算法设计与分析
Algorithms Design & Analysis

第九讲：贪婪算法

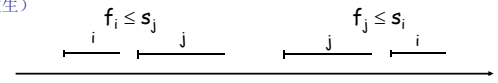
贪婪算法(Greedy Algorithms)

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems(与动态规划方法相似，是更简单的解决优化问题的方法，通常用于求解优化问题)
- Idea: When we have a choice to make, make the one that looks best right now（如有选择，选择眼下看起来最优的那个）
  - Make a locally optimal choice in hope of getting a globally optimal solution（局部最优→全局最优）
- Greedy algorithms don't always yield an optimal solution（贪婪算法不能保证一定得到最优解）
- When the problem has certain general characteristics, greedy algorithms give optimal solutions（对于具有某些特征的问题，贪婪算法有最优解）

作业选择问题 (Activity Selection)

- Schedule n **activities** that require exclusive use of a common resource(对n个作业进行排程，这些作业在执行期间需要专用某个共同的资源)

  $S = \{a_1, \ldots, a_n\}$ – set of activities（作业集合）
- a_i needs resource during period $[s_i, f_i)$（作业执行时间）
  - s_i = **start time（起始时刻）** and f_i = **finish time** of activity a_i **（完成时刻）**
  - $0 \le s_i < f_i < \infty$
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap（如果是下列情况之一，则称作业a_i和a_j没有冲突，在资源共享上可相互谐调。一、 a_i在a_j发生前结束。二、 a_i在a_j结束后发生）



作业选择问题 (Activity Selection Problem)

Select the largest possible set of nonoverlapping (***mutually compatible***) activities.（选出最多个不冲突的作业）

E.g.:

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

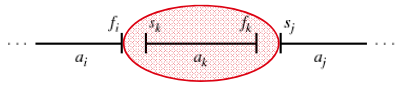
- Activities are sorted in increasing order of finish times（作业按完成的先后顺序排列）
- A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$（可协调作业集合的例子）
- Maximal set of mutually compatible activities:

  $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$（最多可协调作业的集合）

优化基础（Optimal Substructure）

- Define the space of subproblems:(定义子问题空间)

  $S_{ij} = \{ a_k \in S : f_i \le s_k < f_k \le s_j \}$
  - activities that start after a_i finishes and finish before a_j starts（在a_i结束之后和a_j开始之前的所有作业）



- Add fictitious activities $S = S_{0,n+1}$ entire space of activities（完整问题空间）
  - $a_0 = [-\infty, 0)$ （假设作业）
  - $a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$
  - Range for S_{ij} is $0 \le i, j \le n + 1$

问题描述(Representing the Problem)

- In a set S_{ij} we assume that activities are sorted in increasing order of finish times:(问题空间S_{ij},其中的作业按照完成时间递增的顺序排练)

  $f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$
- What happens if $i \ge j$?(如果$i \ge j$，则S_{ij}为空)
  - For an activity $a_k \in S_{ij}$: $f_i \le s_k < f_k \le s_j < f_j$

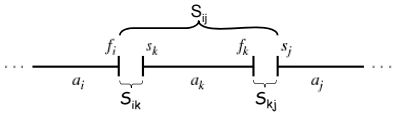
    contradiction with $f_i \ge f_j$! （互相矛盾）

  $\Rightarrow S_{ij} = \varnothing$ (the set S_{ij} must be empty!)
- We only need to consider sets S_{ij} with

  $0 \le i < j \le n + 1$（因此只考虑这种情况）

优化基础（Optimal Substructure）

- Subproblem:（子问题）
  - Select a maximum size subset of mutually compatible activities from set S_{ij}（从S_{ij}挑选互不冲突的最多作业的集合）
- Assume that a solution to the above subproblem includes activity a_k (S_{ij} is non-empty)（假定集合中有作业a_k）

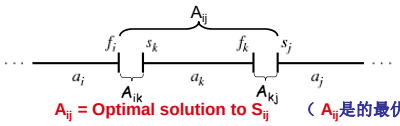


Solution to S_{ij} = (Solution to S_{ik}) ∪ {a_k} ∪ (Solution to S_{kj})（解的组成）

$|$Solution to $S_{ij}|$ = $|$Solution to $S_{ik}|$ + 1 + $|$Solution to $S_{kj}|$（数目）

7

优化结构（Optimal Substructure）



A_{ij} = Optimal solution to S_{ij}　（A_{ij}是的最优解S_{ij}）

- **Claim: Sets A_{ik} and A_{kj} must be optimal solutions**（则A_{ik}和A_{kj}必然是最优解，反证法）
- **Assume ∃ A_{ik}' that includes more activities than A_{ik}**

  Size[A_{ij}'] = Size[A_{ik}'] + 1 + Size[A_{kj}] > Size[A_{ij}]

⇒ **Contradiction: we assumed that A_{ij} is the maximum # of activities taken from S_{ij}**（如果存在更优的A_{ik}'，则有Size[A_{ij}'] > Size[A_{ij}]，与A_{ij}是最优解矛盾）

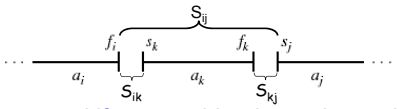
8

递归解（Recursive Solution）

- Any optimal solution (associated with a set S_{ij}) contains within it optimal solutions to subproblems S_{ik} and S_{kj} (S_{ij}的最优解由子问题S_{ik}和S_{kj}的最优解组成）
- c[i, j] = size of maximum-size subset of mutually compatible activities in S_{ij} (c[i, j]表示S_{ij}的最优解集合中作业的数目）
- If $S_{ij} = \varnothing \Rightarrow$ c[i, j] = 0 ($i \geq j$)

9

递归解（Recursive Solution）



If $S_{ij} \neq \varnothing$ and if we consider that a_k is used in an optimal solution (maximum-size subset of mutually compatible activities of S_{ij})（$S_{ij} \neq \varnothing$，a_k是最优解集合中的一个作业）

$$c[i, j] = c[i,k] + c[k, j] + 1$$

10

递归解（Recursive Solution）

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

- There are j – i – 1 possible values for k (k有j – i – 1 种选择)
  - k = i+1, …, j – 1
  - a_k cannot be a_i or a_j　(from the definition of S_{ij})

    $S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$（由$S_{ij}$定义可以保证，$a_k$不是$a_i$，也不是$a_j$）
  - We check all the values and take the best one（穷举k，选最优）

We could now write a dynamic programming algorithm（动态规划）

11

定理（Theorem）

Let $S_{ij} \neq \varnothing$ and a_m the activity in S_{ij} with the earliest finish time:（S_{ij}是非空的作业集，是a_m其中最先完成的一个）

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

Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}（a_m是某个最优解中的一个作业）
   - There exists some optimal solution that contains a_m
2. $S_{im} = \varnothing$（S_{im}为空，S_{mj}是唯一的非空子问题）
   - Choosing a_m leaves S_{mj} the only nonempty subproblem

12

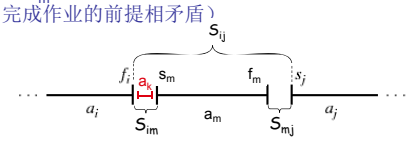
反证法证明第2点（Proof）

2. Assume $\exists\, a_k \in S_{im}$

   $f_i \le s_k < f_k \le s_m < f_m$

   $\Rightarrow$ $f_k < f_m$ contradiction !

   a_m did not have the earliest finish time （与a_m是最先完成作业的前提相矛盾）

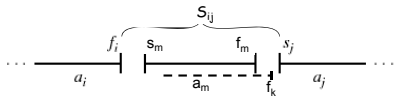


$\Rightarrow$ There is no $a_k \in S_{im} \Rightarrow S_{im} = \varnothing$

证明第1点（Proof）

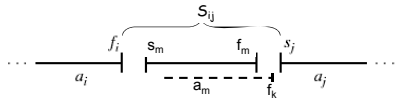
1. **a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}（a_m是某个最优解中的一个作业）**

- A_{ij} = optimal solution for activity selection from S_{ij} (A_{ij}是S_{ij}的最优解）
  - Order activities in A_{ij} in increasing order of finish time（ A_{ij}中的作业按完成时间递增顺序排列）
  - Let a_k be the first activity in $A_{ij} = \{a_k, \ldots\}$ （ a_k是A_{ij}中的第一个作业）
- If $a_k = a_m$ Done!（如果$a_k = a_m$，证明完毕）



证明（Proof）

- Otherwise, replace a_k with a_m (resulting in a set A_{ij}')（否则，用a_m取代a_k，得到A_{ij}'）
  - since $f_m \le f_k$ the activities in A_{ij}' will continue to be compatible （因为$f_m \le f_k$， A_{ij}'中的作业不会冲突）
  - A_{ij}' will have the same size with $A_{ij} \Rightarrow a_m$ is used in some maximum-size subset（ A_{ij}'是一个最优解）



贪婪法（Greedy Approach）

- To select a maximum size subset of mutually compatible activities from set S_{ij}:（从S_{ij}选择不冲突的最多作业）
  - Choose $a_m \in S_{ij}$ with earliest finish time (greedy choice)（选择S_{ij}中最先完成的作业a_m，此即贪婪选择）
  - Add a_m to the set of activities used in the optimal solution（将a_m加入到最优解集合）
  - Solve the same problem for the set S_{mj} （继续求解S_{mj}）
- From the theorem
  - By choosing a_m we are guaranteed to have used an activity included in an optimal solution （由定理保证选择a_m，是将其加入到了一个可能的最优解）

    $\Rightarrow$ We do not need to solve the subproblem S_{mj} before making the choice!（在作出选择时不必求解S_{mj}）
  - The problem has the **GREEDY CHOICE** property（贪婪选择特性）

定理用途（Why is the Theorem Useful?）

	Dynamic programming （动态规划）	Using the theorem （使用定理）
Number of subproblems in the optimal solution （子问题数目）	2 subproblems: S_{ik}, S_{kj} （2个）	1 subproblem: S_{mj} $S_{im} = \varnothing$（一个子问题，另一为空）
Number of choices to consider（选择情况）	j – i – 1 choices （ j – i – 1 个选择）	1 choice: the activity with the earliest finish time in S_{ij} （1 个选择）

- Making the greedy choice (the activity with the earliest finish time in S_{ij})（贪婪选择：每次选择S_{ij}中最早完成的作业）
  - Reduce the number of subproblems and choices（减少了子问题的数目）
  - Solve each subproblem in a top-down fashion（可以自顶向下解决问题）

初始问题（ Original problem ）

- The original problem: find the maximum subset of mutually compatible activities for $S = S_{0, n+1}$（在$S = S_{0, n+1}$中求不冲突的最多作业）
- Activities are sorted by increasing finish time（递增顺序）

  $a_0, a_1, a_2, a_3, \ldots, a_{n+1}$
- We always choose an activity with the earliest finish time （选择最先完成的作业）
  - Greedy choice maximizes the unscheduled time remaining（使得有最多的节余时间选择其他作业）
  - Finish time of activities selected is strictly increasing（严格递增）

递归贪婪算法（A Recursive Greedy Algorithm）

```
REC-ACT-SEL (s, f, i, n)
1.  m ← i + 1
2.  while m ≤ n and s_m < f_i ►在S_{i,n+1}确定第一个与a_i不冲突的事件
3.        do m ← m + 1
4.  if m ≤ n
5.    then return {a_m} ∪ REC-ACT-SEL(s, f, m, n)
6.  else return ∅
```

- Activities are ordered in increasing order of finish time （作业按完成时间递增顺序排列）
- Running time: Θ(n) – each activity is examined only once （运行开销）
- **Initial call:** REC-ACT-SEL(s, f, 0, n) (注意初始调用形式)

增量算法（An Incremental Algorithm）

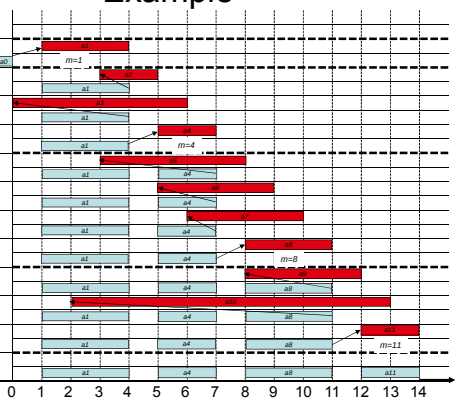
Alg.: GREEDY-ACTIVITY-SELECTOR(s, f, n)

1. $A \leftarrow \{a_1\}$
2. $i \leftarrow 1$
3. **for** $m \leftarrow 2$ **to** n 作业am与ai不冲突
4. **do if** $s_m \geq f_i$ ► activity a_m is compatible with a_i
5. **then** $A \leftarrow A \cup \{a_m\}$
6. $i \leftarrow m$ ► a_i is most recent addition to A
7. **return** A

- Assumes that activities are ordered in increasing order of finish time （递增顺序）
- Running time: Θ(n) – each activity is examined only once（运行开销）

Example

k	s_k	f_k
0	-	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	-



贪婪算法步骤

1. Determined the optimal substructure of the problem（确定问题的优化结构）
2. Developed a recursive solution（得到递归解）
3. Proved that one of the optimal choices is the greedy choice（证明某个最优选择是贪婪选择）
4. Showed that all but one of the subproblems resulted by making the greedy choice are empty（贪婪选择将产生唯一一个非空子问题）
5. Developed a recursive algorithm that implements the greedy strategy（用递归算法实现贪婪策略）
6. Converted the recursive algorithm to an iterative one（将递归算法转换为迭代算法）

动态规划与贪婪算法比较

- **Dynamic programming**
  - **We make a choice at each step（每步选择）**
  - **The choice depends on solutions to subproblems（选择与子问题解相关）**
  - **Bottom up solution, from smaller to larger subproblems（自底向上，即从规模小的子问题逐步求解规模大的子问题）**
- **Greedy algorithm**
  - **Make the greedy choice and THEN （首先作出贪婪选择）**
  - **Solve the subproblem arising after the choice is made （求解贪婪选择后产生的唯一子问题）**
  - **The choice we make may depend on previous choices, but not on solutions to subproblems（后续贪婪选择与前面的选择有关，但与子问题的解无关）**
  - **Top down solution, problems decrease in size（自顶向下，问题规模逐步缩小）**

Huffman 编码(Huffman Codes)

- Idea:（利用字符出现的频率，建立表示每个字符的最优方法，用于文件压缩）
  - Use the frequencies of occurrence of characters to build a optimal way of representing each character

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- **Variable-Length Codes（变长编码）**
  - Assign short codewords to frequent characters and long codewords to infrequent characters （频率高的字符被赋予短编码，频率低的字符被赋予长编码）
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

前缀码（Prefix Codes）

- Prefix codes:
  - Codes for which no codeword is also a prefix of some other codeword （一个字符的编码必须不能是另一个字符编码的前缀）
  - Better name would be “prefix-free codes”（更贴切的名字）
- We can achieve optimal data compression using prefix codes（利用前缀码可以进行文件压缩）

25

编码（Encoding）

Our next goal is to develop a code that represents a given text as compactly as possible.（通过合理的编码尽可能的压缩文件）

A standard encoding is ASCII, which represents every character using 7 bits: （标准ASCII码）

例：“An English sentence”

1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E)
1101110 (n) 1100111 (g) 1101100 (l) 1101001 (i)
1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s)
1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e)
1101110 (n) 1100011 (c) 1100101 (e)= 133 bits ≈ 17 bytes

Of course, this is wasteful because we can encode 12 characters in 4 bits: （上例中共有12个字符，因此等长编码只需要4 bits）

‹space› = 0000 A = 0001 E = 0010 c = 0011 e = 0100
g = 0101 h = 0110 i = 0111 l = 1000 n = 1001 s = 1010
t = 1011

Then we encode the phrase as

0001 (A) 1001 (n) 0000 ( ) 0010 (E) 1001 (n) 0101 (g)
1000 (l) 0111 (i) 1010 (s) 0110 (h) 0000 ( ) 1010 (s)
0100 (e) 1001 (n) 1011 (t) 0100 (e) 1001 (n) 0011 (c)
0100 (e)

This requires 76 bits ≈ 10 bytes

An even better code is given by the following encoding: （利用前缀码将获得更好的表示）

‹space› = 000 A = 0010 E = 0011 s = 010 c = 0110
g = 0111 h = 1000 i = 1001 l = 1010 t = 1011 e = 110
n = 111

Then we encode the phrase as

0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g)
1010 (l) 1001 (i) 010 (s) 1000 (h) 000 ( ) 010 (s) 110 (e)
111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c) 110 (e)

This requires 65 bits ≈ 9 bytes

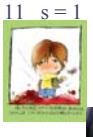
为什么要采用前缀码（Why Prefix Codes?）

Prefix codes:

No character is the prefix of another character.（一个字符的编码必须不能是另一个字符编码的前缀）

Code: a = 01 m = 10 n = 111 o = 0 r = 11 s = 1 t = 0011

“You are a star in my mind.”
“1 0011 01 11”

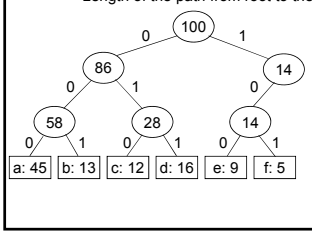


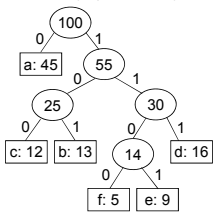
100110111 = 10 0 11 0 111 = “moron”
“You are a moron in my mind.”

Non-prefix codes are ambiguous.（非前缀码导致二义性）

前缀码表示（Prefix Code Representation）

- Binary tree whose leaves are the given characters（二分树，叶节点表示字符）
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”（0表示左，1表示右，从根到叶即为字符编码）
- Length of the codeword （从根到叶路径长度即是编码长度）
  - Length of the path from root to the character leaf (depth of node)

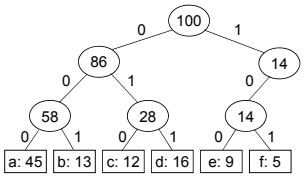




30

最优编码（Optimal Codes）

- An optimal code is always represented by a **full binary tree（最优编码通过完整二分树表示）**
  - Every non-leaf has two children（每个非叶节点都有两个子节点）
  - Fixed-length code is not optimal, variable-length is（定长编码不是最优的，变长编码是的）



31

最优编码（Optimal Codes）

- How many bits are required to encode a file?（编码一个文件需要多少位？）
  - Let C be the alphabet of characters （字符集）
  - Let $f(c)$ be the frequency of character c （频率）
  - Let $d_T(c)$ be the depth of c's leaf in the tree T corresponding to a prefix code （字符编码长度）

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$ the cost of tree T

32

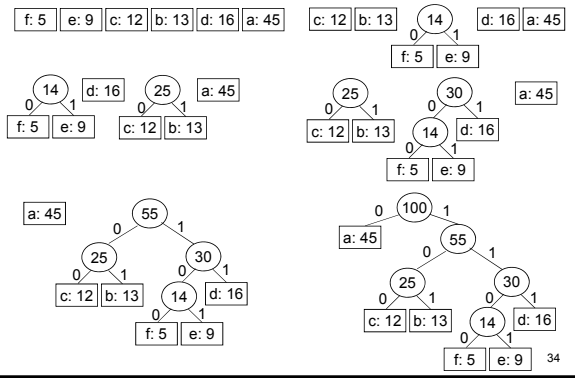
构造Huffman 编码（Constructing a Huffman Code）

- A greedy algorithm that constructs an optimal prefix code called a Huffman code （贪婪算法，最优Huffman 编码）
- Assume that: 假设
  - C is a set of n characters （字符C集含有n个字符）
  - Each character has a frequency $f(c)$ （字符频率）
  - The tree T is built in a bottom up manner （自底向上构建完整二分树T）
- Idea:
  - Start with a set of $|C|$ leaves （从叶节点开始）
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies （每次合并频率最小的两个叶节点，生成的新节点的频率为两者之和）
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects （在新节点和其他节点中再找出两个频率最小的节点，合并生成新节点）

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

33

Example



34

Huffman 编码算法（Building a Huffman Code）

Alg.: HUFFMAN(C) Running time: $O(n \lg n)$

```
1. n ← |C|
2. Q ← C                                   ← O(n)
3. for i ← 1 to n − 1                      ┐
4.     do allocate a new node z            │
5.        left[z] ← x ← EXTRACT-MIN(Q)     │ O(nlgn)
6.        right[z] ← y ← EXTRACT-MIN(Q)    │
7.        f[z] ← f[x] + f[y]               │
8.        INSERT (Q, z)                    ┘
9. return EXTRACT-MIN(Q)
```

35

贪婪选择特性Greedy Choice Property

Lemma: Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. （C字符集，字符$c \in C$频率为$f[c]$， x和y是字符集中频率最低的两个字符）

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.（存在C的最优编码，字符x和y的编码具有相同的长度且只有最后一位不同）

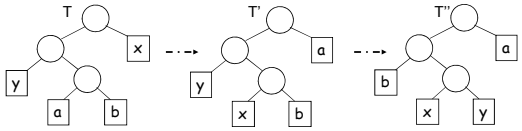
36

贪婪选择证明（Proof of the Greedy Choice）

- Idea:（思路：构建出这样的一棵树T）
  - Consider a tree T representing an arbitrary optimal prefix code（考虑表示任意最优前缀码的树T）
  - Modify T to make a tree representing another optimal prefix code in which x and y will appear as sibling leaves of maximum depth（修改T，使之表示另一最优前缀码，之中x和y是具有最大深度的同胞叶节点）

⇒ The codes of x and y will have the same length and differ only in the last bit（ x和y的编码长度相同且只有最后一位不同）

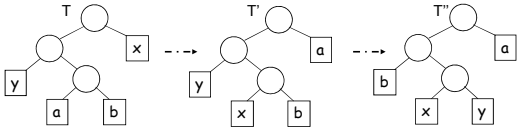
Proof of the Greedy Choice (cont.)



- a, b – two characters, sibling leaves of maximum depth in T（a和b，树T中的两个最大深度的同胞页节点）
- Assume: f[a] ≤ f[b] and f[x] ≤ f[y]（不失一般性，可假设f[a] ≤ f[b] 和 f[x] ≤ f[y]）
- f[x] and f[y] are the two lowest leaf frequencies, in order （ f[x] 和 f[y]是最低的两个频率）

  ⇒ f[x] ≤ f[a] and f[y] ≤ f[b]
- Exchange the positions of a and x (T') and of b and y (T'')（交换）

Proof of the Greedy Choice (cont.)



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

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

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

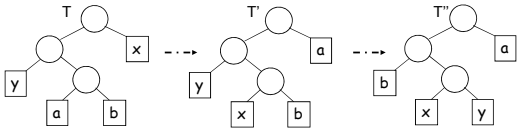
$$= \underbrace{(f[a] - f[x])}_{\geq 0}\ \underbrace{(d_T(a) - d_T(x))}_{\geq 0}$$

x is a minimum frequency leaf

a is a leaf of maximum depth

≥ 0（证明交换后生成的新树代价降低）

Proof of the Greedy Choice (cont.)



B(T) – B(T') ≥ 0

Similarly, exchanging y and b does not increase the cost（同理，交换y和b不会增加代价）

⇒ B(T') – B(T'') ≥ 0 ⇒ B(T'') ≤ B(T)

and since T is optimal ⇒ B(T) ≤ B(T'')（又，T是最优的，所以B(T) ≤ B(T'')）

⇒ B(T) = B(T'') ⇒ T'' is an optimal tree, in which x and y are sibling leaves of maximum depth（ T''即是满足要求的树）

讨论（Discussion）

- Greedy choice property:
  - Building an optimal tree by mergers can begin with the greedy choice: merging the two characters with the lowest frequencies（构造最优前缀码树可以通过贪婪选择合并的方法：每次合并两个频率最小的字符）
  - The cost of each merger is the sum of frequencies of the two items being merged（合并产生新节点的频率是两个被合并的节点的频率之和）
  - Of all possible mergers, HUFFMAN chooses the one that incurs the least cost（对于所有的合并，均是选择了最小代价）

背包问题（The Knapsack Problem）

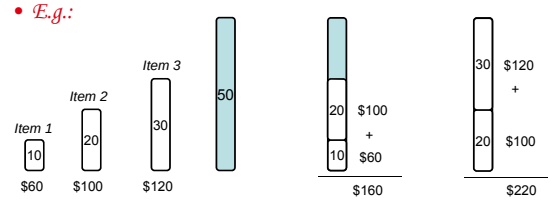
- **The 0-1 knapsack problem（0/1背包问题）**
  - Items must be taken entirely or left behind（要么拿走，要么放弃）
- **The fractional knapsack problem（部分背包问题）**
  - The thief can take fractions of items（可部分那走）

0/1背包问题(The 0-1 Knapsack Problem)

- **Thief has a knapsack of capacity W(背包的可容重量为W)**
- **There are n items: for i-th item value v_i and weight w_i (n个物件，第i个的价值为v_i，重量为w_i)**
- **Goal:**
  - **find x_i such that for all x_i = {0, 1}, i = 1, 2, .., n**
    $\sum w_i x_i \le W$ and （不超重）
    $\sum x_i v_i$ is maximum （价值最大）

0-1 Knapsack - Greedy Strategy

- *E.g.:*



- None of the solutions involving the greedy choice (item 1) leads to an optimal solution
  - The greedy choice property does not hold（**如果你作出贪婪选择，即装入平均价值最高的物件，你最终不可能获得最大化的价值总量**）

部分背包问题(Fractional Knapsack Problem)

- Knapsack capacity: W **(背包的可容重量为W)**
- There are n items: the i-th item has value v_i and weight w_i**(n个物件，第i个的价值为v_i，重量为w_i)**
- Goal:
  - find x_i such that for all $0 \le x_i \le 1$ (分数,部分物件), i = 1, 2, .., n
    $\sum w_i x_i \le W$ and (不超重)
    $\sum x_i v_i$ is maximum (价值最大)

部分背包问题(Fractional Knapsack Problem)

Greedy strategy :

- **Pick the item with the maximum value per pound v_i/w_i(挑选出单位重量价值最高的物件)**
- **If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound(在容量许可下,尽量转载单位重量价值最高的物件，直至拿光）**
- **It is good to order items based on their value per pound（然后依次装载其余物件中价值最高的物件，物件价值递减排列）**

$$\frac{v_1}{w_1} \ge \frac{v_2}{w_2} \ge \ldots \ge \frac{v_n}{w_n}$$

部分背包问题(Fractional Knapsack Problem)

Alg.: Fractional-Knapsack (W, v[n], w[n])

1. While w > 0 and as long as there are items remaining(背包有容量)
2. pick item with maximum v_i/w_i（选择最有价值之物）
3. $x_i \leftarrow \min(1, w/w_i)$（拿走整个，还是部分）
4. remove item i from list(从其余物件中装载)
5. $w \leftarrow w - x_i w_i$

- w – the amount of space remaining in the knapsack(容量空余)

Fractional Knapsack - Example

- *E.g.:*