Chapter 16
Greedy Algorithms

Chi-Yeh Chen
陳奇業
成功大學資訊工程學系

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

國立成功大學 九十週年
90th Anniversary of NCKU

Introduction

- Similar to dynamic programming.

  Use 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 do not always yield optimal solutions, but for many problems they do.*

Activity selection problem

NCKU
National Cheng Kung University
1931

藏行顯光
成就共好
Achieve Securely
Prosper Mutually
國立成功大學 九十週年
90th Anniversary of NCKU

Activity selection

- n activities require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

  Set of actives $S = \{a_1, \ldots, a_n\}$.

- a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i = start time and f_i = finish time.

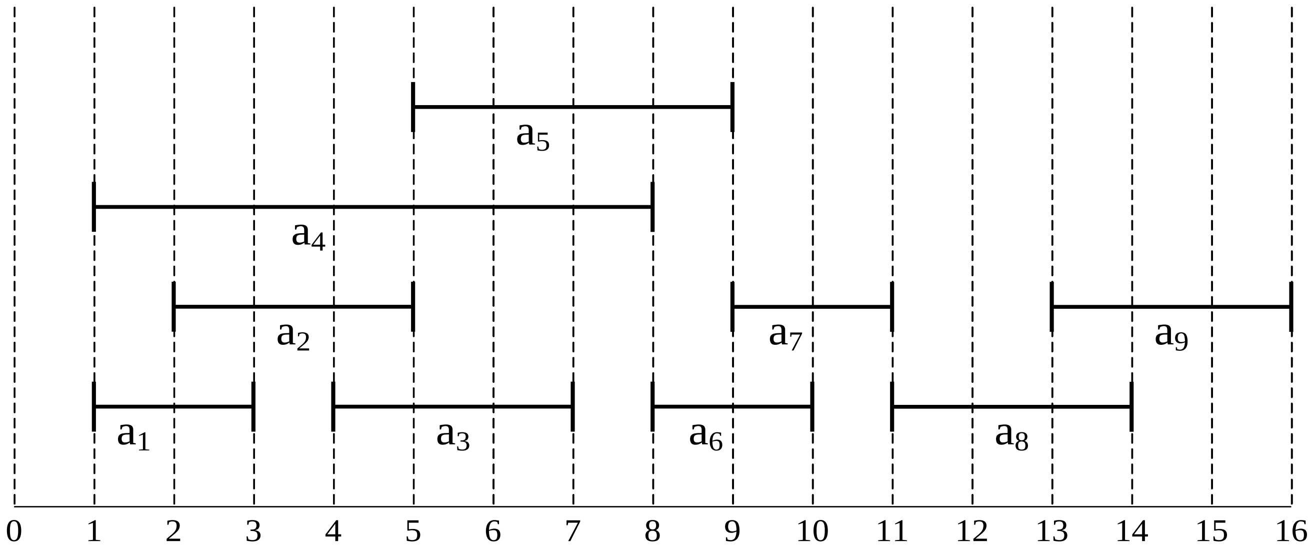
- ***Goal:*** Select the largest possible set of nonoverlapping (mutually compatible) activities

- Assume $a, b \in \mathbb{R}$ and $a < b$

$$(a, b) = \{x | a < x < b\}$$
$$[a, b] = \{x | a \le x \le b\}$$
$$[a, b) = \{x | a \le x < b\}$$
$$(a, b] = \{x | a < x \le b\}$$
$$(a, \infty) = \{x | x > a\}$$
$$[a, \infty) = \{x | x \ge a\}$$
$$(-\infty, b) = \{x | x < b\}$$
$$(-\infty, b] = \{x | x \le b\}$$
$$(-\infty, \infty) = \mathbb{R}$$
$$[a, a] = \{a\}$$

- ***Example*** ： S sorted by finish time:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



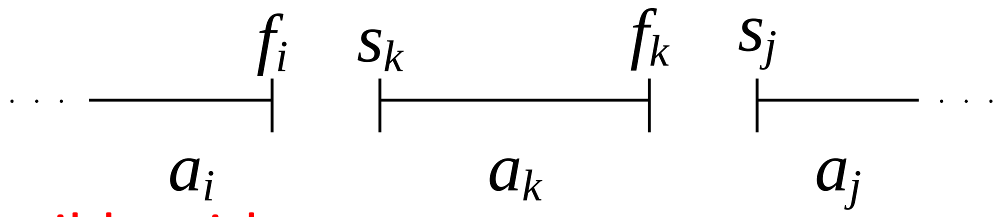
- Maximum-size mutually compatible set: $[a_{1,}a_3, a_6, a_8]$.

  Not unique: also $[a_{2,}a_5, a_7, a_9]$

Optimal substructure of activity selection

- $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$

  $=$ activities that start after a_i finishes and finish before a_j starts.



- Activities in S_{ij} are compatible with
  - All activities that finish by f_i , and
  - All activities that start no earlier than s_j.

To represent the entire problem, add fictitious activities:

- $a_0 = [-\infty, 0)$
- $a_{n+1} = [\infty, "\infty + 1")$

- We don't care about $-\infty$ in a_0 or "$\infty + 1$" in a_{n+1}.

  Then $S = S_{0,n+1}$

  Range for S_{ij} is $0 \le i, j \le n + 1$.

- Assume that activities are sorted by monotonically increasing finish time：

  $f_0 \le f_1 \le f_2 \le \ \dots \le f_n \le \ f_{n+1}$

  Then $i \ge j \Rightarrow S_{ij} = \emptyset$.

  - If there exist $a_k \in S_{ij}$:

    $f_i \le s_k < f_k \le s_j < \ f_j \Rightarrow f_i < \ f_j$.

  - But $i \ge j \Rightarrow f_i \ge \ f_j$, Contradiction.

  So only need to worry about S_{ij} with $0 \le i < j \le n + 1$.

  All other S_{ij} are $\emptyset$.

  Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:

  - S_{ik} (start after a_i finishes, finish before a_k starts)
  - S_{kj} (start after a_k finishes, finish before a_j starts)

- Let $A_{ij} =$ CĆÇÅǼAÆ ĊCÆDÇÅCB ÇC S_{ij}.

  So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{ki}$, assuming:
  - S_{ij} is nonempty, and
  - We know $a_{\boldsymbol{k}}$

Recursive solution to activity selection

$c[i,j] =$ ĊÅÈÃ CÄ ÆAĐÅÆDÆṁĊÅÈÃ ĊDÀĊÃÇ CÄ ÆDÇDAÆÆE ÁCÆĆAÇÅÀÆÃĊ ÅB S_{ij}.

- $i \geq j \Rightarrow$
  - If $S_{ij} = \emptyset \Rightarrow c[i,j] = 0$.
- $i < j \Rightarrow$

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

- ***Theorem***

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time :

$f_m = \min\{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. S_{im} = Ø , so that choosing a_m leaves S_{mi} as only nonempty subproblem.

Proof

1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij},

   Order activities in A_{ij} in monotonically increasing order of finish time.

   Let a_k be the first activity in A_{ij} .

   If $a_k = a_m$, done (a_m is used in a maximum-size subest).

   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m since $s_k \leq s_m \leq f_m \leq f_k$).

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

- ***Theorem***

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time :

$f_m = \min\{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.

2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mi} as only nonempty subproblem.

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$.

   Then $a_k \in S_{ij}$ and it has an earlier finish time than f_m , which

   contradicts our choice of a_m.

   Therefore, there is no $a_k \in S_{im} \Longrightarrow S_{im} = \emptyset$.

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

- ***Claim***

Activities in $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ are disjoint.

Proof

Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \leq f_k$

(so a_m doesn't overlap anything else in A'_{ij}). ◆(claim)

Since $|A'_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}. ◆(theorem)

This is great :

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	*j* – *i* – 1	1

- How we can solve top down:
- To solve a problem S_{ij}
  - Choose $a_m \in S_{ij}$ with earliest finish time: ***the greedy choice***
  - Then solve S_{mj}
- What are the subproblems?
  - Original problem is $S_{0,n+1}$
  - Suppose our first choice is a_{m1}
  - Then next subproblem is $S_{m1,n+1}$
  - Suppose next choice is a_{m2}
  - Nextsuproblem is $S_{m2,n+1}$
  - And so on

- ***Easy recursive algorithm:***

  Assumes activities already sorted by monotonically increasing finish time.

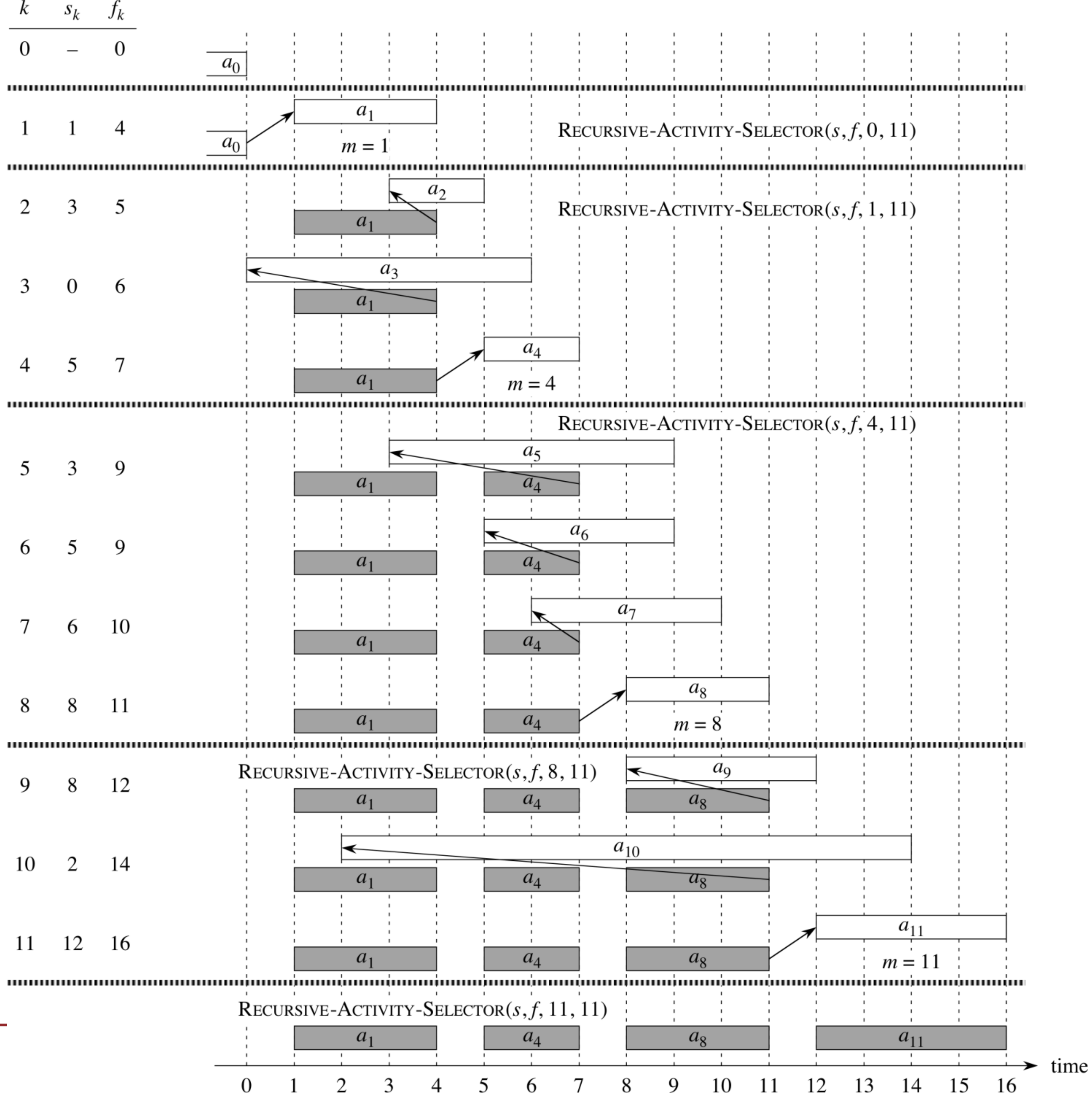
  (If not, then sort in $O(n \lg n)$ time)

  Return an optimal solution for $S_{i,n+1}$:

```
REC-ACTIVITY-SELECTOR(s, f, i, n)

m ← i + 1
while m ≤ n and s_m < f_i do
      ▶ Find first activity in S_{i,n+1}
      m ← m + 1
if m ≤ n then
      return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
else
      return
```

- ***Initial call:*** REC-ACTIVITY-SELECTOR(s, f, 0, n)
- ***Time:*** $\Theta(n)$ — each activity examined exactly once.



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
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
m = 1
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
m = 4
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
m = 8
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time

Can make this iterative. It’s already almost tail recursive.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
A ← {a_1}
i ← 1
for m ← 2 to n do
      if s_m ≥ f_i then
            A ← A ∪ {a_m}
            i ← m                    ▶ a_i is most recent addition to A
return A
```

Time: $\Theta(n)$.

Elements of the greedy strategy

NCKU
National Cheng Kung University
1931

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

- Greedy Strategy (typical streamline steps):
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there's always an optimal solution that make the greedy choice, so that the greedy choice is always safe. (greedy-choice property)
  3. Show that greedy choice and optimal solution to subprblem $\Rightarrow$ optimal solution to the problem. (optimal substructure)

- No general way to tell if a greedy algorithm is optimal, but two key ingredients are
  1. greedy-choice property and
  2. optimal substructure.

- **Greedy-choice property**

  A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. i.e. the greedy-choice is the optimal choice.

- **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*.

- **Optimal substructure**

  Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

- **Greedy vs. dynamic programming**

  The knapsack problem is a good example of the difference.

- ***0-1 knapsack problem***
  - n items.
  - Item i is worth v_i, weighs w_i pounds.
  - Find a most valuable subset of items with total weight $\leq W$.
  - Have to either take an item or not take it — can't take part of it.

- ***Fractional knapsack problem***
  - Like the 0-1 knapsack problem, but can take fraction of an item.
  - Both have optimal substructure.
  - But the fractional knapsack problem has the greedy-choice property, and 0-1 knapsack problem does not.
  - To solve the fractional problem, rank items by value/weight: $v_i \,/\, w_i$.

Let $v_i \,/\, w_i \geq v_{i+1} \,/\, w_{i+1}$ for all i.

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

FRACTIONAL-KNAPSACK(v, w, W)

```
load ← 0
i ← 1
while load < W and i ≤ n do
    if w_i ≤ W − load then
        take all of item i
    else
        take (W − load)/w_i of item i
    add what was taken to load
    i ← i + 1
```

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

Greedy don't work for 0-1 knapsack problem

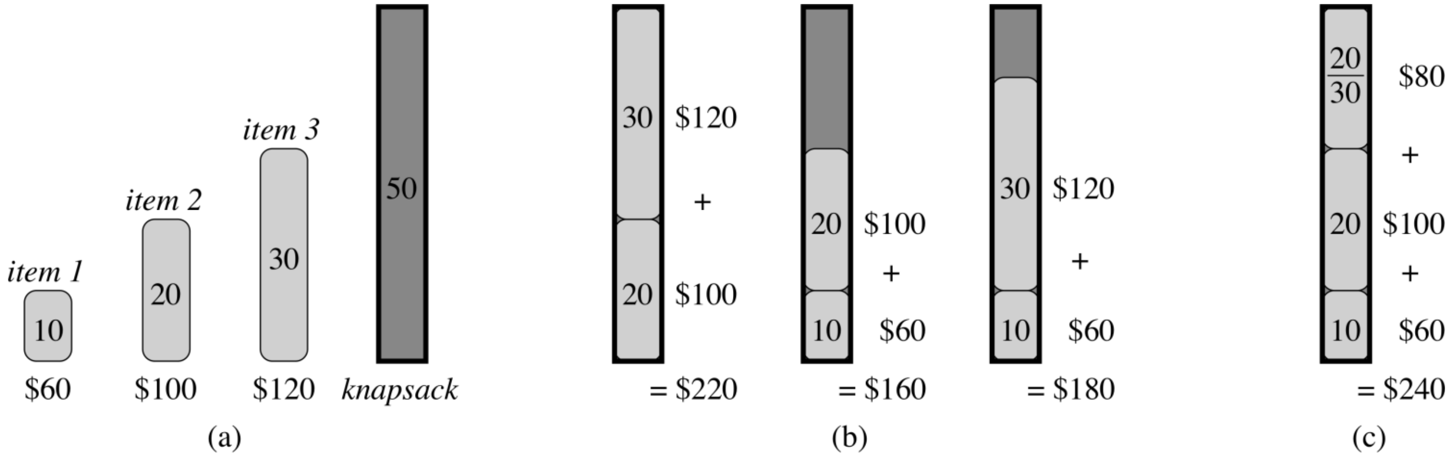
$W = 50$

- Greedy solution :
  - Take items 1 and 2.
  - value = 160, weight = 30.

i	1	2	3
v_i / w_i	6	5	4

Have 20 pounds of capacity left over.

- Optimal solution :
  - Take items 2 and 3.
  - value = 220, weight = 50.
  - No leftover capacity.



Huffman codes

NCKU
National Cheng Kung University

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

Huffman codes

	a	b	c	d	e	f
Frequency (in hundred)	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 code: no codeword is also a prefix of some other codeword.

- Can be shown that the optimal data compression achievable by a character code can always be achieved with prefix codes.

- Simple encoding and decoding.

- An optimal code for a file is always represented by a binary tree.

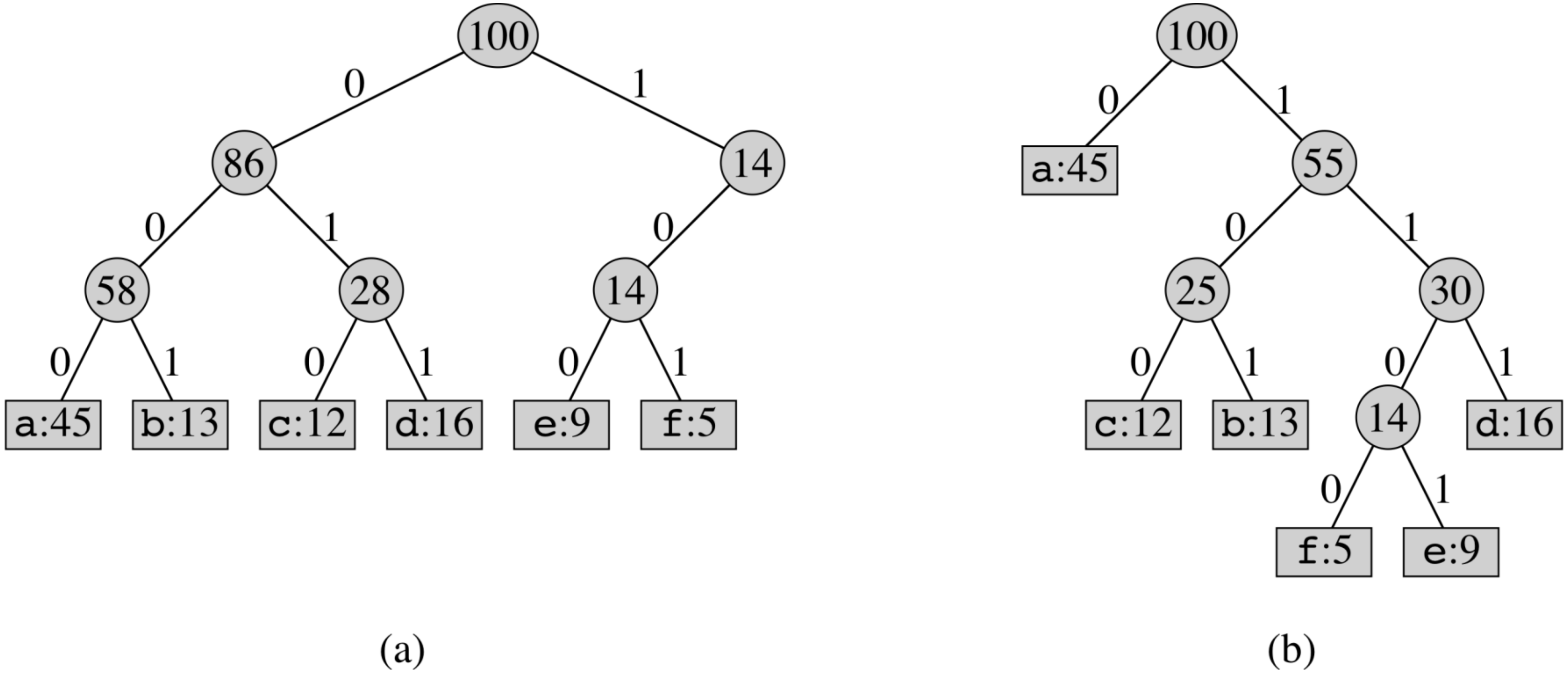
Constructing a Huffman code

HUFFMAN(C)

$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ ***to*** $n-1$ **do**
    allocate a new node z
    $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)
    $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)
    $f[z] \leftarrow f[x] + f[y]$
    INSERT(Q, Z)
return EXTRACT-MIN(Q)

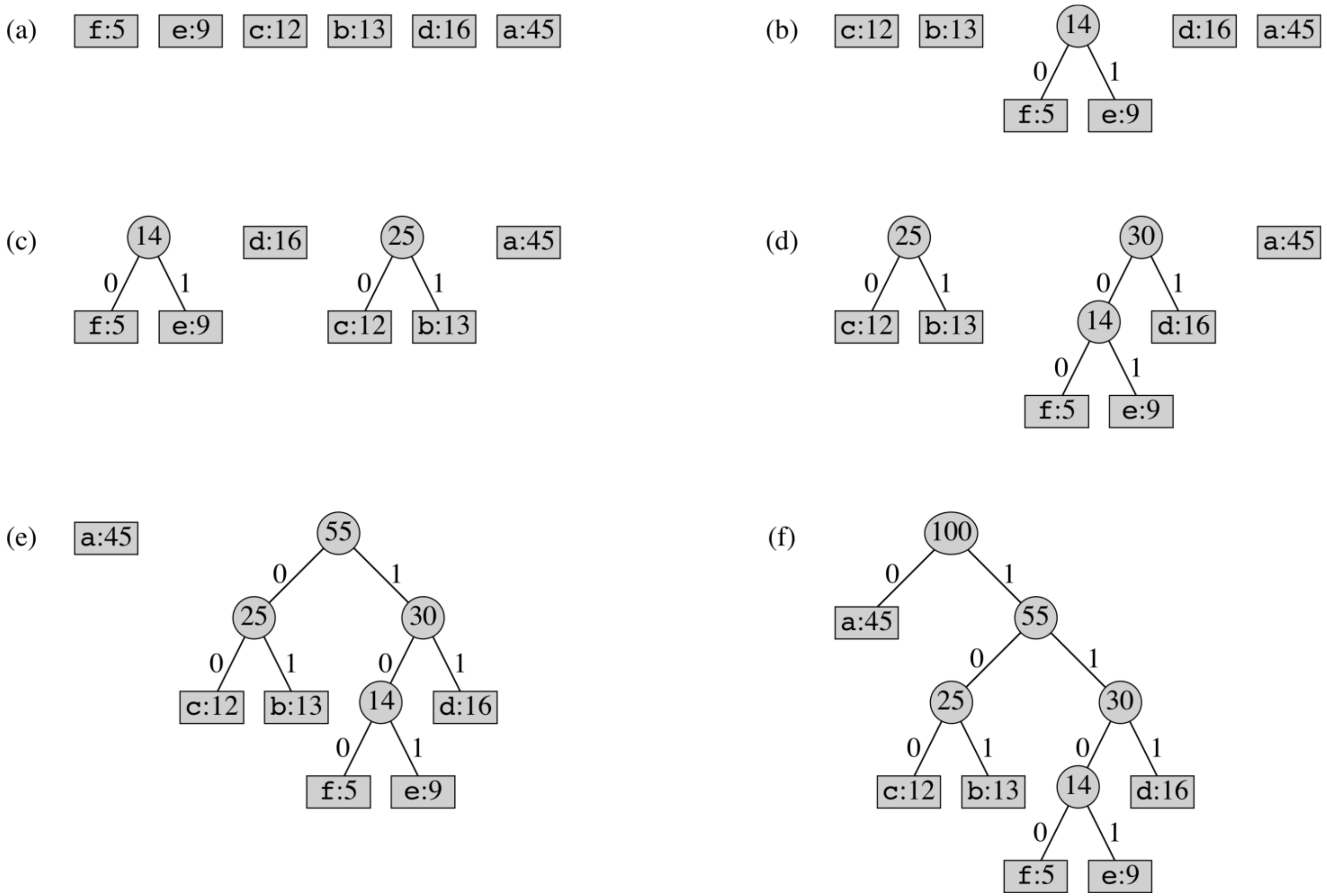
Complexity: $O(n \lg n)$

Tree correspond to the coding schemes



$B(T) = \sum_{c \in C} f(c)\, d_T(c)$ which we define as the cost of tree T

The steps of Huffman’s algorithm



Correction of Huffman’ s algorithm

The next lemma shows that the greedy-choice property holds. (The greedy-choice is the optimal choice.)

Lemma 16.2.

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be the two characters in C having the lowest frequencies. Then there exists an optimal prefix code in C in which the codeword for x and y having the same length and differ only in the last bit.

Proof.

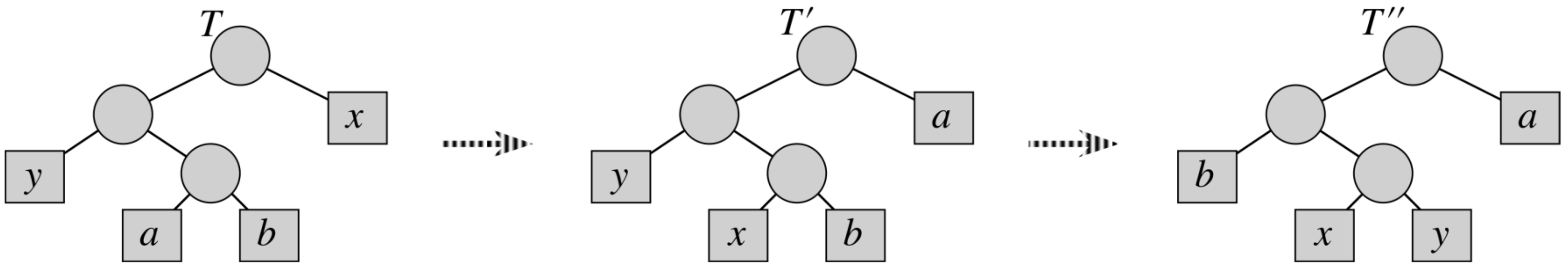
The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T. Without loss of generality, we assume that $f[a] \leq f[b]$ and $f[x] \leq f[y]$. Since $f[x]$ and $f[y]$ are the two lowest leaf frequencies, in order, and $f[a]$ and $f[b]$ are two arbitrary frequencies, in order, we have $f[x] \leq f[a]$ and $f[y] \leq f[b]$.

In the remainder of the proof, it is possible that we could have $f[x] = f[a]$ or $f[y] = f[b]$. However, if we had $f[x] = f[b]$, then we would also have $f[a] = f[b] = f[x] = f[y]$, and the lemma would be trivially true. Thus, we will assume that $f[x] \neq f[b]$, which means that $x \neq b$.

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

We exchange the positions in T and a and x to produce a tree T', and then we exchange the positions in T' of b and y to produce a tree T'' in which x and y are sibling leaves of maximum depth.



The difference in cost between T and T' is

$$\begin{aligned} B(T) - B(T') &= \textstyle\sum_{c\in C} f[c]\cdot d_T(c) - \sum_{c\in C} f[c]\cdot d_{T'}(c) \\ &= f[x]\cdot d_T(x) + f[a]\cdot d_T(a) - f[x]\cdot d_{T'}(x) - f[a]\cdot d_{T'}(a) \\ &= f[x]\cdot d_T(x) + f[a]\cdot d_T(a) - f[x]\cdot d_T(a) - f[a]\cdot d_T(x) \\ &= (f[a] - f[x])\big(d_T(a) - d_T(x)\big) \\ &\geq 0. \end{aligned}$$

because both $f[a] - f[x]$ and $d_T(a) - d_T(x)$ are nonnegative. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

藏行顯光 成就共好 Achieve Securely Prosper Mutually

Correction of Huffman's algorithm

The next lemma shows that the problem of construction optimal prefix codes has the optimal-substructure property.

(Just show that optimal solution to subproblem and greedy choice ⇒ optimal solution to problem.)

Lemma 16.3.

Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x and y removed and character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $f[z] = f[x]+f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof.

We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T'. For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $f[c] \cdot d_T(c) = f[c] \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f[x] \cdot d_T(x) + f[y] \cdot d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z] \cdot d_{T'}(z) + (f[x] + f[y]).$$

from which we conclude that

$$B(T) = B(T') + f[x] + f[y]$$

or, equivalently,

$$B(T') = B(T) - f[x] - f[y].$$

$$B(T) = \sum_{c \in C} f[c] \cdot d_T(c) = \sum_{c \in C-\{x,y\}} f[c] \cdot d_T(c) + f[x] \cdot d_T(x) + f[y] \cdot d_T(y)$$

$$= \sum_{c \in C-\{x,y\}} f[c] \cdot d_T(c) + f[z] \cdot d_{T'}(z) + (f[x] + f[y])$$

$$= \sum_{c \in C-\{x,y\}} f[c] \cdot d_{T'}(c) + f[z] \cdot d_{T'}(z) + (f[x] + f[y])$$

$$= \sum_{c \in C'} f[c] \cdot d_{T'}(c) + f[x] + f[y]$$

$$= B(T') + f[x] + f[y]$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality, T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $f[z] = f[x]+f[y]$. Then

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C.

Theorem 16.4

Theorem 16.4.

Procedure HUFFMAN produces an optimal prefix code.

藏行顯光
成就共好

Achieve Securely
Prosper Mutually

國立成功大學 九十週年
90th Anniversary of NCKU

1931
國立成功大學
National Cheng Kung University