GREEDY ALGORITHM

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

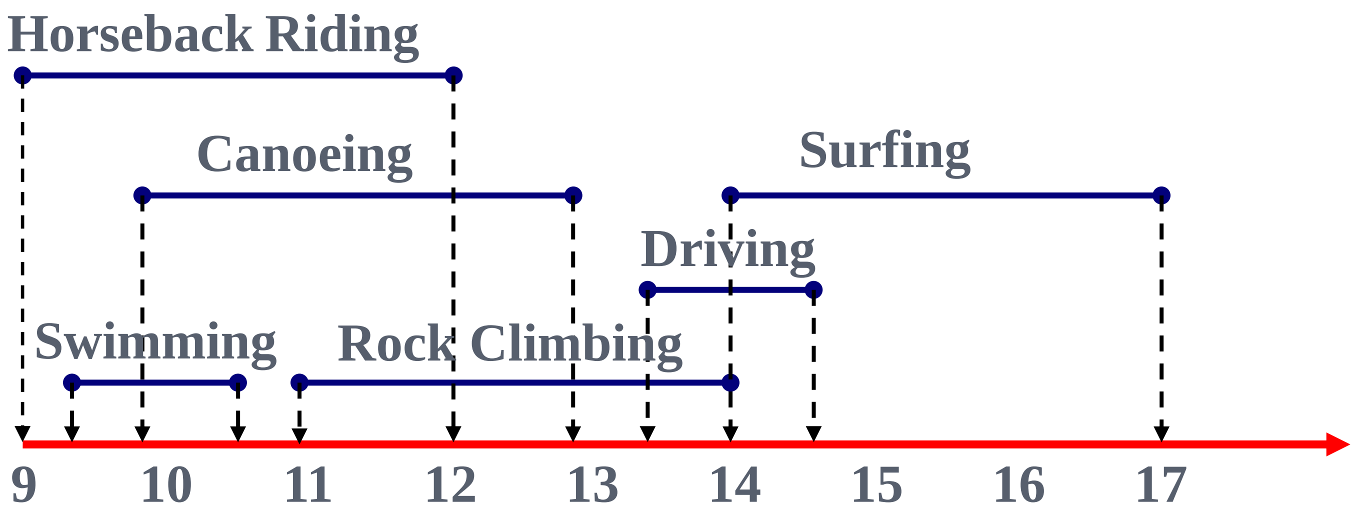
Greedy Algorithms

- 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.

OUTLINE

- Activity-Selection Problem
- Basic Elements of the GA
  - Knapsack Problem
- Data Compression (Huffman) Codes

First example: Activity Selection



- **How to make an arrangement to have the more activities?**

➢ **S1. Shortest activity first**
  - **Swimming, Driving**

➢ **S2. First starting activity first**
  - **Horseback Riding, Driving**

➢ **S3. First finishing activity first**
  - **Swimming, Rock Climbing, Surfing**

An Activity-selection Problem

- *n activities* require *exclusive* use of a common resource.

Example: scheduling the use of a classroom.

- Set of activities $S = \{a_1, a_2, . . . , a_n\}$.
- 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.
- Other objectives: Maximize income rental fees, …

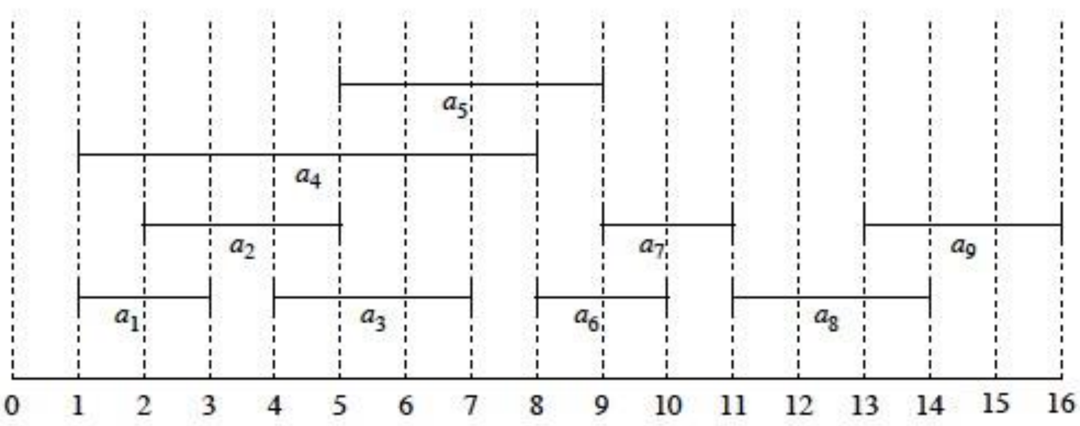
AN ACTIVITY-SELECTION PROBLEM






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

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



Maximum-size mutually compatible set:

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

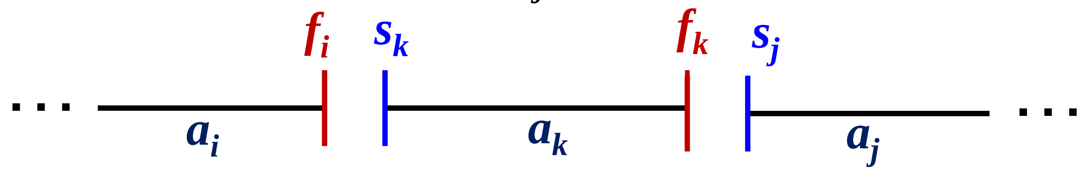
Not unique: also

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

Optimal Substructure of Activity Selection

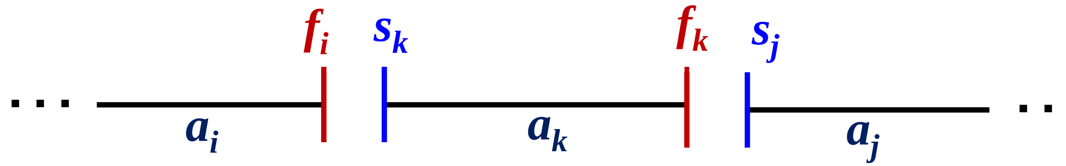
Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ = 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
  - 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 "$\infty+1$" in a_{n+1}.
- Then $S = S_{0,n+1}$. Range for S_{ij} is $0 \leq i, j \leq n+1$.

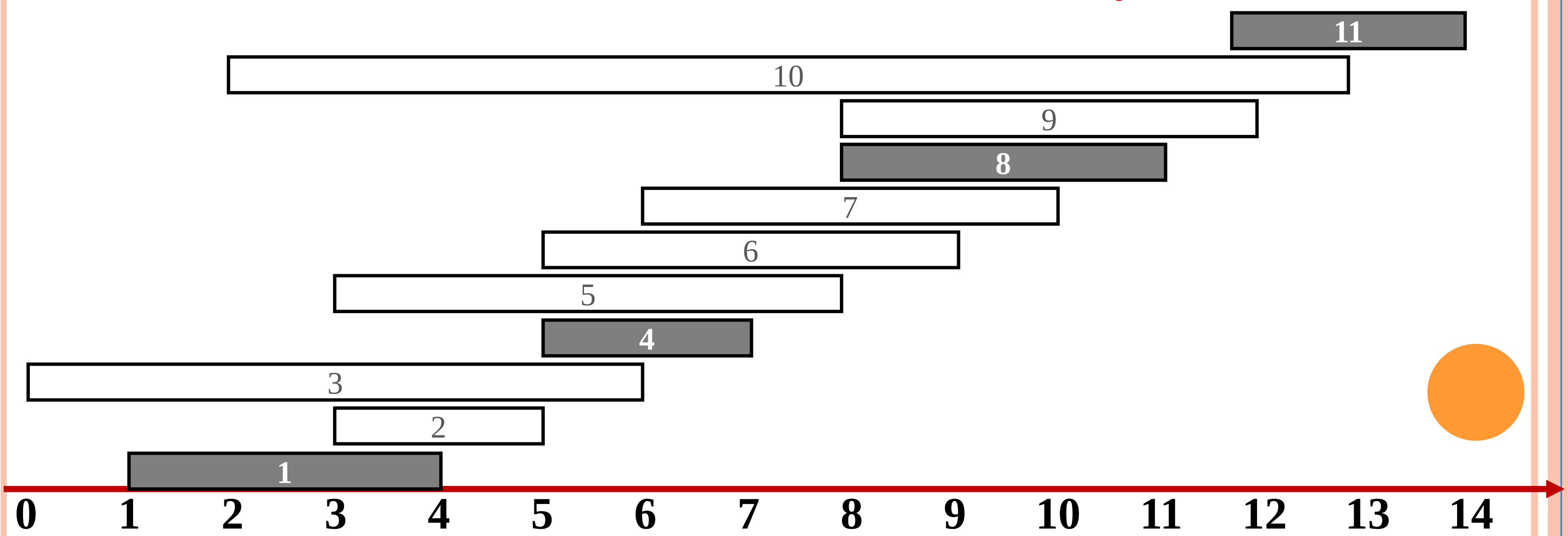
OPTIMAL SUBSTRUCTURE OF ACTIVITY SELECTION



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:

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



Optimal Substructure of Activity Selection

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

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

Proof

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

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

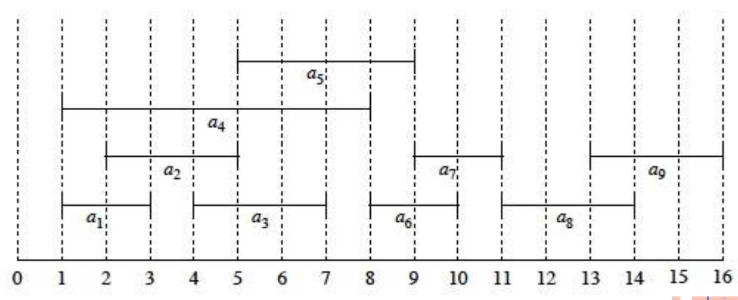
But $i \ge j \Rightarrow f_i \ge f_j$. Contradiction. ■

- So only need to worry about

$$S_{ij} \text{ with } 0 \le i < j \le n + 1.$$

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

Optimal Substructure of Activity Selection

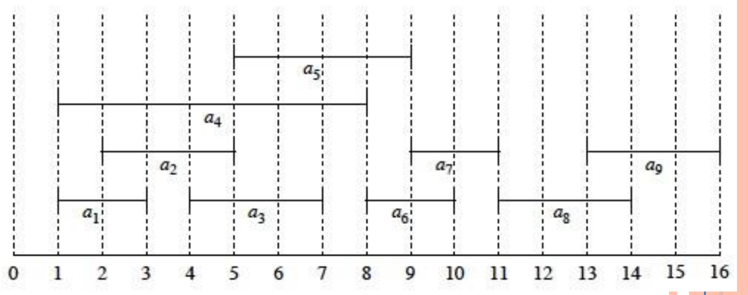


- Suppose that a solution to S_{ij} includes a_k :
  - Have 2 sub-prob
    - S_{ik} (*start* after a_i finishes, *finish* before a_k starts)
    - S_{kj} (*start* after a_k finishes, *finish* before a_j starts)
- Solution to S_{ij} = (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj})

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 of Activity Selection



- 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.
  - We can use usual cut-and-paste argument to solve it.
- Let A_{ij} = optimal solution to S_{ij} , we have

$$A_{ij} = A_{ik} \cup \{a_k\} \cup \mathrm{A}_{kj} , \qquad (2)$$

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

A Recursive Solution

- Let $c[i, j\,]$ = size of maximum-size subset of mutually compatible activities in S_{ij} .

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

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

$$c[i,j] = \begin{cases} 0\,, & \text{if } S_{ij} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\}\,, & \text{if } S_{ij} \neq \emptyset \end{cases} \qquad (3)$$

Why this range of k?

Because $S_{ij} = \{a_k \in S\colon f_i \leq s_k < f_k \leq s_j\,\} \Rightarrow a_k$ can't be a_i or a_j .

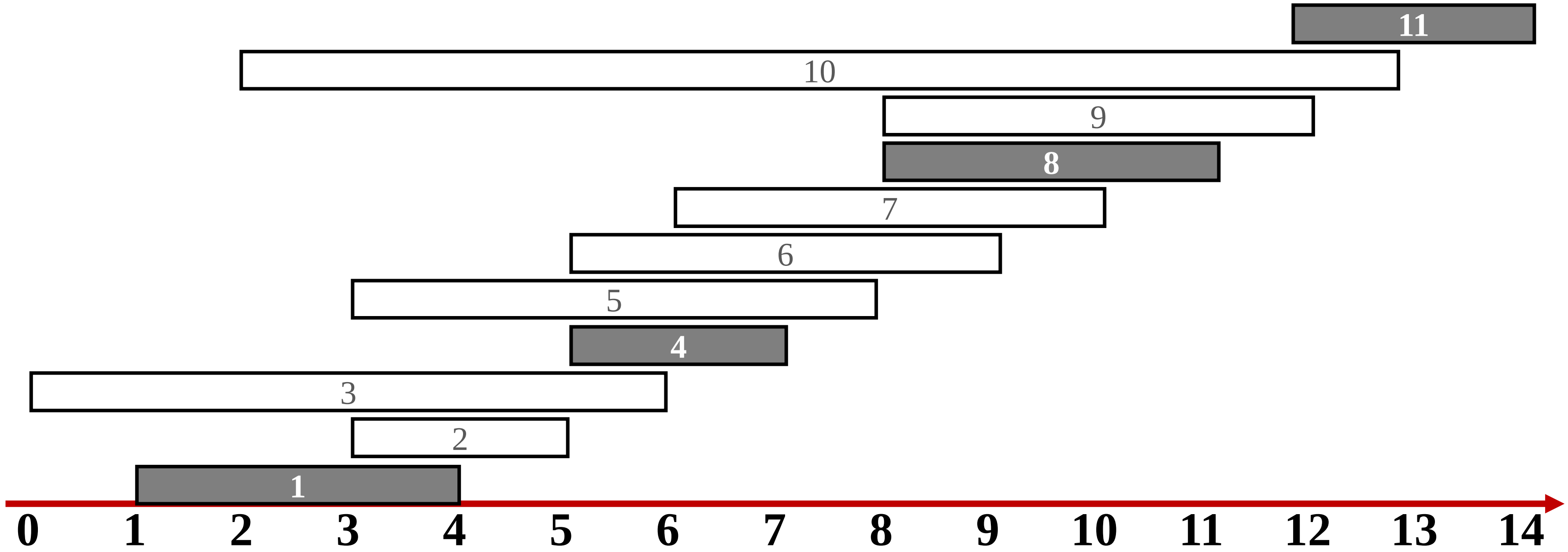
Converting a DP solution to a Greedy Solution

$$c[i, j] = \begin{cases} 0\,, & \text{if } S_{ij} = \varnothing \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k, j] + 1\}\,, & \text{if } S_{ij} \neq \varnothing \end{cases} \qquad (3)$$

- 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?

Converting a DP solution to a Greedy Solution

- Can we simplify our solution?

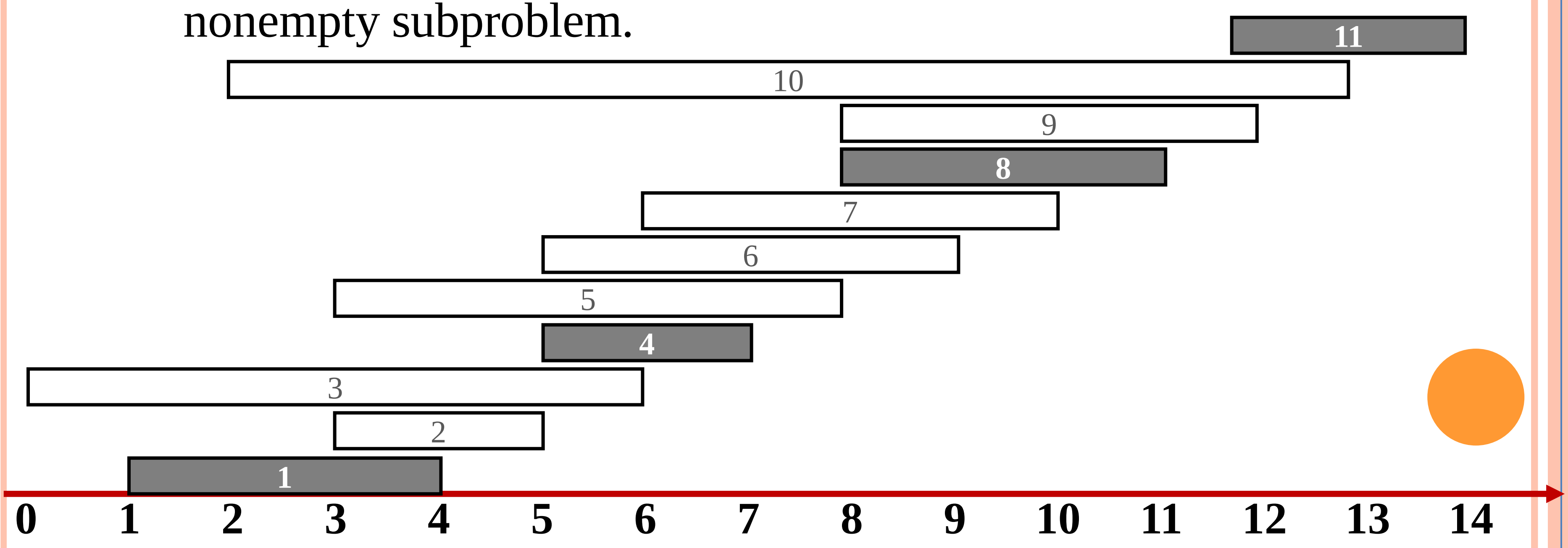


Converting a DP Solution to a Greedy Solution

- **Theorem 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} .
  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.



Converting a DP Solution to a Greedy Solution

- **Theorem 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} .
  2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

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 $a_l \in S_{im} \Rightarrow S_{im} = \emptyset$. ■

Converting a DP Solution to a Greedy Solution

- **Theorem 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} .

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

Converting a DP Solution to a Greedy Solution

- **Theorem 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} .

Proof 1:

- 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, and 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} .

CONVERTING A DP SOLUTION TO A GREEDY SOLUTION

$$c[i, j] = \begin{cases} 0\,, & \text{if } S_{ij} = \varnothing \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k, j] + 1\}\,, & \text{if } S_{ij} \neq \varnothing \end{cases} \qquad (3)$$

- **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} .
  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

- This theorem is great:

	before theorem	after theorem
# of sub-prob in optimal solution	**2 (S_{ik}, S_{kj})**	**1**
# of choices to consider	**$O(j - i - 1)$ ($i<k<j$)**	**1**

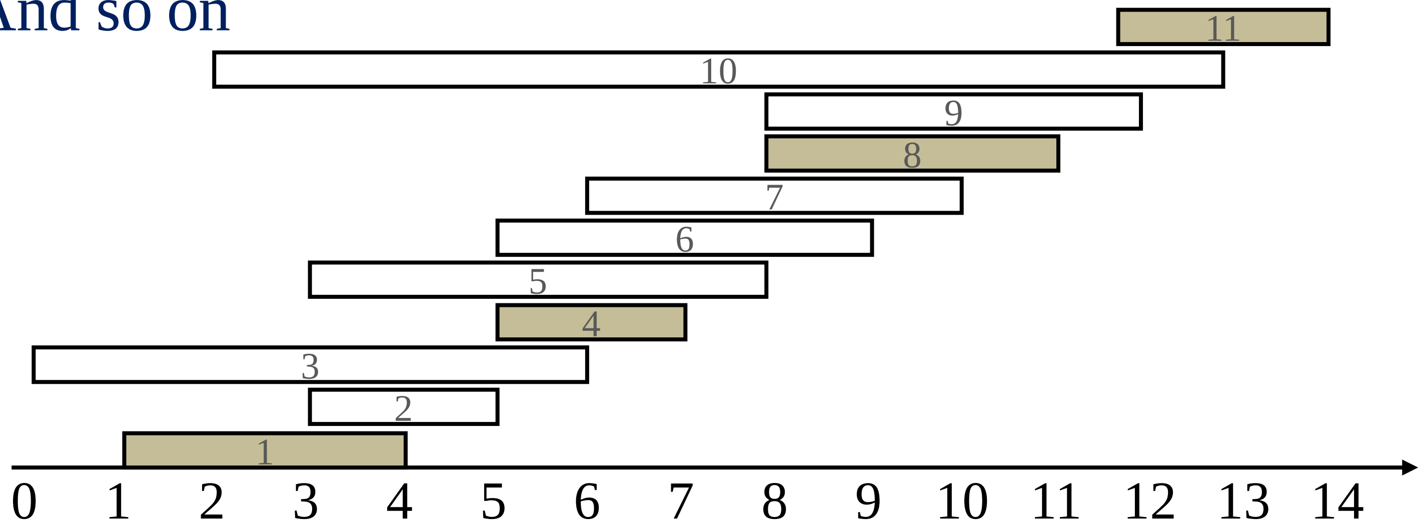
Converting a DP Solution to a Greedy Solution

- **Theorem 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}.
  2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

- 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.
  - Then solve S_{mj}.

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?)
    - $\Rightarrow$ Next subproblem is $S_{m2,n+1}$
  - And so on



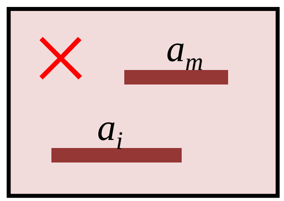
Converting a DP Solution to a Greedy Solution

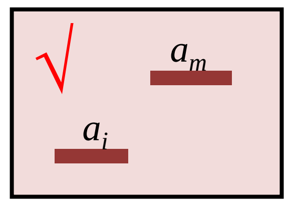
- What are the subproblems?
  - Original problem is $S_{0,n+1}$ 〔 $a_0 = [-\infty, 0)$; $a_{n+1} = [\infty, \text{“}\infty+1\text{”})$ 〕
  - Suppose our first choice is a_{m1} (in fact, it is a_1)
  - Then next subproblem is $S_{m1,n+1}$
  - Suppose next choice is a_{m2} (it must be a_2?)

    $\Rightarrow$ Next subproblem is $S_{m2,n+1}$
  - And so on
- Each subproblem is $S_{mi,n+1}$.
- The subproblems chosen have finish times that increase.
- **Therefore**, we can consider each activity just once, in <u>monotonically increasing order</u> of finish time.

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($\boldsymbol{s}$, f, i, n)

$m \leftarrow i+1$

while $m \le n$ and $s_m < f_i$ // Find first activity a_m in $S_{i,n+1}$.

  do $m \leftarrow m+1$

if $m \le n$

  then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, $\boldsymbol{m}$, n)

else return $\emptyset$

A Recursive Greedy Algorithm

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

$m \leftarrow i+1$

while $m \leq n$ and $s_m < f_i$ // Find first activity a_m in $S_{i,n+1}$.

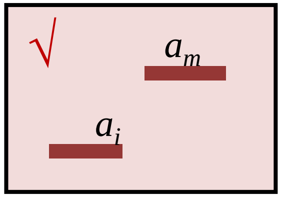
    do $m \leftarrow m+1$

if $m \leq n$

    then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(***s***, *f*, *m*, *n)*

else return *Ø*

- ***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.

A RECURSIVE GREEDY ALGORITHM

REC-ACTIVITY-SELECTOR(***s***, *f*, *i*, *n*)
1 $m \leftarrow i+1$
2 **while** $m \le n$ and $s_m < f_i$ // Find first activity in $S_{i,n+1}$.
3 **do** $m \leftarrow m+1$
4 **if** $m \le n$
5 **then return** $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(***s***, *f*, *m*, *n*)
6 **else return** *Ø*

- **Time: $\Theta(n)$** — each activity examined exactly once.

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

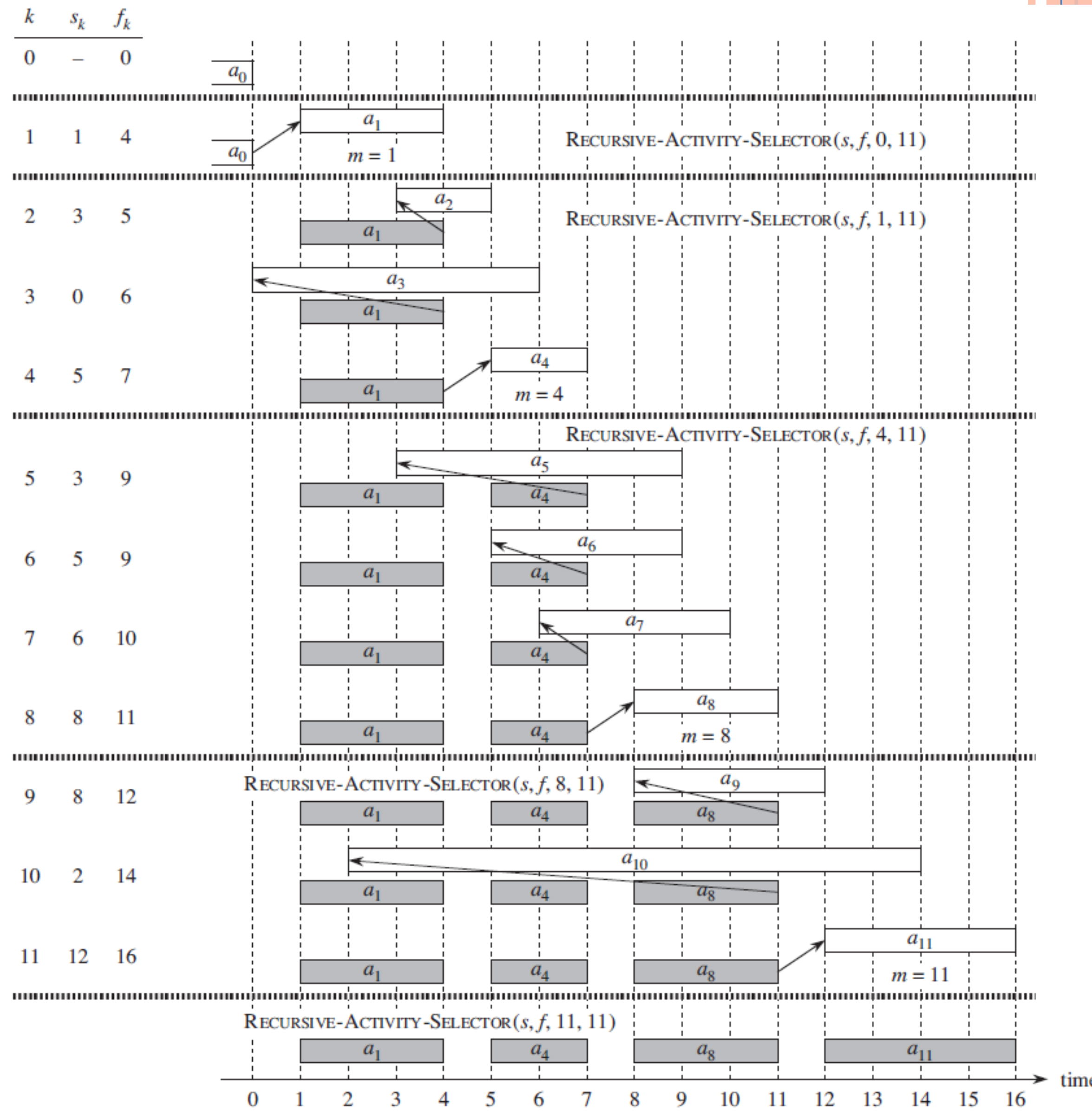
Because: $n - \Sigma m_k = 1$, **then** $\Sigma m_k = n - 1$, $\Sigma m_k + T(1) = \Theta(n)$

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 (**needs $s_m \geq f_i$**).

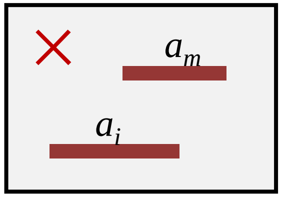
- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

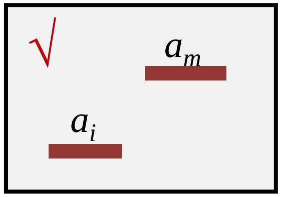


An Iterative Greedy Algorithm

- 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 ← {a_1}
i ← 1
for m ← 2 to n
    do if s_m ≥ f_i
        then A ← A ∪ {a_m}
            i ← m    // a_i is most recent addition to A
return A
```





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*.
- Greedy Algorithm: Simpler, more efficient.

OUTLINE

- Activity-selection problem
- Basic elements of the GA
  - Knapsack problem
- Data compression (Huffman) codes

ELEMENTS OF THE GREEDY STRATEGY

- The choice that seems best at the moment is chosen.
- 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.
  5. Develop a recursive greedy algorithm.
  6. Convert it to an iterative algorithm.

Elements of the Greedy Strategy

- These steps looked like dynamic programming.
- Typically, we streamline these steps.
- 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 $S_i = \{a_k \in S : f_i \leq s_k \}$, and show the greedy choice:

$$\left.\begin{array}{c}\text{The } \textit{first } a_m \text{ to finish in } S_i \\ \text{Combined with optimal solution to } S_m\end{array}\right\} \Rightarrow \begin{array}{c}\text{An optimal} \\ \text{solution to } S_i \text{ .}\end{array}$$

Elements of the Greedy Strategy

- **Typical streamlined steps**

1. Cast the optimization problem as one form
    - Make a choice and only one subproblem is left to solve.
2. Prove that there's always an optimal solution to the original optimization that makes the greedy choice, so that the greedy choice is always **safe**.
3. Demonstrate optimal substructure
    - Having made the greedy choice.
    - What remains is a subproblem with the property:

An optimal solution to the subproblem

The greedy choice we have made

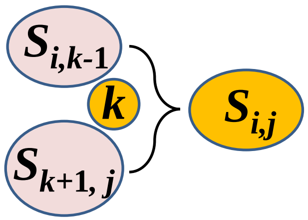
$\Rightarrow$ Optimal solution to the original problem.

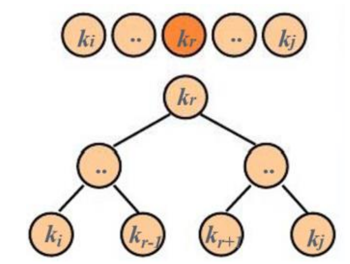
Elements of the Greedy Strategy

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

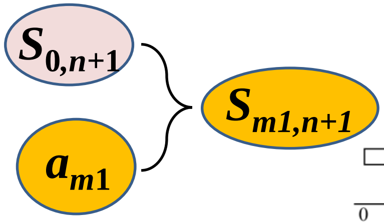
GREEDY-CHOICE PROPERTY

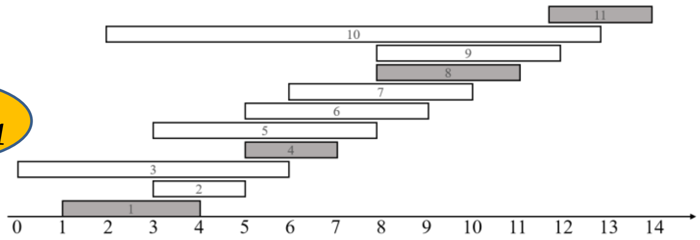
- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.





- *Dynamic Programming*
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblem**s**. Solve subproblems first.
  - Solve *bottom-up*.





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

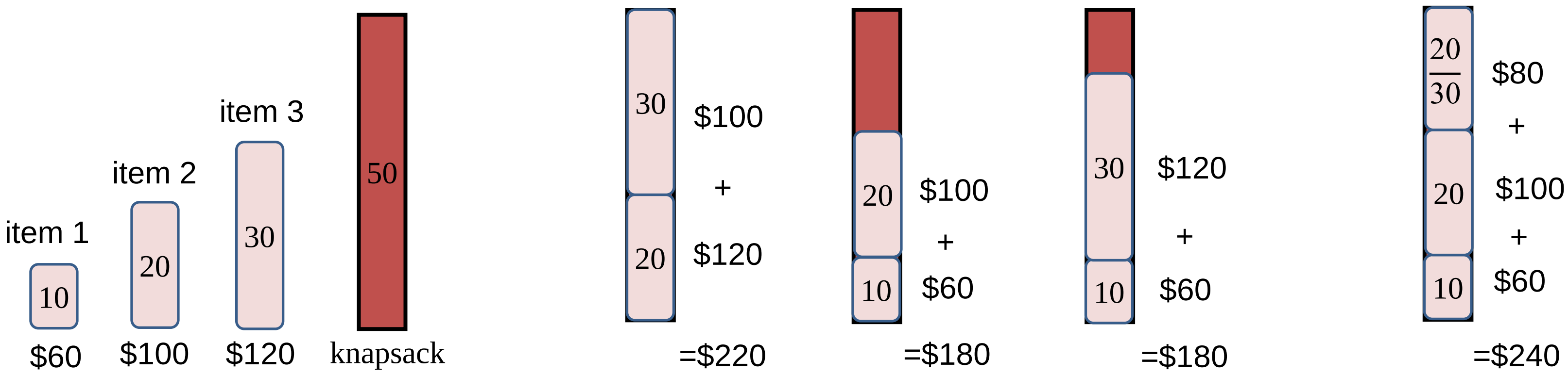
Greedy-Choice Property

- 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.
  - An appropriate data structure (often a priority queue).

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 $\Rightarrow$ optimal solution to problem.

GREEDY *v.s.* DYNAMIC PROGRAMMING



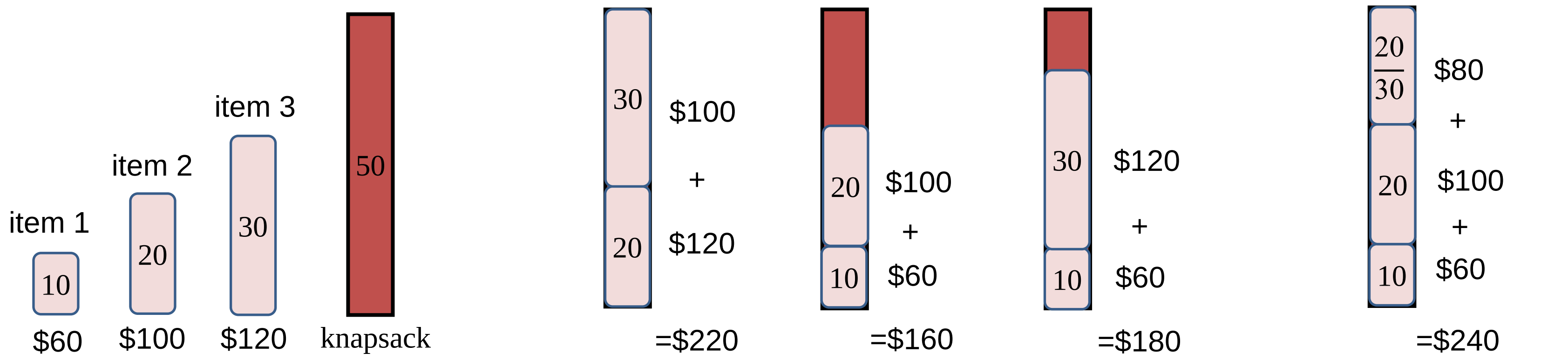
knapsack problem **0-1 knapsack problem** **Fractional knapsack problem**

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

GREEDY *v.s.* DYNAMIC PROGRAMMING

- *0-1 knapsack problem*
- *Fractional knapsack problem*
- Both have optimal substructure property.
  - 0-1 : <u>choose the most valuable</u> load j that <u>weighs</u> $w_j \leq W$, remove j, choose the most valuable load i that weighs $w_i \leq W\text{-}w_j$
  - 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 $W\text{-}w$ that the thief can take from the n-1 original items plus $w_j\text{-}w$ pounds from item j.
- However, the fractional problem has the greedy-choice property, and the 0-1 problem does not.

GREEDY *v.s.* DYNAMIC PROGRAMMING



knapsack problem **0-1 knapsack problem** **Fractional knapsack problem**

- Fractional knapsack problem has the greedy-choice property.
- 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 .
- Time: $O(nlg\ n)$ to sort, $O(n)$ to greedy choice thereafter.

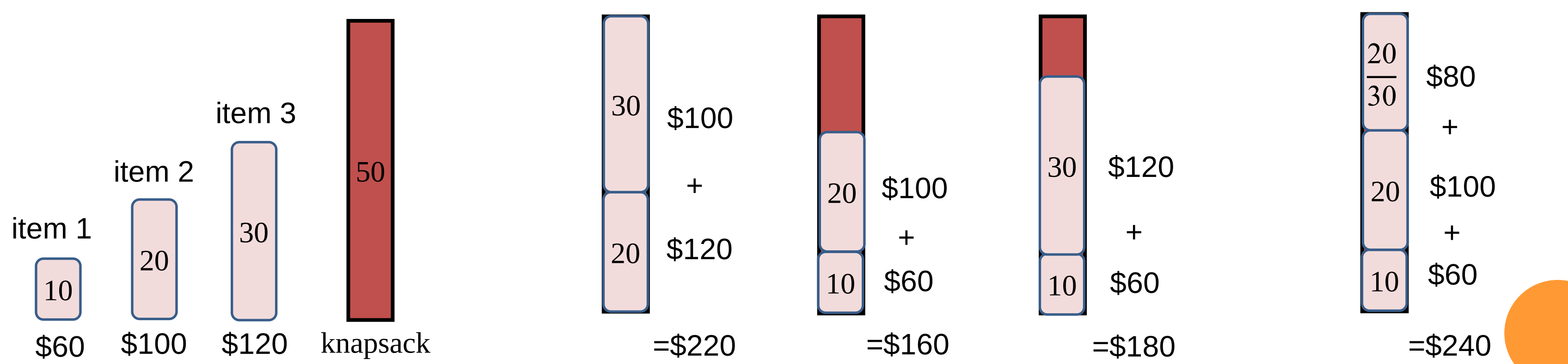
```
FRACTIONAL-KNAPSACK(v, w, W)
load ← 0
i ← 1
while load < W and i ≤ n
      do if wi ≤ W - load
                 then take all of item i
            else take W-load of wi from
                 item i
      add what was taken to load
      i ← i + 1
```

GREEDY *v.s.* 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.



knapsack problem

0-1knapsack problem

Fractional knapsack problem

OUTLINE

- Activity-selection problem
- Basic elements of the GA

  Knapsack problem
- Data compression (Huffman) codes

Huffman Codes

- Huffman codes: widely used and very effective technique for compressing data.
  - savings of 20% to 90%
- Consider the data being compressed 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.

HUFFMAN CODES

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

  Only six different characters (*a*-*f*) appear. The frequency table is shown as:

	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?

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***.

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

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.

PREFIX CODES

- Prefix codes (prefix-free codes): no codeword is a prefix of some other codeword.

	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 the codewords representing each character.
  - For example, "*abc*", with the variable-length prefix code as $0 \cdot 101 \cdot 100 = 0101100$, where we use '$\cdot$' to denote concatenation.
- Prefix codes simplify decoding.

PREFIX CODES

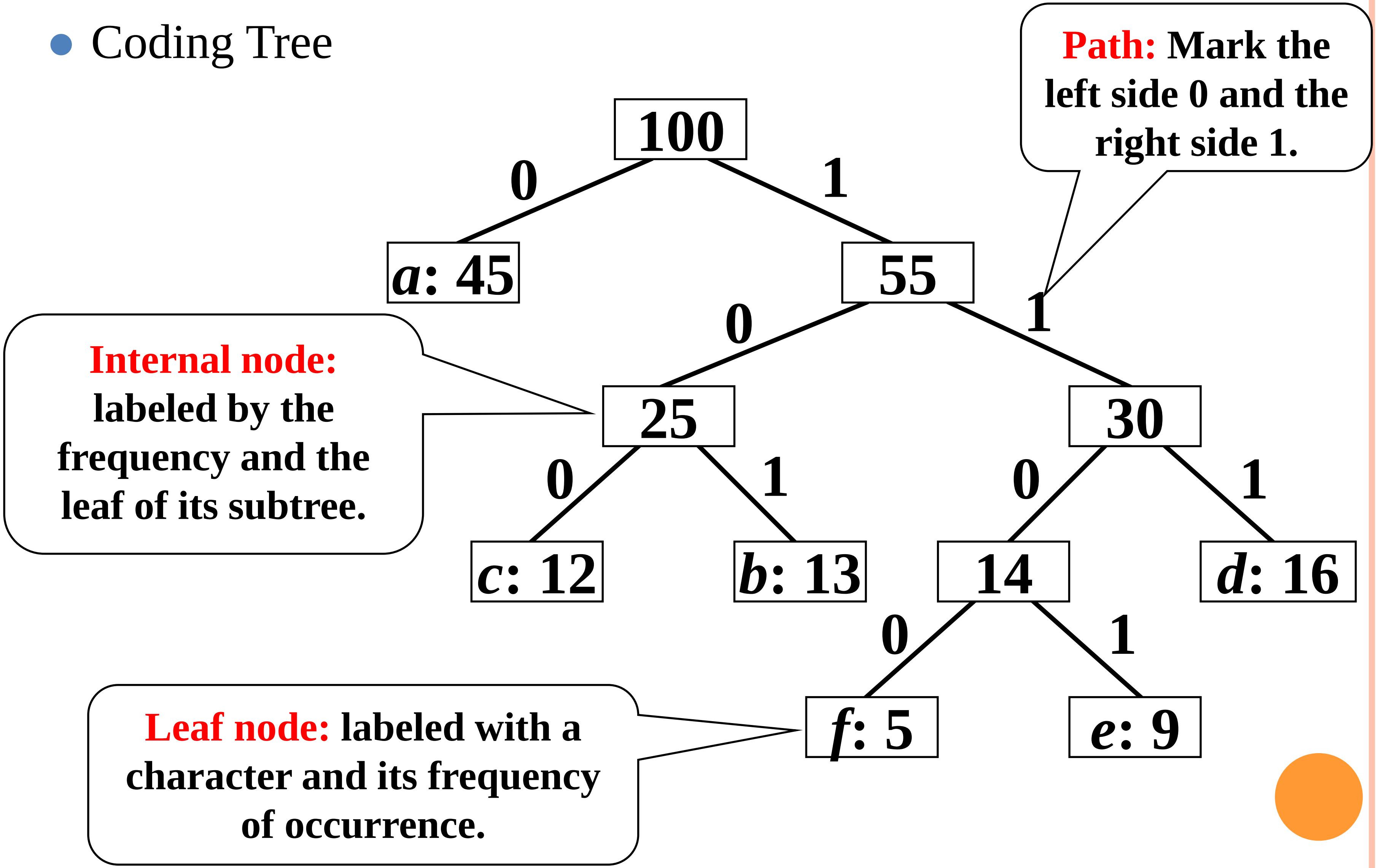
- Prefix codes (prefix-free codes): no codeword is a prefix of some other codeword.

	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.
  - 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".

HUFFMAN CODES

- Coding Tree



HUFFMAN CODES

- The cost of the tree $\boldsymbol{T}$
  - For each **character** c in the alphabet C
  - Let the **attribute** $f(c)$ denote the frequency of c in the file
  - Let $d_{\boldsymbol{T}}(c)$ denote **the depth** of c's leaf in the tree.
  - The **number of bits** required to encode a file is thus:

$$\textit{minimize} \quad \boldsymbol{B(T)} = \sum_{\boldsymbol{c \in C}} \boldsymbol{f(c) \cdot d_T(c)}$$

Huffman Codes

- How to build a optimal coding tree?
  - Input: alphabet $\boldsymbol{C} = \{ \boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_n} \}$

    frequency table $\mathbf{F} = \{ \boldsymbol{f}(\boldsymbol{c_1}), \boldsymbol{f}(\boldsymbol{c_2}), \ldots, \boldsymbol{f}(\boldsymbol{c_n}) \}$
  - Output: a code tree with minimal $\boldsymbol{B}(\boldsymbol{T})$

Greedy algorithm:

The two nodes with the lowest frequency are iteratively selected to generate a subtree until a tree is formed.

Correctness of Huffman's Algorithm

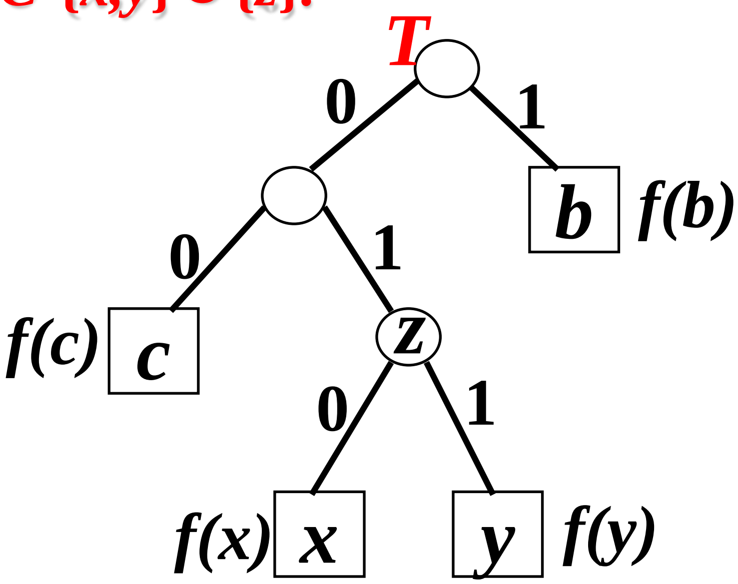
- **Correctness** of Huffman's algorithm

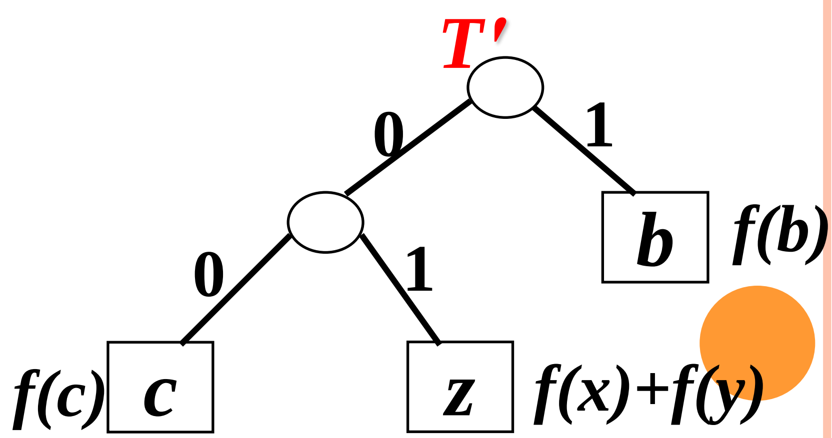
  To prove that the greedy algorithm HUFFMAN is correct, we need to prove

  - **Optimal-substructure property**: if the tree constructed by **merging two nodes** is optimal it must have been constructed from an optimal tree for the subproblem.
  - **Greedy choice property**: there exists an optimal prefix code where two characters having the lowest frequencies in *C* are encoded with **equal length strings** that differ only in the last bit, as they are leaf nodes.

OPTIMAL SUBSTRUCTURE

- **Lemma 1**
  - Let ***T*** be an optimized prefix tree of the alphabet C, $\forall c \in C$, $f(c)$ is the frequency of **occurrence of** c in the file.
  - Let x and y be any two adjacent **leaf** nodes in T, and z be their **parent** node, then z is a character with its frequency $\boldsymbol{f(z) = f(x)+f(y)}$.
  - $\boldsymbol{T' = T - \{x,y\}}$ is the **optimized** prefix encoding tree of alphabet of $\boldsymbol{C' = C - \{x,y\} \cup \{z\}}$.





Proof of Lemma 1

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

(1) $\forall v \in C-\{x,y\}$, $d_T(v)=d_{T'}(v)$, $f(v)d_T(v) = f(v)d_{T'}(v)$.

(2) Because $d_T(x) = d_T(y) = d_{T'}(z)+1$, **and then**

$$
\begin{aligned}
& f(x)d_T(x) + f(y)d_T(y) \\
&= (f(x)+f(y))(d_{T'}(z)+1) \\
&= (f(x)+f(y))\, d_{T'}(z) + (f(x)+f(y))
\end{aligned}
$$

Because $f(x) + f(y) = f(z)$,

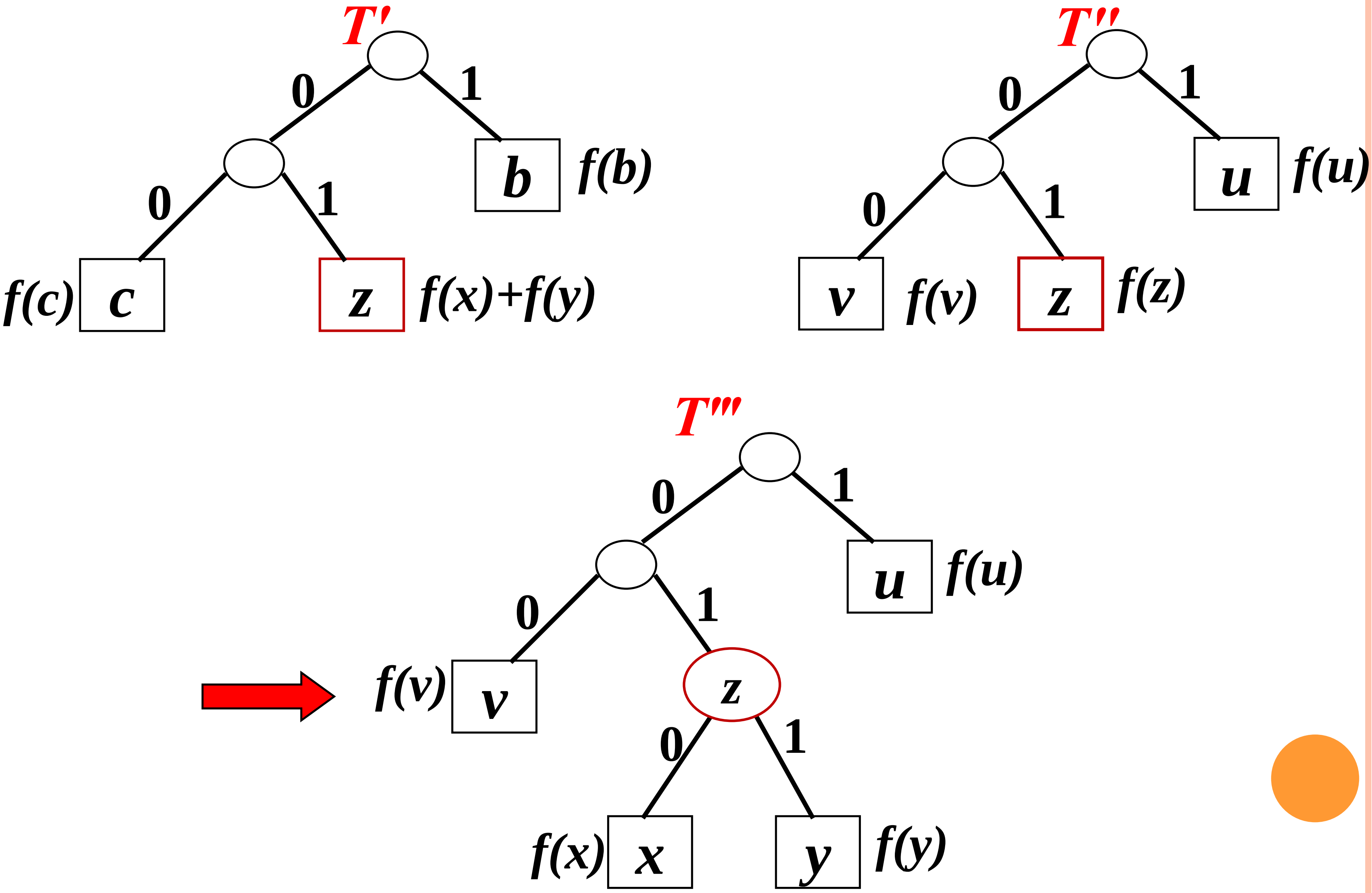
$$f(x)d_T(x) + f(y)d_T(y) = f(z)d_{T'}(z) + (f(x)+f(y)).$$

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

We now prove the lemma by contradiction.

- Suppose that T′ <u>does not</u> represent an optimal prefix code for C′.
- Then there exists another optimal tree T″ such that B(T″) < B(T′).
- Because *z* is character of C′, so it is a leaf of tree T″.
- Adding nodes *x* and *y* to T″, as child nodes of *z*, then we get a prefix coding tree of C, T‴:

Proof of Lemma 1

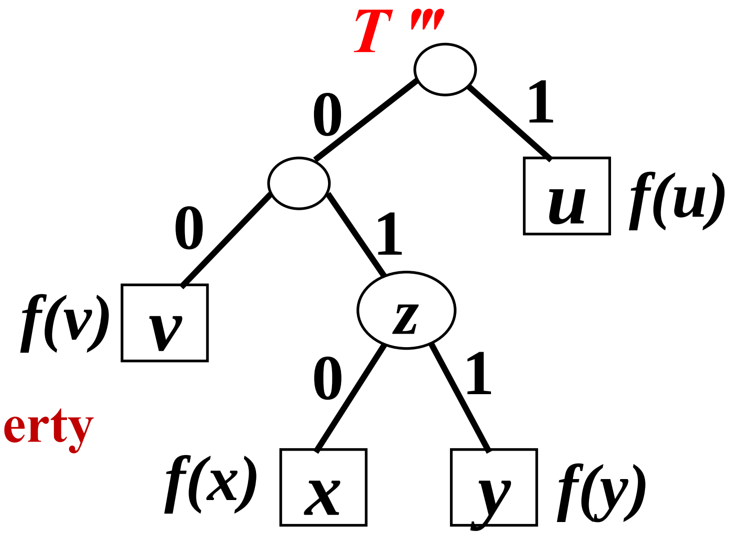


PROOF OF LEMMA 1

- The cost of T‴:

$$B(T''') = \ldots\ldots + (f(x)+f(y))\,(d_{T''}(z)+1)$$
$$= \ldots\ldots + f(z)d_{T''}(z) + (f(x)+f(y)) \quad (d_{T''}(z)=d_{T'}(z))$$
$$= B(T'') + f(x)+f(y) < B(T')+f(x)+f(y) = B(T)$$

- 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′.



Optimal-substructure property Proved!!!

THE GREEDY-CHOICE

Lemma 2

Let C be an alphabet in which each character $c \in C$ has frequency $f(x)$.

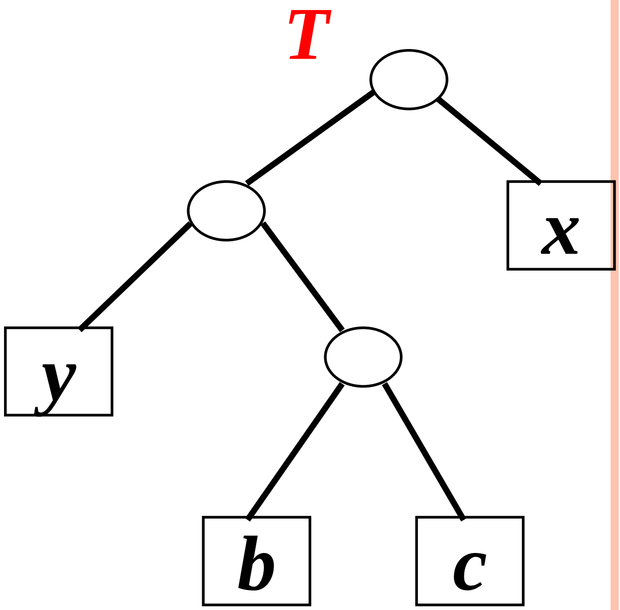
Let ***x* and *y*** be two characters in C having **the lowest frequencies**.

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.

Greedy Choice Property

Proof of Lemma 2

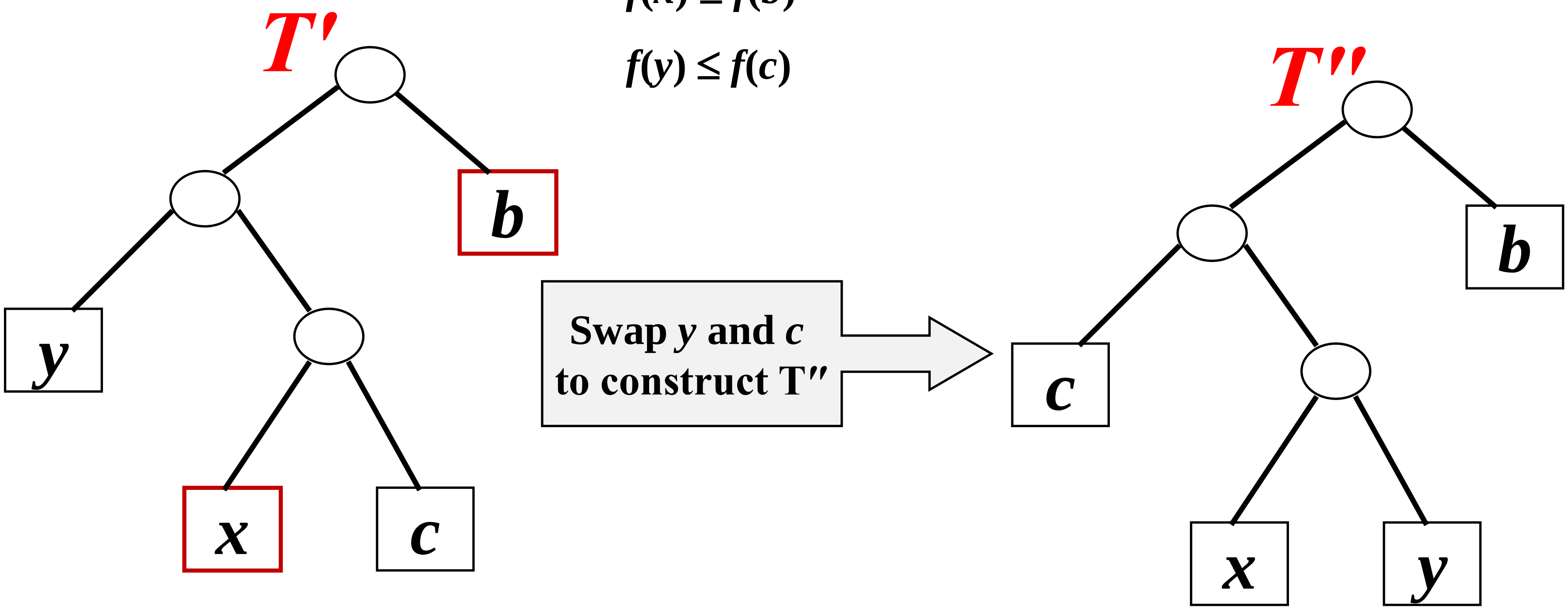
- Taking the tree T represent an optimal prefix code tree of C. Let b and c be two characters that are sibling leaves of maximum depth in T.
- Without loss of generality, we assume that $f(b) \le f(c)$ and $f(x) \le f(y)$.
- Since $f(x)$ and $f(y)$ are the two lowest leaf frequencies, in order, and $f(b)$, $f(c)$ are two arbitrary frequencies, in order, we have $f(x) \le f(b)$ and $f(y) \le f(c)$.
- We exchange the positions in T of b and x to produce a tree T':



Proof of Lemma 2

$f(x) \leq f(b)$

$f(y) \leq f(c)$



PROOF OF LEMMA 2

Proof that tree T″ is an optimal prefix code tree:

$B(T) - B(T')$

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

$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$

$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x)$

$= (f(b) - f(x)) (d_T(b) - d_T(x)).$

$\because$ $f(b) \geq f(x)$, $d_T(b) \geq d_T(x)$ (because b is a leaf of maximum depth in T)

$\therefore$ $B(T) - B(T') \geq 0 \Rightarrow B(T) \geq B(T')$

$f(x) \leq f(b)$

$f(y) \leq f(c)$

Similarly $B(T') \geq B(T'')$. So $B(T) \geq B(T'')$.

Since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$.

Thus, T'' *is* an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows.

CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Theorem 2**

Procedure HUFFMAN produces an optimal prefix code.

Proof

Since Lemma 1 and Lemma 2 are valid, and Huffman algorithm performs local optimization selection according to the rules determined by **greedy selectivity** of Lemma 2, Huffman algorithm generates an optimized prefix coding tree.