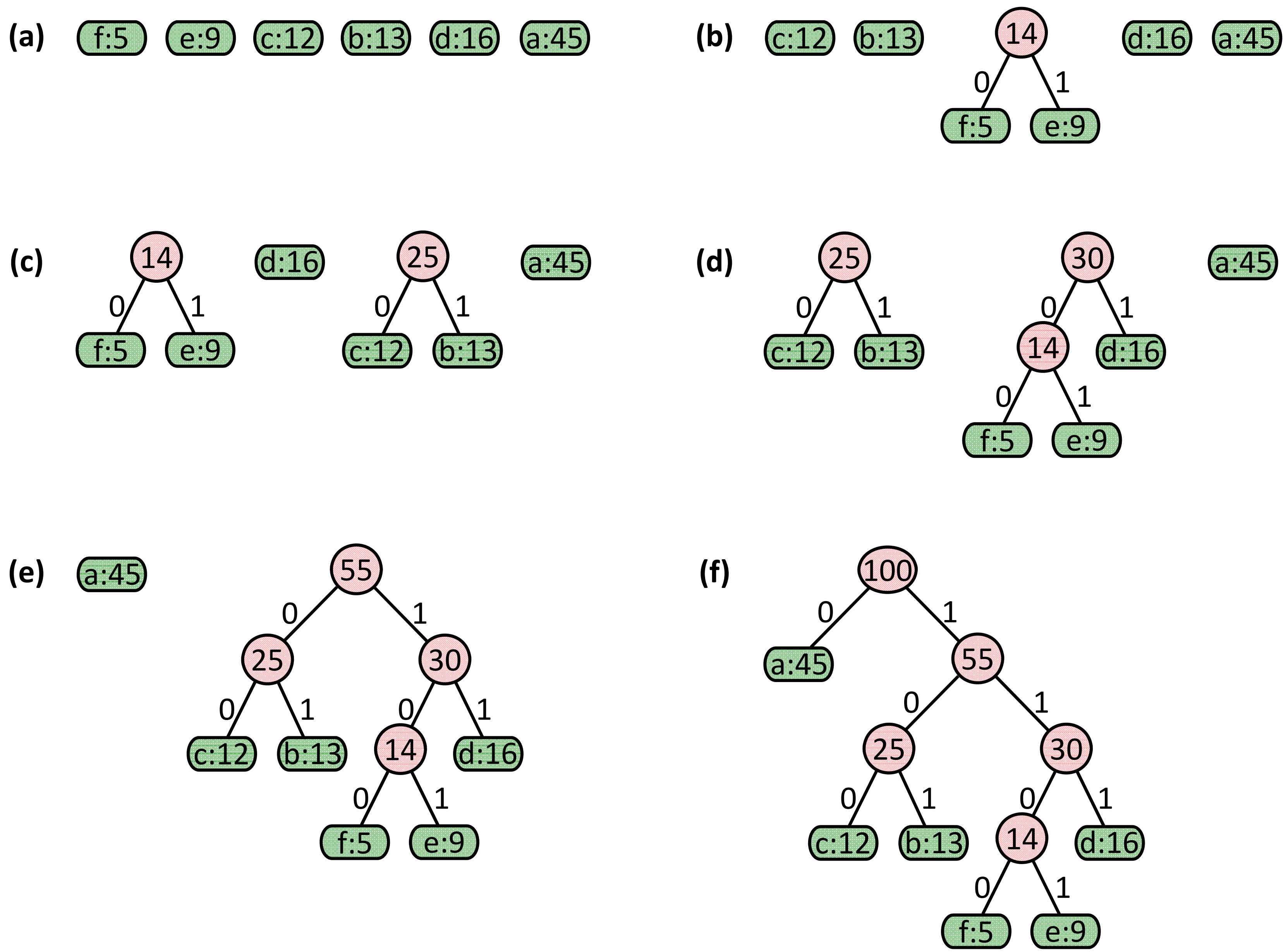
An example



The tree corresponding to the fixed-length code a = 000, ..., f = 101.
The code is not optimal



The tree corresponding to the optimal prefix code a = 0, b = 101, ..., f = 1100.


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9

HUFFMAN pseudocode

```
HUFFMAN(C)
1.    n ← |C|                                   } O(1)
2.    Q ← C                                     } O(n)
3.    for i ← 1 to n − 1
4.        do allocate a new node z
5.            left[z] ← x ← EXTRACT-MIN(Q)
6.            right[z] ← y ← EXTRACT-MIN(Q)         } (n−1) · O(lg n)
7.            f [z] ← f [x] + f [y]
8.            INSERT(Q, z)
9.    return EXTRACT-MIN(Q)   /* Return the root of the tree. */   } O(1)
```

- Line 2 initializes the min-priority queue Q with the characters in C.

- Time: $O(n \lg n)$.
- Correctness: omitted.