Greedy Algorithms

Dr. Zhenyu He

Autumn 2008

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 (过度的杀伤威力).**
- **Greedy Algorithm: Simpler, more efficient**

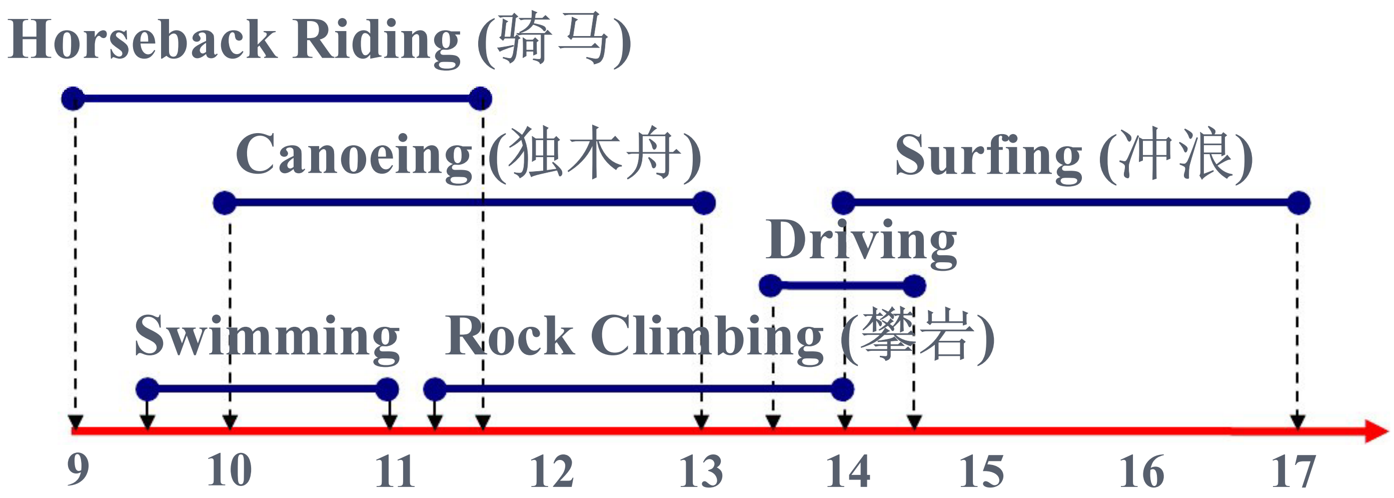
16 Greedy Algorithms

- 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）
  - 16.5, unit-time tasks scheduling （有限期作业调度）

16 Greedy Algorithms

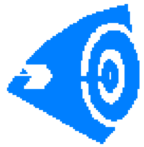
- **The greedy method is quite powerful and works well for a wide range of problems:**
  - **minimum-spanning-tree algorithms (Chap 23)**
    （最小生成图）
  - **shortest paths from a single source (Chap 24)**
    （最短路径）
  - **set-covering heuristic (Chap 35).**
    （集合覆盖）
  - …

First 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






- ***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）

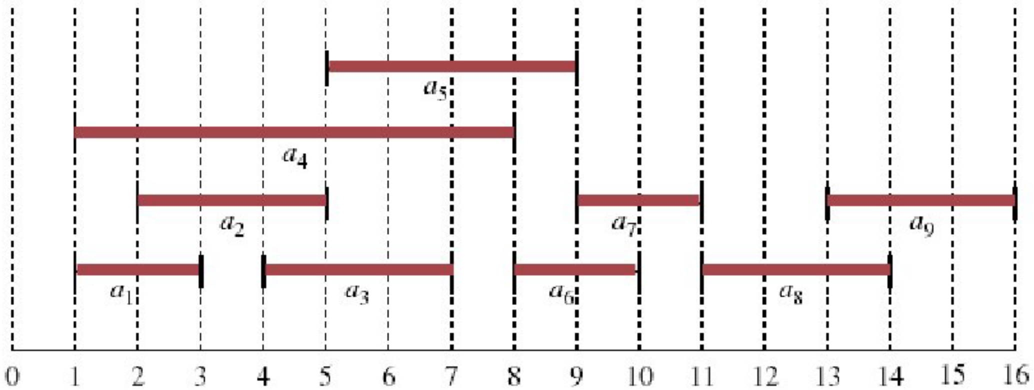
playground

  - **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

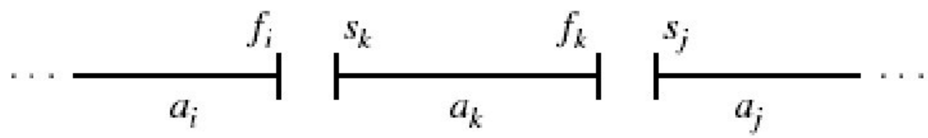
$\{a_2, a_5, a_7, a_9\}$.

16.1.1 Optimal substructure of activity selection

Space of subproblems

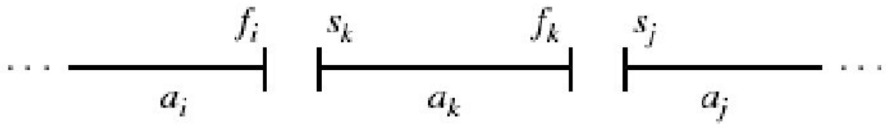
- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$

  = 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 $0 \le i, j \le n+1$.**

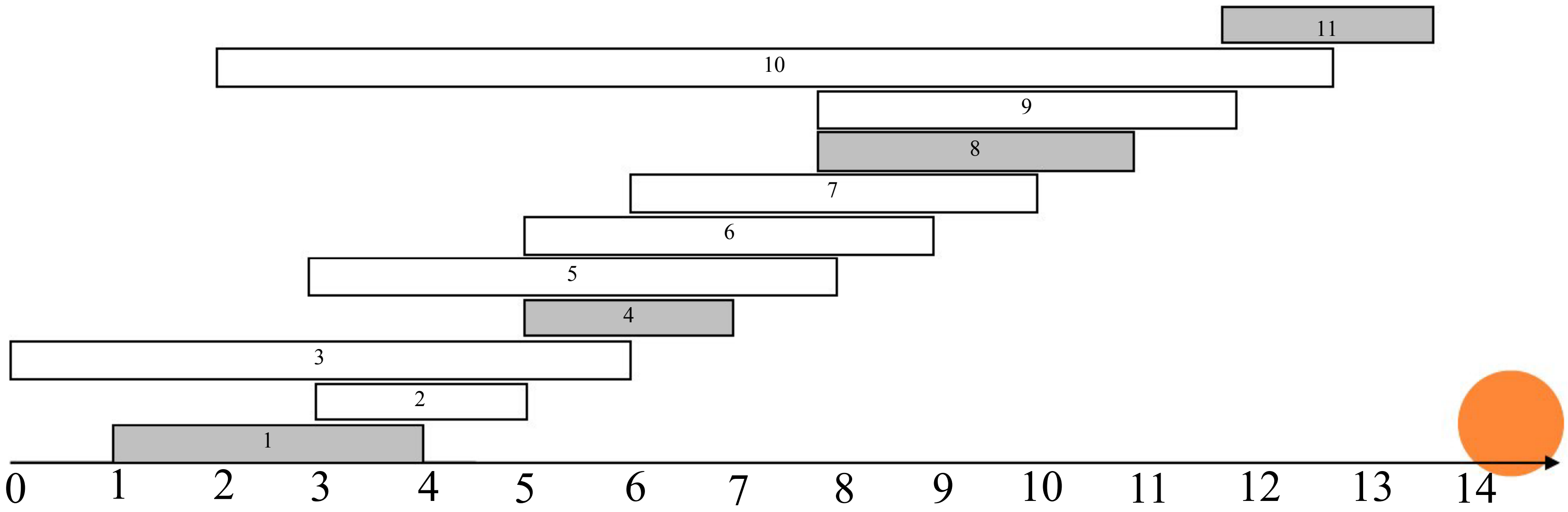
16.1.1 Optimal substructure of activity selection



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）

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



16.1.1 Optimal substructure of activity selection

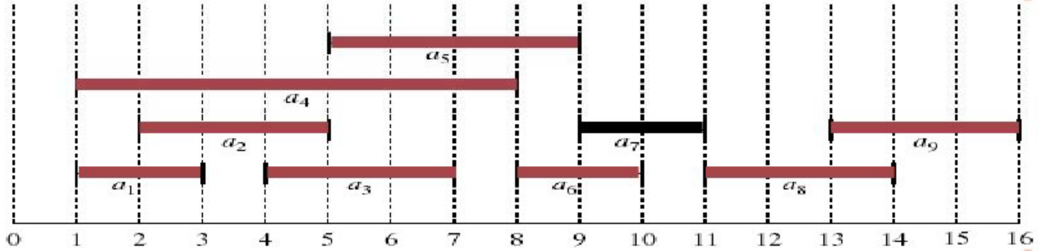
- if $f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$ (if $i \le j$, then $f_i \le f_j$) (16.1)
  Then $i \ge j \Rightarrow S_{ij} = \emptyset$

  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 ∅

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}) ∪ {a_k} ∪ (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）
  $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0$.
- **If $S_{ij} \neq \emptyset$, 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**

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

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.

16.1.3 Converting a DP solution to a greedy solution

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

- **It may be easy to design an algorithm to the problem based on recurrence (16.3).**
  - **Direct recursion algorithm (complexity?)**
  - **Dynamic programming algorithm (complexity?)**
- **Can we simplify our solution?**

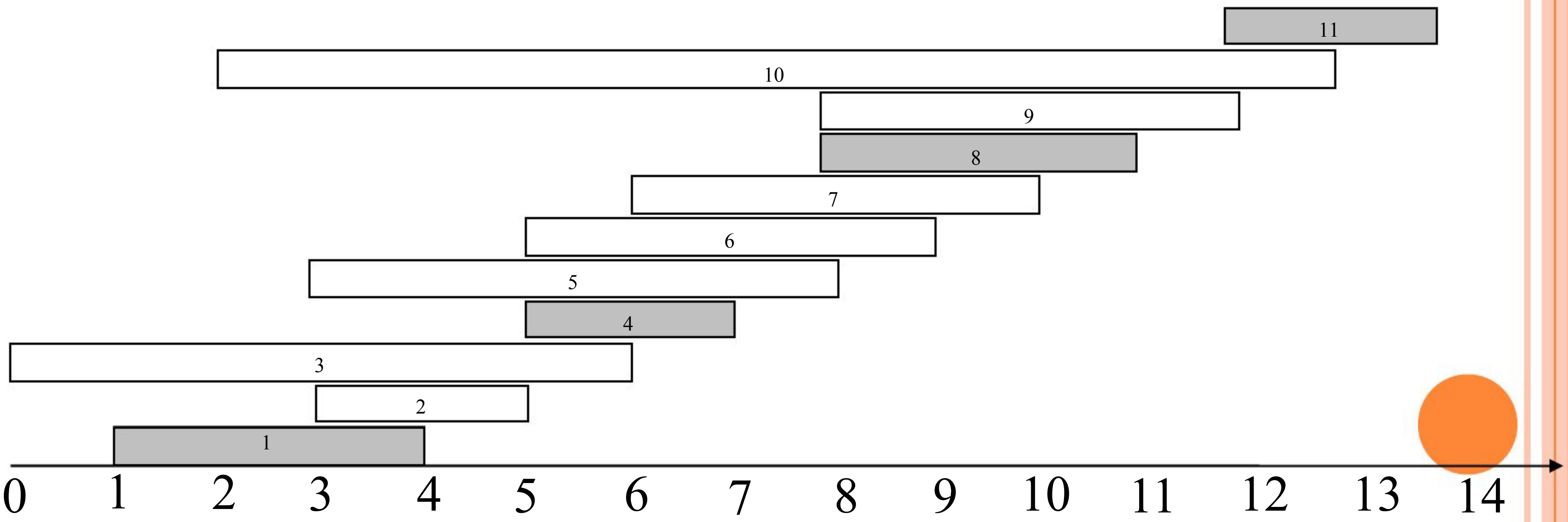
16.1.3 Converting a DP solution to a greedy solution

- **Theorem 16.1**

  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}.（a_m 包含在某个最大相容活动子集中）

  2. $S_{im} = \emptyset$, 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 $S_{ij} \neq$ |, 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. $S_{im} =$ |, 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 $f_i < s_l < f_l \leq s_m < f_m \Rightarrow f_l < f_m$ 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

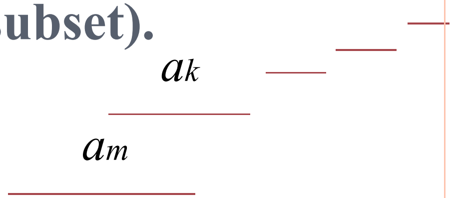
- **Theorem 16.1** Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

  1. a_m **is used in some maximum-size subset of mutually compatible activities of** S_{ij}. （a_m包含在某个最大相容活动子集中）

16

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

$$c[i,j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \emptyset, \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \emptyset. \end{cases} \tag{16.3}$$

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

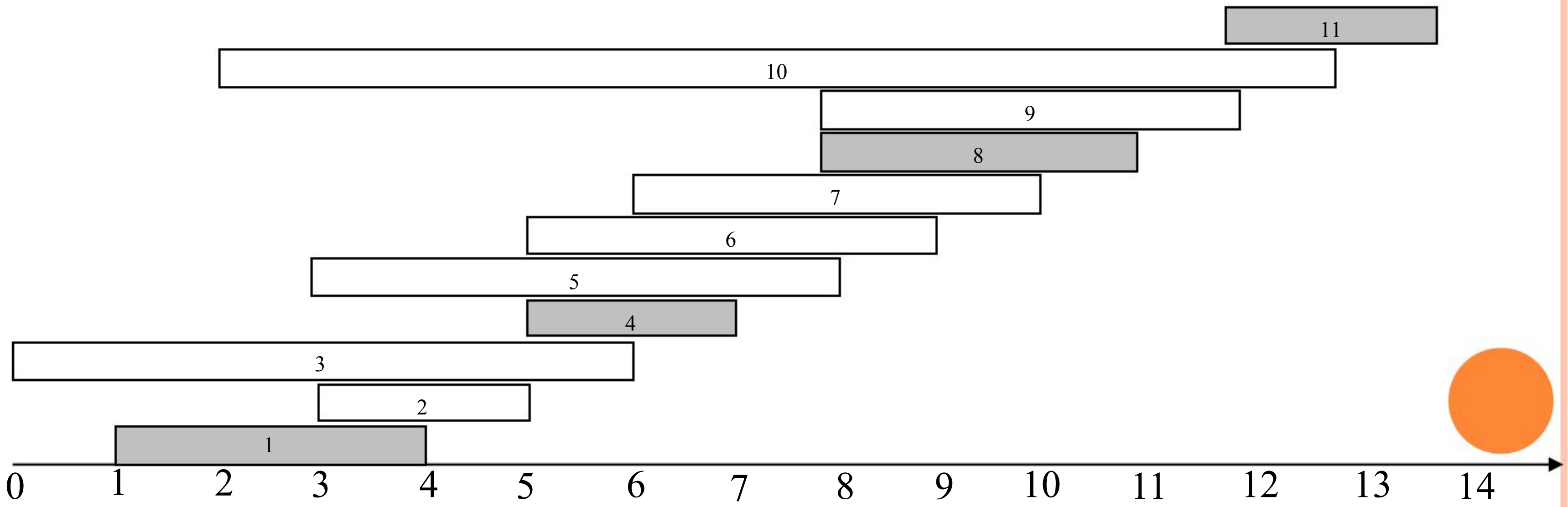
  1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.（**a_m**包含在某个最大相容活动子集中）

  2. $S_{im} = |$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. 18
  （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**
  - **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 $S_{0,n+1}$ （ $a_0 = [-\infty, 0)$; $a_{n+1} = [\infty, \text{“}\infty+1\text{”})$ ）
  - 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

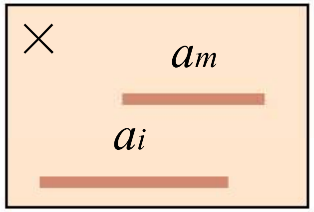


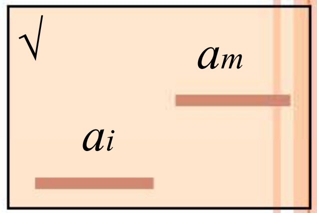
16.1.3 Converting a DP solution to a greedy solution

- **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

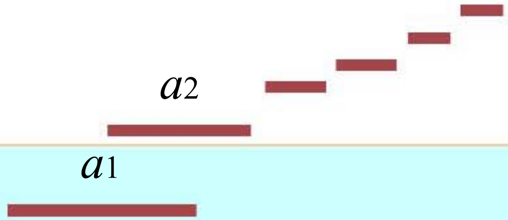
- **Original problem is $S_{0,n+1}$**
- **Each subproblem is $S_{m_i,n+1}$**
- **Assumes activites already sorted by monotonically increasing finish time. (If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:**



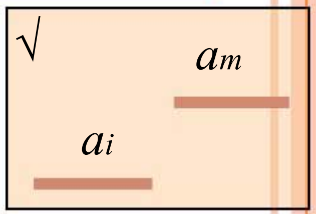



```
REC-ACTIVITY-SELECTOR(s, f, i, n)
   m ← i+1
   while m ≤ n and s_m < f_i   // Find first activity in S_{i,n+1}.
       do m ← m+1
   if m ≤ n
       then return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
   else return
```

16.1.4 A recursive greedy algorithm

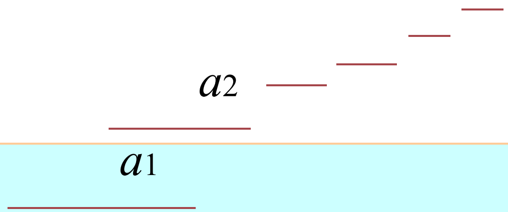



```
REC-ACTIVITY-SELECTOR(s, f, i, n)
m ← i+1
while m≤n and s_m< f_i    // Find first activity in S_{i,n+1}.
    do m ← m+1
if m≤ n
    then return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return |
```

- ***Initial call:*** REC-ACTIVITY-SELECTOR(s, *f*, 0, *n*).
- ***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≤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 first activity in S_{i,n+1}.
      do m ← m+1
if m≤ n
      then return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return
```

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

$$T(n) = m_1 + T(n - m_1) = m_1 + m_2 + T(n - m_1 - m_2)$$

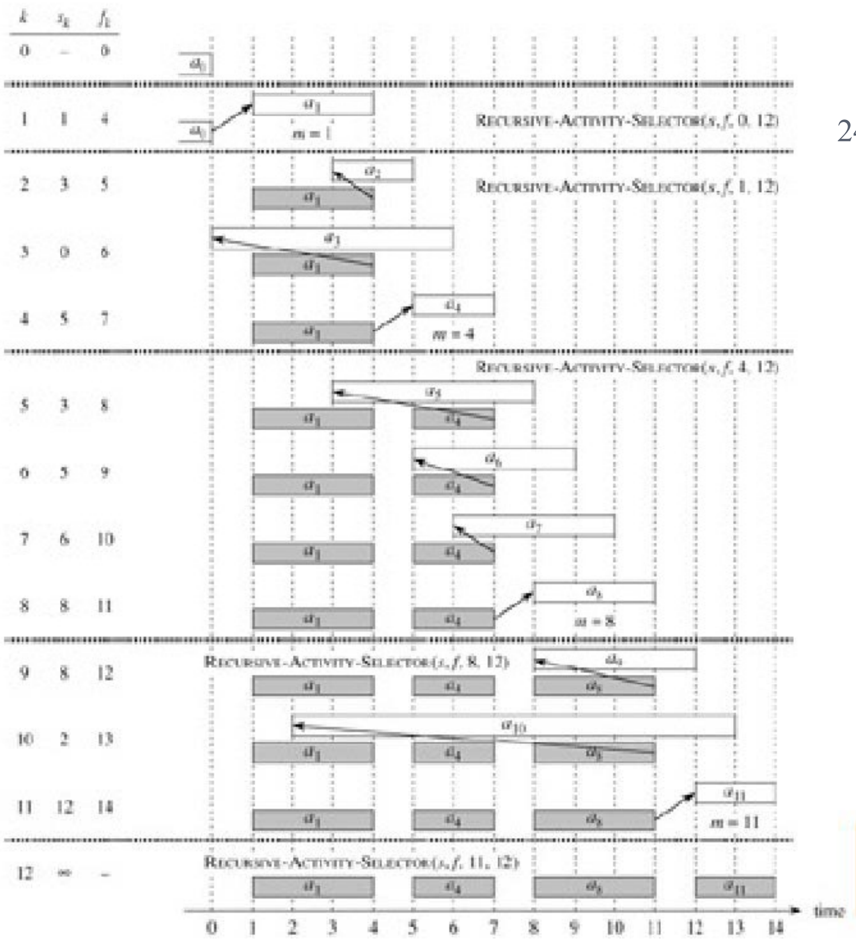
$$= m_1 + m_2 + m_3 + T(n - m_1 - m_2 - m_3) = \ \ldots$$

$$= \sum m_k + T(n - \sum m_k)$$

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

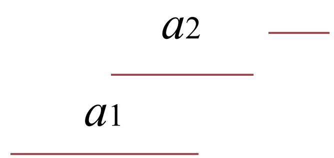
16.1.4 A recursive greedy algorithm

- ***Initial call:*** REC-ACTIVITY-SELECTOR(s, *f*, 0, *n*).
- ***Idea:*** The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ 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.



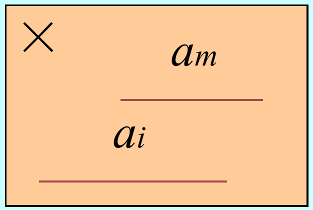
16.1.5 An iterative greedy algorithm

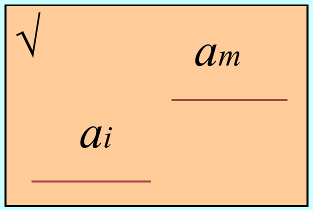
- **REC-ACTIVITY-SELECTOR is almost "tail recursive“.**
- **We easily can convert the recursive procedure to an iterative one. (Some compilers perform this task automatically)**



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

```
A ← {a1}
i ← 1
for m ← 2 to n
    do if sm ≥ fi
          then A ← A ∪ {am}
               i ← m   // ai 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.**
- **Typically, we streamline these steps （简化这些步骤）**
- **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** （简化这些步骤）

  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

  （最优问题为：做一个选择，留下一个待解的子问题）

  2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.

  （证明存在一个基于贪婪选择的最优解，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.

  （说明由贪婪选择和子问题的最优解 ⇒ 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**

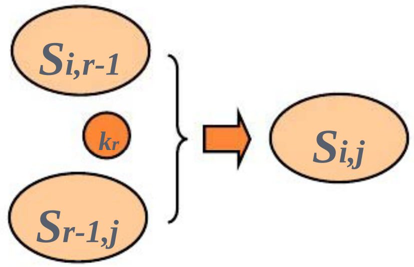
（没有一般化的规则来说明贪婪算法是否最优，但有两个基本要点）

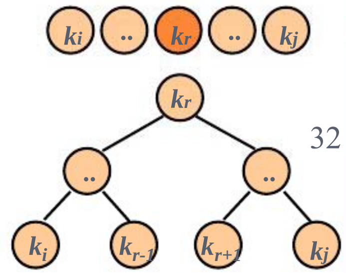
1. greedy-choice property （贪婪选择属性）

2. optimal substructure

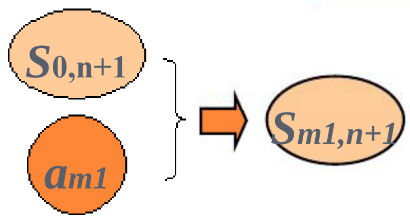
16.2.1 Greedy-choice property

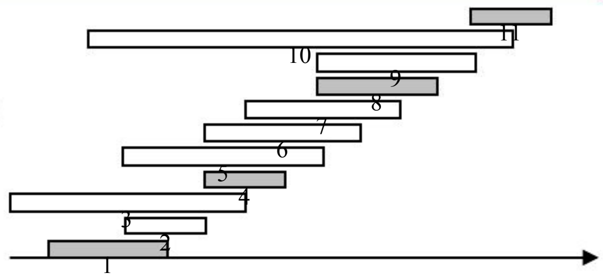
- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. （通过局部最优解可导出全局最优解）





- *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*.





- *Greedy*
  - Make a choice at each step.
  - Make the choice *before* solving the subproblems. （先作选择，再解子问题）
  - 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**

16.2.2 Optimal substructure

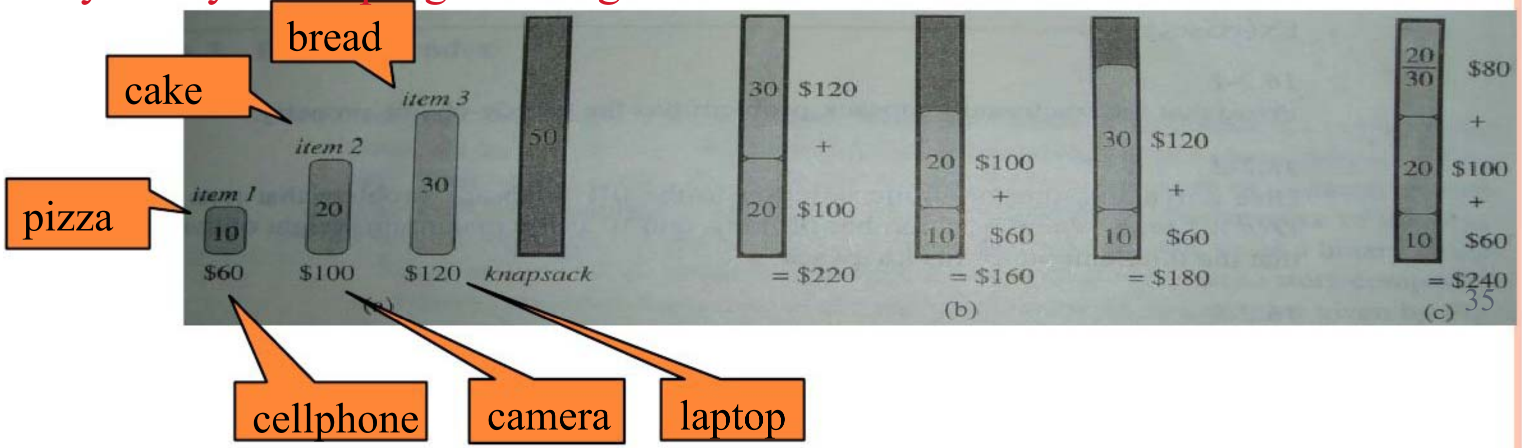
- ***optimal substructure:*** **an optimal solution to the problem contains within it optimal solutions to subproblems.**

- **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 Greedy vs. dynamic programming



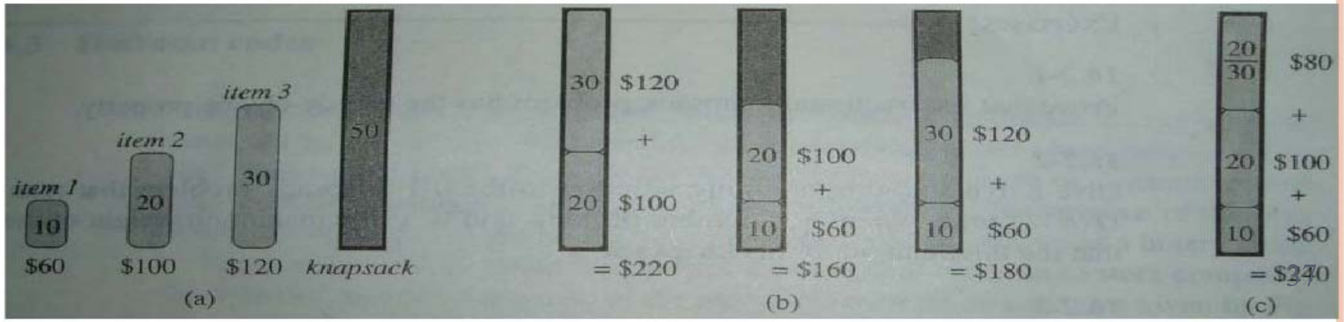
- *0-1 knapsack problem* （ 0-1背包问题，小偷问题）
  - ◆ *n* items （n 个物品）
  - ◆ Item *i* is worth $\$v_i$, weighs w_i P（物品i 价值v_i，重w_i）
  - ◆ Find a most valuable subset of items with total weight ≤W.
    （背包的最大负载量为W，如何选取物品，使得背包装的物品价值最大）
  - ◆ Have to either take an item or not take it—can't take part of it.
    （每个物品是一个整体，不能分割，因此，要么选取该物品，要么不选取）
- *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 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 : choose the most valuable load j that weighs $w_j \le W$, remove j , choose the most valuable load i that weighs $w_i \le W-w_j$**
  - **fractional: choose a weight w from item j (part of j), then remove the part, the remaining load is the most valuable load weighing at most $W-w$ that the thief can take from the $n-1$ original items plus w_j-w pounds from item j.**

    （若先选 **item *j*** 的一部分，其重 ***w***，则接下来的最优挑选方案为从余下的 ***n-1*** 个物品〔除 ***j*** 外〕和 ***j*** 的另外重 ***wj*-w** 的部分中挑选，其重量不超过 ***W-w*** ）
- **But the fractional problem has the greedy-choice property, and the 0-1 problem does not.**

16.2.3 Greedy vs. dynamic programming



- **Fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem 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
      do if wi ≤ W - load
            then take all of item i
            else take W-load of wi from item i
      add what was taken to load
i ←i + 1
```

- ***Time: O(nlg n)* to sort, *O(n)* to greedy choice thereafter.**

16.2.3 Greedy vs. dynamic programming

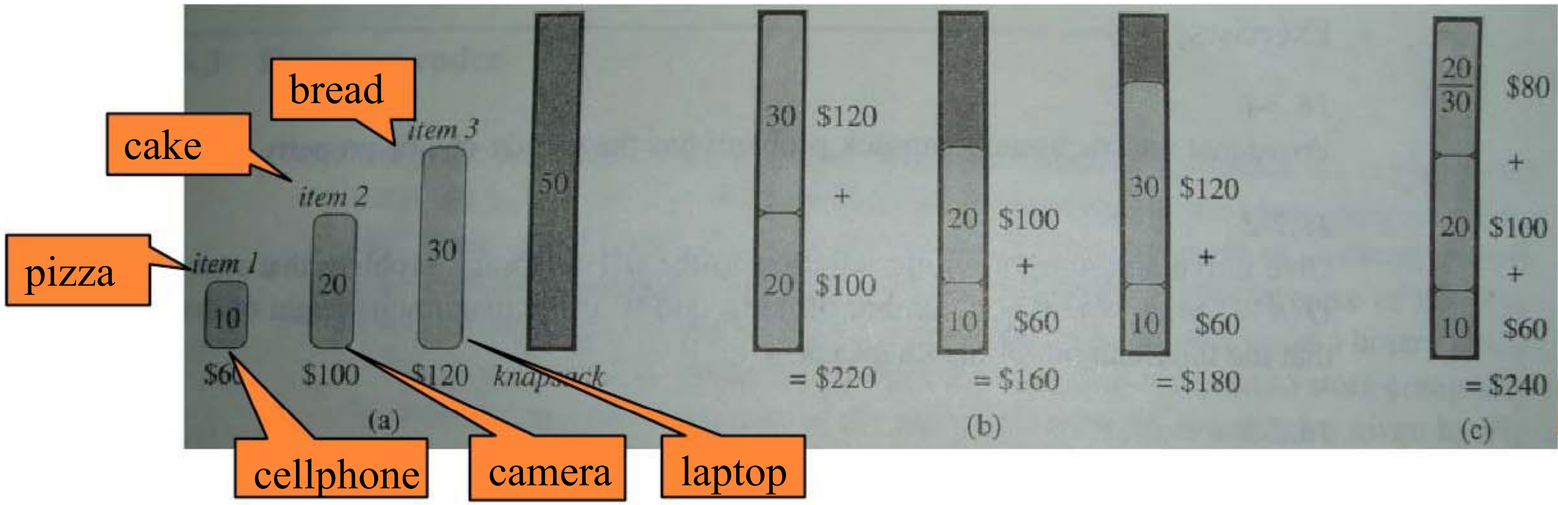
i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

- **0-1 knapsack problem has not the greedy-choice property**
- ***W*** **= 50.**
- **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）

16.3 Huffman codes

- Huffman codes: widely used and very effective technique for compressing data.
  - savings of 20% to 90%
- Consider the data to be a sequence of characters
  - Abaaaabbbdcffeaaeaeec

- 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. 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）

$$(45\cdot 1 + 13\cdot 3 + 12\cdot 3 + 16\cdot 3 + 9\cdot 4 + 5\cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

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 approximately 25%. 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. （字首码，前缀代码，前置代码〔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**（编码）**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（解码）

16.3.1 Prefix codes

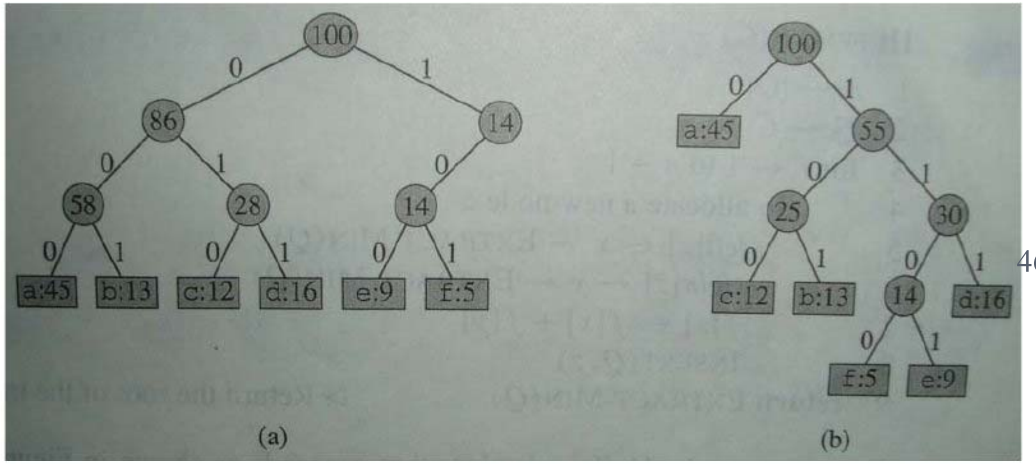
- *prefix codes* (prefix-free codes): no codeword is a prefix of some other codeword. （字首码，前缀代码，前置代码〔前缀无关码〕：没有字码是其他字码的前缀）

	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”.

- **Decoding**
  - **the process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off.**

    （为了使初始字码容易识别，解码过程需要一种前置无关码的方便表示）

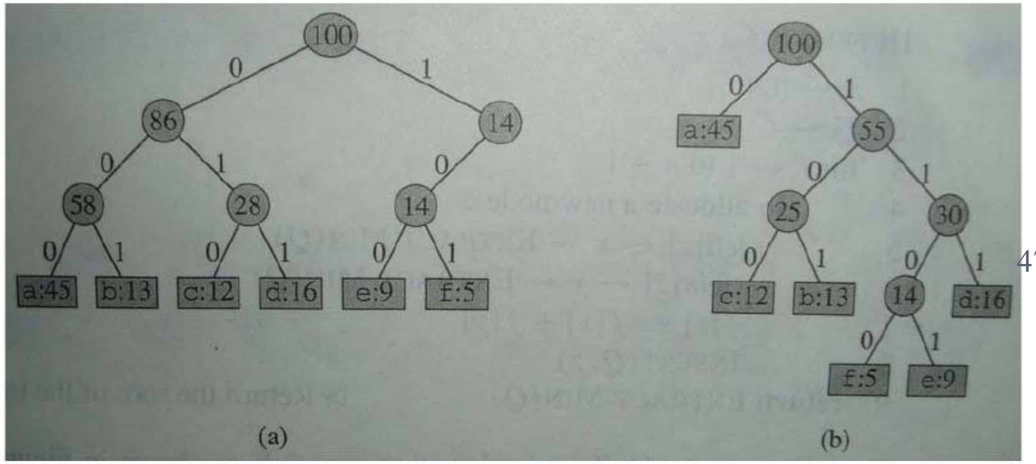
  - **A binary tree whose leaves are the given characters provides one such representation.**

    （二叉树是一种方便的表示方法，树叶为给定字符）

16.3.1 Prefix codes

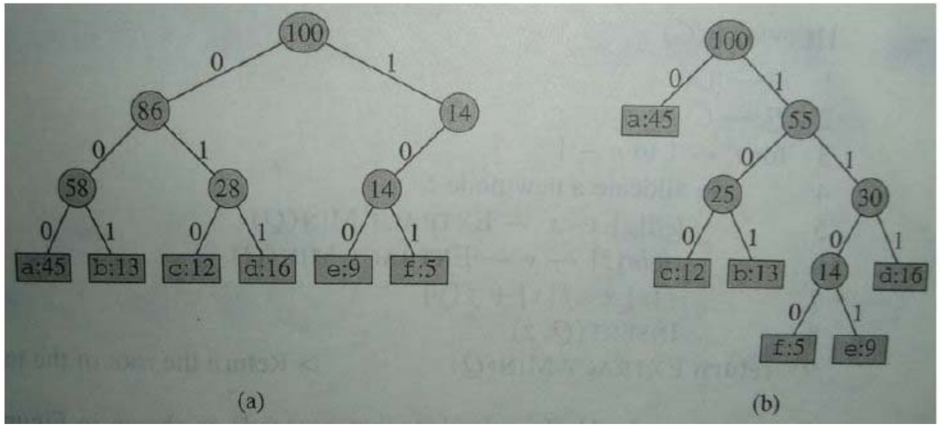
a	b	c	d	e	f
0	101	100	111	1101	1100

001011101
uniquely as 0·0·101·1101,
which decodes to "aabe".



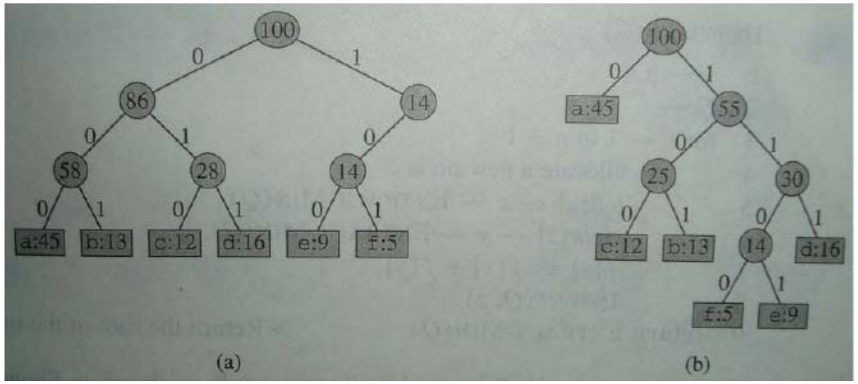
- **Decoding**
  - **We interpret the binary codeword for a character as the path from the root to that character.**
    （将字符的二叉码表示解释为一条路径：从树根到树叶）
  - **not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.**
    （不是二叉搜索树）

16.3.1 Prefix codes



- 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

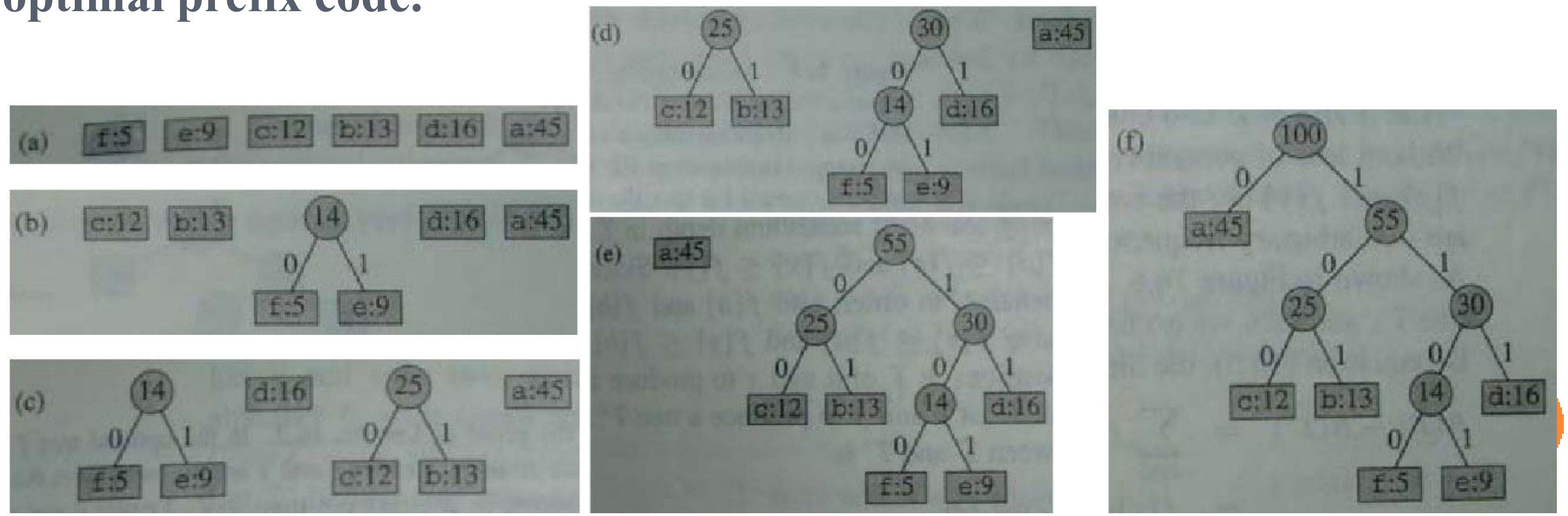
```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n - 1
        do allocate(分配) a new node z
            left[z] ← x ← EXTRACT-MIN (Q)
            right[z] ← y ← EXTRACT-MIN (Q)
            f [z] ← f [x] + f [y]
              INSERT(Q, z)
return EXTRACT-MIN(Q)  //return the root of the tree.
```


- ***C*: set of *n* characters, *c* ∈*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

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

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n - 1
    do 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
```

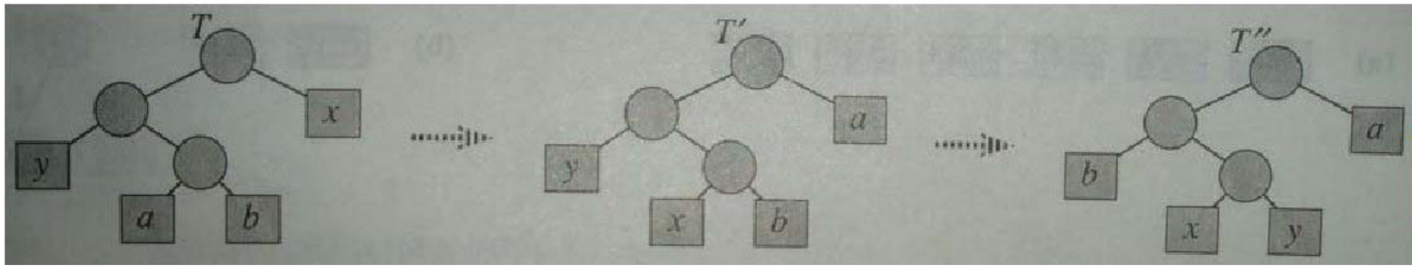


16.3.3 Correctness of Huffman's algorithm

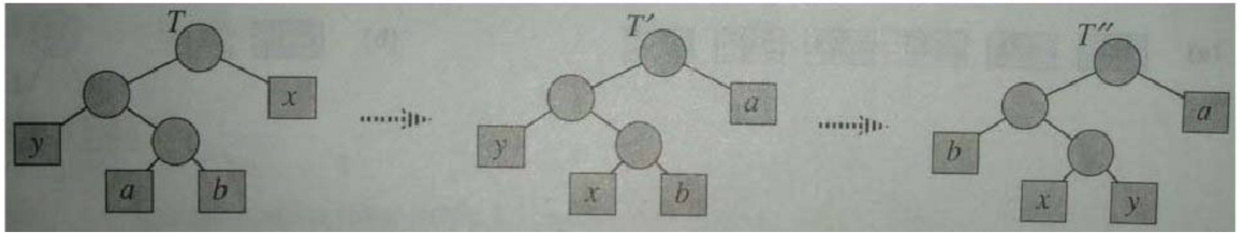
- **Problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.**
- **Lemma 16.2 (greedy-choice property)**

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.

Proof : Let a and b are sibling leaves of maximum depth in T. Assume that $f[a] \leq f[b]$, $f[x] \leq f[y]$. $f[x]$ and $f[y]$ are the two lowest leaf frequencies, $f[a]$, $f[b]$ are two arbitrary frequencies, in order,$\Rightarrow f$[x]≤f[a], $f[y] \leq f[b]$. 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

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a)$$

$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x)$$

$$= (f[x] - f[a])d_T(x) + (f[a] - f[x])d_T(a)$$

$$= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0$$

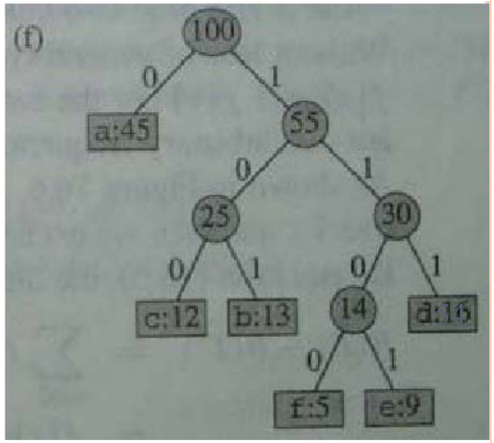
Similarly, $B(T')-B(T'') \geq 0$, therefore, $B(T'') \leq B(T)$.

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

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。）

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

Suppose that T does not represent an optimal prefix code for C. Then there exists a tree T'' such that $B(T'') < B(T)$. 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

- **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）**.**

Solution 16.2-2

- **Optimal substructure:** 背包的最大负荷 ***W*，*有 n*** 件物品，最优解 ***S*** 中的最大项数为 ***i***，则 ***S'=S-{i}必定是最大负荷为 W-w*** $_i$，物品项为1,…,i-1的最优解，且v(S)=v(S')+v$_i$
- ***c[i,w]:*** 物品项1,…,i，最大负荷 **w** 时的最优值

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ c[i - 1, w] & \text{if } w_i > w, \\ \max(v_i + c[i - 1, w - w_i], c[i - 1, w]) & \text{if } i > 0 \text{ and } w \geq w_i . \end{cases}$$

Solution 16.2-2

- 输入：W, n, v = <v_1, v_2, . . . , v_n>, w = <w_1, w_2, . . . ,w_n>
- 储存c[i, j] 在表 **c[0 . . n, 0 . .W]** *中（行优先）*， **c[n,W]** 为原始最大值

```
DYNAMIC-0-1-KNAPSACK(v,w, n, W)
for w ← 0 to W
   do c[0,w] ← 0
for i ← 1 to n
   do c[i, 0] ← 0
      for w ← 1 to W
         do if wi ≤ w
            then if vi + c[i-1,w-wi] > c[i-1,w]
                    then c[i,w] ← vi + c[i-1,w-wi]
                    else c[i,w] ← c[i-1,w]
            else c[i,w] ← c[i-1,w]
```


- **Tracing** 最优解：If ***c[i,w]=c[i-1,w]***, **item *i* is !**∈最优解，继续 **tracing with *c[i-1,w]*, else item *i*** ∈ 最优解，继续 **tracing with *c[i-1,w-w*$_i$*]*.**

Solution 16.2-4

The optimal strategy is the obvious greedy one. Starting will a full tank of gas, Professor Midas should go to the farthest gas station he can get to within *n* miles of Newark. Fill up there. Then go to the farthest gas station he can get to within *n* miles of where he filled up, and fill up there, and so on.