

# CS473 - Algorithms I

---

## Lecture 11 Greedy Algorithms

*View in slide-show mode*

# CS473 - Algorithms I



## Activity Selection Problem

# Activity Selection Problem

- We have:
  - ▣ A set of activities with fixed start and finish times
  - ▣ One shared resource (only one activity can use at any time)
- Objective: Choose the **max number** of compatible activities

*Note: Objective is to maximize the number of activities, not the total time of activities.*

- Example:
  - Activities**: Meetings with fixed start and finish times
  - Shared resource**: A meeting room
  - Objective**: Schedule the max number of meetings

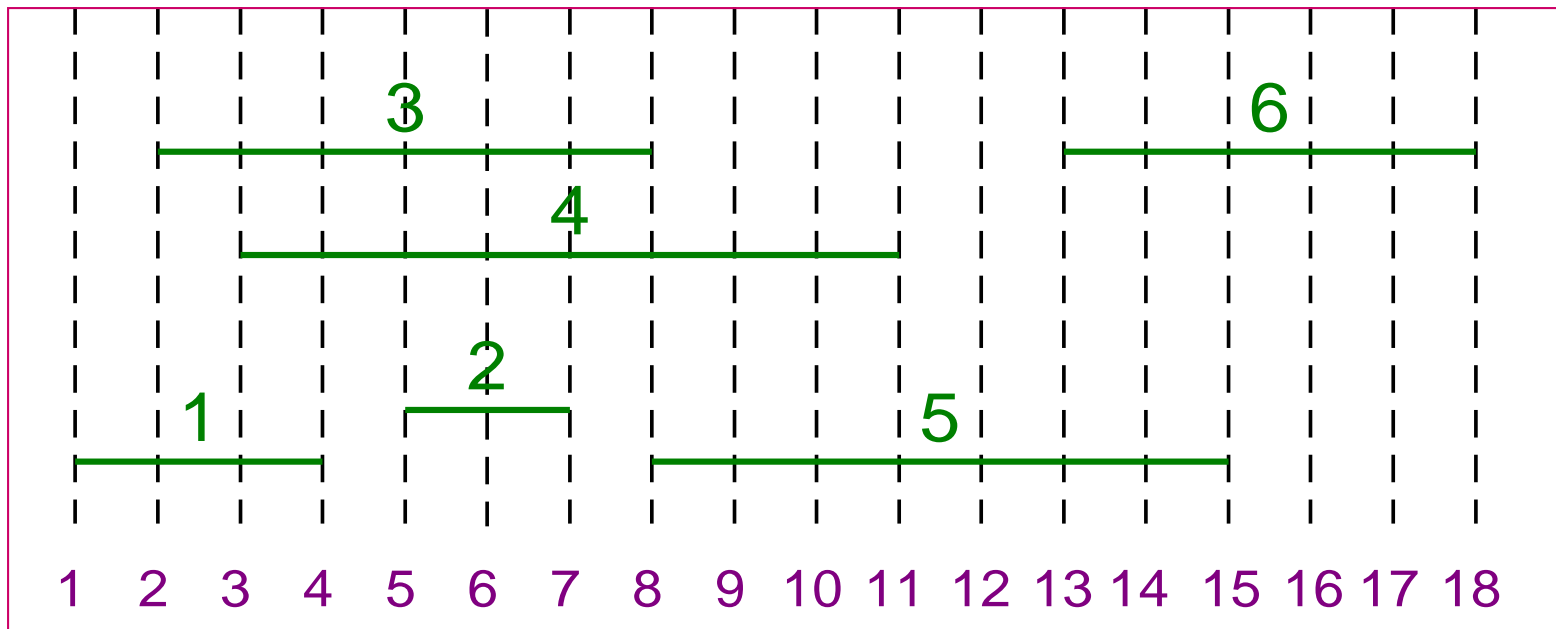
# Activity Selection Problem

---

- **Input**: a set  $S = \{a_1, a_2, \dots, a_n\}$  of  $n$  activities
  - $s_i$  : Start time of activity  $a_i$ ,
  - $f_i$  : Finish time of activity  $a_i$Activity  $i$  takes place in  $[s_i, f_i)$
- **Aim**: Find max-size subset  $A$  of mutually *compatible* activities
  - Max number of activities, not max time spent in activities
  - Activities  $i$  and  $j$  are **compatible** if intervals  $[s_i, f_i)$  and  $[s_j, f_j)$  do not overlap, i.e., either  $s_i \geq f_j$  or  $s_j \geq f_i$

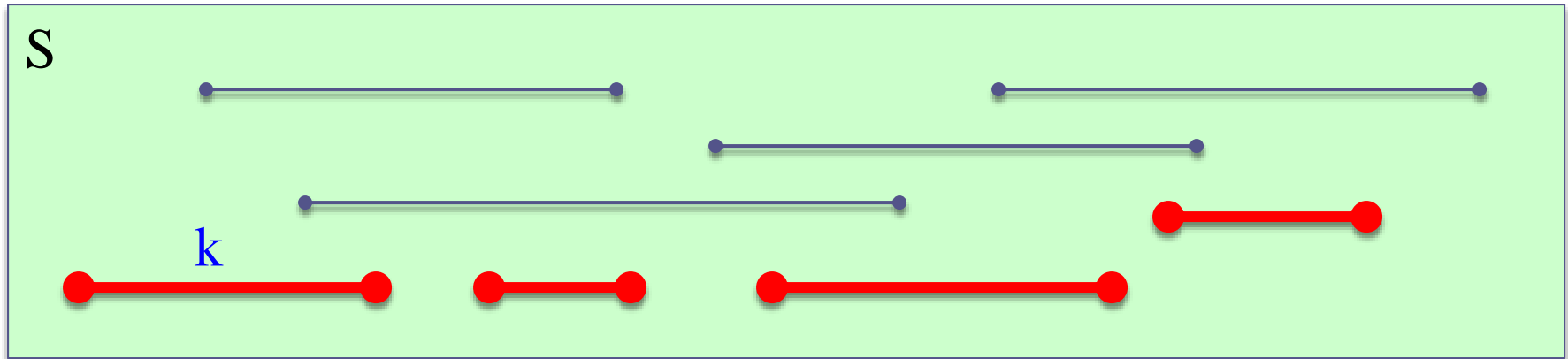
# Activity Selection Problem: An Example

$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$



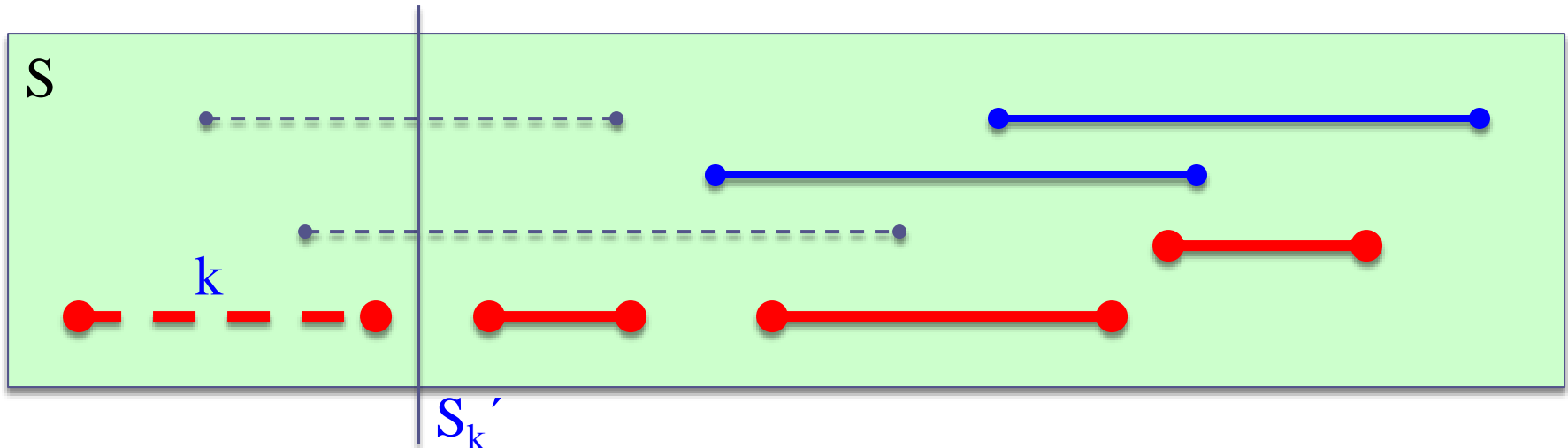
# Optimal Substructure Property

- Consider an optimal solution  $A$  for activity set  $S$ .
- Let  $k$  be the activity in  $A$  with the **earliest finish time**



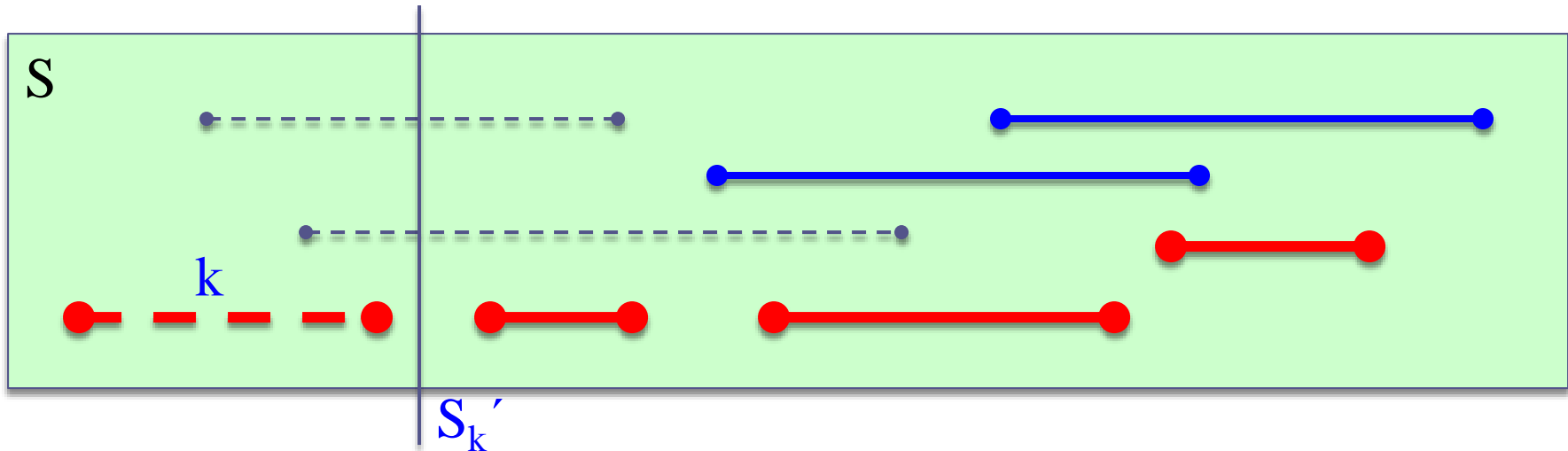
# Optimal Substructure Property

- Consider an optimal solution  $A$  for activity set  $S$ .
- Let  $k$  be the activity in  $A$  with the **earliest finish time**
- Now, consider the **subproblem**  $S_k'$  that has the activities that start after  $k$  finishes, i.e.  $S_k' = \{a_i \in S : s_i \geq f_k\}$
- What can we say about the optimal solution to  $S_k'$  ?



# Optimal Substructure Property

- Consider an optimal solution  $A$  for activity set  $S$ .
- Let  $k$  be the activity in  $A$  with the **earliest finish time**
- Now, consider the **subproblem**  $S_k'$  that has the activities that start after  $k$  finishes, i.e.  $S_k' = \{a_i \in S : s_i \geq f_k\}$
- $A - \{k\}$  is an optimal solution for  $S_k'$ . Why?





# Optimal Substructure

---

**Theorem:** Let  $k$  be the activity with the earliest finish time in an optimal soln  $A \subseteq S$  then

$A - \{k\}$  is an optimal solution to subproblem

$$S'_k = \{a_i \in S : s_i \geq f_k\}$$

**Proof** (by contradiction):

▷ Let  $B'$  be an optimal solution to  $S'_k$  and

$$|B'| > |A - \{k\}| = |A| - 1$$

▷ Then,  $B = B' \cup \{k\}$  is compatible and

$$|B| = |B'| + 1 > |A|$$

Contradiction to the optimality of  $A$

**Q.E.D.**

# Optimal Substructure

- Recursive formulation: Choose the first activity  $k$ , and then solve the remaining subproblem  $S_k'$
- How to choose the first activity  $k$ ?  
DP, memoized recursion?  
i.e. choose the  $k$  value that will have the max size for  $S_k'$
- DP would work,  
but is it necessary to try all possible values for  $k$ ?

# Greedy Choice Property

- Assume (without loss of generality)  $f_1 \leq f_2 \leq \dots \leq f_n$ 
  - ▣ If not, sort activities according to their finish times in non-decreasing order
- Greedy choice property: a sequence of locally optimal (greedy) choices  $\Rightarrow$  an optimal solution
- How to choose the first activity **greedily** without losing optimality?

# Greedy Choice Property - Theorem

Let activity set  $S = \{a_1, a_2, \dots, a_n\}$ , where  $f_1 \leq f_2 \leq \dots \leq f_n$

*Theorem*: There exists an optimal solution  $A \subseteq S$  such that  $a_1 \in A$

*In other words, the activity with the earliest finish time is guaranteed to be in an optimal solution.*

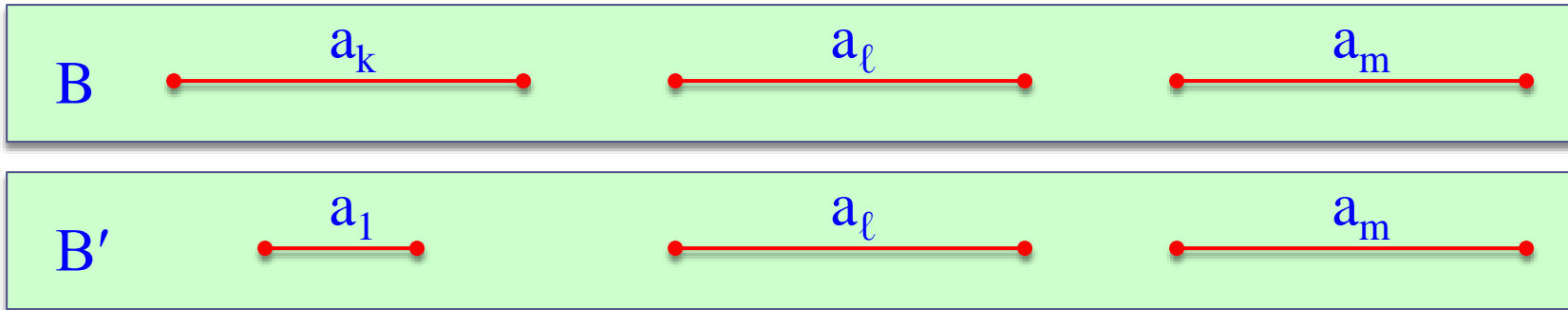
# Greedy Choice Property - Proof

**Theorem**: There exists an optimal solution  $A \subseteq S$  such that  $a_1 \in A$

**Proof**: Consider an arbitrary optimal solution  $B = \{a_k, a_\ell, a_m, \dots\}$ , where  $f_k < f_\ell < f_m < \dots$

**If  $k = 1$** , then  $B$  starts with  $a_1$ , and the proof is complete

**If  $k > 1$** , then create another solution  $B'$  by replacing  $a_k$  with  $a_1$ . Since  $f_1 \leq f_k$ ,  $B'$  is guaranteed to be valid, and  $|B'| = |B|$ , hence also optimal



# Greedy Algorithm

□ So far, we have:

- Optimal substructure property: If  $A = \{a_k, \dots\}$  is an optimal solution, then  $A - \{a_k\}$  must be optimal for subproblem  $S_k'$ , where  $S_k' = \{a_i \in S: s_i \geq f_k\}$

*Note:  $a_k$  is the activity with the earliest finish time in  $A$*

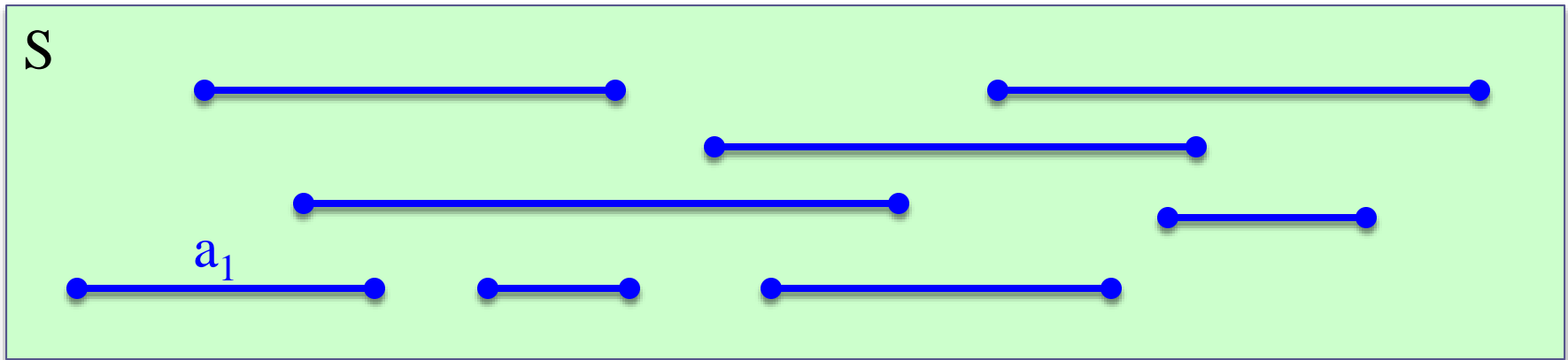
- Greedy choice property: There is an optimal solution  $A$  that contains  $a_1$

*Note:  $a_1$  is the activity with the earliest finish time in  $S$*

# Greedy Algorithm

□ Basic idea of the greedy algorithm:

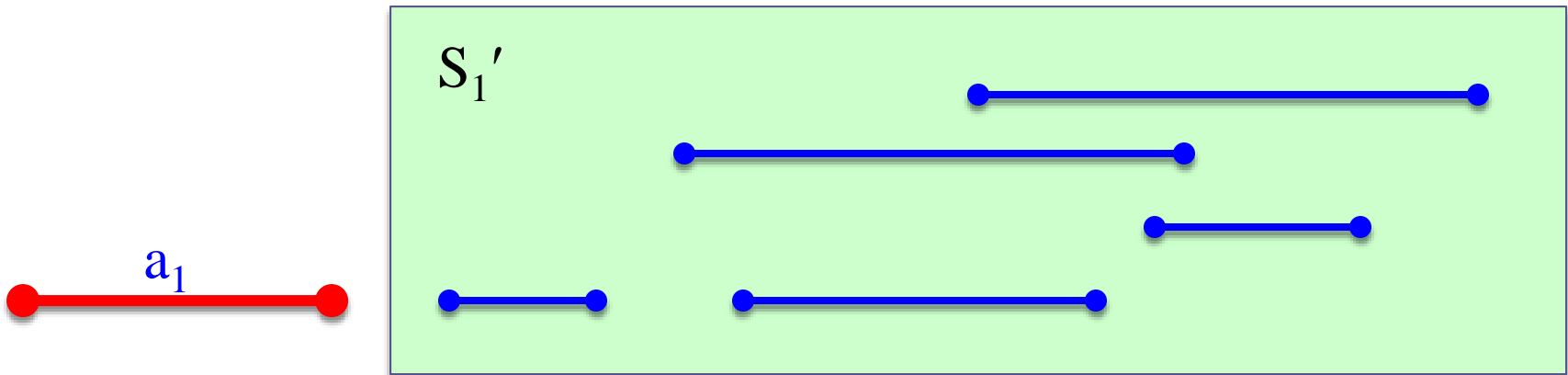
1. Add  $a_1$  to  $A$
2. Solve the remaining subproblem  $S_1'$ , and then append the result to  $A$



# Greedy Algorithm

□ Basic idea of the greedy algorithm:

1. Add  $a_1$  to  $A$
2. Solve the remaining subproblem  $S_1'$ , and then append the result to  $A$





# Greedy Algorithm for Activity Selection

$j$ : specifies the index of most recent activity added to  $A$

$f_j = \text{Max}\{f_k : k \in A\}$ , max finish time of any activity in  $A$ ; because activities are processed in non-decreasing order of finish times

Thus, “ $s_i \geq f_j$ ” checks the compatibility of  $i$  to current  $A$

Running time:  $\Theta(n)$  assuming that the activities were already sorted

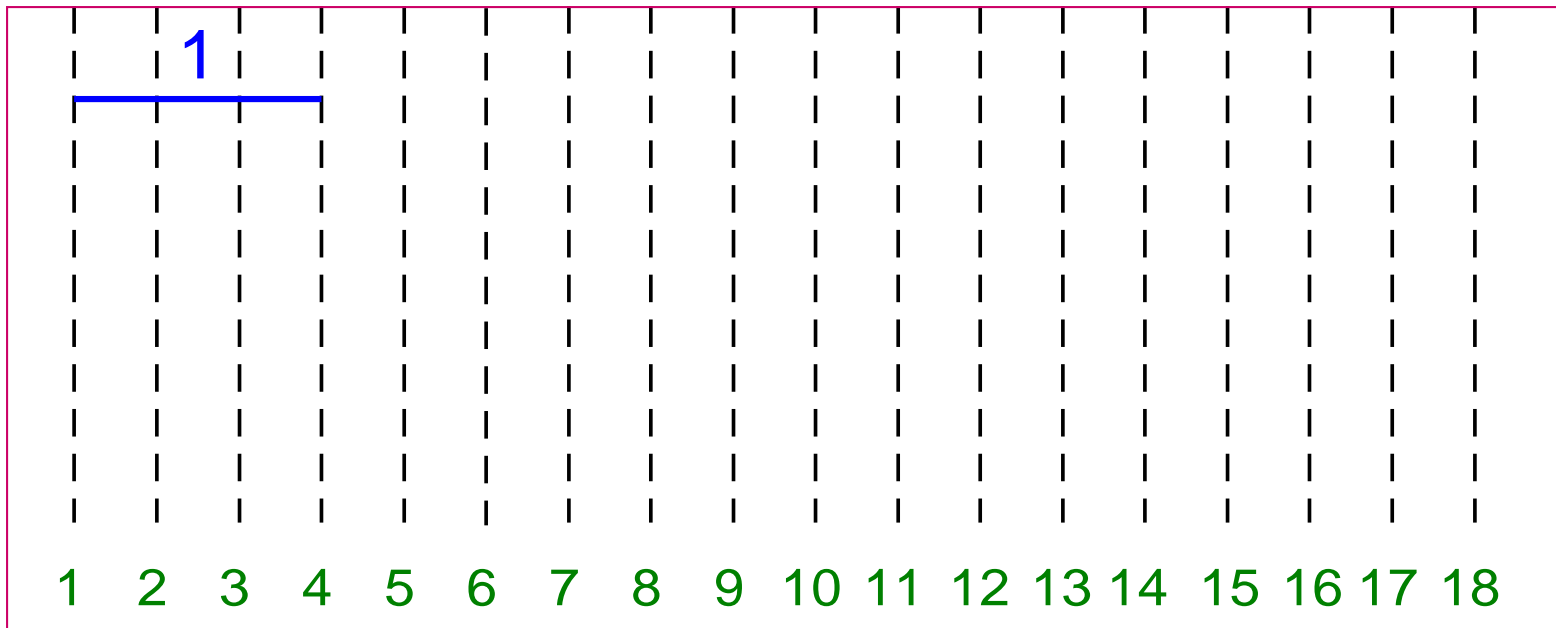
```
GAS (s, f, n)
  A ← {1}
  j ← 1
  for i ← 2 to n do
    if  $s_i \geq f_j$  then
      A ← A ∪ {i}
      j ← i
  return A
```

# Activity Selection Problem: An Example

$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$

---

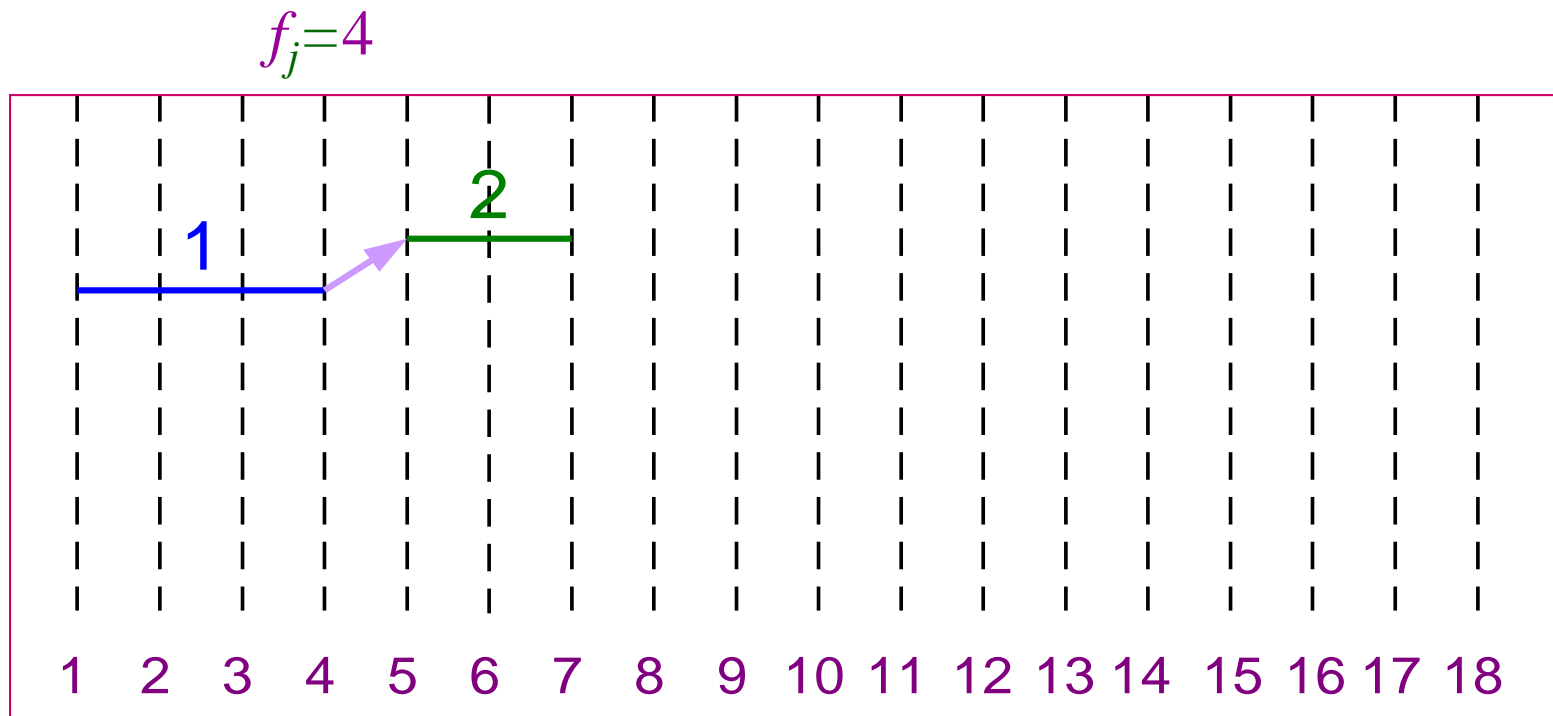
$$f_j = 0$$



# Activity Selection Problem: An Example

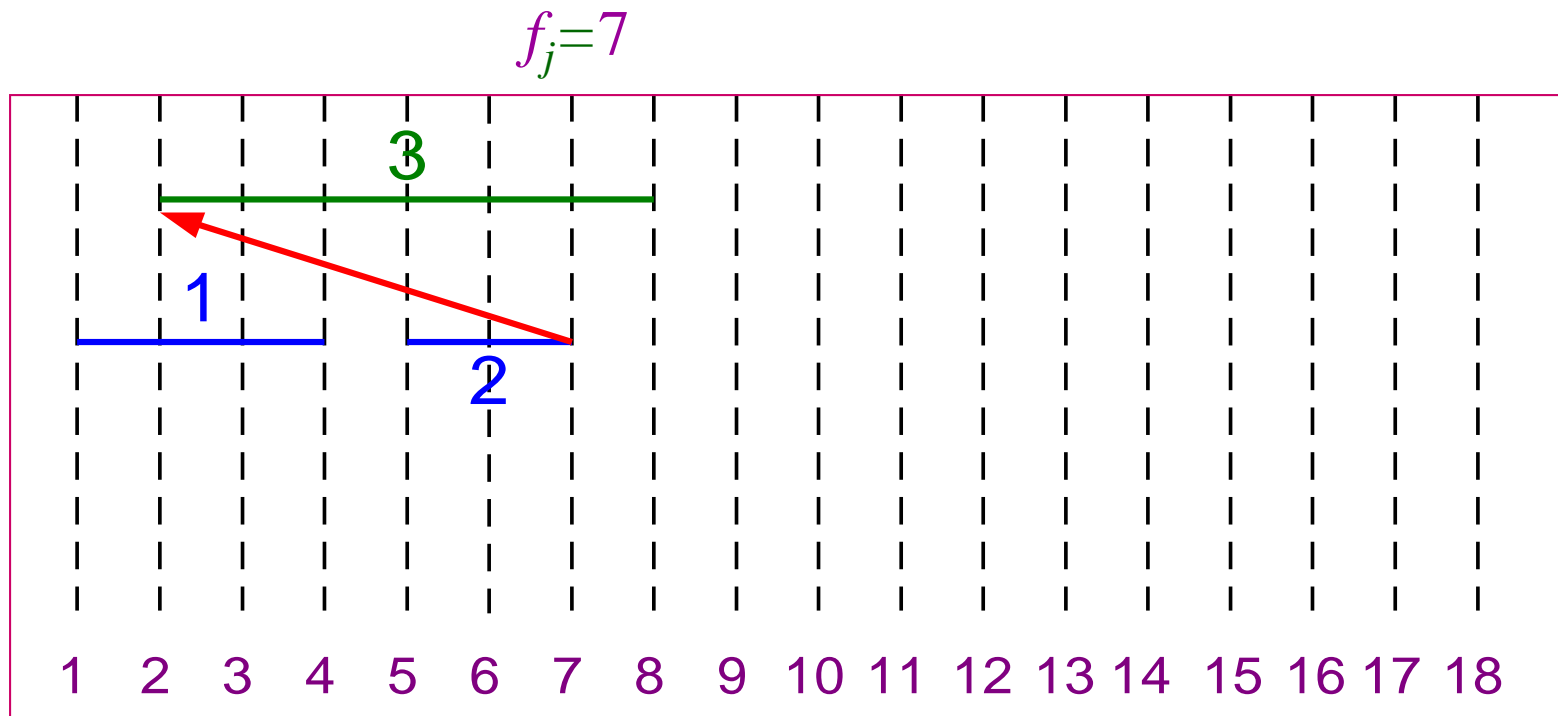
$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$

---



# Activity Selection Problem: An Example

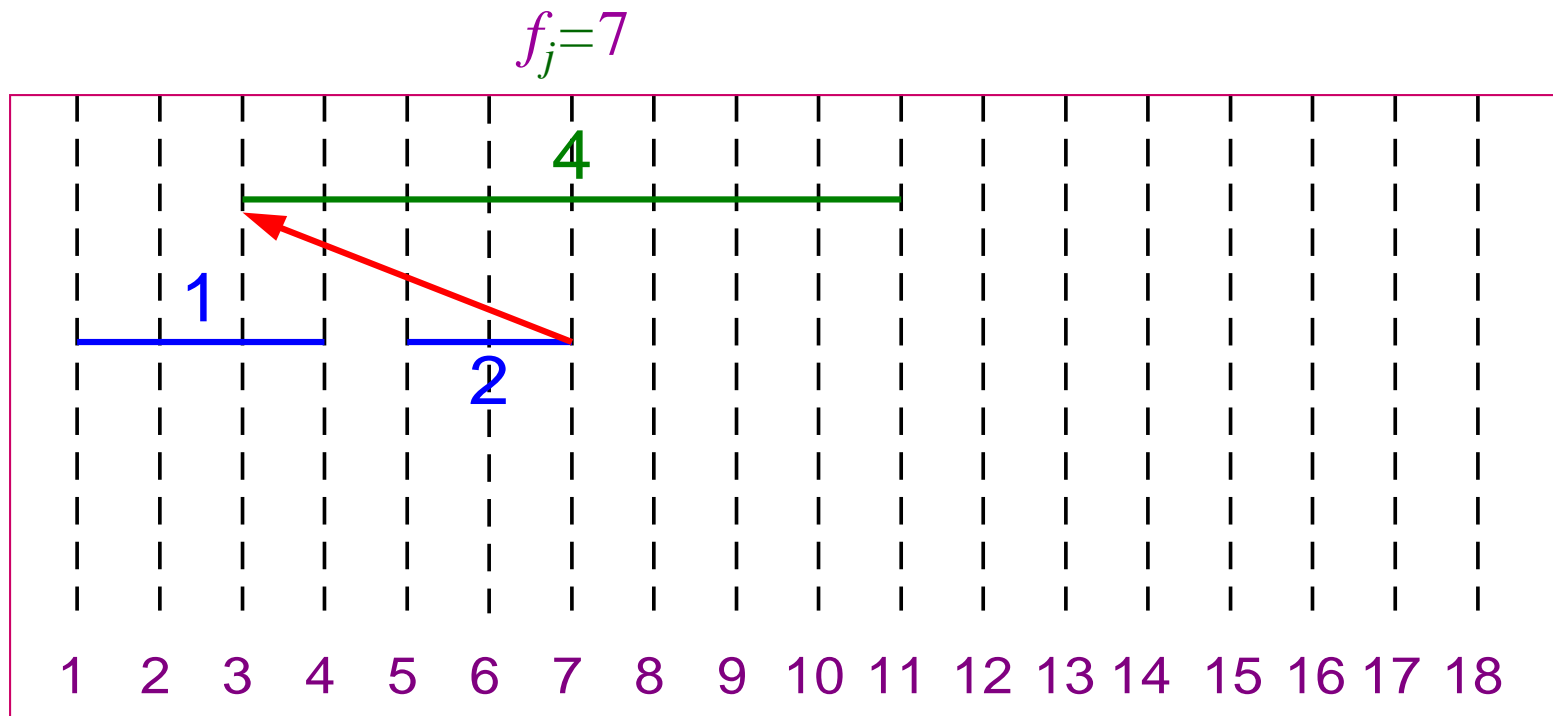
$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$



# Activity Selection Problem: An Example

$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$

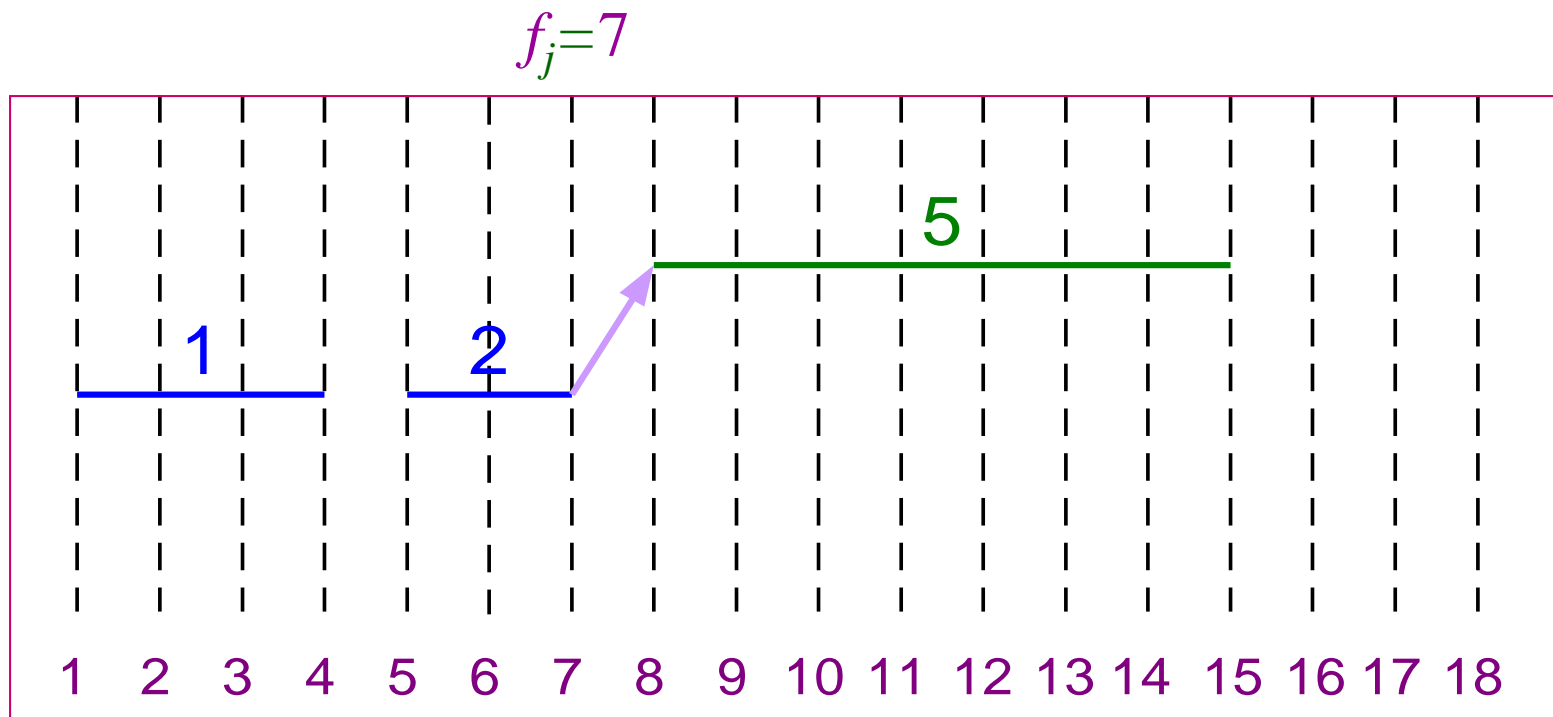
---



# Activity Selection Problem: An Example

$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$

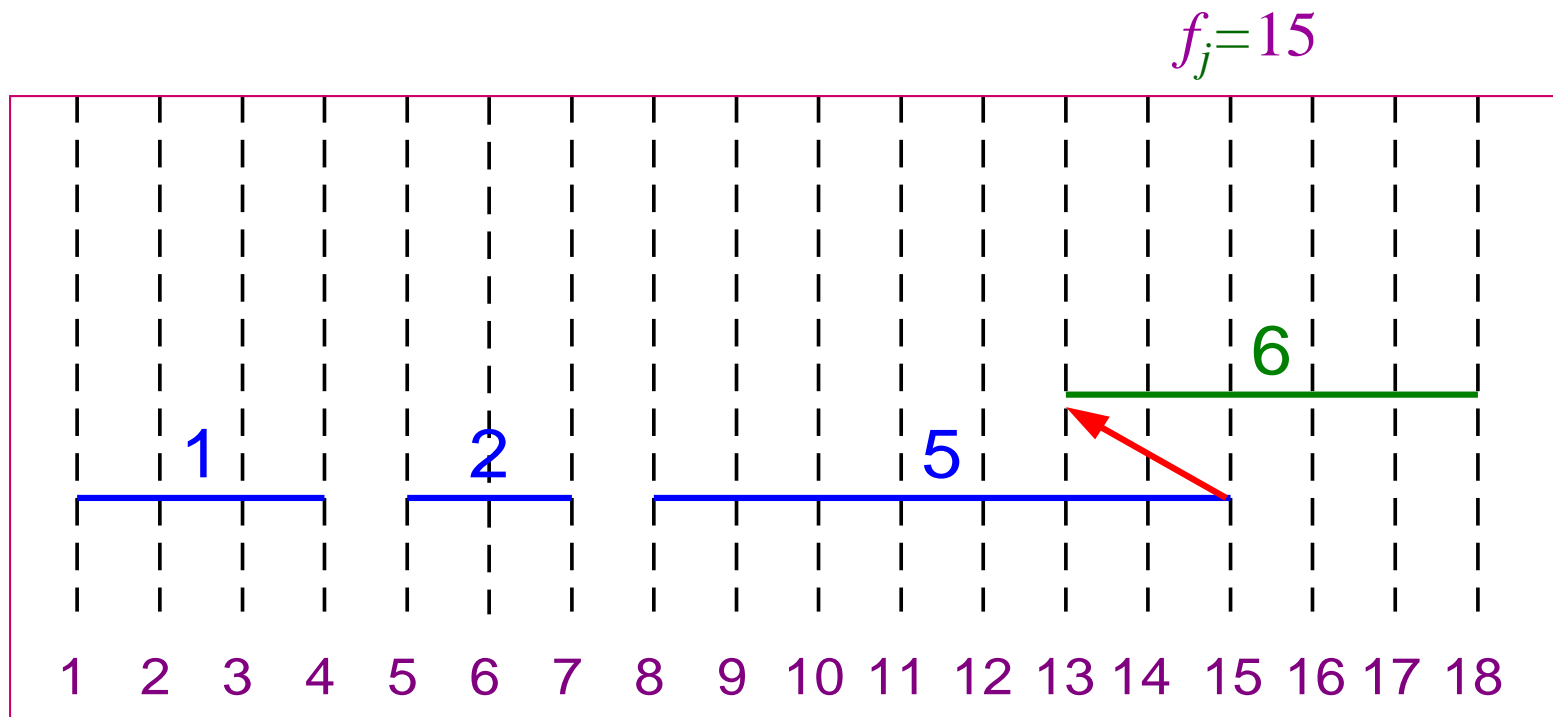
---



# Activity Selection Problem: An Example

$$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$$

---

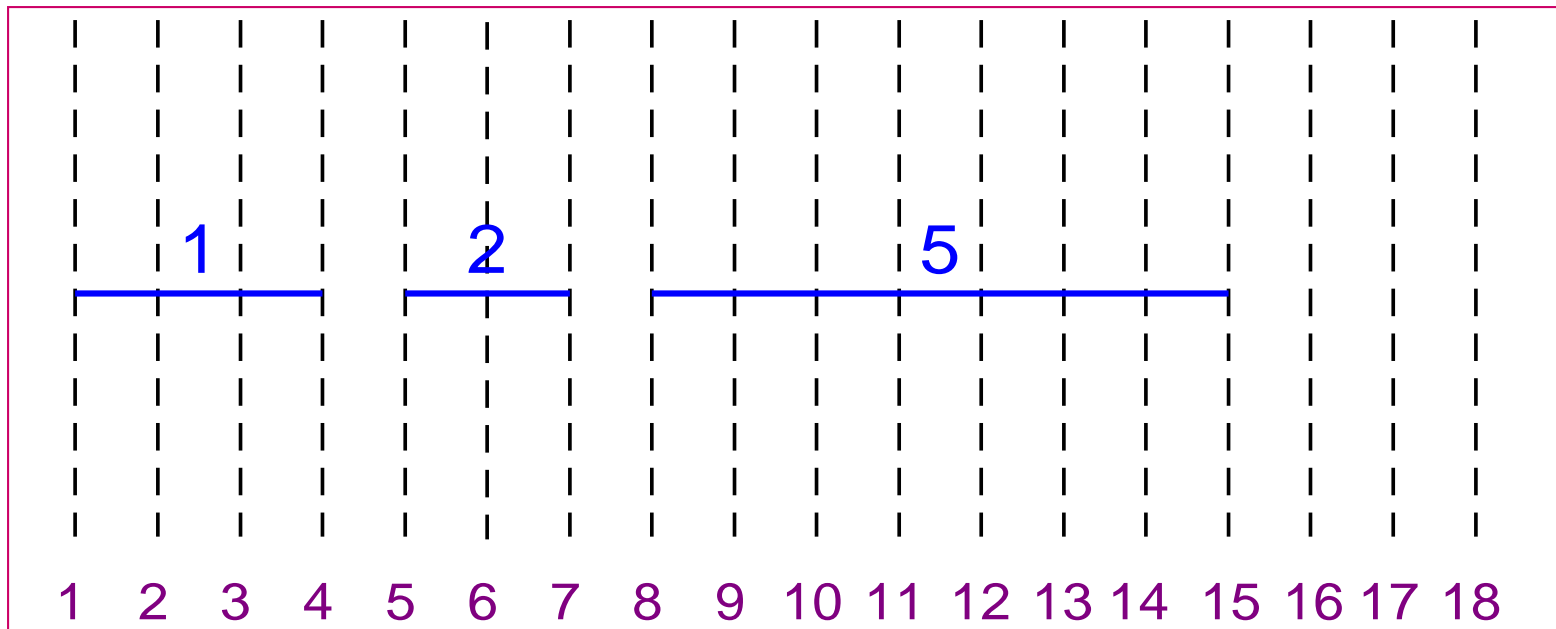


# Activity Selection Problem: An Example

$S = \{ [1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18) \}$

---

$A = \{ 1, 2, 5 \}$





# CS473 - Algorithms I



## Comparison of DP and Greedy Algorithms

# Reminder: DP-Based Matrix Chain Order

$$m_{ij} = \min_{i \leq k < j} \{m_{ik} + m_{k+1,j} + p_{i-1}p_k p_j\}$$

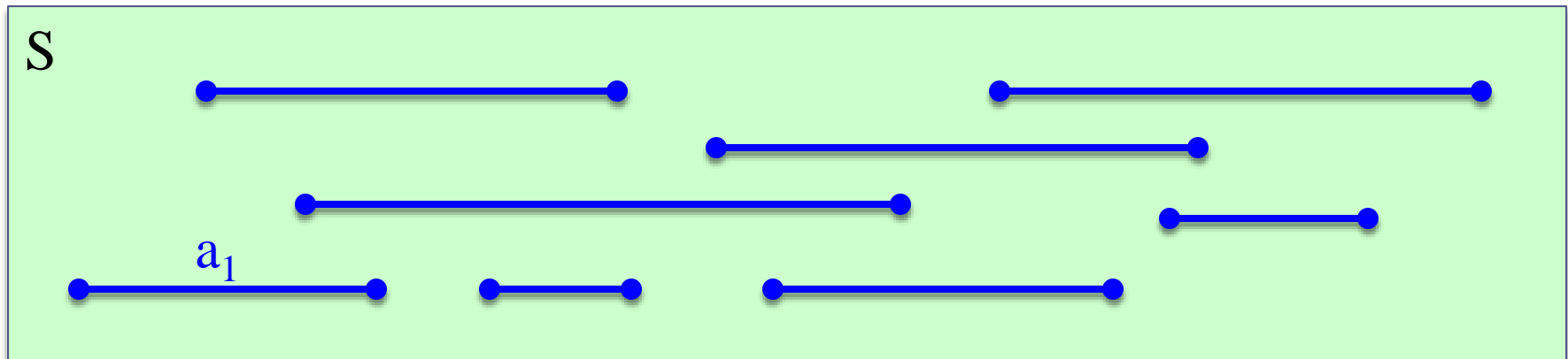
- We don't know ahead of time which **k** value to choose.
- We first need to compute the results of subproblems  $m_{ik}$  and  $m_{k+1,j}$  before computing  $m_{ij}$
- The selection of **k** is done based on the **results of the subproblems**.

# Greedy Algorithm for Activity Selection

1. Make a greedy selection in the beginning:

Choose  $a_1$  (the activity with the earliest finish time)

2. Solve the remaining subproblem  $S_1'$  (all activities that start after  $a_1$ )

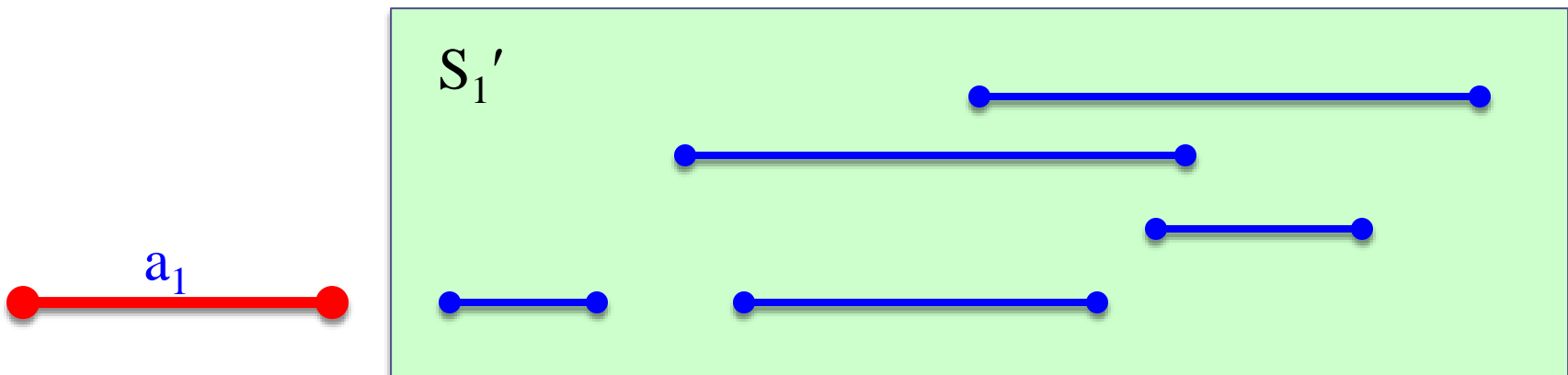


# Greedy Algorithm for Activity Selection

1. Make a greedy selection in the beginning:

Choose  $a_1$  (the activity with the earliest finish time)

2. Solve the remaining subproblem  $S_1'$  (all activities that start after  $a_1$ )



# Greedy vs Dynamic Programming

---

- Optimal substructure property exploited by both **Greedy** and **DP** strategies
- **Greedy Choice Property**: A sequence of locally optimal choices  $\Rightarrow$  an optimal solution
  - We make the choice that seems best at the moment
  - Then solve the subproblem arising after the choice is made
- **DP**: We also make a choice/decision at each step, but the choice may depend on the optimal solutions to subproblems
- **Greedy**: The choice may depend on the choices made so far, but it cannot depend on any future choices or on the solutions to subproblems

# Greedy vs Dynamic Programming

---

- **DP** is a bottom-up strategy
- **Greedy** is a top-down strategy
  - each greedy choice in the sequence iteratively reduces each problem to a similar but smaller problem

# Proof of Correctness of Greedy Algorithms

---

- Examine a globally optimal solution
- Show that this soln can be modified so that
  - 1) A greedy choice is made as the first step
  - 2) This choice reduces the problem to a similar but smaller problem
- Apply induction to show that a greedy choice can be used at every step
- Showing (2) reduces the proof of correctness to proving that the problem exhibits optimal substructure property

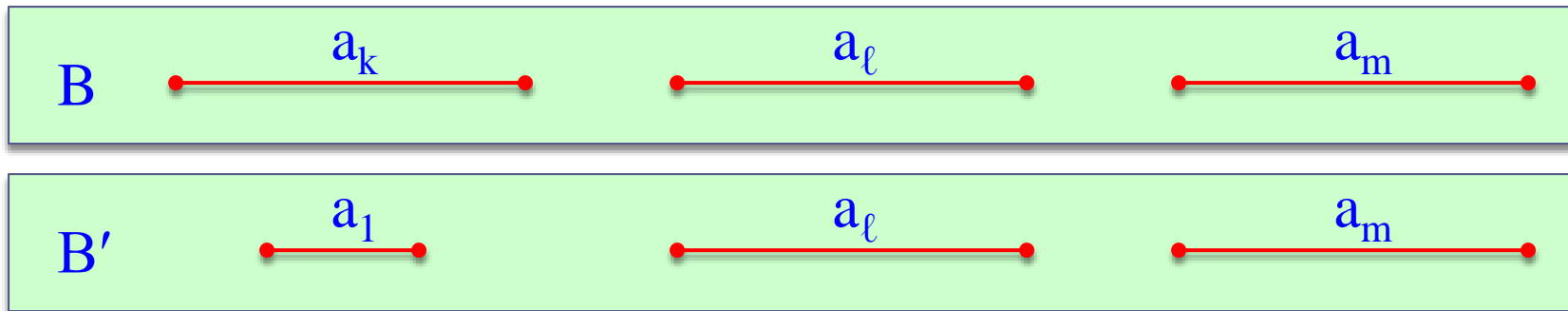
# Greedy Choice Property - Proof

**Theorem**: There exists an optimal solution  $A \subseteq S$  such that  $a_1 \in A$

**Proof**: Consider an arbitrary optimal solution  $B = \{a_k, a_\ell, a_m, \dots\}$ , where  $f_k < f_\ell < f_m < \dots$

**If  $k = 1$** , then  $B$  starts with  $a_1$ , and the proof is complete

**If  $k > 1$** , then create another solution  $B'$  by replacing  $a_k$  with  $a_1$ . Since  $f_1 \leq f_k$ ,  $B'$  is guaranteed to be valid, and  $|B'| = |B|$ , hence also optimal





# Elements of Greedy Strategy

---

- How can you judge whether
- A greedy algorithm will solve a particular optimization problem?

## Two key ingredients

- Greedy choice property
- Optimal substructure property

# Key Ingredients of Greedy Strategy

---

- **Greedy Choice Property:** A globally optimal solution can be arrived at by making locally optimal (greedy) choices
- In **DP**, we make a choice at each step but the choice may depend on the solutions to subproblems
- In **Greedy Algorithms**, we make the choice that seems best at that moment then solve the subproblems arising after the choice is made
  - The choice may depend on choices so far, but it cannot depend on any future choice or on the solutions to subproblems
- DP solves the problem bottom-up
- Greedy usually progresses in a top-down fashion by making one greedy choice after another reducing each given problem instance to a smaller one

# Key Ingredients: Greedy Choice Property

---

- We must prove that a greedy choice at each step yields a globally optimal solution
- The proof examines a globally optimal solution
- Shows that the soln can be modified so that a **greedy choice made as the first step** reduces the problem to a similar but smaller subproblem
- Then **induction** is applied to show that a greedy choice can be used at each step
- Hence, this induction proof reduces the proof of correctness to demonstrating that an optimal solution must exhibit **optimal substructure** property

# Key Ingredients: Greedy Choice Property

- How to prove the greedy choice property?
  1. Consider the greedy choice  $c$
  2. Assume that there is an optimal solution  $B$  that doesn't contain  $c$ .
  3. Show that it is possible to convert  $B$  to another optimal solution  $B'$ , where  $B'$  contains  $c$ .
- Example: Activity selection algorithm
  - Greedy choice:  $a_1$  (the activity with the earliest finish time)
  - Consider an optimal solution  $B$  without  $a_1$
  - Replace the first activity in  $B$  with  $a_1$  to construct  $B'$
  - Prove that  $B'$  must be an optimal solution

# Key Ingredients: Optimal Substructure

---

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems

Example: Activity selection problem  $S$

If an optimal solution  $A$  to  $S$  begins with activity  $a_1$  then the set of activities

$$A' = A - \{a_1\}$$

is an optimal solution to the activity selection problem

$$S' = \{a_i \in S : s_i \geq f_1\}$$

# Key Ingredients: Optimal Substructure

---

- Optimal substructure property is exploited by both Greedy and dynamic programming strategies
- Hence one may
  - Try to generate a dynamic programming soln to a problem when a greedy strategy suffices → inefficient
  - Or, may mistakenly think that a greedy soln works when in fact a DP soln is required → incorrect

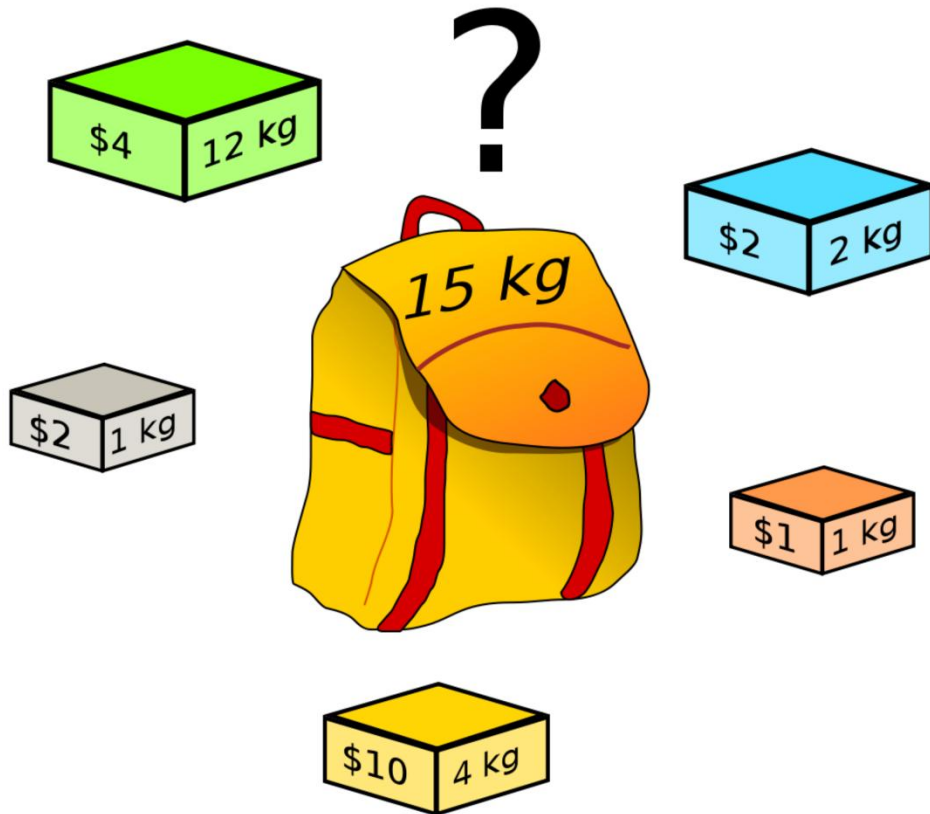
Example: Knapsack Problems( $S, w$ )

# CS473 - Algorithms I



## Knapsack Problems

# Knapsack Problem



*Image source: Wikimedia Commons*

- Each item  $i$  has:
  - weight  $w_i$
  - value  $v_i$
- A thief has a knapsack of weight capacity  $W$
- Which items to choose to maximize the value of the items in the knapsack?



# Knapsack Problem: Two Versions

## □ The 0-1 knapsack problem:

Each item is discrete.

Each item either chosen as a whole or not chosen.

Examples: TV, laptop, gold bricks, etc.

## □ The fractional knapsack problem:

Can choose fractional part of each item.

If item  $i$  has weight  $w_i$ , we can choose any amount  $\leq w_i$

Examples: Gold dust, silver dust, rice, etc.

# Knapsack Problems

---

- **The 0-1 Knapsack Problem**( $S, W$ )
  - A thief robbing a store finds  $n$  items  $S = \{I_1, I_2, \dots, I_n\}$ , the  $i$ th item is worth  $v_i$  dollars and weighs  $w_i$  pounds, where  $v_i$  and  $w_i$  are integers
  - He wants to take as valuable a load as possible, but he can carry at most  $W$  pounds in his knapsack, where  $W$  is an integer
  - The thief cannot take a fractional amount of an item
- **The Fractional Knapsack Problem** ( $S, W$ )
  - The scenario is the same
  - But, the thief can take fractions of items rather than having to make binary (0-1) choice for each item

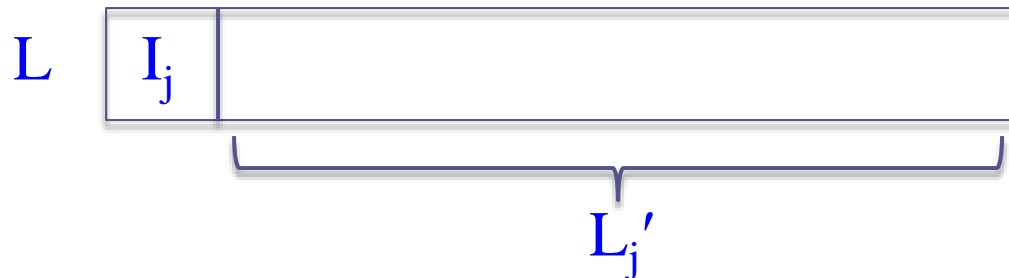
# Optimal Substructure Property for the 0-1 Knapsack Problem (S, W)

- Consider an optimal load  $L$  for the problem (S, W).
- Let  $I_j$  be an item chosen in  $L$  with weight  $w_j$
- Assume we remove  $I_j$  from  $L$ , and let:

$$L_j' = L - \{I_j\}$$

$$S_j' = S - \{I_j\}$$

$$W_j' = W - w_j$$



What can we say about the optimal substructure property?

# Optimal Substructure Property for the 0-1 Knapsack Problem (S, W)

$$L_j' = L - \{I_j\}$$

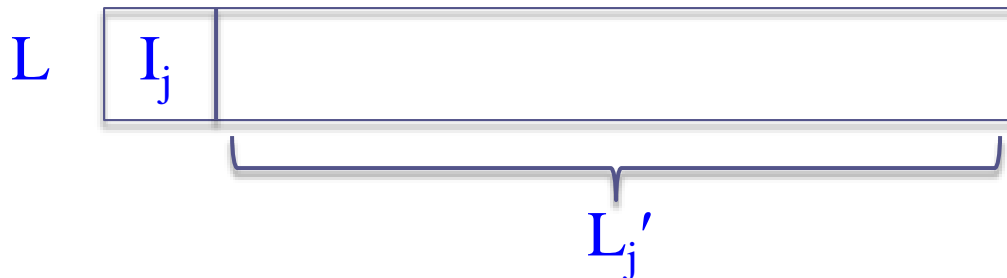
$$S_j' = S - \{I_j\}$$

$$W_j' = W - w_j$$

Optimal substructure property:

$L_j'$  must be an optimal solution for  $(S_j', W_j')$

Why?



# Optimal Substructure Property for the 0-1 Knapsack Problem $(S, W)$

$$L_j' = L - \{I_j\}$$

$$S_j' = S - \{I_j\}$$

$$W_j' = W - w_j$$

Optimal substructure:  $L_j'$  must be an optimal solution for  $(S_j', W_j')$

Proof: By contradiction, assume there is a solution  $B_j'$  for  $(S_j', W_j')$ , which is better than  $L_j'$ .

We can construct a solution  $B$  for the original problem  $(S, W)$  as:  $B = B_j' \cup \{I_j\}$ .

The total value of  $B$  is now higher than  $L$ , which is a contradiction because  $L$  is optimal for  $(S, W)$ .

Q.E.D.

# Optimal Substructure Property for the Fractional Knapsack Problem (S, W)

- Consider an optimal solution  $L$  for (S, W)
- If we remove a weight  $0 < w \leq w_j$  of item  $j$  from optimal load  $L$

The remaining load

$$L_j' = L - \{w \text{ pounds of } I_j\}$$

must be a most valuable load weighing at most

$$W_j' = W - w$$

pounds that the thief can take from

$$S_j' = S - \{I_j\} \cup \{w_j - w \text{ pounds of } I_j\}$$

- That is,  $L_j'$  should be an optimal soln to the  
Fractional Knapsack Problem( $S_j'$ ,  $W_j'$ )

# Knapsack Problems

- Two different problems:
  - ▣ 0-1 knapsack problem
  - ▣ Fractional knapsack problem
- The problems are similar.
- Both problems have optimal substructure property.
- Which algorithm to solve each problem?

# Fractional Knapsack Problem

- Can we use a greedy algorithm?
- Greedy choice: Take as much as possible from the item with the largest value per pound  $v_i/w_i$

- Does the greedy choice property hold?

Let  $j$  be the item with the largest value per pound  $v_j/w_j$

Need to prove that there is an optimal load that has as much  $j$  as possible.

*Proof: Consider an optimal solution  $L$  that does not have the maximum amount of item  $j$ . We could replace the items in  $L$  with item  $j$  until  $L$  has maximum amount of  $j$ .  $L$  would still be optimal, because item  $j$  has the highest value per pound.*

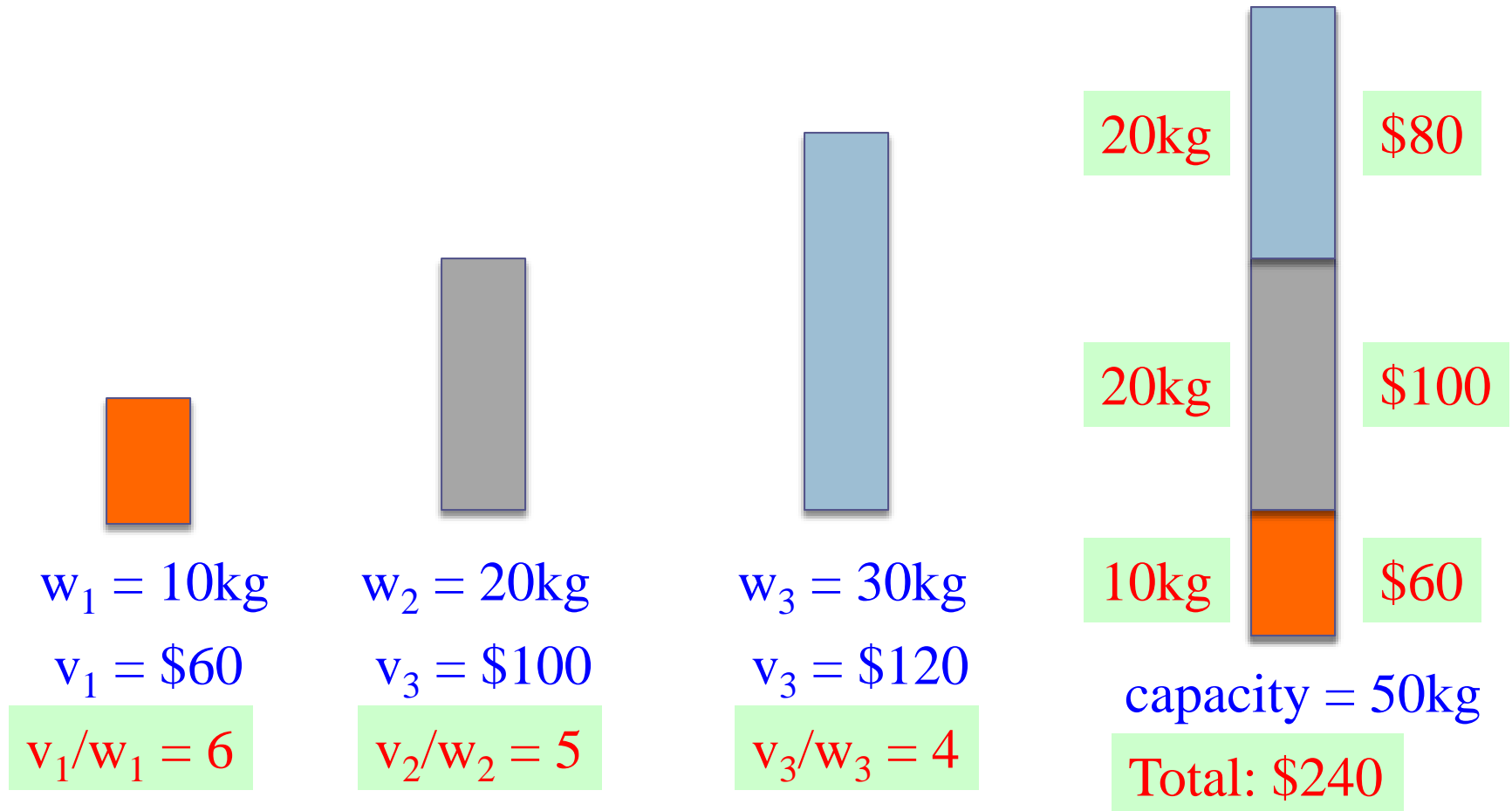


# Greedy Solution to Fractional Knapsack

---

- 1) Compute the value per pound  $v_i / w_i$  for each item
  - 2) The thief begins by taking, as much as possible, of the item with the greatest value per pound
  - 3) If the supply of that item is exhausted before filling the knapsack, then he takes, as much as possible, of the item with the next greatest value per pound
  - 4) Repeat (2-3) until his knapsack becomes full
- Thus, by sorting the items by value per pound the greedy algorithm runs in  $O(n \lg n)$  time

# Fractional Knapsack Problem: Example



# 0-1 Knapsack Problem

Can we use the same greedy algorithm?

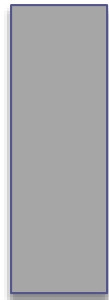
Is there a better solution?



$$w_1 = 10\text{kg}$$

$$v_1 = \$60$$

$$v_1/w_1 = 6$$



$$w_2 = 20\text{kg}$$

$$v_2 = \$100$$

$$v_2/w_2 = 5$$



$$w_3 = 30\text{kg}$$

$$v_3 = \$120$$

$$v_3/w_3 = 4$$

20kg

\$100

10kg

\$60

capacity = 50kg

Total: \$160



# 0-1 Knapsack Problem

The optimal solution for this problem is:

This solution cannot be obtained using the greedy algorithm



$$w_1 = 10\text{kg}$$

$$v_1 = \$60$$

$$v_1/w_1 = 6$$



$$w_2 = 20\text{kg}$$

$$v_2 = \$100$$

$$v_2/w_2 = 5$$



$$w_3 = 30\text{kg}$$

$$v_3 = \$120$$

$$v_3/w_3 = 4$$

30kg

\$120

20kg

\$100

capacity = 50kg

Total: \$220

# 0-1 Knapsack Problem

---

- When we consider an item  $I_j$  for inclusion we must compare the solutions to two subproblems
    - Subproblems in which  $I_j$  is included and excluded
  - The problem formulated in this way gives rise to many **overlapping subproblems** (a key ingredient of DP)
- In fact, dynamic programming can be used to solve the 0-1 Knapsack problem

# 0-1 Knapsack Problem

---

- A thief robbing a store containing  $n$  articles  $\{a_1, a_2, \dots, a_n\}$ 
  - The value of  $i$ th article is  $v_i$  dollars ( $v_i$  is integer)
  - The weight of  $i$ th article is  $w_i$  kg ( $w_i$  is integer)
- Thief can carry at most  $W$  kg in his knapsack
- Which articles should he take to maximize the value of his load?
- Let  $K_{n,W} = \{a_1, a_2, \dots, a_n : W\}$  denote 0-1 knapsack problem
- Consider the solution as a sequence of  $n$  decisions
  - i.e.,  $i$ th decision: whether thief should pick  $a_i$  for optimal load

# Optimal Substructure Property

□ Notation:  $K_{n,W}$ :

The items to choose from:  $\{a_1, \dots, a_n\}$

The knapsack capacity:  $W$

□ Consider an optimal load  $L$  for problem  $K_{n,W}$

□ Let's consider two cases:

1)  $a_n$  is in  $L$

2)  $a_n$  is not in  $L$

# Optimal Substructure Property

## □ Case 1: If $a_n \in L$

What can we say about the optimal substructure?

$L - \{a_n\}$  must be optimal for  $K_{n-1, W-w_n}$

$K_{n-1, W-w_n}$ :

The items to choose from  $\{a_1, \dots, a_{n-1}\}$

The knapsack capacity:  $W - w_n$

## □ Case 2: If $a_n \notin L$

What can we say about the optimal substructure?

$L$  must be optimal for  $K_{n-1, W}$

$K_{n-1, W}$ :

The items to choose from  $\{a_1, \dots, a_{n-1}\}$

The knapsack capacity:  $W$



# Optimal Substructure Property

- In other words, optimal solution to  $K_{n,W}$  contains an optimal solution to:

either:  $K_{n-1, W-w_n}$  (if  $a_n$  is selected)

or:  $K_{n-1, W}$  (if  $a_n$  is not selected)

# Recursive Formulation

$c[i, w]$ : The value of an optimal solution to  $K_{i,w}$   
where  $K_{i,w}: \{a_1, \dots, a_i: w\}$

$$c[i, w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w], & \text{if } w_i > w \\ \max\{v_i + c[i-1, w - w_i], c[i-1, w]\} & \text{o/w} \end{cases}$$

# 0-1 Knapsack Problem

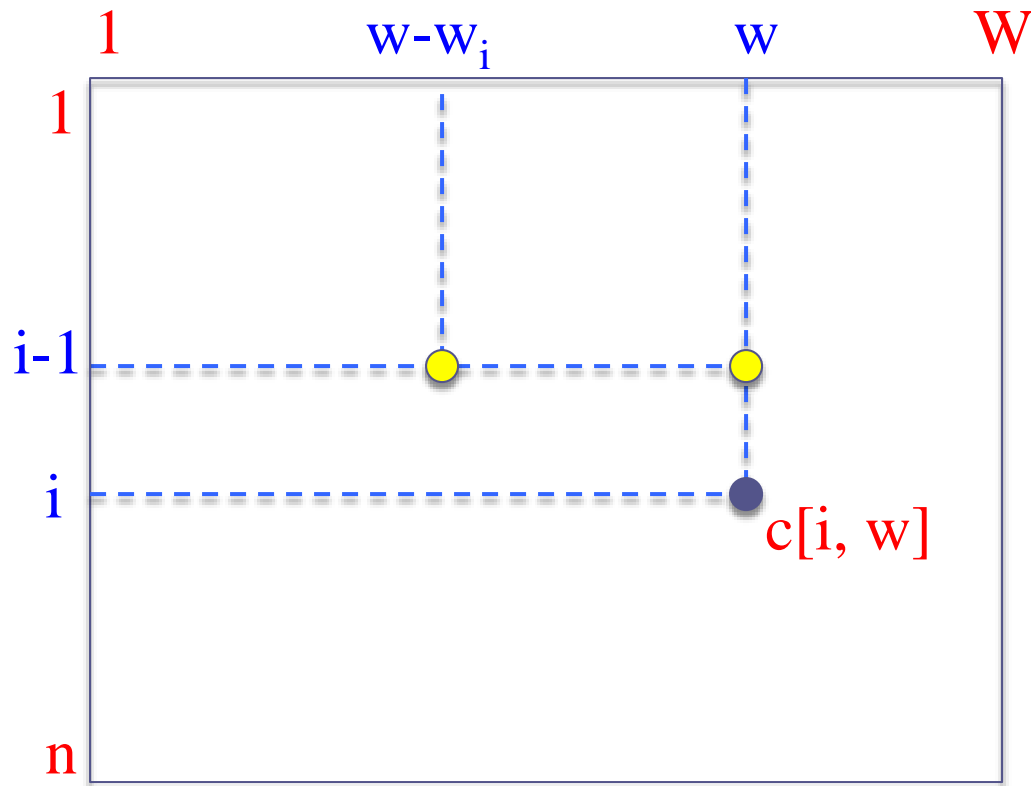
---

## Recursive definition for value of optimal soln:

This recurrence says that an optimal solution  $S_{i,w}$  for  $K_{i,w}$

- either contains  $a_i \Rightarrow c(S_{i,w}) = v_i + c(S_{i-1,w-w_i})$
- or does not contain  $a_i \Rightarrow c(S_{i,w}) = c(S_{i-1,w})$
- If thief decides to pick  $a_i$ 
  - He takes  $v_i$  value and he can choose from  $\{a_1, a_2, \dots, a_{i-1}\}$  up to the weight limit  $w - w_i$  to get  $c[i-1, w - w_i]$
- If he decides not to pick  $a_i$ 
  - He can choose from  $\{a_1, a_2, \dots, a_{i-1}\}$  up to the weight limit  $w$  to get  $c[i-1, w]$
- The better of these two choices should be made

# Bottom-up Computation



Need to process:

$c[i, w]$

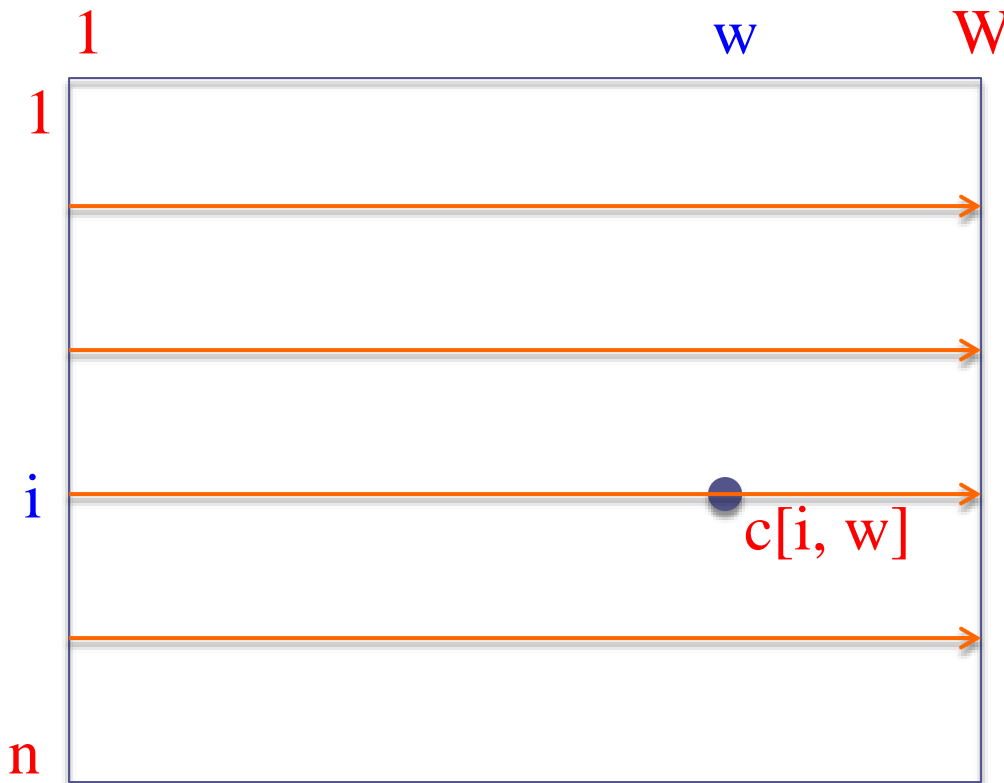
after computing:

$c[i-1, w]$ ,

$c[i-1, w-w_i]$

for all  $w_i < w$

# Bottom-up Computation



```
for i ← 1 to n
  for w ← 1 to W
    ...
    ...
    c[i, w] =
```

# DP Solution to 0-1 Knapsack

**KNAP0-1**( $v$ ,  $w$ ,  $n$ ,  $W$ )

**for**  $\omega \leftarrow 0$  **to**  $W$  **do**

$c[0, \omega] \leftarrow 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$c[i, 0] \leftarrow 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

**for**  $\omega \leftarrow 1$  **to**  $W$  **do**

**if**  $w_i \leq \omega$  **then**

$c[i, \omega] \leftarrow \max\{v_i + c[i-1, \omega - w_i], c[i-1, \omega]\}$

**else**

$c[i, \omega] \leftarrow c[i-1, \omega]$

**return**  $c[n, W]$

$c$  is an  $(n+1) \times (W+1)$   
array;  $c[0..n : 0..W]$

**Note:** table is computed  
in row-major order

Run time:  $T(n) = \Theta(nW)$

# Constructing an Optimal Solution

- No extra data structure is maintained to keep track of the choices made to compute  $c[i, w]$

i.e. The choice of whether choosing item  $i$  or not

- Possible to understand the choice done by comparing  $c[i, w]$  with  $c[i-1, w]$

If  $c[i, w] = c[i-1, w]$  then it means item  $i$  was assumed to be not chosen for the best  $c[i, w]$

# Finding the Set $S$ of Articles in an Optimal Load

**SOLKNAP0-1**( $a, v, w, n, W, c$ )

$i \leftarrow n$ ;  $\omega \leftarrow W$

$S \leftarrow \emptyset$

**while**  $i > 0$  **do**

**if**  $c[i, \omega] = c[i-1, \omega]$  **then**

$i \leftarrow i-1$

**else**

$S \leftarrow S \cup \{a_i\}$

$\omega \leftarrow \omega - w_i$

$i \leftarrow i-1$

**return**  $S$



# CS473 - Algorithms I



## Huffman Codes

# Huffman Codes for Compression

- Widely used and very effective for data compression
- Savings of 20% - 90% typical  
*(depending on the characteristics of the data)*
- In summary: Huffman's greedy algorithm uses a table of frequencies of character occurrences to build up an optimal way of representing each character as a binary string.

# Binary String Representation - Example

- Consider a data file with:

- ▣ 100K characters
- ▣ Each character is one of {a, b, c, d, e, f}

- Frequency of each character in the file:

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

- Binary character code: Each character is represented by a unique binary string.
- Intuition: Frequent characters  $\Leftrightarrow$  shorter codewords  
Infrequent characters  $\Leftrightarrow$  longer codewords

# Binary String Representation - Example

	a	b	c	d	e	f
frequency	45K	13K	12K	16K	9K	5K
fixed-length	000	001	010	011	100	101
variable-length(1)	0	101	100	111	1101	1100
variable-length(2)	0	10	110	1110	11110	11111

How many total bits needed for fixed-length codewords?

$$100K * 3 = 300K \text{ bits}$$

How many total bits needed for variable-length(1) codewords?

$$45K * 1 + 13K * 3 + 12K * 3 + 16K * 3 + 9K * 4 + 5K * 4 = 224K$$

How many total bits needed for variable-length(2) codewords?

$$45K * 1 + 13K * 2 + 12K * 3 + 16K * 4 + 9K * 5 + 5K * 5 = 241K$$

# Prefix Codes

- Prefix codes: No codeword is also a prefix of some other codeword
- Example:

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

- It can be shown that:  
*Optimal data compression is achievable with a prefix code*
- In other words, optimality is not lost due to prefix-code restriction.

# Prefix Codes: Encoding

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

- Encoding: Concatenate the codewords representing each character of the file
- Example: Encode file “abc” using the codewords above  
 $abc \Rightarrow 0.101.100 \Rightarrow 0101100$

*Note: “.” denotes the concatenation operation. It is just for illustration purposes, and does not exist in the encoded string*

# Prefix Codes: Decoding

- Decoding is quite simple with a prefix code
- The first codeword in an encoded file is unambiguous  
*because no codeword is a prefix of any other*
- *Decoding algorithm:*
  1. Identify the initial codeword
  2. Translate it back to the original character
  3. Remove it from the encoded file
  4. Repeat the decoding process on the remainder of the encoded file.

# Prefix Codes: Decoding - Example

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

Example: Decode encoded file 001011101

001011101  $\Rightarrow$  0.01011101  $\Rightarrow$  0.0.1011101

0.0.101.1101  $\Rightarrow$  0.0.101.1101  $\Rightarrow$  aabe



# Prefix Codes

---

Convenient representation for the prefix code:  
a binary tree whose leaves are the given characters

Binary codeword for a character is the path from the root to that character in the binary tree

“0” means “go to the left child”

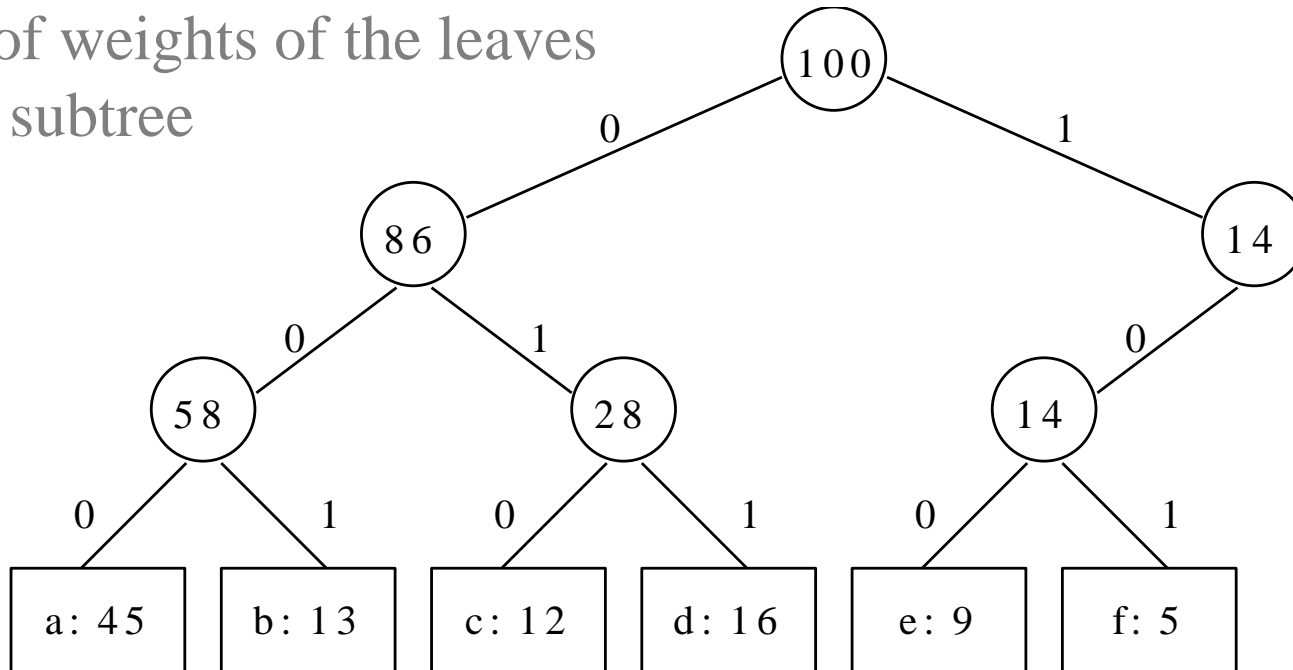
“1” means “go to the right child”

# Binary Tree Representation of Prefix Codes

---

Weight of an internal node:

sum of weights of the leaves  
in its subtree

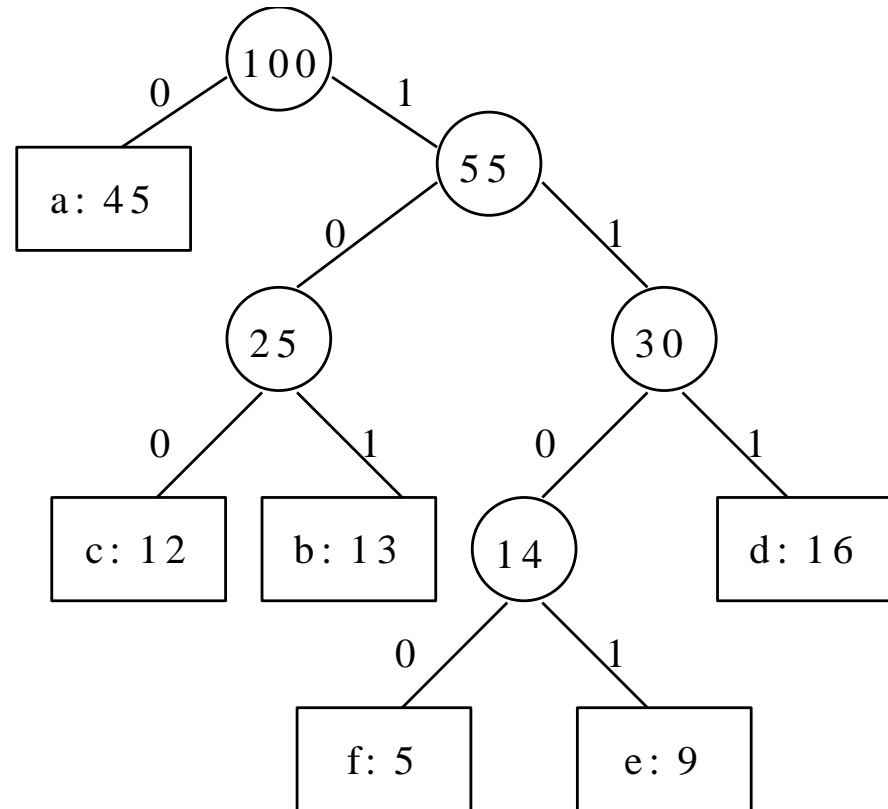


The binary tree corresponding to the **fixed-length** code

# Binary Tree Representation of Prefix Codes

Weight of an internal node:  
sum of weights of the leaves  
in its subtree

The binary tree corresponding  
to the **optimal variable-length**  
code



An optimal code for a file is always represented by a **full binary tree**

# Full Binary Tree Representation of Prefix Codes

---

Consider an **FBT** corresponding to an optimal prefix code

It has  $|C|$  leaves (external nodes)

One for each letter of the alphabet where  $C$  is the alphabet from which the characters are drawn

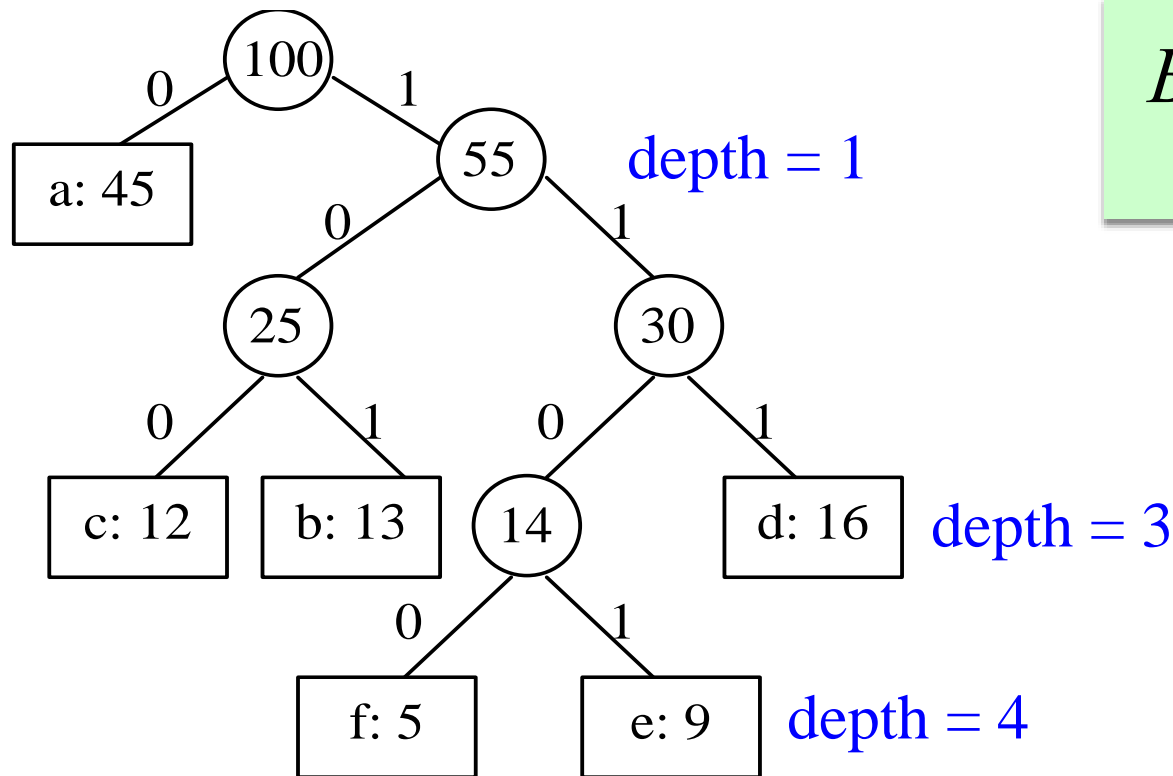
**Lemma:** An **FBT** with  $|C|$  external nodes has exactly  $|C|-1$  internal nodes

# Full Binary Tree Representation of Prefix Codes

- Consider an FBT  $T$ , corresponding to a prefix code.
- Notation:
  - $f(c)$ : frequency of character  $c$  in the file
  - $d_T(c)$ : depth of  $c$ 's leaf in the FBT  $T$
  - $B(T)$ : the number of bits required to encode the file
- What is the length of the codeword for  $c$ ?
  - $d_T(c)$ , same as the depth of  $c$  in  $T$
- How to compute  $B(T)$ , cost of tree  $T$ ?

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

# Cost Computation - Example



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

$$\begin{aligned} B(T) &= 45 * 1 + 12 * 3 + \\ &\quad 13 * 3 + 16 * 3 + \\ &\quad 5 * 4 + 9 * 4 \\ &= 224 \end{aligned}$$

# Prefix Codes

---

**Lemma:** Let each internal node  $i$  is labeled with the sum of the weight  $w(i)$  of the leaves in its subtree

Then  $B(T) = \sum_{c \in C} f(c) d_T(c) = \sum_{i \in I_T} w(i)$  where

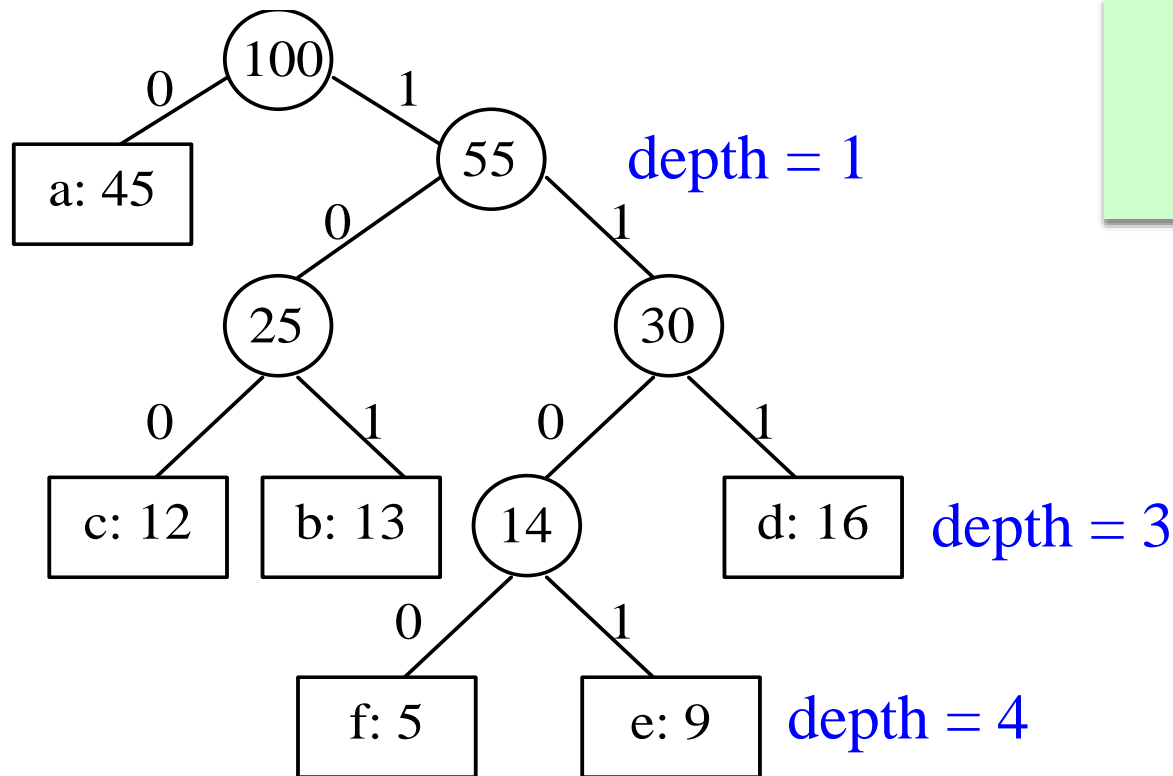
$I_T$  denotes the set of internal nodes in  $T$

**Proof:** Consider a leaf node  $c$  with  $f(c)$  &  $d_T(c)$

Then,  $f(c)$  appears in the weights of  $d_T(c)$  internal node along the path from  $c$  to the root

Hence,  $f(c)$  appears  $d_T(c)$  times in the above summation

# Cost Computation - Example



$$B(T) = \sum_{i \in I_T} w(i)$$

$$\begin{aligned} B(T) &= 100 + 55 + \\ &\quad 25 + 30 + 14 \\ &= 224 \end{aligned}$$



# Constructing a Huffman Code

---

**Problem Formulation**: For a given character set  $C$ , construct an optimal prefix code with the minimum total cost

**Huffman** invented a **greedy algorithm** that constructs an optimal prefix code called a **Huffman code**

The greedy algorithm

- builds the **FBT** corresponding to the optimal code in a **bottom-up** manner
- begins with a set of  $|C|$  leaves
- performs a sequence of  $|C|-1$  “**merges**” to create the final tree

# Constructing a Huffman Code

---

A **priority queue**  $Q$ , keyed on  $f$ , is used to identify the two **least-frequent** objects to merge

The result of the **merger** of two objects is a **new object**

- inserted into the priority queue according to its frequency
- which is the sum of the frequencies of the two objects merged

# Constructing a Huffman Code

---

HUFFMAN( $C$ )

$n \leftarrow |C|$

$Q \leftarrow \text{BUILD-HEAP}(C)$

for  $i \leftarrow 1$  to  $n - 1$  do

$z \leftarrow \text{ALLOCATE-NODE}()$

$x \leftarrow \text{left}[z] \leftarrow \text{EXTRACT-MIN}(Q)$

$y \leftarrow \text{right}[z] \leftarrow \text{EXTRACT-MIN}(Q)$

$f[z] \leftarrow f[x] + f[y]$

    INSERT( $Q, z$ )

return EXTRACT-MIN( $Q$ )     $\Delta$  only one object left in  $Q$

Priority queue is implemented as a binary heap

Initiation of  $Q$  (BUILD-HEAP):  $O(n)$  time

EXTRACT-MIN & INSERT take  $O(\lg n)$  time on  $Q$  with  $n$  objects

$$T(n) = \sum_{i=1}^n \lg i = O(\lg(n!)) = O(n \lg n)$$

# Constructing a Huffman Code - Example

Start with one leaf node for each character

The 2 nodes with the least frequencies: **f** & **e**

Merge **f** & **e** and create an internal node

Set the internal node frequency to  $5 + 9 = 14$

f: 5

e: 9

c: 12

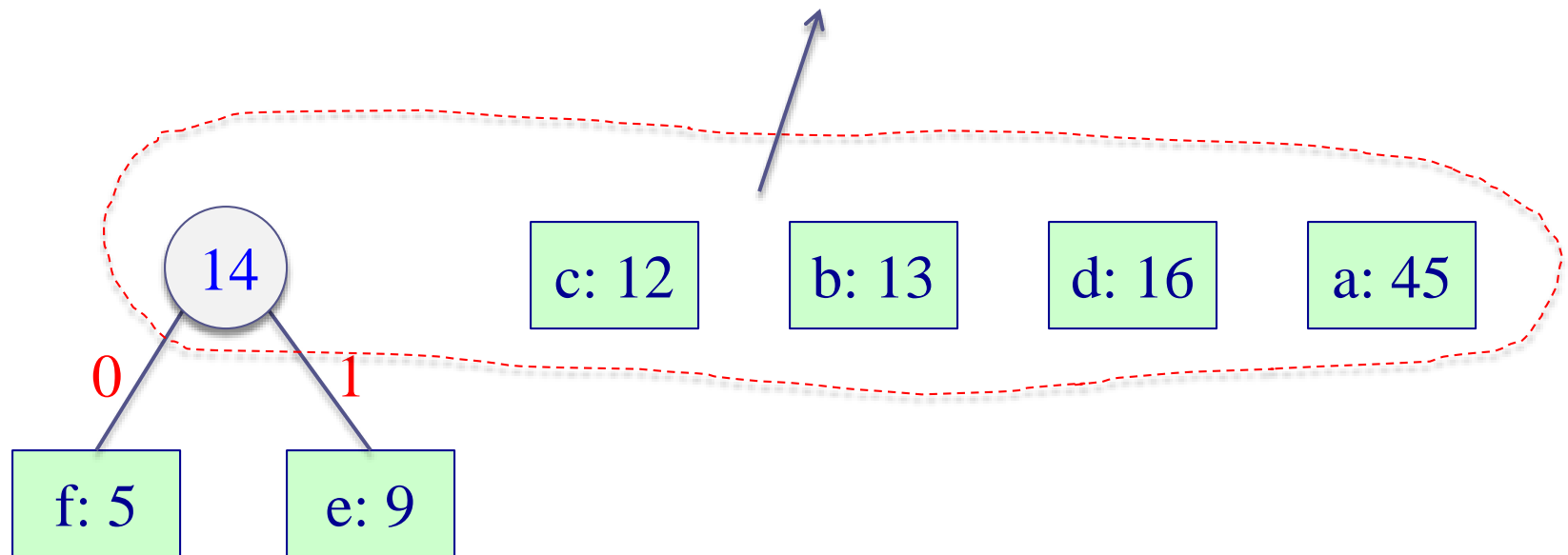
b: 13

d: 16

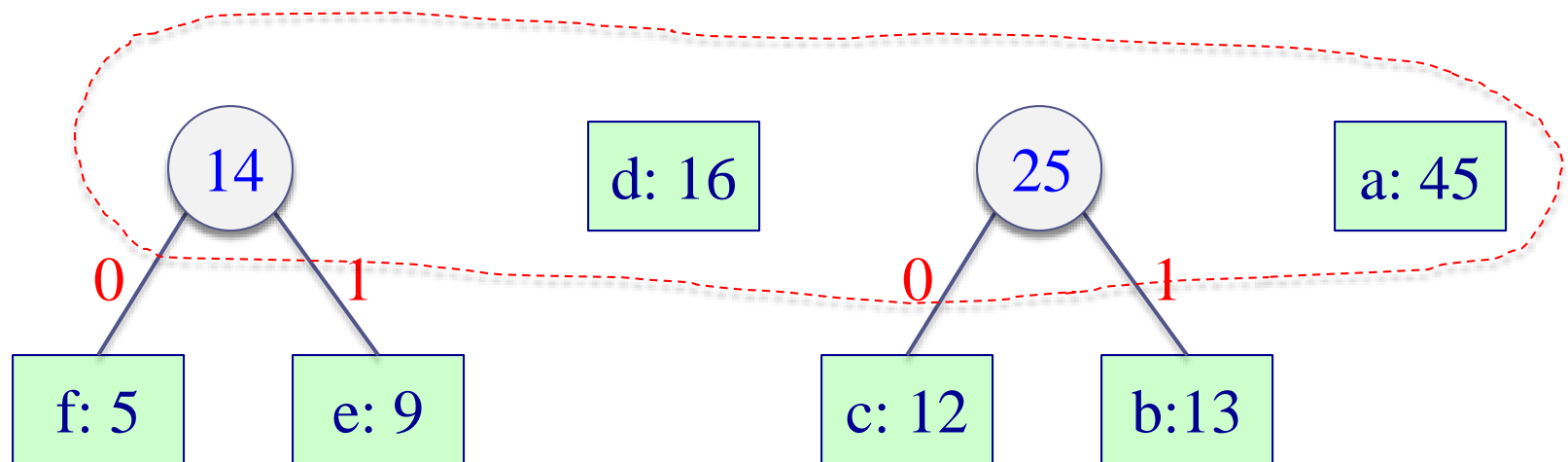
a: 45

# Constructing a Huffman Code - Example

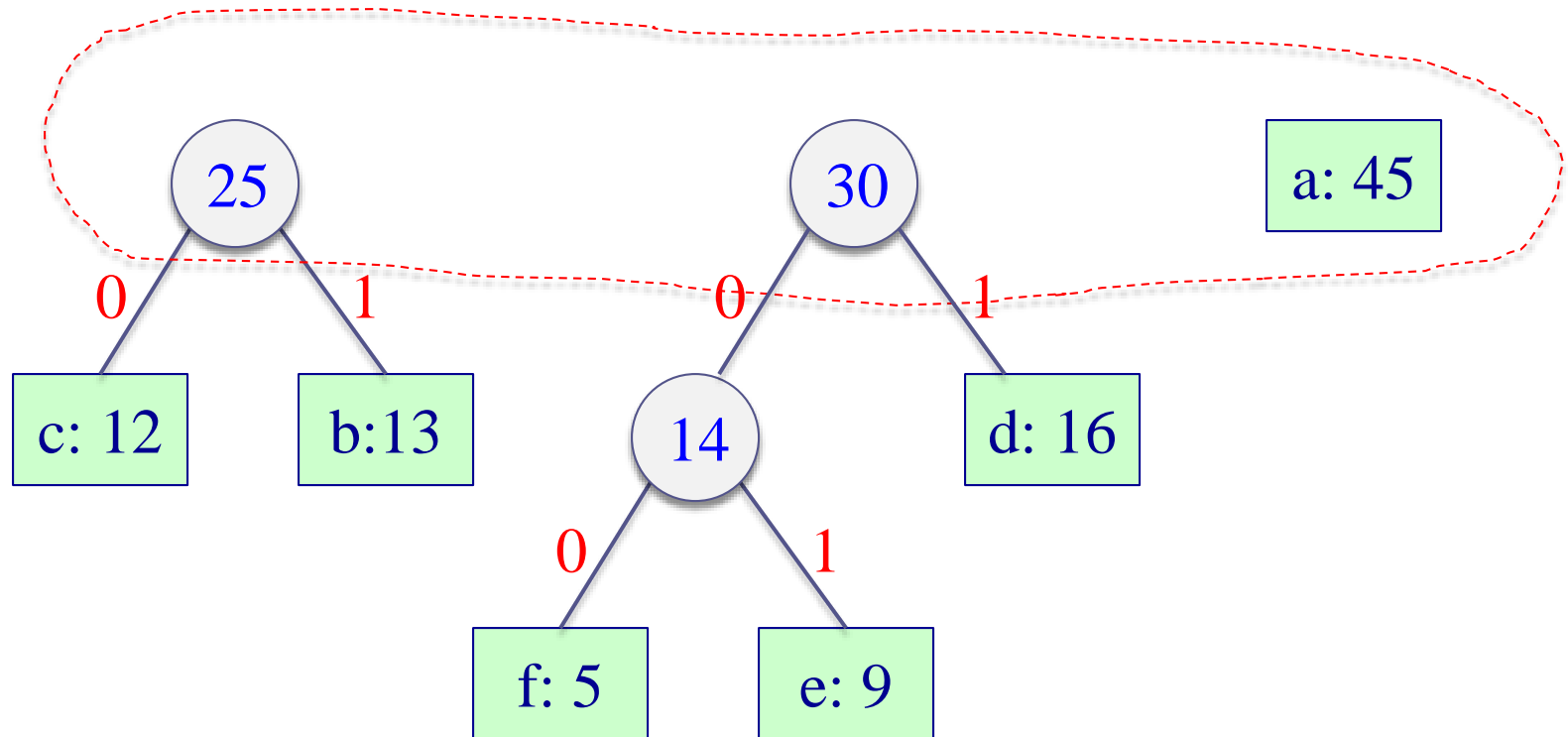
The 2 nodes with least frequency: b & c



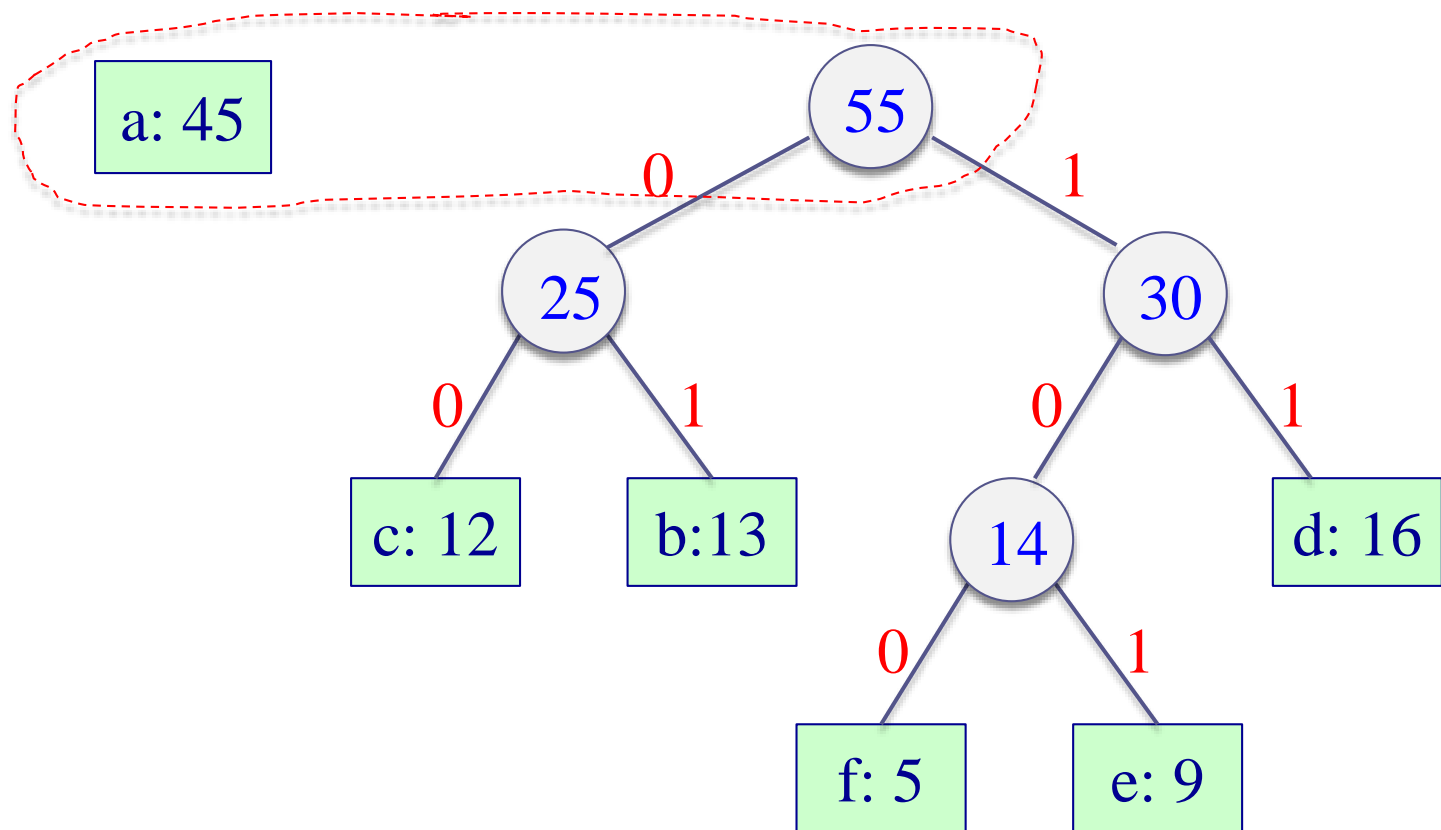
# Constructing a Huffman Code - Example



# Constructing a Huffman Code - Example

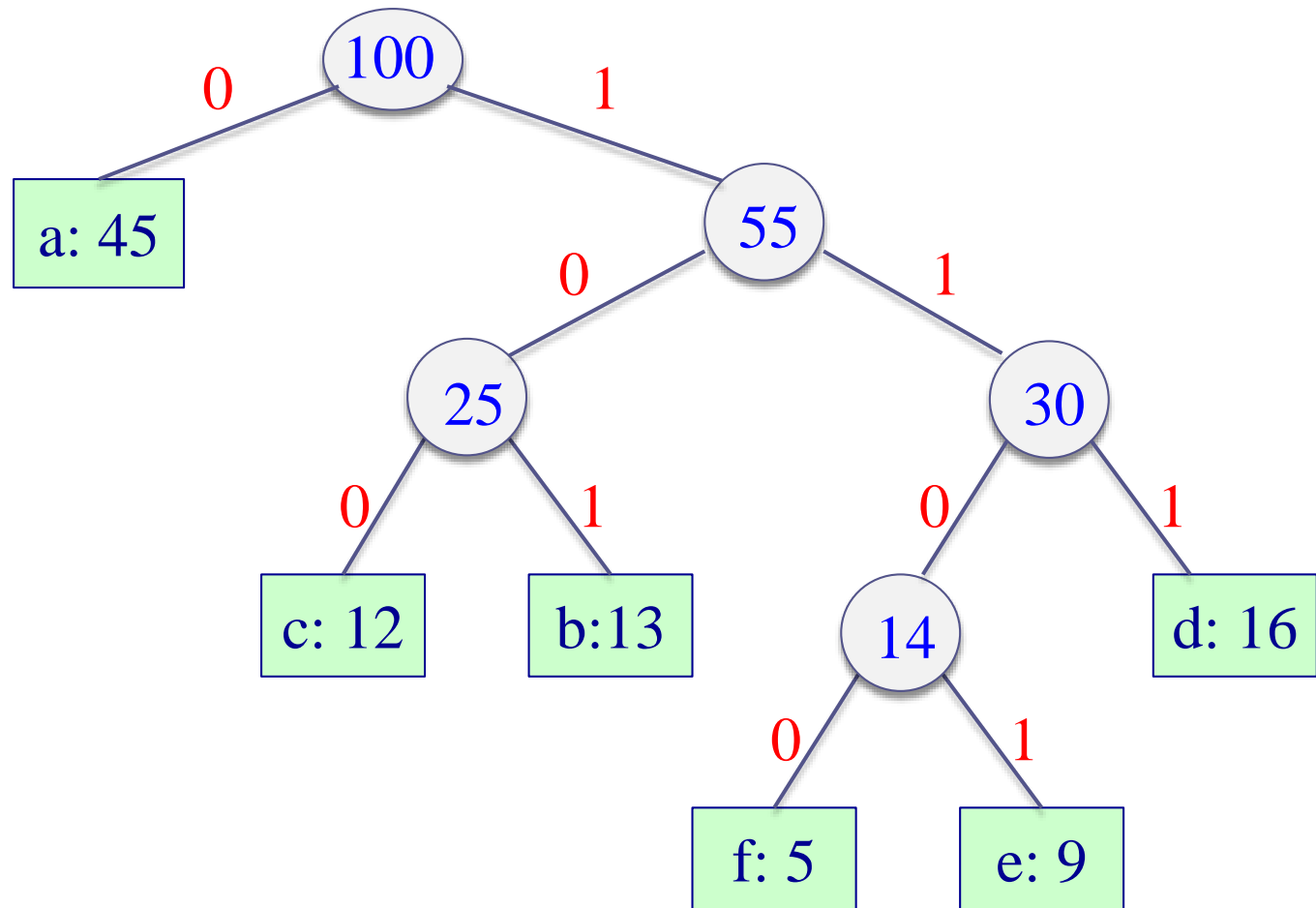


# Constructing a Huffman Code - Example





# Constructing a Huffman Code - Example



# Correctness Proof of Huffman's Algorithm

- We need to prove:
  - ▣ The greedy choice property
  - ▣ The optimal substructure property
- What is the greedy step in Huffman's algorithm?

Merging the two characters with the lowest frequencies
- We will first prove the greedy choice property

# Greedy Choice Property

**Lemma 1**: Let  $x$  &  $y$  be two characters in  $C$  having the **lowest frequencies**.

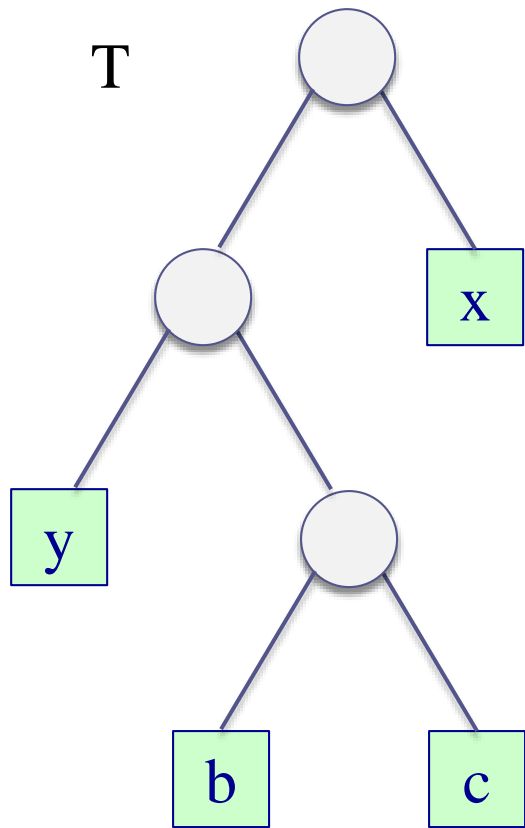
Then,  $\exists$  an **optimal prefix code** for  $C$  in which the codewords for  $x$  &  $y$  have the same length and differ only in the last bit

Note: If  $x$  &  $y$  are merged in Huffman's algorithm, their codewords are guaranteed to have the same length and they will differ only in the last bit. Lemma 1 states that there exists an optimal solution where this is the case.

# Greedy Choice Property - Proof

- Outline of the proof:
  - Start with an arbitrary optimal solution
  - Convert it to an optimal solution that satisfies the greedy choice property.
  
- *Proof*: Let **T** be an arbitrary optimal solution where:
  - b & c** are the sibling leaves with the **max depth**
  - x & y** are the characters with the **lowest frequencies**

# Greedy Choice Property - Proof



Reminder:

$b$  &  $c$  are the nodes with max depth  
 $x$  &  $y$  are the nodes with min freq.

Without loss of generality, assume:

$$f(x) \leq f(y)$$

$$f(b) \leq f(c)$$

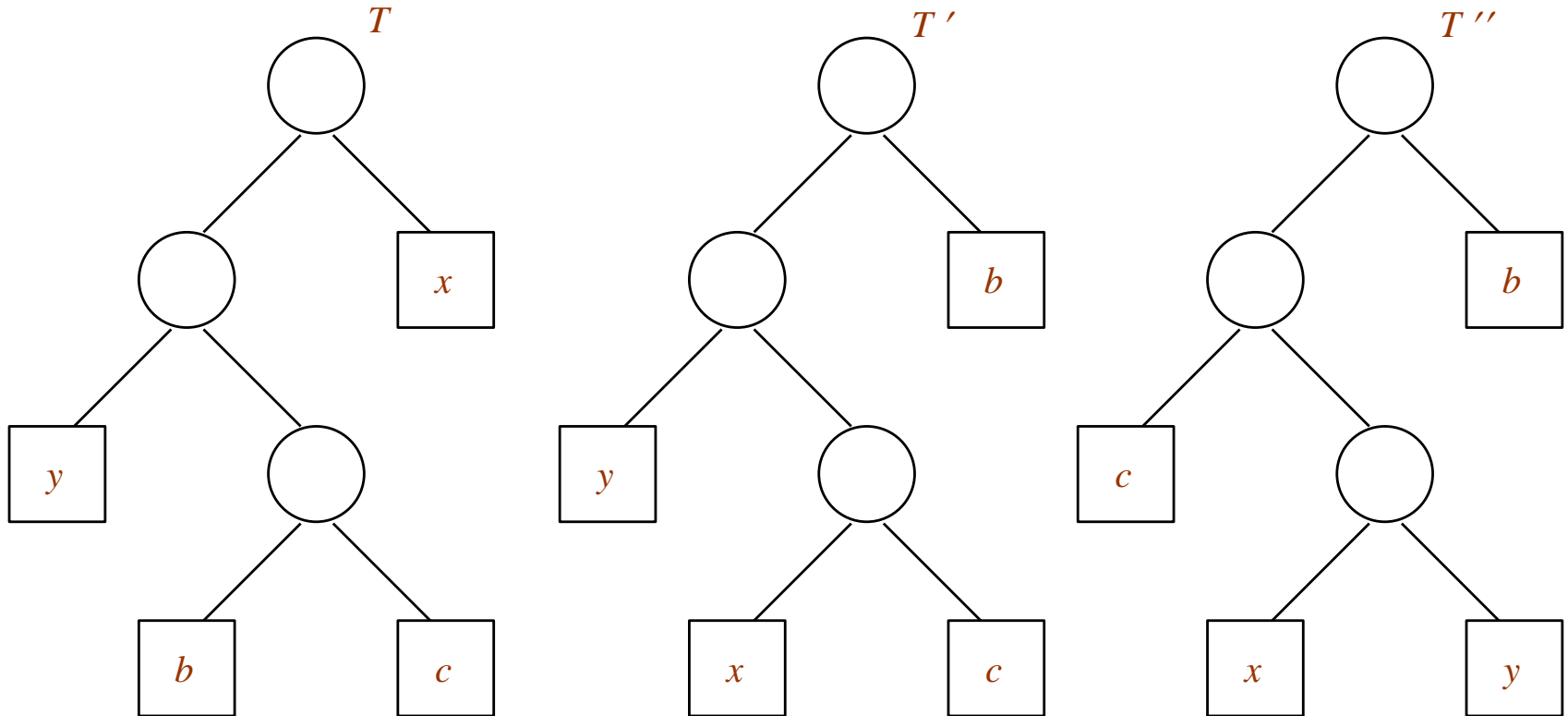
Then, it must be the case that:

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

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

# Greedy Choice Property - Proof

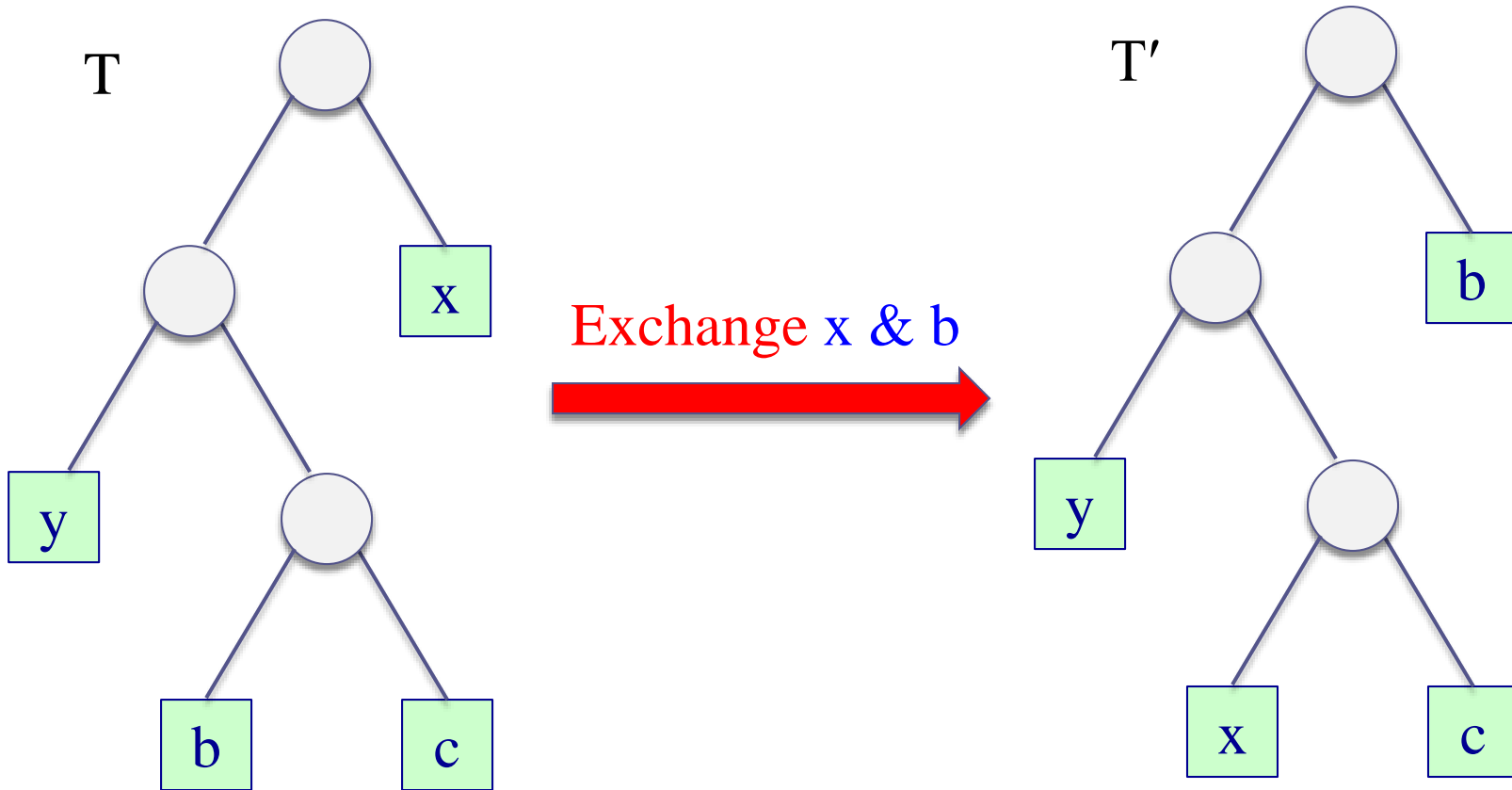
---



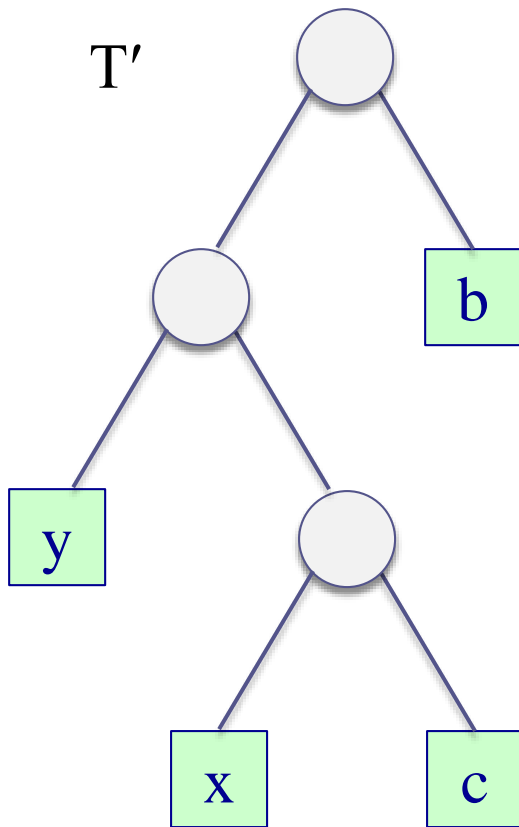
$T \Rightarrow T'$  : exchange the positions of the leaves  $b$  &  $x$

$T' \Rightarrow T''$  : exchange the positions of the leaves  $c$  &  $y$

# Greedy Choice Property - Proof



# Greedy Choice Property - Proof



**Reminder:** Cost of tree T':

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

How does  $B(T')$  compare to  $B(T)$ ?

**Reminder:**  $f(x) \leq f(b)$

$$d_{T'}(x) = d_T(b) \text{ and } d_{T'}(b) = d_T(x)$$



# Greedy Choice Property - Proof

**Reminder:**  $f(x) \leq f(b)$

$$d_{T'}(x) = d_T(b) \text{ and } d_{T'}(b) = d_T(x)$$

The difference in cost between T and T':

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

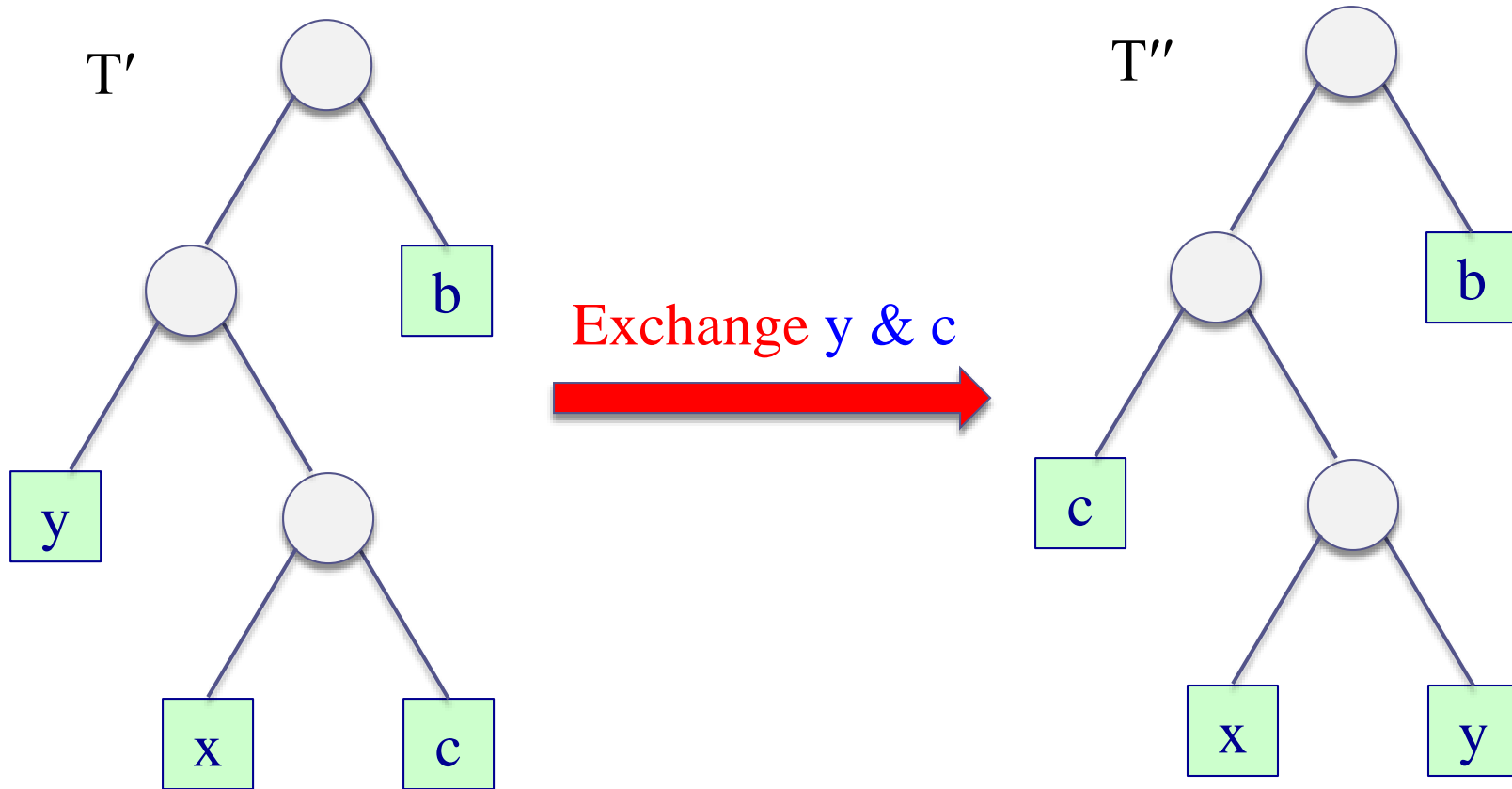
# Greedy Choice Property - Proof

$$B(T) - B(T') = (f[b] - f[x])(d_T(b) - d_T(x))$$

Since  $f[b] - f[x] \geq 0$  and  $d_T(b) \geq d_T(x)$   
therefore  $B(T') \leq B(T)$

In other words,  $T'$  is also optimal

# Greedy Choice Property - Proof



# Greedy Choice Property - Proof

- We can similarly show that

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

which implies  $B(T'') \leq B(T)$

- Since  $T$  is optimal  $\Rightarrow B(T'') = B(T) \Rightarrow T''$  is also optimal

- Note:  $T''$  contains our greedy choice:

Characters  $x$  &  $y$  appear as **sibling leaves of max-depth** in  $T''$

- Hence, the proof for the greedy choice property is complete

# Greedy-Choice Property of Determining an Optimal Code

---

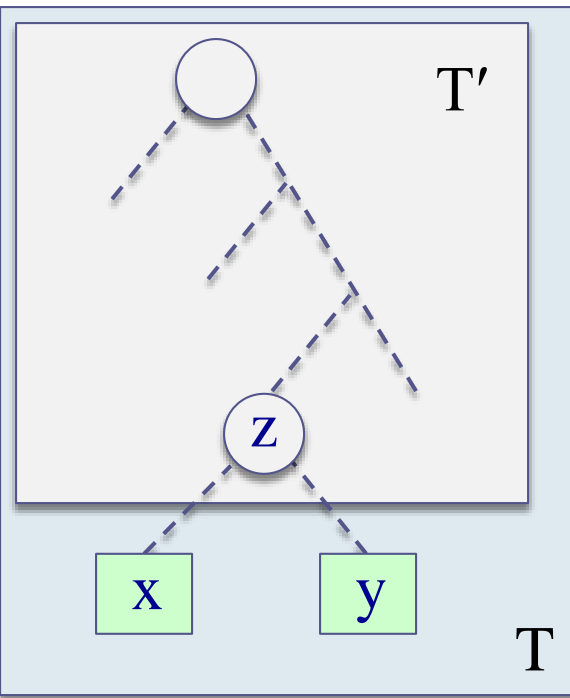
**Lemma 1** implies that

process of building an optimal tree by mergers can begin with the greedy choice of merging those two characters with the lowest frequency

We have already proved that  $B(T) = \sum_{i \in I_T} w(i)$ , that is, the total cost of the tree constructed is the **sum of the costs of its mergers (internal nodes) of all possible mergers**

At each step **Huffman chooses** the merger that incurs the **least cost**

# Optimal Substructure Property



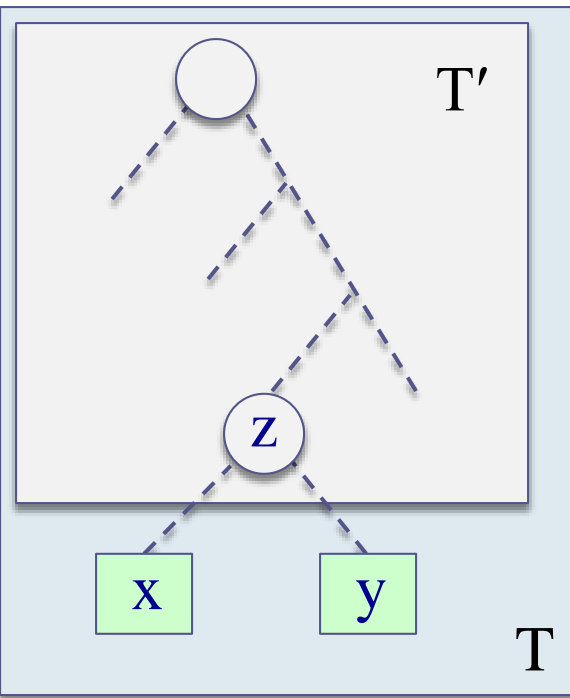
Consider an optimal solution  $T$  for alphabet  $C$ . Let  $x$  and  $y$  be any two sibling leaf nodes in  $T$ . Let  $z$  be the parent node of  $x$  and  $y$  in  $T$ .

Consider the subtree  $T'$  where  $T' = T - \{x, y\}$ . Here, consider  $z$  as a new character, where

$$f[z] = f[x] + f[y]$$

Optimal substructure property:  $T'$  must be optimal for the alphabet  $C'$ , where  $C' = C - \{x, y\} \cup \{z\}$

# Optimal Substructure Property - Proof



Reminder:

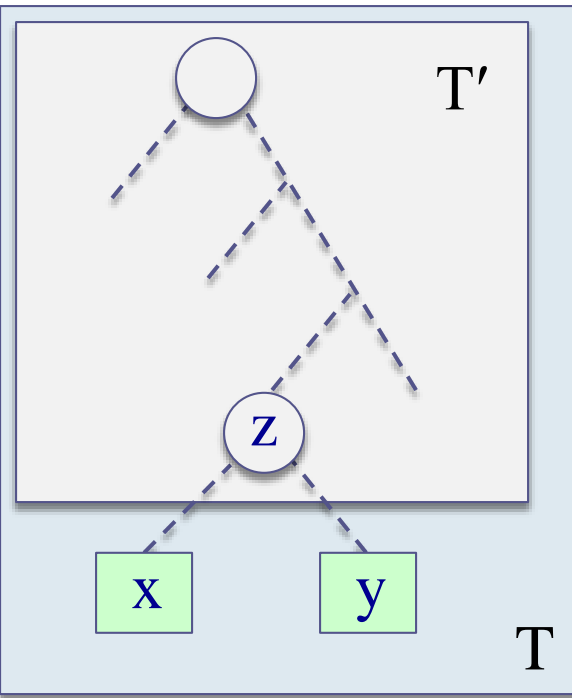
$$B(T) = \sum_{c \in C} f[c] d_T(c)$$

Try to express  $B(T)$  in terms of  $B(T')$ .

Note: All characters in  $C'$  have the same depth in  $T$  and  $T'$ .

$$B(T) = B(T') - \text{cost}(z) + \text{cost}(x) + \text{cost}(y)$$

# Optimal Substructure Property - Proof



Reminder:

$$B(T) = \sum_{c \in C} f[c] d_T(c)$$

$$\begin{aligned} B(T) &= B(T') - \text{cost}(z) + \text{cost}(x) + \text{cost}(y) \\ &= B(T') - f[z] \cdot d_T(z) + f[x] \cdot d_T(x) + f[y] \cdot d_T(y) \\ &= B(T') - f[z] \cdot d_T(z) + (f[x] + f[y]) (d_T[z] + 1) \\ &= B(T') - f[z] \cdot d_T(z) + f[z] (d_T[z] + 1) \\ &= B(T') + f[z] \end{aligned}$$

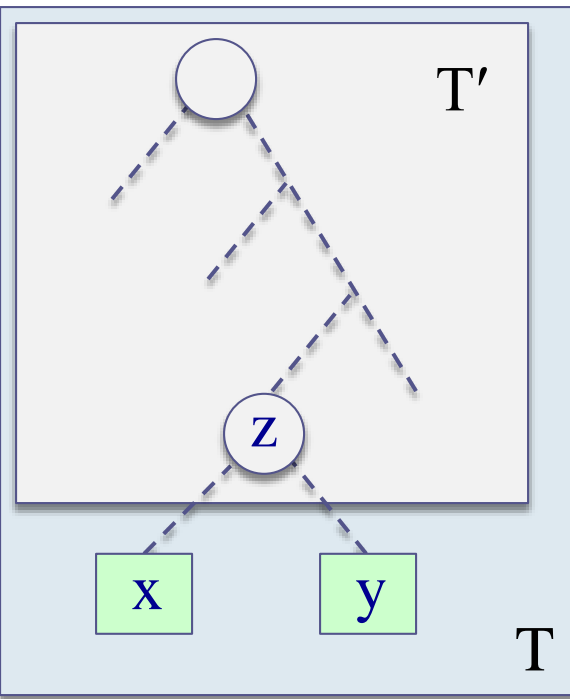
$$d_T(x) = d_T(z) + 1$$

$$d_T(y) = d_T(z) + 1$$

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



# Optimal Substructure Property - Proof



We want to prove that  $T'$  is optimal for

$$C' = C - \{x, y\} \cup \{z\}$$

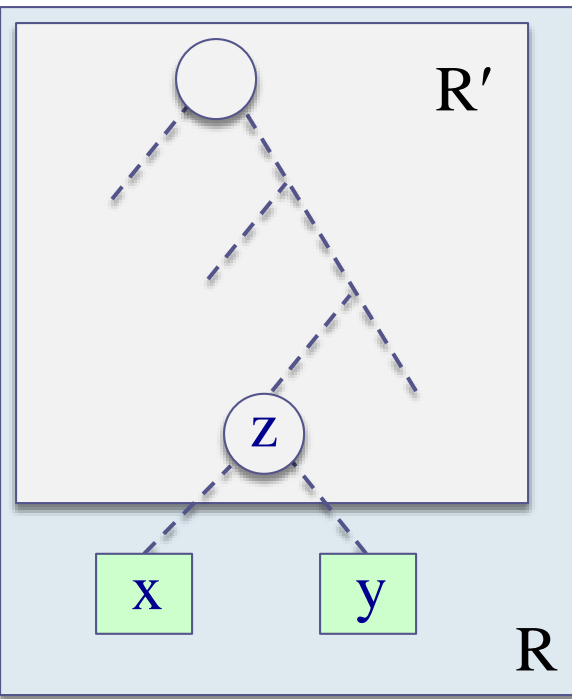
Assume by contradiction that there exists another solution for  $C'$  with smaller cost than  $T'$ . Call this solution  $R'$ :

$$B(R') < B(T')$$

Let us construct another prefix tree  $R$  by adding  $x$  &  $y$  as children of  $z$  in  $R'$

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

# Optimal Substructure Property - Proof



Let us construct another prefix tree  $R$  by adding  $x$  &  $y$  as children of  $z$  in  $R'$ .

We have:

$$B(R) = B(R') + f[x] + f[y]$$

In the beginning, we assumed that:

$$B(R') < B(T')$$

So, we have:

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

**Contradiction!  $\Rightarrow$  Proof complete**

# Greedy Algorithm for Huffman Coding - Summary

- For the greedy algorithm, we have proven that:
  - ▣ The greedy choice property holds.
  - ▣ The optimal substructure property holds.
  
- So, the greedy algorithm is optimal.