16 Greedy Algorithms

- **Similar to dynamic programming. Used for optimization problems.**
- **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.**
- **Greedy Algorithm: Simpler, more efficient**

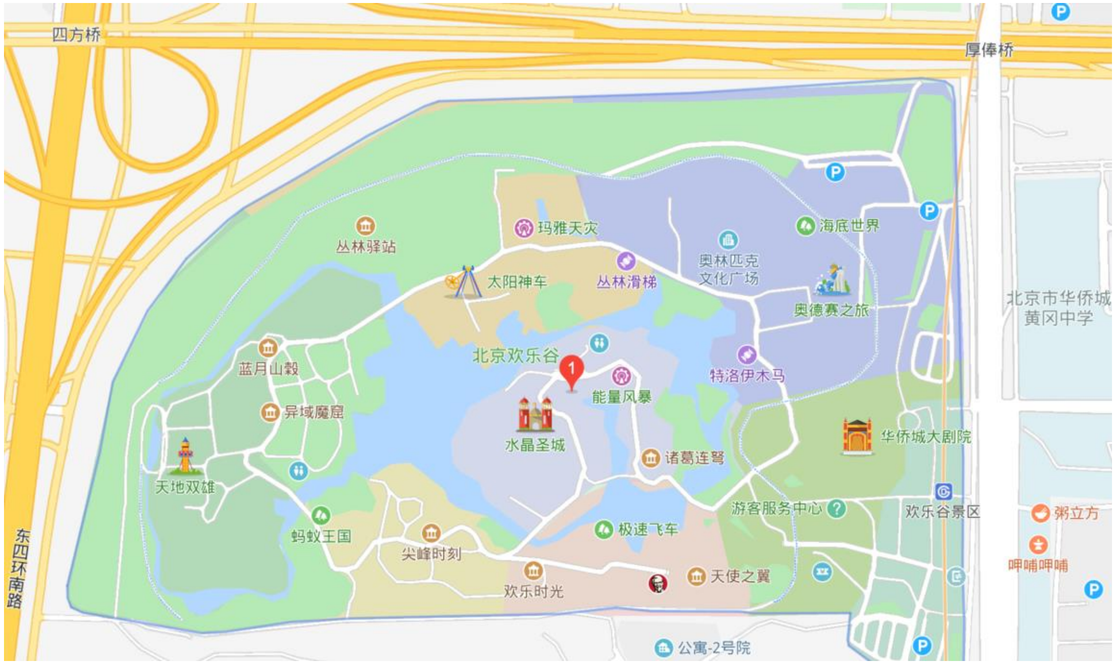
16 Greedy Algorithms

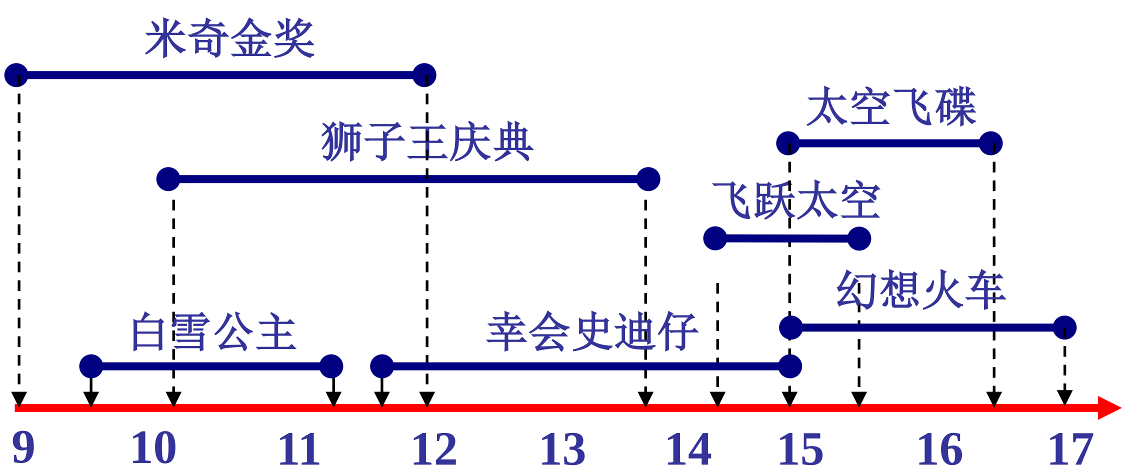
- **Greedy algorithms (GA) do not always yield optimal solutions, but for many problems they do.**
  - **16.1, the activity-selection problem （活动安排）**
  - **16.2, basic elements of the GA; knapsack prob.（贪婪算法的基本特征；背包问题）**
  - **16.3, an important application: the design of data compression (Huffman) codes. （哈夫曼编码）**
  - ***16.4 Matroids and greedy methods**
  - ***16.5, A task-scheduling problem as matroid (unit-time tasks scheduling, 有限期作业调度)**

16 Greedy Algorithms

- **The greedy method is quite powerful and works well for a wide range of problems:**
  - **minimum-spanning-tree algorithms (Chap 23) （最小生成树）**
  - **shortest paths from a single source (Chap 24) （最短路径）**
  - **set-covering heuristic (Chap 35). （集合覆盖）**
  - **…**

Example: 北京欢乐谷游玩的活动安排

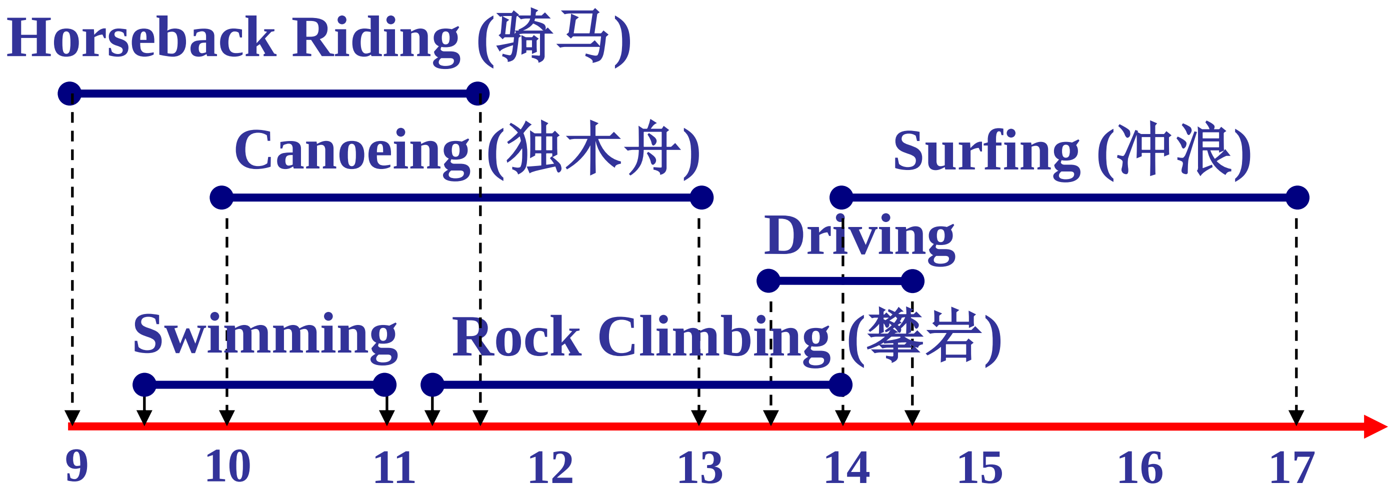




Activity Selection

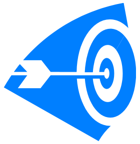
- 欢乐谷
- Disneyland

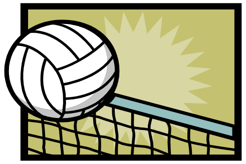
Example: Activity Selection



- How to make an arrangement to have the more activities?
  - S1. Shortest activity first （最短活动优先原则）
    Swimming , Driving
  - S2. First starting activity first （最早开始活动优先原则）
    Horseback Riding , Driving
  - S3. First finishing activity first （最早结束活动优先原则）
    Swimming , Rock Climbing , Surfing

16.1 An activity-selection problem



应用场景：

借体育馆、借会议室

n activities **require**

exclusive **use of a common resource.**

Example, scheduling the use of a classroom.

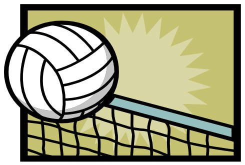
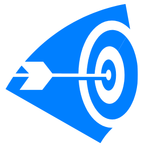
（n 个活动，1 项资源，任一活动进行时需唯一占用该资源）

- Set of activities $S = \{a_1, a_2, \ldots, a_n\}$.
- a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i is start time and f_i is finish time.
- *Goal:* Select the largest possible set of nonoverlapping (*mutually compatible*) activities. （安排一个活动计划，使得相容的活动数目最多）
- Other objectives: Maximize income rental fees，…

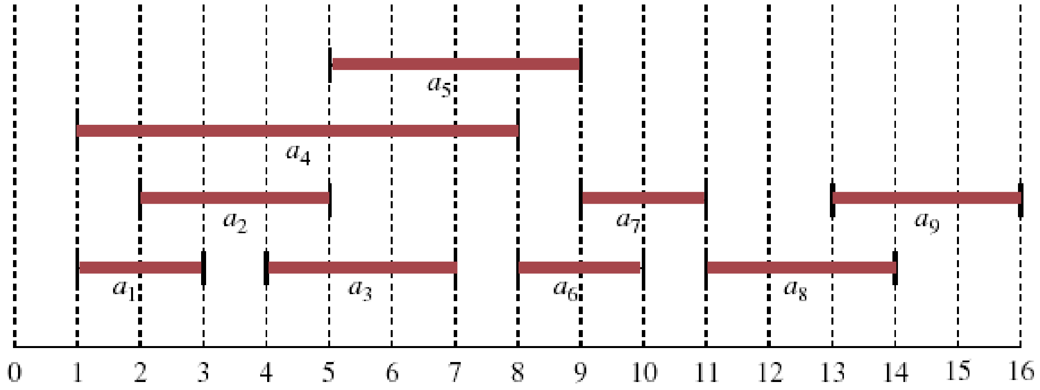
16.1 An activity-selection problem

- ***n activities*** **require *exclusive* use of a common resource.**
  - **Set of activities $S = \{a_1, a_2, \ldots, a_n\}$**
  - **a_i needs resource during period $[s_i, f_i)$**
- ***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:

$\{a_1, a_3, a_6, a_8\}$.

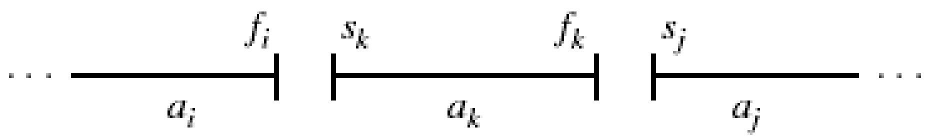
Not unique: also

$\{a_2, a_5, a_7, a_9\}$.

16.1.1 Optimal substructure of activity selection

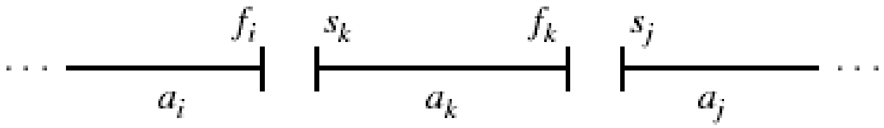
Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
  = activities that start after a_i finishes & finish before a_j starts



- **Activities in S_{ij} are compatible with**
  - **all activities that finish by f_i（完成时间早于f_i 的活动）, and**
  - **all activities that start no earlier than s_j .**
- **To represent the entire problem, add fictitious activities:**
  - $a_0 = [-\infty, 0)$; $a_{n+1} = [\infty, \text{“}\infty+1\text{”})$
  - **We don’t care about $-\infty$ in a_0 or “∞+1” in a_{n+1}.**
- **Then $S = S_{0,n+1}$. Range for S_{ij} is $0 \leq i, j \leq n + 1$.**

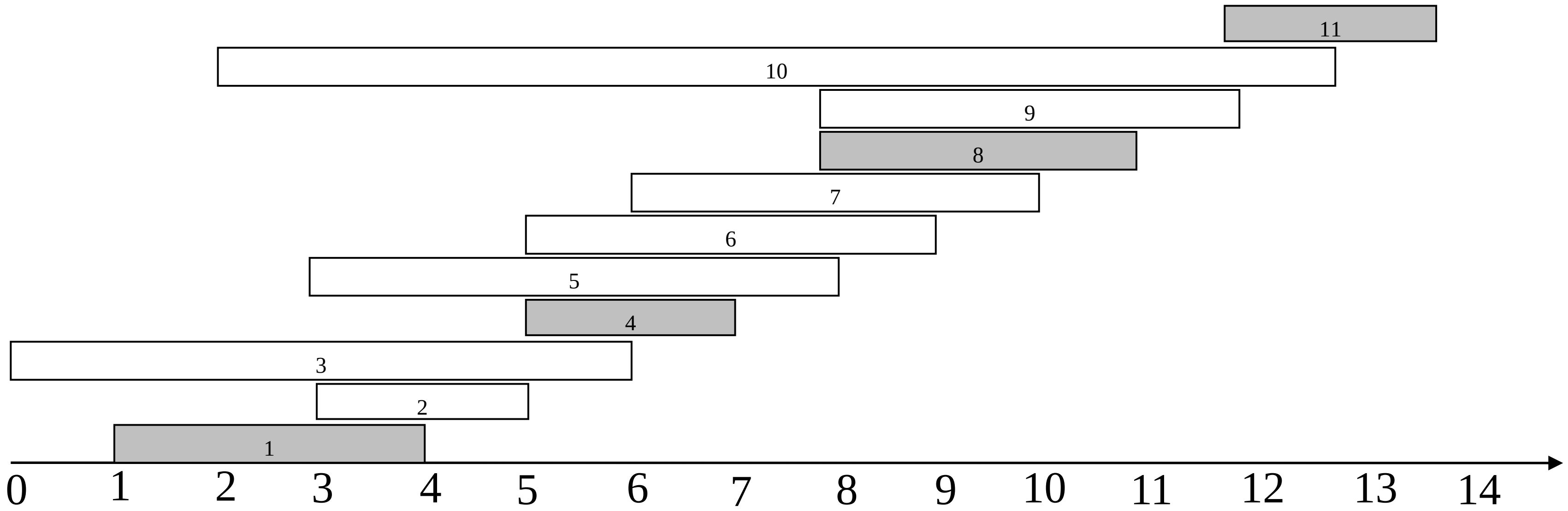
16.1.1 Optimal substructure of activity selection



Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
- Assume that activities are sorted by monotonically increasing finish time （以结束时间单调增的方式对活动进行排序）

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1} \quad (\text{if } i \le j, \text{ then } f_i \le f_j) \qquad (16.1)$$



16.1.1 Optimal substructure of activity selection

if $f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$ **(if $i \le j$, then $f_i \le f_j$)** **(16.1)**

- **Then** $i \ge j \Rightarrow S_{ij} = \varnothing$

  Proof **If there exists $a_k \in S_{ij}$, then**

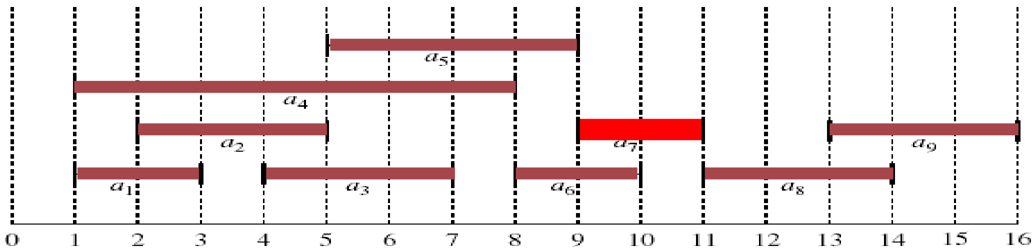
  $$f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j .$$

  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 $\varnothing$.

16.1.1 Optimal substructure of activity selection



- **Suppose that a solution to S_{ij} includes a_k . Have 2 sub-prob**
  - **S_{ik} (start after a_i finishes, finish before a_k starts)**
  - **S_{kj} (start after a_k finishes, finish before a_j starts)**
- **Solution to S_{ij} = (solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (solution to S_{kj})**

  Since a_k is in neither of the subproblems, and the subproblems are disjoint, | solution to S | = | solution to S_{ik} | + 1 + | solution to S_{kj} | .
- **Optimal substructure: If an optimal solution to S_{ij} includes a_k , then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. (use usual cut-and-paste argument).**
- **Let A_{ij} = optimal solution to S_{ij} ,**

  so $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, (16.2)

  assuming: S_{ij} is nonempty; and we know a_k .

16.1.2 A recursive solution

- **Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij} .** （$c[i, j]$ 表示 S_{ij} 相容的最大活动数）

  $i \geq j \Rightarrow S_{ij} = \varnothing \Rightarrow c[i, j] = 0.$
- **If $S_{ij} \neq \varnothing$, suppose that a_k is used in a maximum-size subsets of mutually S_{ij} . Then** $c[i, j] = c[i, k] + 1 + c[k, j]$.
- **But of course we don't know which k to use, and so**

$$c[i, j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

Why this range of k? Because $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\} \Rightarrow a_k$ can't be a_i or a_j (if $k = i$, we have $c[i, j] = c[i, j] + 1$).

核心要素：1. 递归；2. 遍历n次，选k

16.1.3 Converting a DP solution to a greedy solution

$$c[i,j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

- **It may be easy to design an algorithm to the problem based on recurrence (16.3).**

  (1) Direct recursion algorithm? complexity?

  (2) Dynamic programming algorithm? complexity?

- **For (16.3):**
  - **How many choices?**
  - **How many subproblems for a choice?**

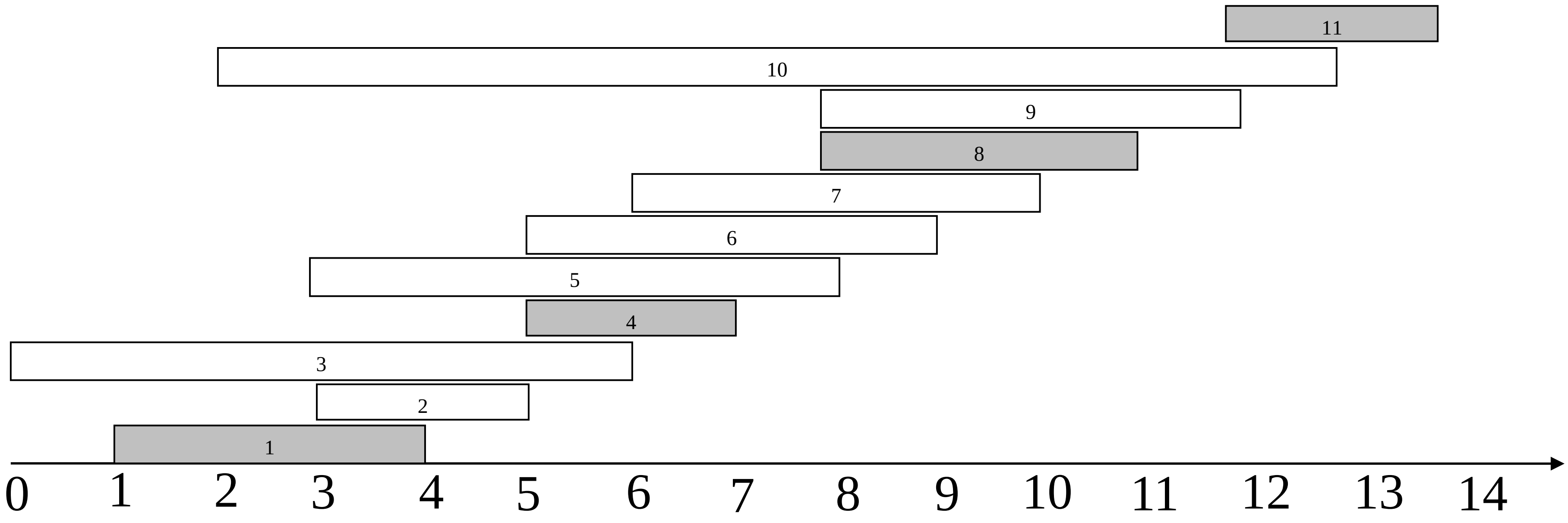
- **Can we simplify our solution?**

16.1.3 Converting a DP solution to a greedy solution

- **Theorem 16.1**

  Let $S_{ij} \neq \varnothing$, 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}.** （a_m包含在某个最大相容活动子集中）
  2. **$S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.** （仅剩下一个非空子问题 S_{mj}）



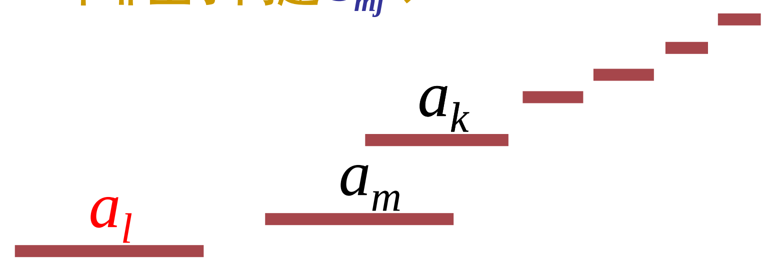
16.1.3 Converting a DP solution to a greedy solution

- **Theorem 16.1**

  Let $S_{ij} \neq \varnothing$, 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. ……

  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. （仅剩下一个非空子问题 S_{mj} ）



Proof

2. Suppose there is some $a_l \in S_{im}$. Then $f_i \leq s_l < f_l \leq s_m < f_m \Rightarrow f_l < f_m$. Then $a_l \in S_{ij}$ and it has an earlier finish time than f_m, which contradicts our choice of a_m. Therefore, there is no $a_l \in S_{im} \Rightarrow S_{im} = \varnothing$.

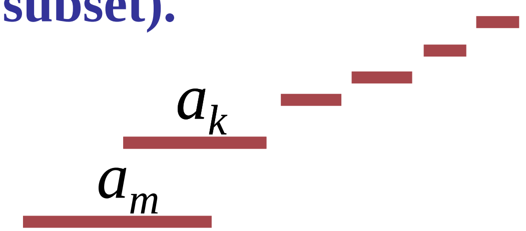
16.1.3 Converting a DP solution to a greedy solution

- **Theorem 16.1** Let $S_{ij} \neq \varnothing$, 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} .（a_m包含在某个最大相容活动子集中）

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 subset).



- Otherwise, construct $B_{ij}=A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m). Activities in B_{ij} are disjoint. (Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish. $f_m \leq f_k \Rightarrow a_m$ doesn't overlap anything else in B_{ij}). Since $|B_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is B_{ij} .

16.1.3 Converting a DP solution to a greedy solution

$$c[i,j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

- **Theorem 16.1** Let $S_{ij} \neq \varnothing$, 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}. （a_m包含在某个最大相容活动子集中）
  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. （仅剩下一个非空子问题 S_{mj}）

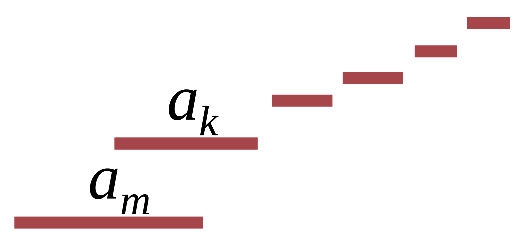
- **This theorem is great:**

	before theorem	after theorem
# of sub-prob in optimal solution	2	1
# of choices to consider	$O(j-i-1)$	1

16.1.3 Converting a DP solution to a greedy solution

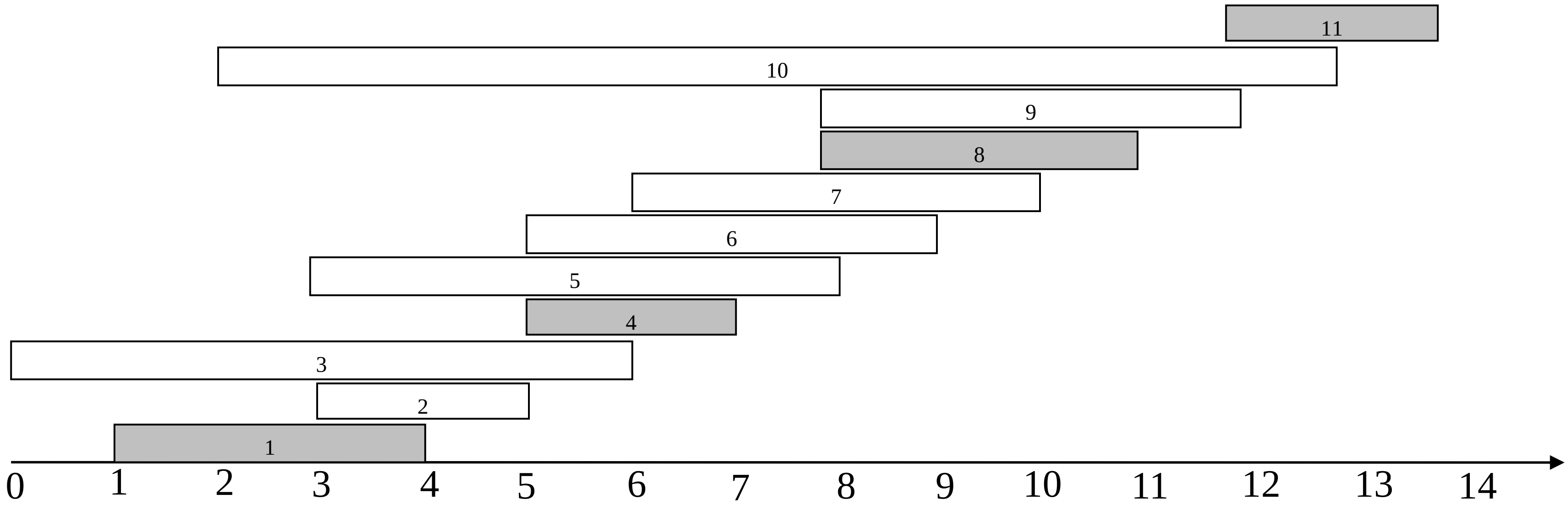
- **Theorem 16.1** Let $S_{ij} \neq \varnothing$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$. Then
  1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}. （a_m包含在某个最大相容活动子集中）
  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. （仅剩下一个非空子问题 S_{mj}）

- **Now we can solve a problem S_{ij} in a top-down fashion (What kind of fashion for DP?)**
  - **Choose $a_m \in S_{ij}$ with earliest finish time: the *greedy choice*. (it leaves as much opportunity as possible for the remaining activities to be scheduled)** （留下尽可能多的时间来安排活动，贪心选择）
  - **Then solve S_{mj}.**



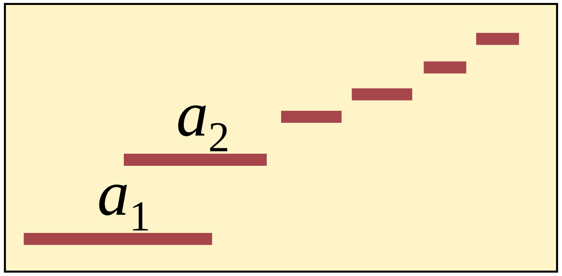
16.1.3 Converting a DP solution to a greedy solution

- **What are the subproblems?**
  - **Original problem is $S_{0,\, n+1}$** （ a_0 = [-∞, 0); a_{n+1} = [∞, "∞+1") ）
  - **Suppose our first choice is a_{m1} (in fact, it is a_1)**
  - **Then next subproblem is $S_{m1,\, n+1}$**
  - **Suppose next choice is a_{m2} (it must be a_2?)**
  - **Next subproblem is $S_{m2,\, n+1}$**
  - **And so on**



16.1.3 Converting a DP solution to a greedy solution

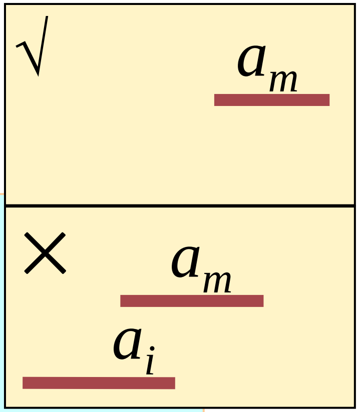
- **What are the subproblems?**
  - **Original problem is $S_{0,\ n+1}$**
  - **Suppose our first choice is a_{m1}**
  - **Then next subproblem is $S_{m1,\ n+1}$**
  - **Suppose next choice is a_{m2}**
  - **Next subproblem is $S_{m2,\ n+1}$**
  - **And so on**



- **Each subproblem is $S_{mi\ ,\ n+1}$.**
- **And the subproblems chosen have finish times that increase.**（所选的子问题，其完成时间是增序排列）
- **Therefore, we can consider each activity just once, in monotonically increasing order of finish time.**

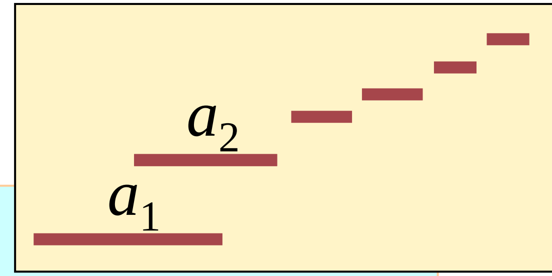
16.1.4 A recursive greedy algorithm

- **Original problem is $S_{0,\,n+1}$**
- **Each subproblem is $S_{mi\,,\,n+1}$**
- **Assumes activites already sorted by monotonically increasing finish time. (If not, then sort in $O(n\lg n)$ time.) Return an optimal solution for $S_{i,\,n+1}$:**

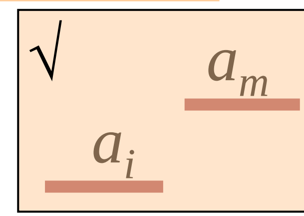



```
REC-ACTIVITY-SELECTOR(s, f, i, n)
  m ← i+1 // initially i = 0, m = 1
  while m ≤ n and s_m < f_i    // Find next activity in S_{i, n+1}.
        m ← m+1
  if m ≤ n
        return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else  return ∅
```

16.1.4 A recursive greedy algorithm

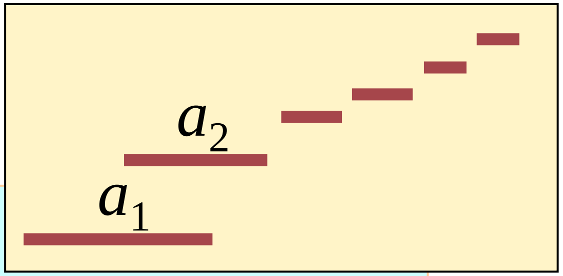


REC-ACTIVITY-SELECTOR(s, f, i, n)
$m \leftarrow i+1$
while $m \le n$ and $s_m < f_i$ // Find next activity in $S_{i,\, n+1}$.
    $m \leftarrow m+1$
if $m \le n$
    return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
else **return** $\emptyset$



- ***Initial call*:** REC-ACTIVITY-SELECTOR(s, f, 0, n).
- ***Idea:*** The **while** loop checks a_{i+1} , a_{i+2} , . . . , a_n until it finds an activity a_m that is compatible with a_i (need $s_m \ge f_i$).
  - If the loop terminates because a_m is found ($m \le n$), then recursively solve $S_{m,\, n+1}$, and return this solution, along with a_m.
  - If the loop never finds a compatible a_m ($m > n$), then just return empty set.

16.1.4 A recursive greedy algorithm




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

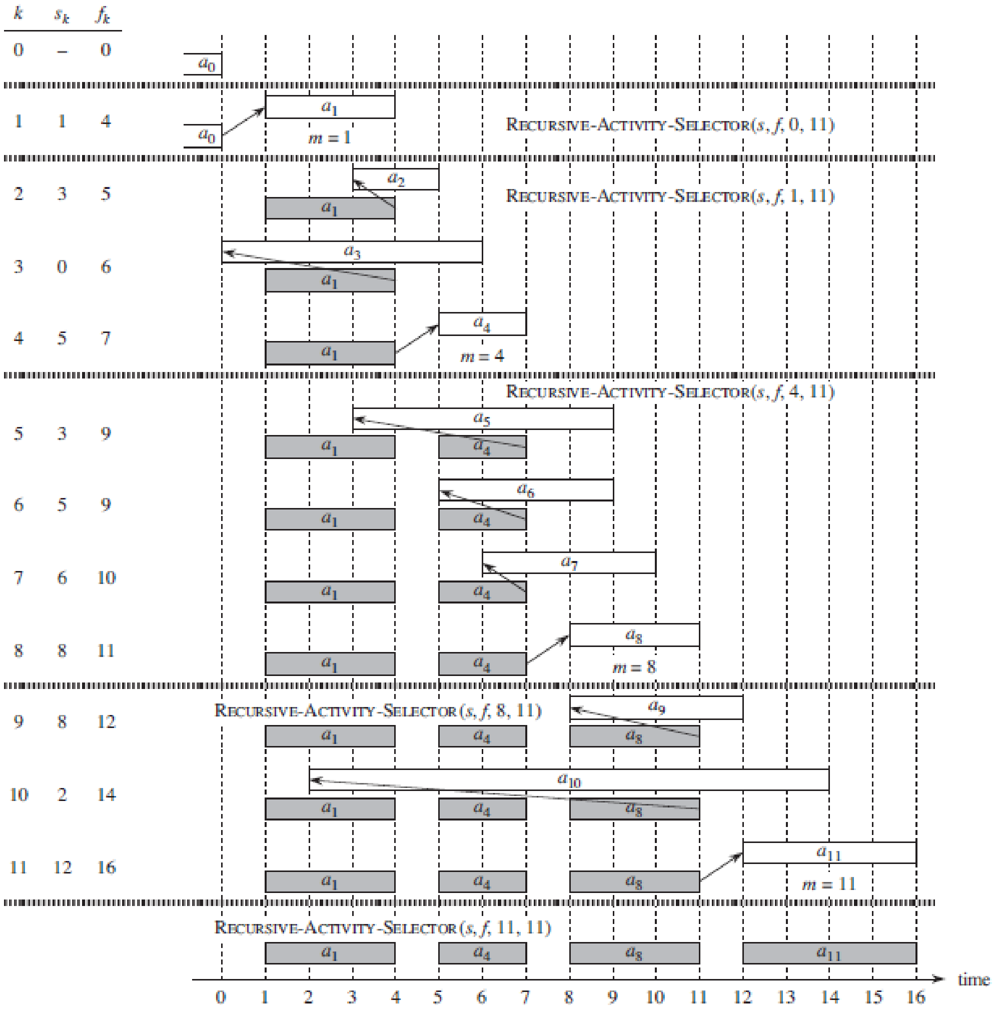
- ***Time:*** **Θ**(n)—each activity examined exactly once.

$$T(n) = m_1 + T(n - m_1) = m_1 + m_2 + T(n - m_1 - m_2)$$
$$= m_1 + m_2 + m_3 + T(n - m_1 - m_2 - m_3) = \cdots$$
$$= \sum m_k + T(n - \sum m_k)$$

basecase: $n - \sum m_k = 1$, then $\sum m_k = n - 1$, $\sum m_k + T(1) = \Theta(n)$

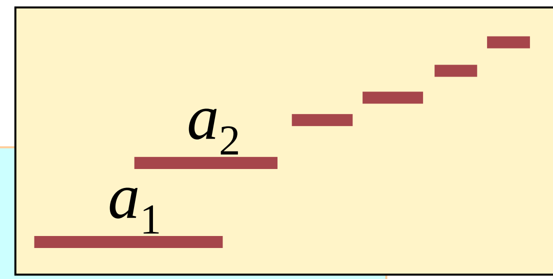
16.1.4 A recursive greedy algorithm

- ***Initial call:*** REC-ACTIVITY-SELECTOR(s, f, 0, n).
- ***Idea:*** The **while** loop checks a_{i+1} , a_{i+2} , . . . , a_n until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).
  - If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
  - If the loop never finds a compatible a_m ($m > n$), then just return empty set.



16.1.5 An iterative greedy algorithm

- **REC-ACTIVITY-SELECTOR is almost "tail recursive“.**
- **We easily can convert the recursive procedure to an iterative one. (Some compilers perform this task automatically)**



GREEDY-ACTIVITY-SELECTOR(s, f, n)

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

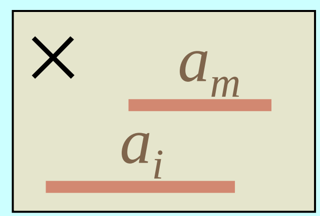
for $m \leftarrow 2$ **to** n

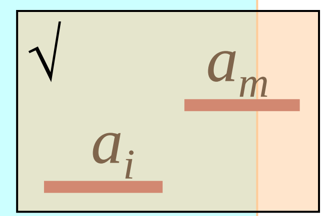
  if $s_m \geq f_i$

    $A \leftarrow A \cup \{a_m\}$

    $i \leftarrow m$ // a_i is most recent addition to A

return A





Review

- **Greedy Algorithm 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*.**
  （希望当前选择是最好的，每一个局部最优选择能产生全局最优选择）

- **Greedy Algorithm: Simpler, more efficient**

16 Greedy Algorithms

- **16.1, the activity-selection problem （活动安排）**
- **16.2**
  basic elements of the GA
  knapsack prob
  （贪婪算法的基本特征；背包问题）
- **16.3, an important application: the design of data compression (Huffman) codes （哈夫曼编码）**

16.2 Elements of the greedy strategy

- **The choice that seems best at the moment is chosen**
  （每次决策时，当前所做的选择看起来是“最好”的）
- **What did we do for activity selection?**

  1. Determine the optimal substructure.

  2. Develop a recursive solution.

  3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice.

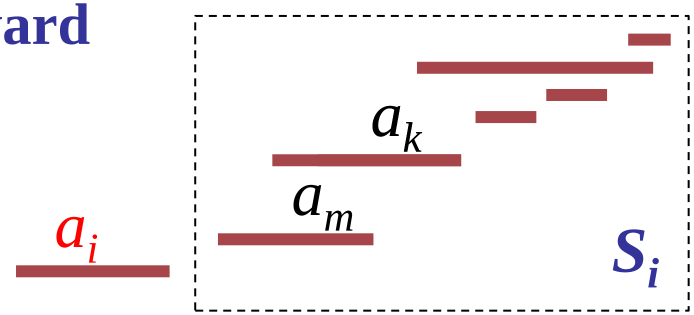
  4. Show that all but one of the subproblems resulting from the greedy choice are empty.（通过贪婪选择，只有一个子问题非空）

  5. Develop a recursive greedy algorithm.

  6. Convert it to an iterative algorithm.

16.2 Elements of the greedy strategy

- These steps looked like dynamic programming.
- Develop the substructure with an eye toward
  - making the greedy choice,
  - leaving just one subproblem.



- For activity selection, we showed that the greedy choice implied that in S_{ij} , only i varied, and j was fixed at n+1,
- So, we could have started out with a greedy algorithm in mind:
  - define $S_i = \{a_k \in S : f_i \leq s_k\}$,（所有在 a_i 结束之后开始的活动）
  - show the greedy choice, first a_m to finish in S_i combined with optimal solution to S_m $\Rightarrow$ optimal solution to S_i

16.2 Elements of the greedy strategy

Typical streamlined steps

1. **Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.**
   快速做选择，且留下尽可能少的子问题，且子问题包括的信息尽可能多

2. **Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.**
   选择是解的一部分【贪婪】，因此贪婪选择是安全的

3. **Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.**
   贪婪选择 + 子问题的最优解 ⇒ 原问题的最优解

16.2 Elements of the greedy strategy

- **No general way to tell if a greedy algorithm is optimal, but two key ingredients are**

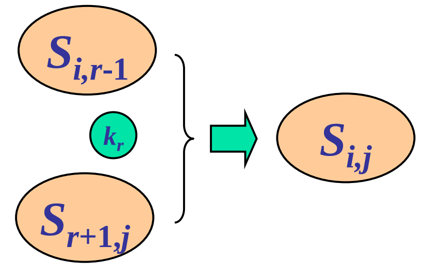
（没有一般化的规则来说明贪婪算法是否最优，但有两个基本要点）

1. greedy-choice property （贪婪选择属性）

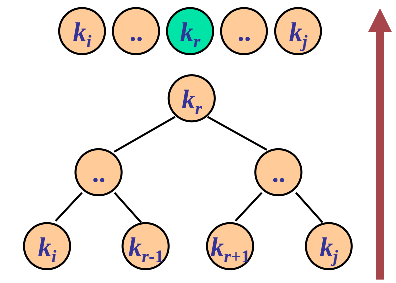
2. optimal substructure

16.2.1 Greedy-choice property

- **A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.**

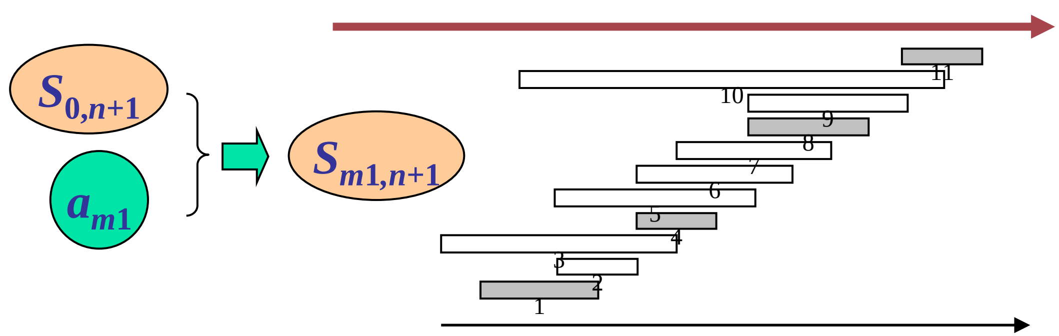


Activity-DP



OBST-DP

- ***Dynamic programming***
  - **Make a choice at each step.**
  - **Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.** （依赖于已知子问题的最优解再作出选择）
  - **Solve *bottom-up*.**



Activity-GA

- ***Greedy***
  - **Make a choice at each step.**
  - **Make the choice *before* solving the subproblems.**
  - **Solve *top-down*.**

16.2.1 Greedy-choice property

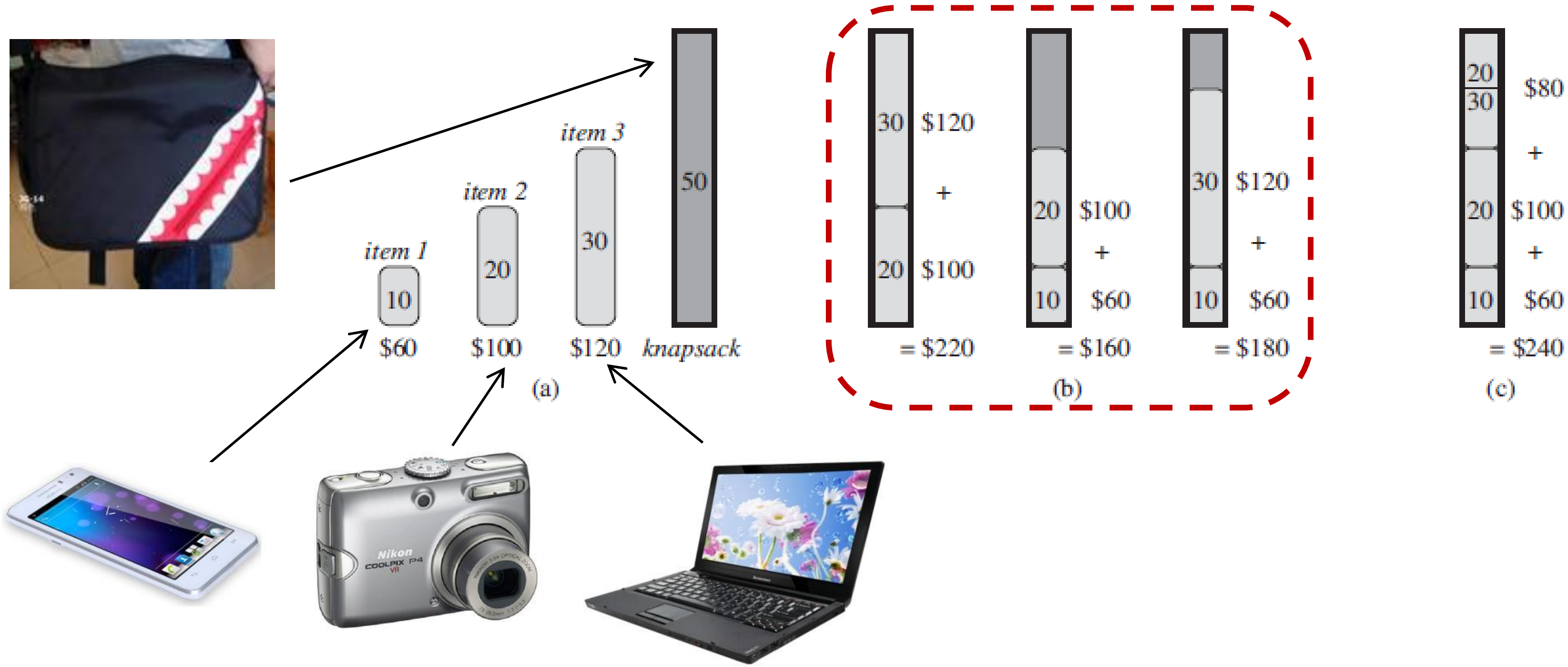
- **We must prove that a greedy choice at each step yields a globally optimal solution. Difficulty! Cleverness may be required!**
- **Typically, Theorem 16.1, shows that the solution (A_{ij}) can be modified to use the greedy choice (a_m), resulting in one similar but smaller subproblem (A_{mj}).**
- **We can get efficiency gains from greedy-choice property. *(For example, in activity-selection, sorted the activities in monotonically increasing order of finish times, needed to examine each activity just once.)***
  - **Preprocess input to put it into greedy order**
  - **Technically, the maximum or minimum is usually the choice.**

16.2.2 Optimal substructure

- ***optimal substructure*: an optimal solution to the problem contains within it optimal solutions to subproblems.**

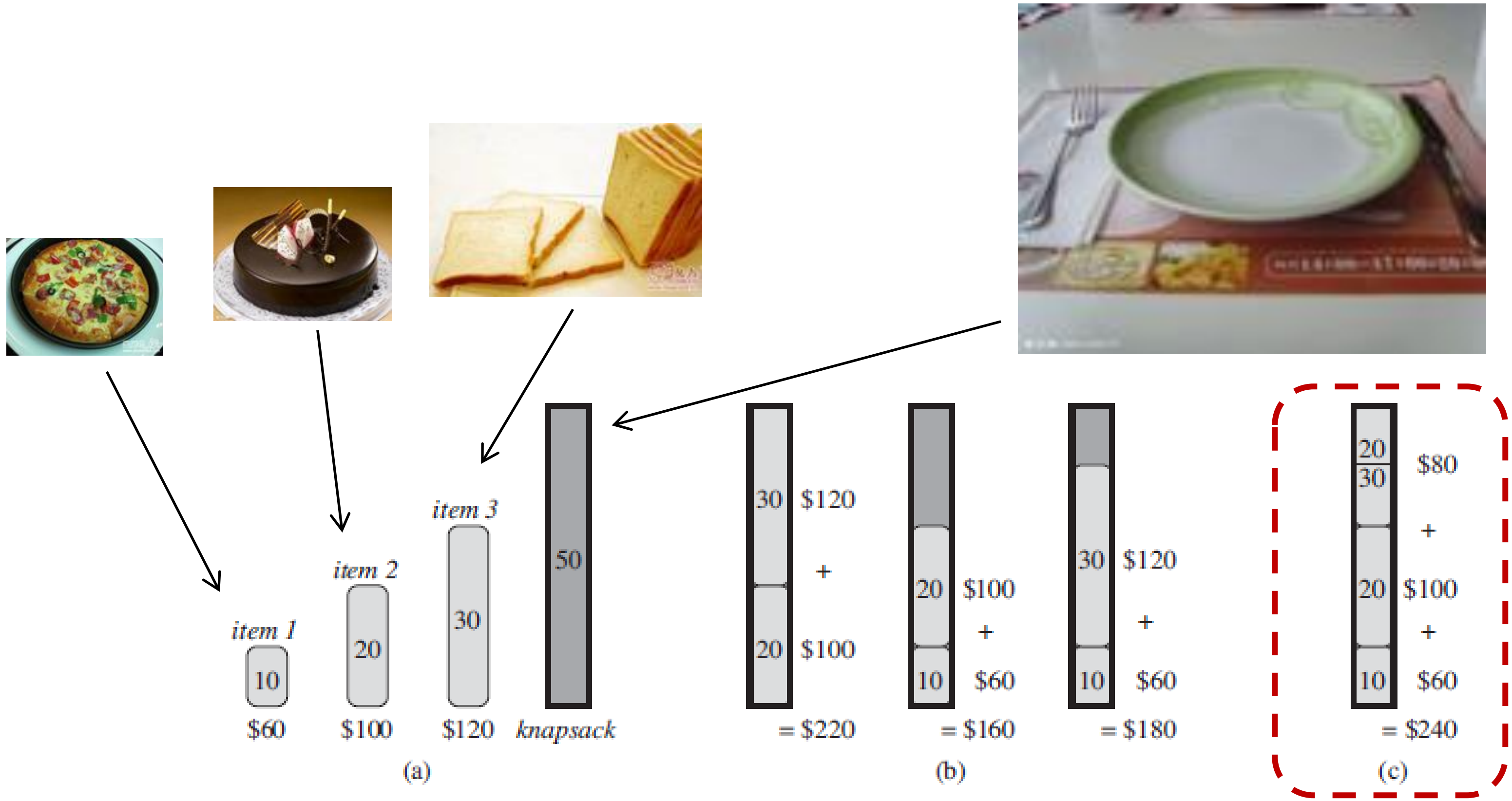
- **Just show that optimal solution to subproblem and greedy choice ⇒ optimal solution to problem.**
  （说明子问题的最优解和贪婪选择
  ⇒ 原问题的最优解）

16.2.3 knapsack: Greedy vs. dynamic programming



- ***0-1 knapsack problem*** （0-1背包问题，小偷问题）
  - ◆ ***n*** **items** （***n*** 个物品）
  - ◆ **Item** ***i*** **is worth** $\$v_i$**, weighs** w_i **P**（物品i 价值v_i，重w_i）
  - ◆ **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.**

16.2.3 knapsack: Greedy vs. dynamic programming

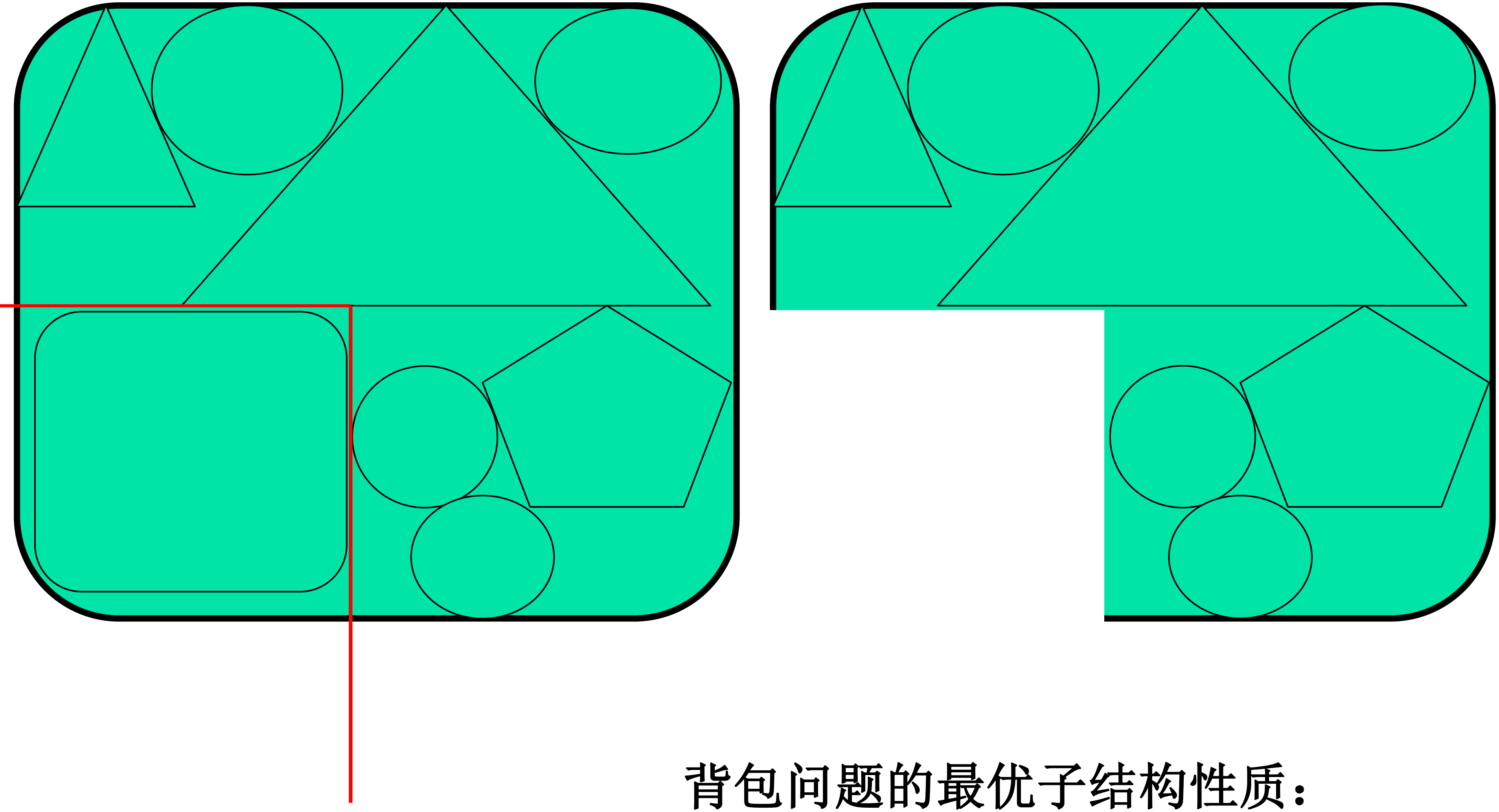


- ***Fractional knapsack problem*** （分数背包问题，小偷问题）
  - **Like the 0-1 knapsack problem, but can take fraction of an item.**

16.2.3 knapsack: Greedy vs. dynamic programming

- ***0-1 knapsack problem*** **（0-1背包问题，小偷问题）**
- ***Fractional knapsack problem*** **（分数背包问题，小偷问题）**

- **Both have optimal substructure property.**
  - **0-1 : ?**
  - **fractional: ?**



背包问题的最优子结构性质：

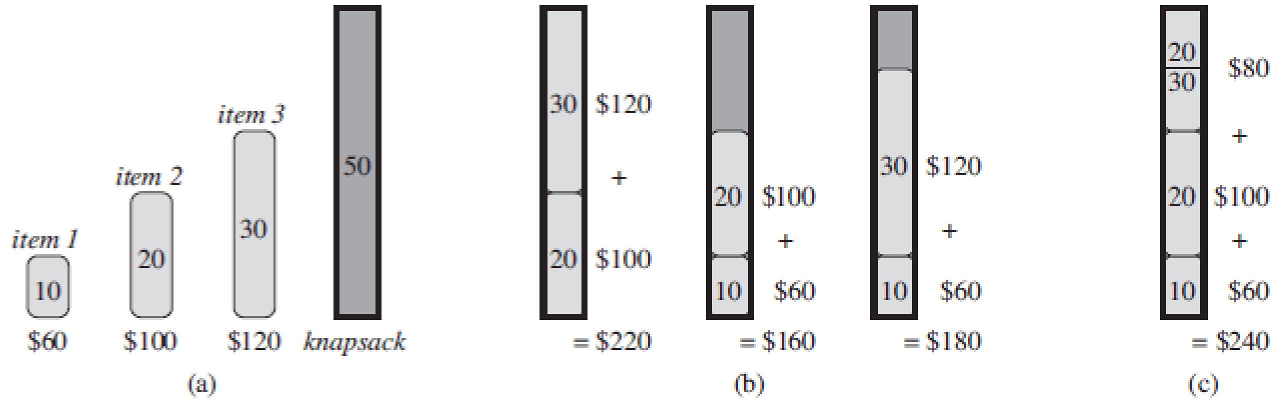
完整的圆角矩形框是一个最优背包，去掉右下角的红色部分剩下的部分是一个子背包，则该子背包也是一个最优背包。

16.2.3 knapsack: Greedy vs. dynamic programming

- ***0-1 knapsack problem*** **（ 0-1背包问题，小偷问题）**
- ***Fractional knapsack problem*** **（分数背包问题，小偷问题）**

- **But the fractional problem has the greedy-choice property, and the 0-1 problem does not.**

16.2.3 knapsack: Greedy vs. dynamic programming



- **Fractional knapsack problem has the greedy-choice property, and the <u>0-1 knapsack problem</u> does not.**
- **To solve the fractional problem, rank decreasingly items by v_i/w_i**
- **Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i**

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

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

16.2.3 knapsack: Greedy vs. dynamic programming

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

0-1 knapsack problem has not the greedy-choice property.

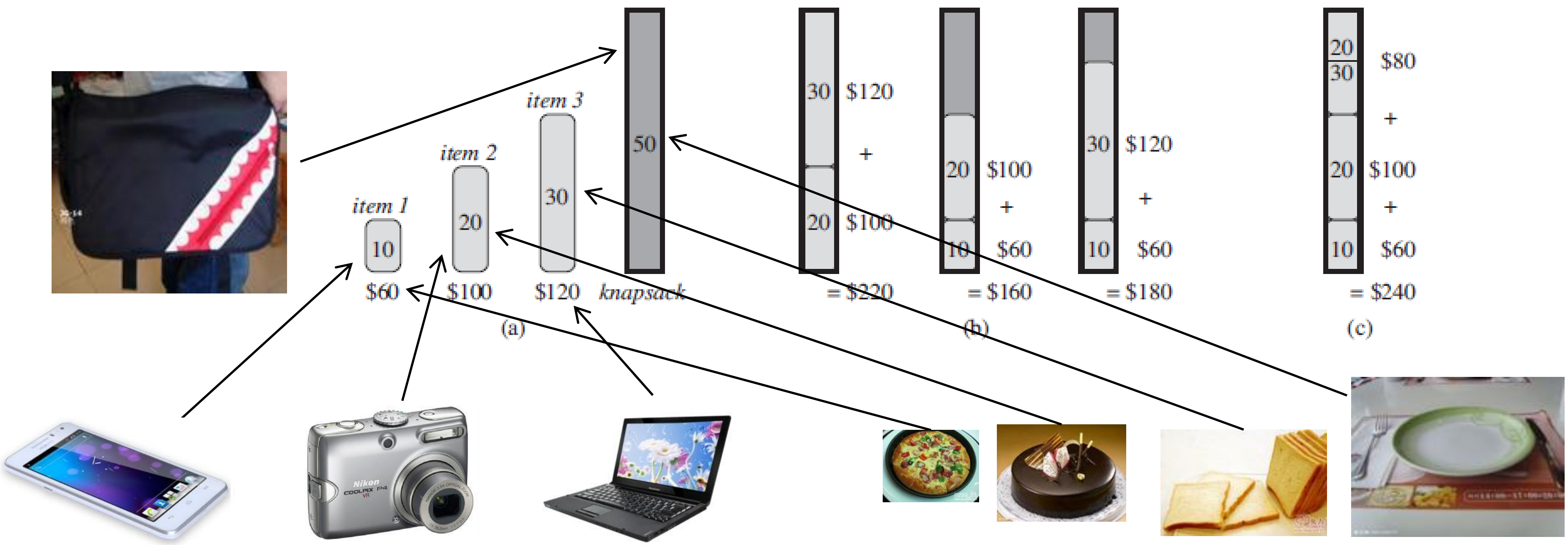
let $W = 50$ for the following example.

- **Greedy solution:**
  - **take items 1 and 2**
  - **value = 160, weight = 30**

20 pounds of capacity leftover.

- **Optimal solution:**
  - **Take items 2 and 3**
  - **value=220, weight=50**

No leftover capacity. **(没有剩余空间)**



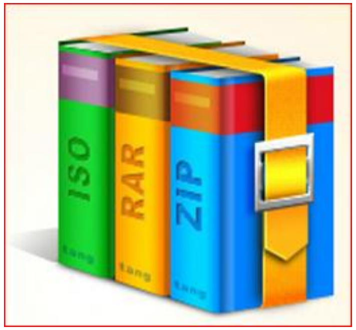
16 Greedy Algorithms

- **16.1, the activity-selection problem （活动安排）**
- **16.2, basic elements of the GA; knapsack prob （贪婪算法的基本特征；背包问题）**
- **16.3, an important application: the design of data compression (Huffman) codes （哈夫曼编码）**

A method for the construction of minimum-redundancy codes
DA Huffman - Proceedings of the IRE, 1952 - ieeexplore.ieee.org
INTRODUCTION Q NE IMPORTANT METHOD of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one sym- bol. If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) ofa message is directly proportional to the number of symbols associated with it. In this paper, the symbol or sequence of symbols associated with a given message will be called the …

16.3 Huffman codes

- Huffman codes: widely used and very effective technique for encoding file or compressing data.
  - savings of 20% to 90%



作业：每人写一个压缩软件

- Consider the data to be a sequence of characters

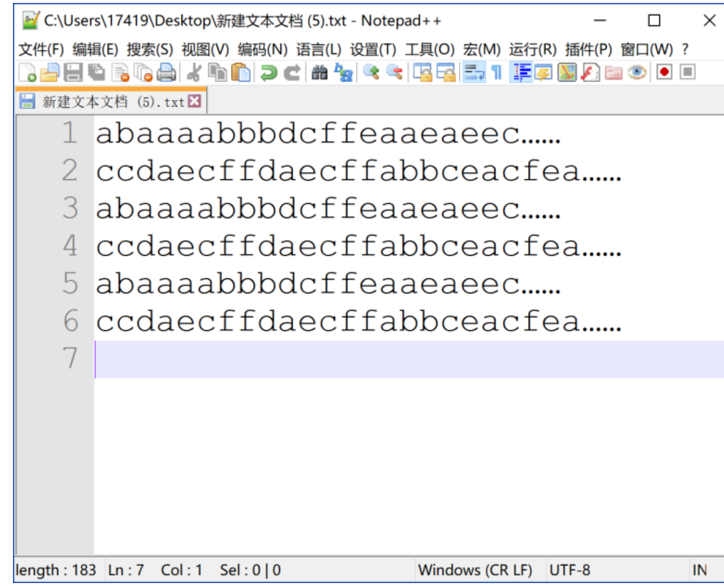
  abaaaabbbdcffeaaeaeec…abaadefe

- Huffman's greedy algorithm:

  uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

  依据字符出现的频率表，使用二进串来建立一种表示字符的最佳方法

16.3 Huffman codes



- **Wish to store compactly 100,000-character data file.**

  Only six different characters appear.

  abaaaabbbdcffeaaeaeec……

  ccdaecffdaecffabbceacfea……

  Frequency table:

	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

- **Many ways (encodes) to represent such a file of information**
- ***binary character code* (or *code* for short): each character is represented by a unique binary string.**
  - ***fixed-length code*: if use 3-bit codeword, the file can be encoded in 300,000 bits. Can we do better?**

16.3 Huffman codes

- **100,000-character data 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

- ***binary character code* (or *code* for short)**
  - ***variable-length code*: by giving frequent characters short codewords and infrequent characters long codewords, here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.** （高频出现的字符以短字码表示；低频→长字码）

    $(45\cdot1 + 13\cdot3 + 12\cdot3 + 16\cdot3 + 9\cdot4 + 5\cdot4)\cdot 1{,}000 = 224{,}000$ bits

16.3 Huffman codes

- **100,000-character data 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

- ***binary character code* (or *code* for short)**
  - ***fixed-length code*: 300,000 bits**
  - ***variable-length code*: 224,000 bits, a savings of 25.3%. In fact, this is an optimal character code for this file.**

16.3.1 Prefix codes

- *prefix codes* (prefix-free codes): no codeword is a prefix of some other codeword.
前缀码〔前缀无关码〕：没有字码是其他字码的前缀

	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

- Encoding is always simple for any binary character code
  - Concatenate（连接） the codewords representing each character. For example, “abc”, with the variable-length prefix code as 0·101·100 = 0101100, where we use ‘·’ to denote concatenation.
- Prefix codes simplify decoding

16.3.1 Prefix codes

- ***prefix codes*** **(prefix-free codes): no codeword is a prefix of some other codeword.**
  前缀码〔前缀无关码〕：没有字码是其他字码的前缀

	a	b	c	d	e	f
Variable-length codeword	0	101	100	111	1101	1100

- **Encoding is always simple for any binary character code**
- **Prefix codes simplify decoding**
  - **Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous（明确的）.**
  - **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.**
  - **Exam: 001011101 uniquely as 0·0·101·1101, which decodes to "aabe".**

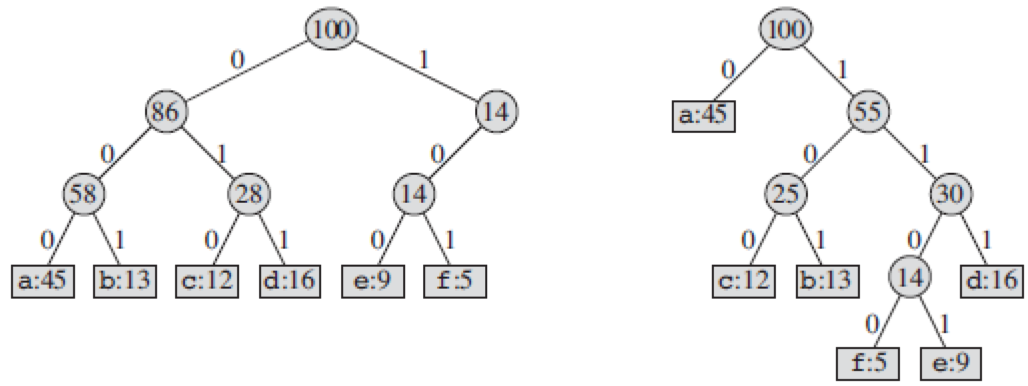
16.3.1 Prefix codes

a	b	c	d	e	f
0	101	100	111	1101	1100

001011101

uniquely as 0·0·101·1101,

which decodes to “aabe”.



(a) (b)

Decoding

- **the process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off.** **（前置无关码方便解码）**

- **A binary tree whose leaves are the given characters provides one such representation.** **（二叉树是一种方便的表示方法，树叶为给定字符，从树根到树叶的过程就是解码过程）**

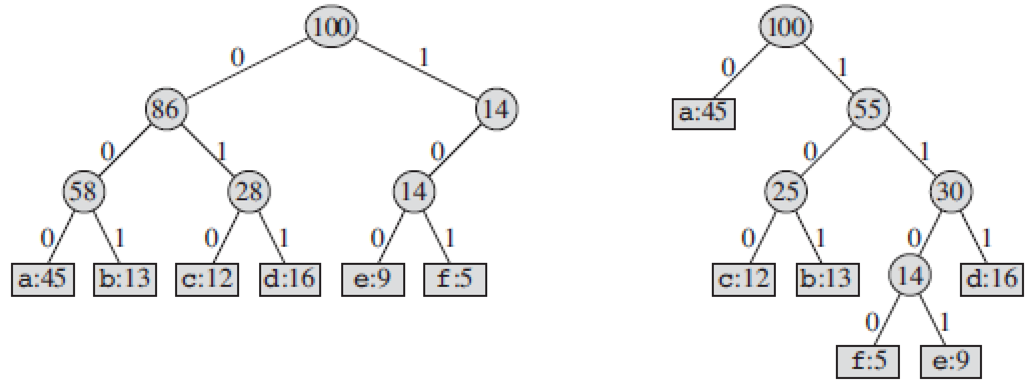
16.3.1 Prefix codes

a	b	c	d	e	f
0	101	100	111	1101	1100

001011101

uniquely as 0·0·101·1101,

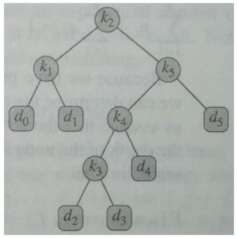
which decodes to “aabe”.



(a) (b)

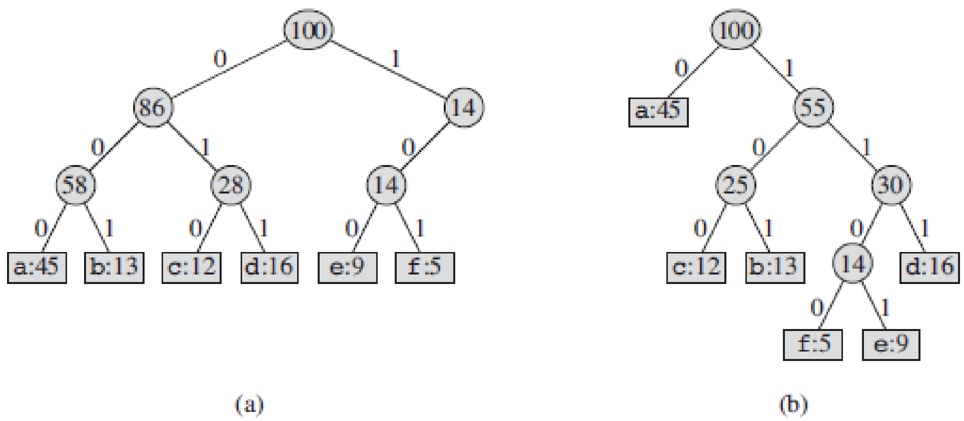
Decoding

- We interpret the binary codeword for a character as the path from the root to that character.（字符的编码为一条从树根到树叶路径）
- It is not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.



注意：不要混淆各种二叉树，如，最大（小）堆、二叉搜索树，哈夫曼树

16.3.1 Prefix codes

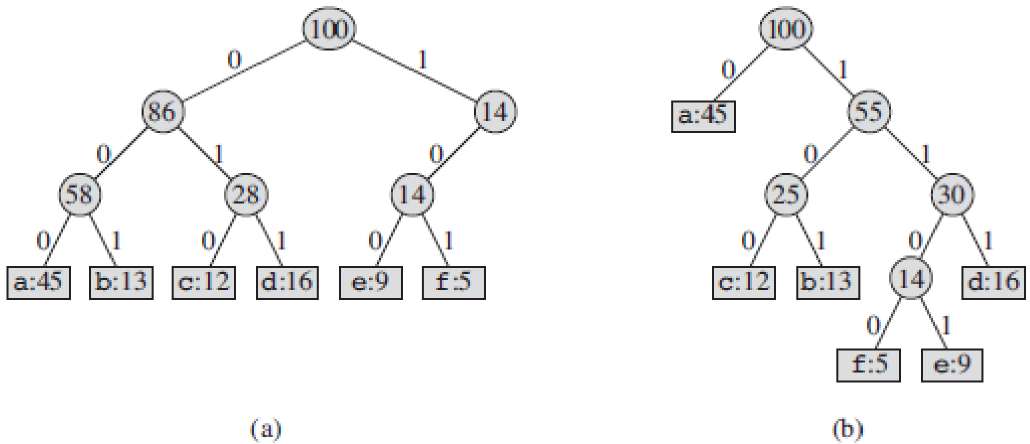


full binary tree 满二叉树

- 国际定义：除叶子节点外，所有节点都有两个孩子。
- 国内定义：除了满足如上定义，所有叶子节点还需要在同一层上。

- **An optimal code for a file is always represented by a *full* binary tree, every nonleaf node has two children (Ex16.3-1). The fixed-length code in our example is not optimal.**
- **We can restrict our attention to full binary trees**
  - ***C* is the alphabet,**
  - **all character frequencies >0**
  - **the tree for an optimal prefix code has |*C*| leaves, one for each letter of *C* , and exactly |*C*|-1 internal nodes.**

16.3.1 Prefix codes



Compute # of bits required to encode a file

- **Given a tree *T* corresponding to a prefix code, for each character *c* in the alphabet *C*,**
  - ***f* (*c*): frequency of *c* in the file**
  - **$d_T(c)$: depth of *c*'s leaf in the tree (length of the codeword for character *c*). Then, # of bits required to encode a file**

$$B(T) = \sum_{c \in C} f(c) d_T(c) \qquad (16.5)$$

which we define as the *cost* of the tree *T*.

16.3.2 Constructing a Huffman code

***Huffman code*:**
a greedy algorithm that constructs an optimal prefix code

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n - 1
        allocate(分配) 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.
```

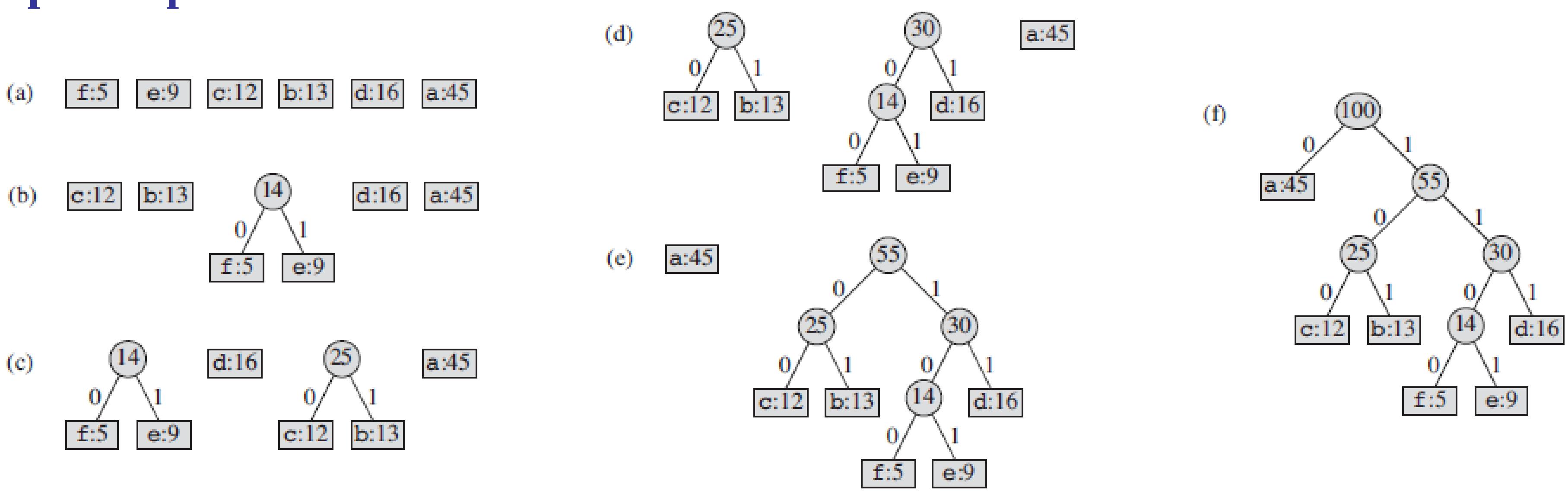
- **C: set of n characters, $c \in C$: an object with frequency $f\,[c]$.**
  - **Build the tree T corresponding to the optimal code.**
  - **Begin with $|C|$ leaves, perform $|C|$-1 "merging" operations.**
  - **A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. Result of the merger is a new object whose frequency is the sum of the frequencies of the two objects that were merged.**

16.3.2 Constructing a Huffman code

```
HUFFMAN(C)
  n ← |C|
  Q ← C
  for i ← 1 to n - 1
        allocate(分配) 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.
```

Running time ?

Example: Huffman's algorithm proceeds. 6 letters, 5 merge steps. The final tree represents the optimal prefix code.



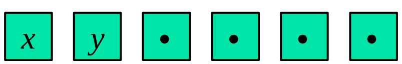
16.3.3 Correctness of Huffman's algorithm*

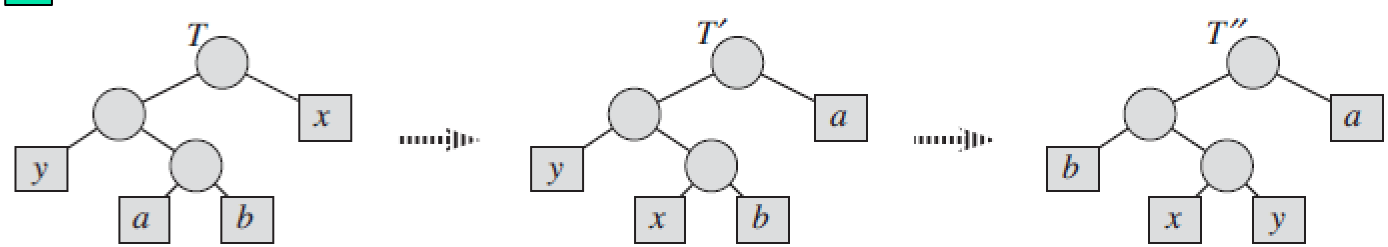
- **Problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.**

- **Lemma 16.2 (greedy-choice property)**

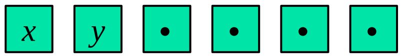
  Let *C* be an alphabet, each character $c \in C$ has frequency $f[c]$. *x* and *y* $\in C$, and having the lowest frequencies. Then there exists an optimal prefix code for *C* in which the codewords for *x* and *y* have the same length and differ only in the last bit.

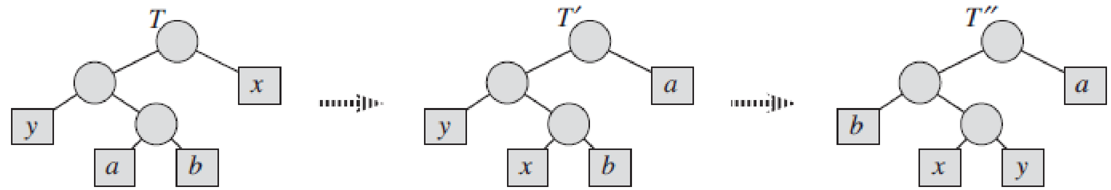
  ***Proof idea*: take the tree *T* representing an arbitrary optimal prefix code, and modify it to make a tree representing another optimal prefix code such that *x* and *y* appear as sibling leaves（姐妹叶） of maximum depth in the new tree.**





16.3.3 Correctness of Huffman's algorithm*





- **Lemma 16.2**

  $c \in C$ has frequency $f[c]$. x , $y \in C$, having the lowest frequencies. Then, exist an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

$$B(T) = \sum_{c \in C} f(c) d_T(c) \qquad (16.5)$$

Proof : Let a and b are sibling leaves of maximum depth in optimal T. Assume that $f[a] \le f[b]$, $f[x] \le f[y]$. $f[x]$ and $f[y]$ are the two lowest leaf frequencies, $f[a]$, $f[b]$ are two arbitrary frequencies, in order,$\Rightarrow f[x] \le f[a]$, $f[y] \le f[b]$. Exchange the positions in T of a and x to produce a tree T' , and then exchange the positions in T' of b and y to produce a tree T'' . By (16.5), we have

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

Similarly, $B(T') - B(T'') \ge 0$,

therefore, $B(T'') \le B(T)$.

Since T is optimal, $B(T) \le B(T'')$.

Then $B(T'') = B(T)$.

Thus, T'' is an optimal tree.

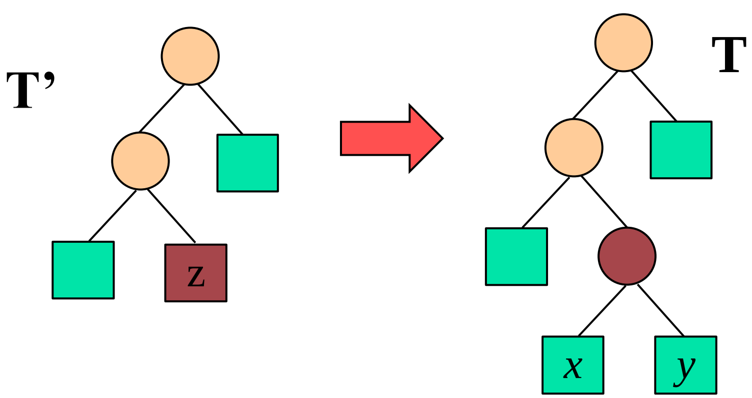
16.3.3 Correctness of Huffman's algorithm*

- **Lemma 16.3 (optimal-substructure property) ?**

  Alphabet *C* , each character *c*∈*C* has frequency *f*[*c*]. *x* and *y* ∈*C* , and having the lowest frequencies. *C'* = *C*-{*x*, *y*}∪{*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 *C*.

  （给定字母表集*C*，每一个字符*c* ∈*C* 的频率为*f* [*c*] 。 *x* 和 *y* ∈*C*有最小频率。从 *C* 中抽取字符*x* 和 *y* ，但增加新字符 *z* 到C中，得到新的字符集*C'*， 即 *C'* = *C*-{*x*, *y*}∪{*z*}。除 *f*[*z*]=*f*[*x*]+*f*[*y*] 以外，*f* 在 *C'* 中的定义与在*C*中相同。若 *T'* 为 *C'* 的最优前缀无关编码，则 *T* 为关于 *C* 的最优前缀无关编码，其中， *T* 为把 *T'* 的叶节点 *z* 代替为以 *x* 和 *y* 作为叶节点的内点变换而来。）

$C:\{c_1, \ldots, c_m, x, y\}$, $C':\{c_1, \ldots, c_m, z\}$,



由optimal **T'** 能构成 optimal **T**

16.3.3 Correctness of Huffman's algorithm*

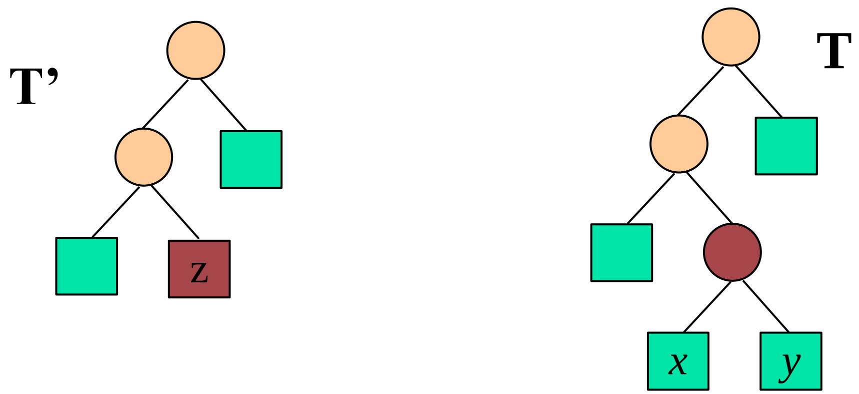
- **Lemma 16.3 (optimal-substructure property)**

Proof : For each $c \in C$-{x, y}, we have $d_T(c) = d_{T'}(c)$, then $f[c]d_T(c) = f[c]d_{T'}(c)$. Since $d_T(x)=d_T(y)=d_{T'}(z)+1$, we have

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

from which we conclude that $B(T) = B(T') + f[x] + f[y]$.

($B(T) = f[x]d_T(x)+f[y]d_T(y) + f[c]d_T(c)$, $B(T') = f[z]d_{T'}(z) + f[c]d_T(c)$)

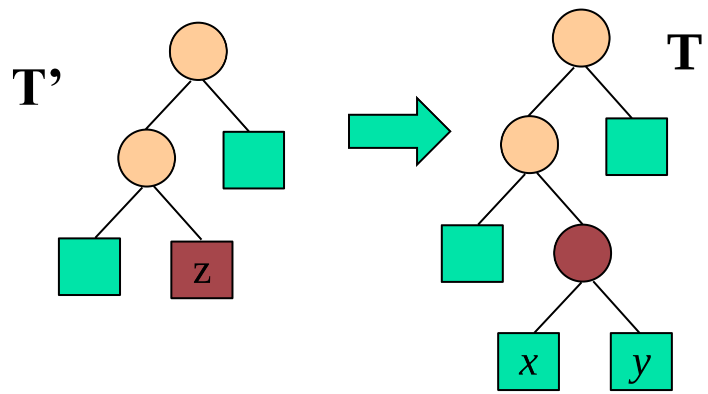


Detail of proof? Next slides.

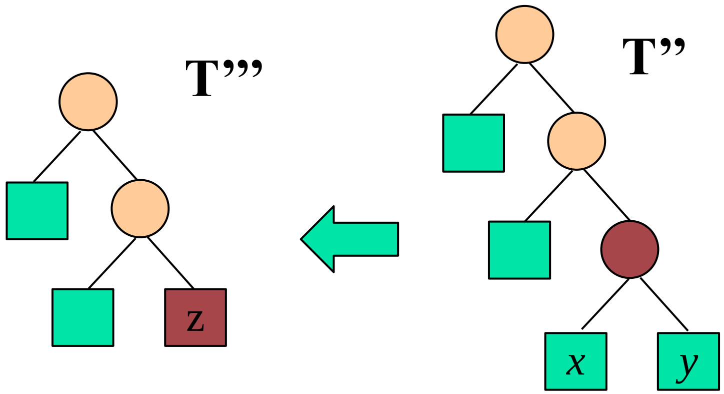
16.3.3 Correctness of Huffman's algorithm*

- **Lemma 16.3 (optimal-substructure property)**

C : $\{c_1, \ldots, c_m, x, y\}$, C': $\{c_1, \ldots, c_m, z\}$, 由optimal **T'** 能构成 optimal **T**



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



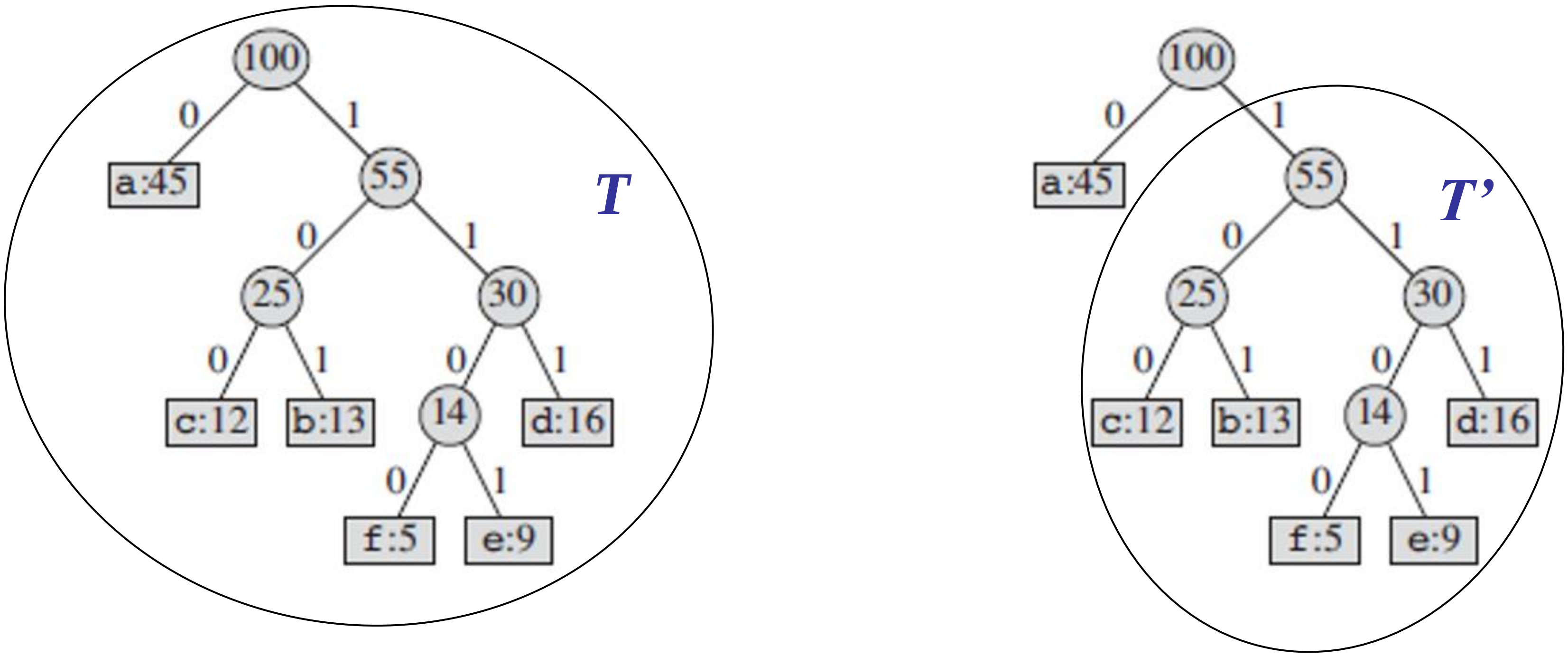
Suppose that T is not optimal, T'' is. Then $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $f[z]=f[x]+f[y]$. Then

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

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.

16.3.3 Correctness of Huffman's algorithm*

- **Lemma 16.3 (optimal-substructure property) 另一种解释**



if *T* optimal, then *T'* optimal

16.3.3 Correctness of Huffman's algorithm*

- **Theorem 16.4**

  Procedure HUFFMAN produces an optimal prefix code.

Proof **Immediate from Lemmas 16.2（每一次选择是贪婪的、是正确的） and Lemmas 16.3（确保由子问题的最优解能导出原问题的最优解）.**

Exercises

$$c[i, j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k, j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

- **It may be easy to design an algorithm to the problem based on recurrence (16.3).**

  (1) Direct recursion algorithm? complexity?

  (2) Dynamic programming algorithm? complexity?

- **For (16.3):**
  - **How many choices?**
  - **How many subproblems for a choice?**
- **Can we simplify our solution?**

16.1.3 Converting a DP solution to a greedy solution

Exercises

16.1-1,

16.1-2（最晚开始优先原则）

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\ W)$ time, where n is number of items and W is the maximum weight of items that the thief can put in his knapsack.

16.3-2（课堂作业）

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

又一些大作业或讨论题提示：

- 活动安排、Huffman code等，是否都能描述为背包问题？
- 本书有多少算法是用的贪心策略（小论文：贪心算法十个经典问题？）
- 哈夫曼编码（用哈夫曼压缩方法，设计一个压缩软件：测试一下，算法导论这本书的压缩率能到多少？）
- OBST（最优二叉搜索树构建（以某本书里的词汇为基础？））

- 活动安排、分数背包
- 0-1背包、钢管切割、ALS、MCM、LCS、最短路径
- 雇佣（雇佣多少人）、取帽子、相同生日
- RSA加密解密、FFT、串匹配、计算几何、最大流

- 算法实验室（问题求解工具、算法效果展示平台、多种算法时间复杂度对比分析、多种算法空间复杂度对比分析、IO导入、……。支持穷举、递归、回溯、分治、DP、贪心；排序、查找；随机；图、树；等等若干方法（算法）的仿真演示。）