

# *Design and Analysis of Algorithms*



**CS3230**  
C23530

## Lecture 9 Greedy Algorithms

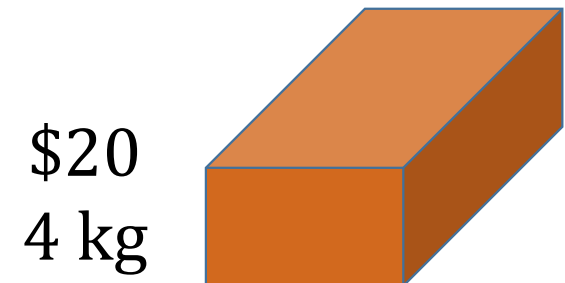
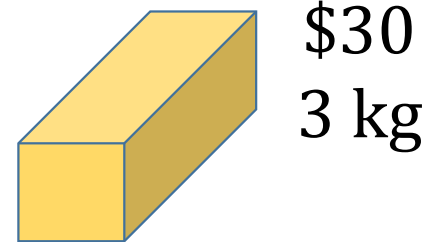
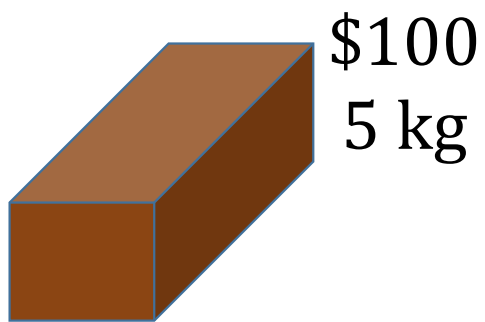
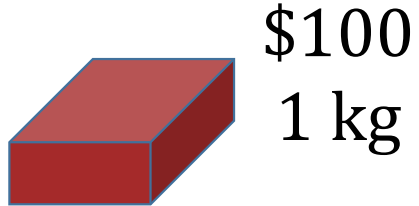
**Warut Suksompong**

# Today: Greedy Algorithms

A very general technique, like divide-and-conquer and dynamic programming

Technique is to recast the problem so that only one subproblem needs to be solved at each step. Beats divide-and-conquer and dynamic programming, **when it works.**

# Fractional Knapsack



# Fractional Knapsack

## Input:

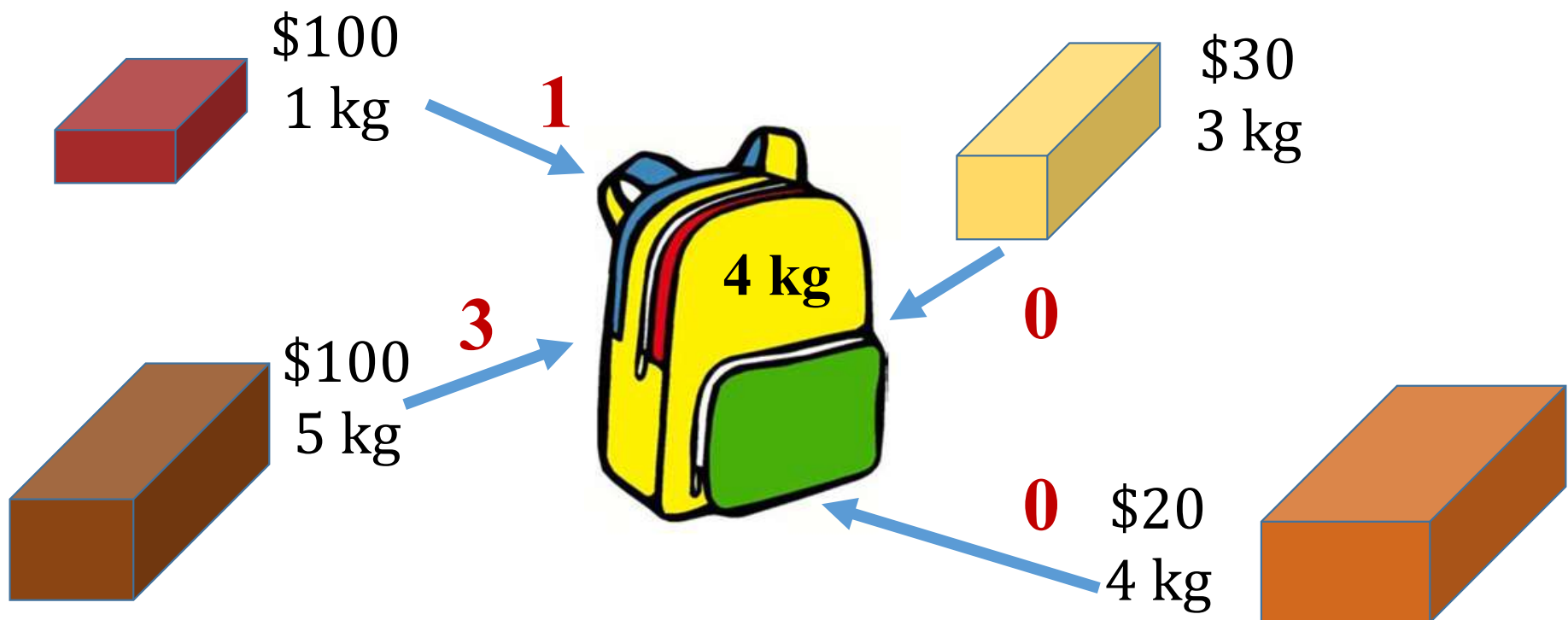
$(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n)$  and  $W$

## Output:

Weights  $x_1, \dots, x_n$  that maximize  $\sum_i v_i \cdot \frac{x_i}{w_i}$  subject to:

$$\sum_i x_i \leq W \text{ and } 0 \leq x_j \leq w_j \text{ for all } j \in \{1, 2, \dots, n\}.$$

# Fractional Knapsack



## Optimal Substructure

If we remove  $w$  kgs of one item  $j$  from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most  $W - w$  kgs that one can take from the  $n - 1$  original items and  $w_j - w$  kgs of item  $j$ .

# Optimal Substructure: Proof

**cut-and-paste** argument

- Let  $X$  be the value of the optimal knapsack. Suppose that the remaining load after removing  $w$  kgs of item  $j$  was not the optimal knapsack weighing at most  $W - w$  kgs that one can take from the  $n - 1$  original items and  $w_j - w$  kgs of item  $j$ .
- This means that there is a knapsack of value  $> X - v_j \cdot \frac{w}{w_j}$  with weight  $\leq W - w$  kgs among the  $n - 1$  other items and  $w_j - w$  kgs of item  $j$ .
- Combining with  $w$  kgs of item  $j$  gives knapsack of value  $> X$  and weight at most  $W$  for original input. **Contradiction!**

# Dynamic Programming?

In integral knapsack problem, we used the optimal substructure to formulate DP for deciding whether to add item  $j$ .

But in this case, we can do better....



# Make a guess!

Suppose we do not know anything about algorithm design and would like to solve the fractional knapsack problem in the real life. What strategy will you use?

1. Will first take the item with **maximum value**, then the item with **second maximum value** and **so on** until exceed the weight (the last chosen item could be fractional)
2. Will first take the item with **minimum weight**, then the item with **second minimum weight** and **so on** until exceed the weight (the last chosen item could be fractional)
3. Will first take the item with **maximum value-per-weight ratio**, then the item with **second maximum value-per-weight ratio** and **so on** until exceed the weight (the last chosen item could be fractional)

## Greedy-choice Property

Let  $j^*$  be an item with the **maximum value/kg**,  $v_j/w_j$ . Then, there exists an optimal knapsack containing  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

## Greedy-choice Property: Why?

**Claim:** Let  $j^*$  be an item with the maximum value/kg,  $v_j/w_j$ . Then, there exists an optimal knapsack containing  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

## Greedy-choice Property: Why?

**Claim:** Let  $j^*$  be an item with the maximum value/kg,  $v_j/w_j$ . Then, there exists an optimal knapsack containing  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

- Suppose an optimal knapsack contains  $x_1$  kgs of item 1,  $x_2$  kgs of item 2, ...,  $x_n$  kgs of item  $n$  such that:

$$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, W)$$

## Greedy-choice Property: Why?

**Claim:** Let  $j^*$  be an item with the maximum value/kg,  $v_j/w_j$ . Then, there exists an optimal knapsack containing  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

- Suppose an optimal knapsack contains  $x_1$  kgs of item 1,  $x_2$  kgs of item 2, ...,  $x_n$  kgs of item  $n$  such that:
$$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, W)$$
- Replace this weight by  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

# Greedy-choice Property: Why?

**Claim:** Let  $j^*$  be an item with the maximum value/kg,  $v_j/w_j$ . Then, there exists an optimal knapsack containing  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .

- Suppose an optimal knapsack contains  $x_1$  kgs of item 1,  $x_2$  kgs of item 2, ...,  $x_n$  kgs of item  $n$  such that:
$$x_1 + x_2 + \dots + x_n = \min(w_{j^*}, W)$$
- Replace this weight by  $\min(w_{j^*}, W)$  kgs of item  $j^*$ .
- Total weight does not change. Total value does not decrease because value/kg of  $j^*$  is maximum. So, knapsack stays optimal.

# Strategy for Greedy Algorithm

- Use greedy-choice property to put  $\min(w_{j^*}, W)$  kgs of item  $j^*$  in knapsack.
- If knapsack weighs  $W$  kgs, we are done.
- Otherwise, use optimal substructure to solve subproblem where all of item  $j^*$  is removed and knapsack weight limit is  $W - w_{j^*}$ .

# Iterative greedy algorithm

**ITER-FRAC-KNAPSACK**( $v, w, W$ ):

$valperkg \leftarrow [1, 2, \dots, n]$

Sort  $valperkg$  using comparison operator  $\preceq$  where  $i \preceq j$  if  $\frac{v[i]}{w[i]} \leq \frac{v[j]}{w[j]}$

**for**  $i = n$  to 1:

**if**  $W = 0$ : **break**

$j \leftarrow valperkg[i]$

$k \leftarrow \min(w[j], W)$

**print** “ $k$  kgs of item  $j$ ”

$W \leftarrow W - k$

**return**



# Iterative greedy algorithm

**ITER-FRAC-KNAPSACK**( $v, w, W$ ):

$valperkg \leftarrow [1, 2, \dots, n]$

Sort  $valperkg$  using comparison operator  $\preceq$  where  $i \preceq j$  if  $\frac{v[i]}{w[i]} \leq \frac{v[j]}{w[j]}$

**for**  $i = n$  to 1:

**if**  $W = 0$ : **break**

$j \leftarrow valperkg[i]$

$k \leftarrow \min(w[j], W)$

**print** “ $k$  kgs of item  $j$ ”

$W \leftarrow W - k$

**return**

Total time =  $O(n \log n)$

# Paradigm for greedy algorithms

1. Cast the problem where we have to **make a choice and are left with one subproblem** to solve.
2. Prove that there is always an **optimal solution to the original problem that makes the greedy choice**, so the greedy choice is safe.
3. Use **optimal substructure** to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem.

# The Simplest Greedy Algorithm

- **Problem:** Given  $n$  items with positive weights  $w_1, \dots, w_n$ , find a set  $S$  of  $k$  items that maximizes the total weight of items in  $S$ .

# The Simplest Greedy Algorithm

- **Problem:** Given  $n$  items with positive weights  $w_1, \dots, w_n$ , find a set  $S$  of  $k$  items that maximizes the total weight of items in  $S$ .
- **Optimal Substructure:**
  - Suppose  $S$  is any optimal solution, and  $S$  contains item  $i$ .
  - **Claim:**  $S - \{i\}$  is optimal for the sub-problem with  $i$ -th item removed and size limit  $k - 1$ .
  - **Cut & Paste Proof:** If  $T$  is optimal for the sub-problem and weighs more than  $S - \{i\}$ , then  $T \cup \{i\}$  would weigh more than  $S$ . Contradiction!

# The Simplest Greedy Algorithm

- **Problem:** Given  $n$  items with positive weights  $w_1, \dots, w_n$ , find a set  $S$  of  $k$  items that maximizes the total weight of items in  $S$ .
- **Greedy-choice Property:**
  - Suppose the  $i$ -th item has the maximum weight
  - **Claim:** There exists an optimal solution that contains item  $i$ .
  - **Proof:** Suppose there is an optimal solution  $S$  that does not contain  $i$ . Then, replace any item in  $S$  with  $i$ . The total weight does not decrease; so new set is also optimal and contains  $i$ .

# The Simplest Greedy Algorithm

- **Problem:** Given  $n$  items with positive weights  $w_1, \dots, w_n$ , find a set  $S$  of  $k$  items that maximizes the total weight of items in  $S$ .
- **Greedy Algorithm:**
  - Output the top  $k$  heaviest items.
  - **Correctness:** By greedy-choice property, can assume that heaviest item in  $S$ .  
By optimal substructure, can combine with solution to remaining subproblem.
  - Via sorting:  $O(n \log n)$ . Via QuickSelect:  $O(n)$ .

# Minimum spanning trees

**Input:** A connected, undirected graph  $G = (V, E)$  with weight function  $w : E \rightarrow \mathbb{R}^+$ .

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

# Minimum spanning trees

**Input:** A connected, undirected graph  $G = (V, E)$  with weight function  $w : E \rightarrow \mathbb{R}$ .

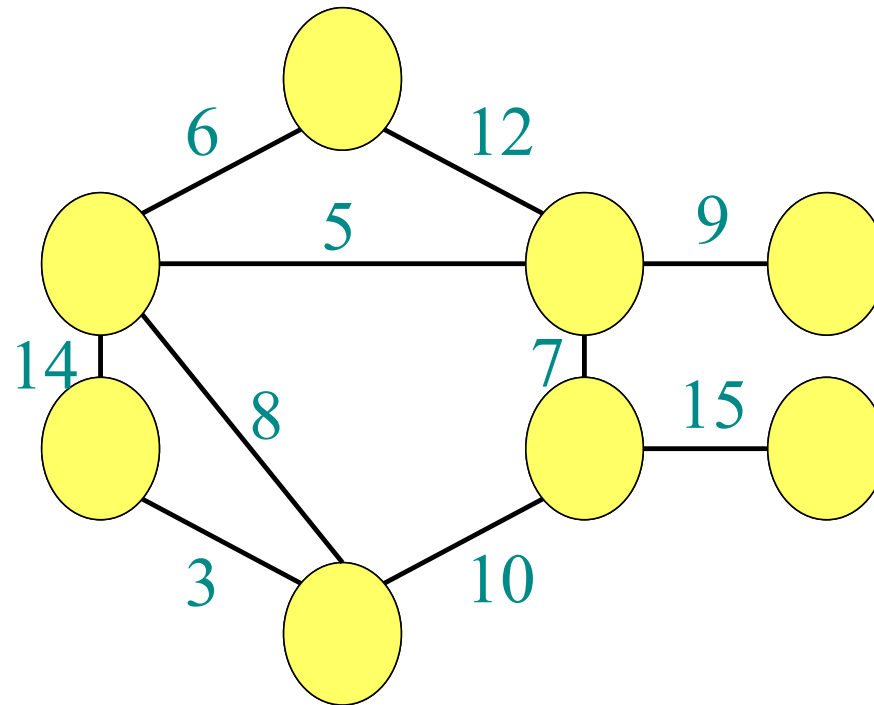
- For simplicity, assume that all edge weights are distinct.

**Output:** A *spanning tree*  $T$  — a tree that connects all vertices — of minimum weight:

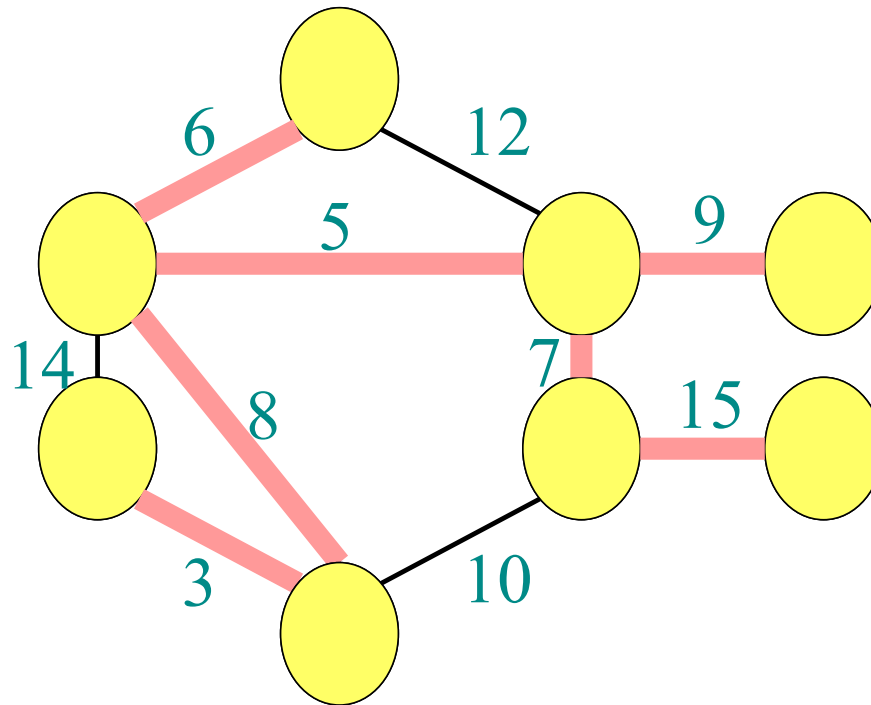
$$w(T) = \sum_{(u,v) \in T} w(u,v).$$



## Example of MST

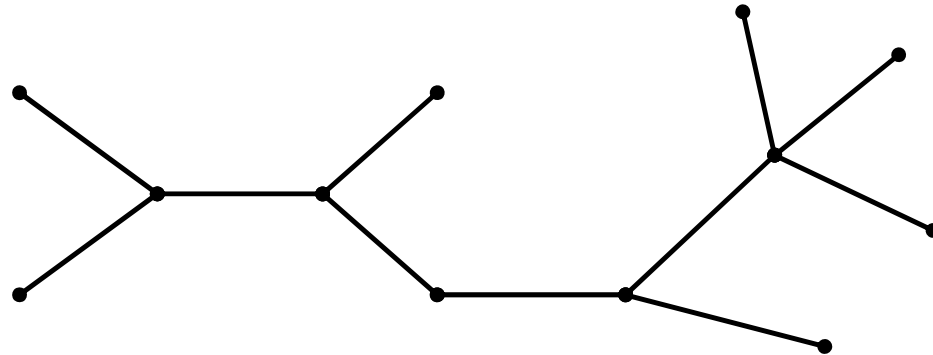


## Example of MST



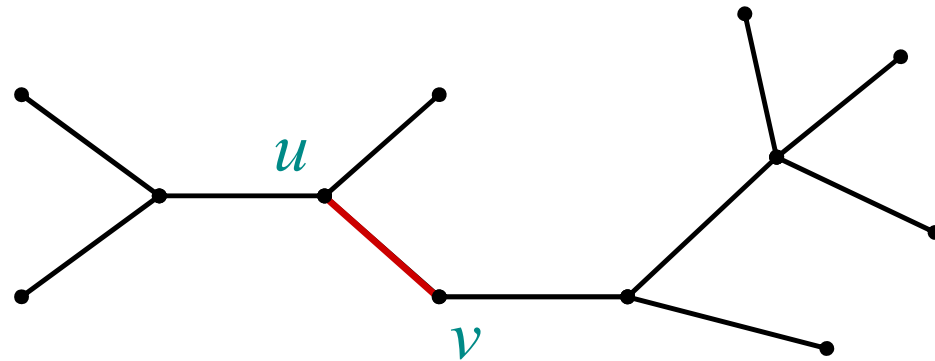
# Optimal substructure

MST  $T$ :  
(Other edges of  $G$   
are not shown.)



# Optimal substructure

MST  $T$ :  
(Other edges of  $G$   
are not shown.)

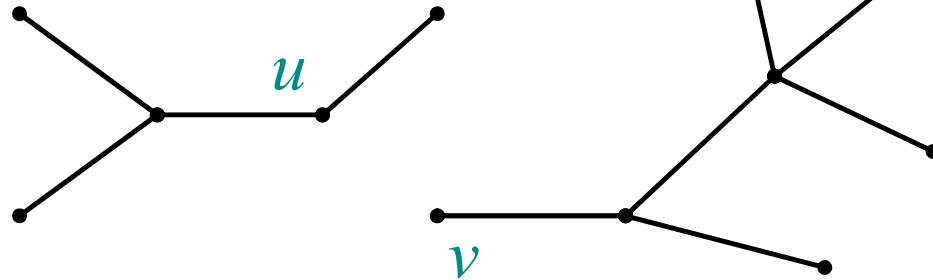


Remove any edge  $(u, v) \in T$ .

# Optimal substructure

MST  $T$ :

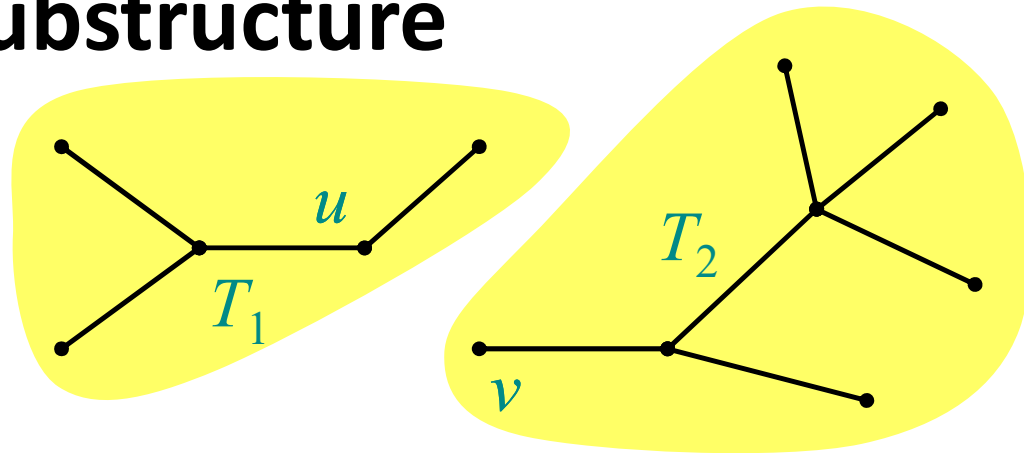
(Other edges of  $G$   
are not shown.)



Remove any edge  $(u, v) \in T$ .

# Optimal substructure

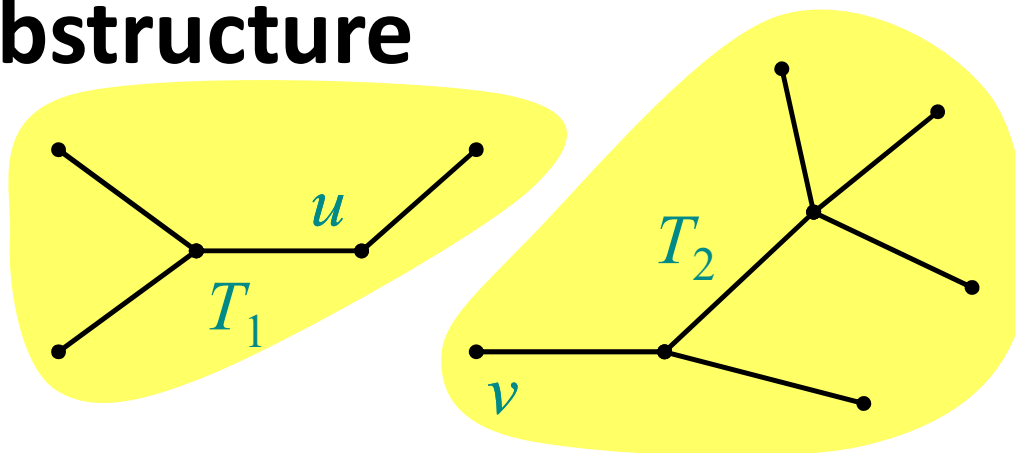
MST  $T$ :  
(Other edges of  $G$   
are not shown.)



Remove any edge  $(u, v) \in T$ . Then,  $T$  is partitioned into two subtrees  $T_1$  and  $T_2$ .

# Optimal substructure

MST  $T$ :  
(Other edges of  $G$   
are not shown.)



Remove any edge  $(u, v) \in T$ . Then,  $T$  is partitioned into two subtrees  $T_1$  and  $T_2$ .

**Theorem.** The subtree  $T_1$  is an MST of  $G_1 = (V_1, E_1)$ , the subgraph of  $G$  *induced* by the vertices of  $T_1$ :

$$V_1 = \text{vertices of } T_1,$$
$$E_1 = \{ (x, y) \in E : x, y \in V_1 \}.$$

Similarly for  $T_2$ .

# Proof of optimal substructure

*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If  $T_1'$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T_1' \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . 



# Proof of optimal substructure

*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If  $T_1'$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T_1' \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . □

Do we also have overlapping subproblems?

- Yes.

# Proof of optimal substructure

*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If  $T_1'$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T_1' \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . 

Do we also have overlapping subproblems?

- Yes.

Great, then dynamic programming may work!


- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.

# Hallmark for “greedy” algorithms

***Greedy-choice property***

*A locally optimal choice  
is globally optimal.*

# Hallmark for “greedy” algorithms



***Greedy-choice property***  
*A locally optimal choice  
is globally optimal.*

**Theorem.** Let  $T$  be the MST of  $G = (V, E)$ ,  
and let  $A \subseteq V$ . Suppose that  $(u, v) \in E$  is the  
least-weight edge connecting  $A$  to  $V - A$ .  
Then,  $(u, v) \in T$ .

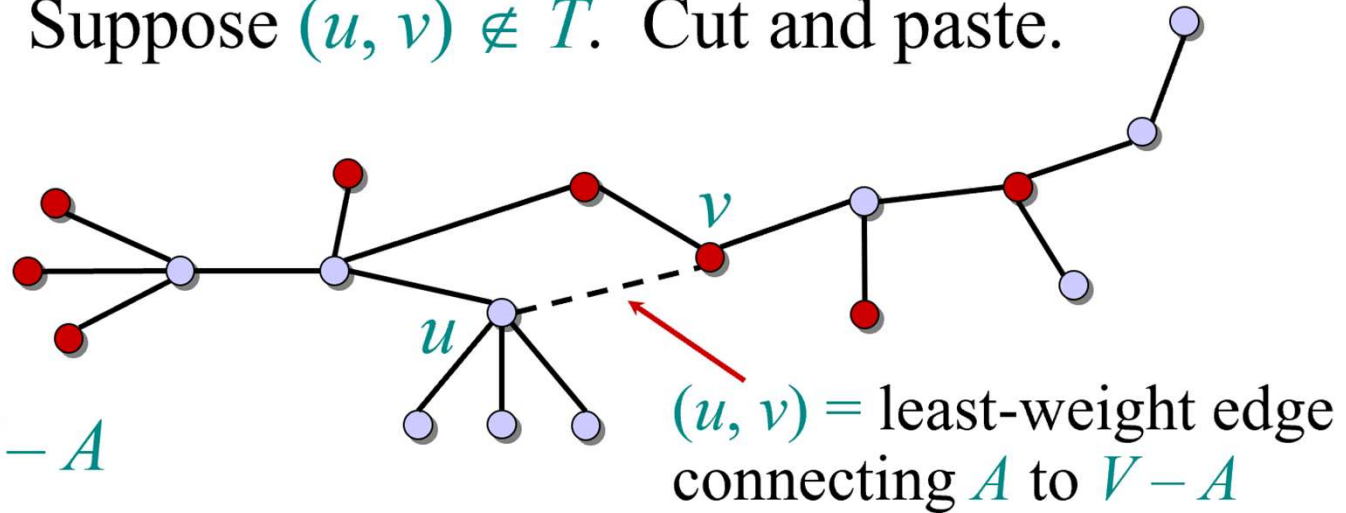
# Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

$T$ :

$\circ \in A$

$\bullet \in V - A$



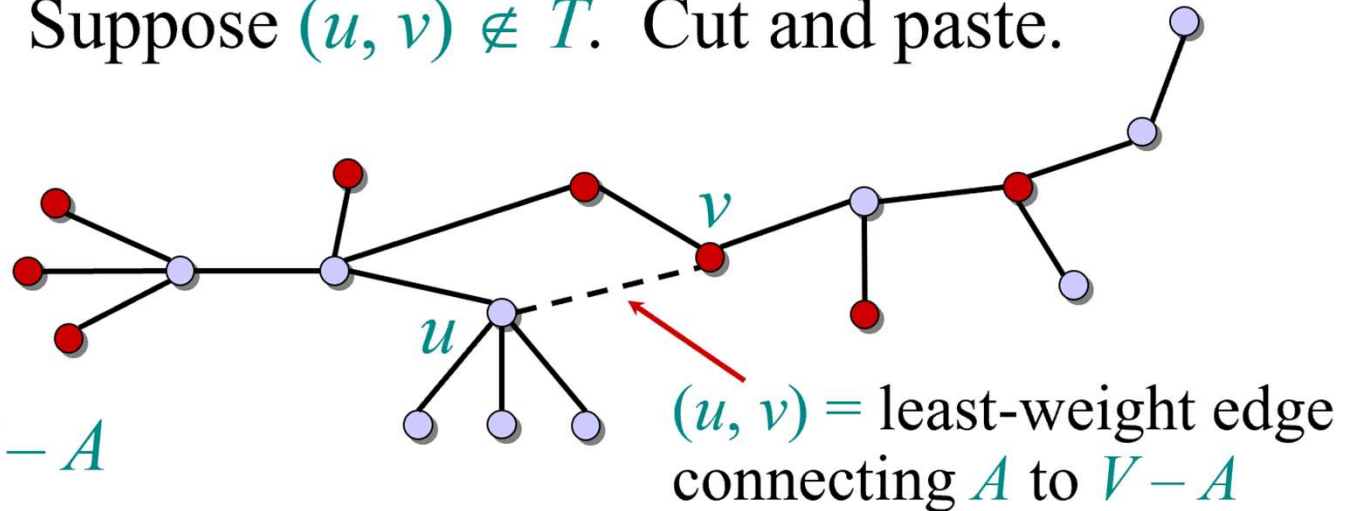
## Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

$T$ :

$\circ \in A$

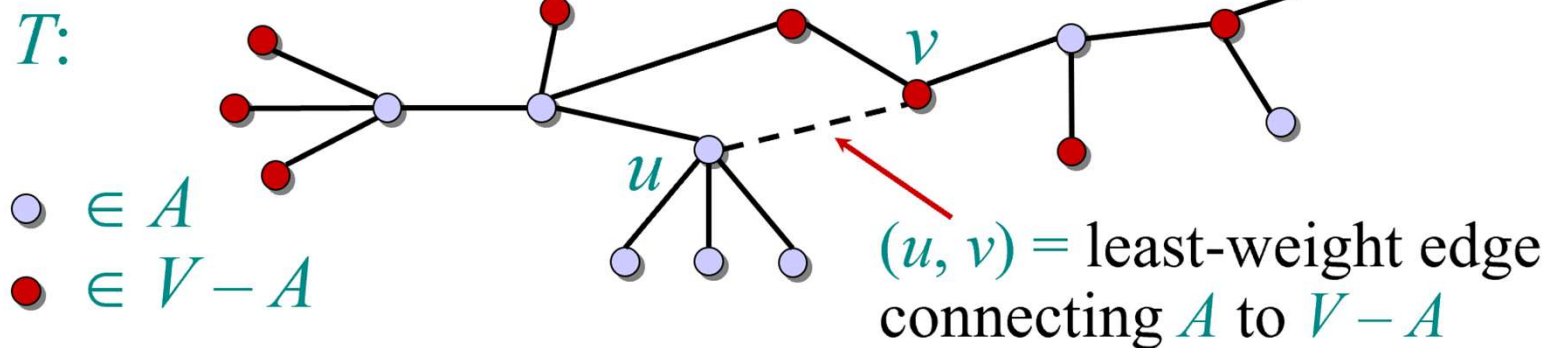
$\bullet \in V - A$



Consider the unique simple path from  $u$  to  $v$  in  $T$ .

## Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

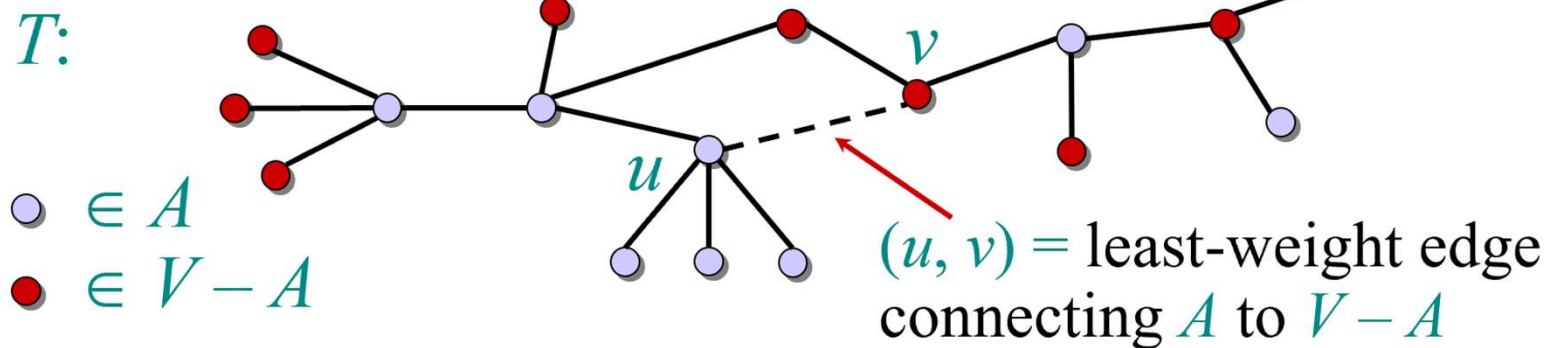


Consider the unique simple path from  $u$  to  $v$  in  $T$ .

Swap  $(u, v)$  with the first edge on this path that connects a vertex in  $A$  to a vertex in  $V - A$ .

## Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.



Consider the unique simple path from  $u$  to  $v$  in  $T$ .

Swap  $(u, v)$  with the first edge on this path that connects a vertex in  $A$  to a vertex in  $V - A$ .

A lighter-weight spanning tree than  $T$  results. □



# Prim's algorithm

**IDEA:** Build the tree one vertex at a time. At each step, add a least-weight edge from the tree to some vertex outside the tree.

$Q \leftarrow V$

$key[v] \leftarrow \infty$  for all  $v \in V$

$key[s] \leftarrow 0$  for some arbitrary  $s \in V$

**while**  $Q \neq \emptyset$

**do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$

**for each**  $v \in \text{Adj}[u]$

**do if**  $v \in Q$  and  $w(u, v) < key[v]$

**then**  $key[v] \leftarrow w(u, v)$       $\triangleright$  DECREASE-KEY

$\pi[v] \leftarrow u$

At the end,  $\{(v, \pi[v])\}$  forms the MST.

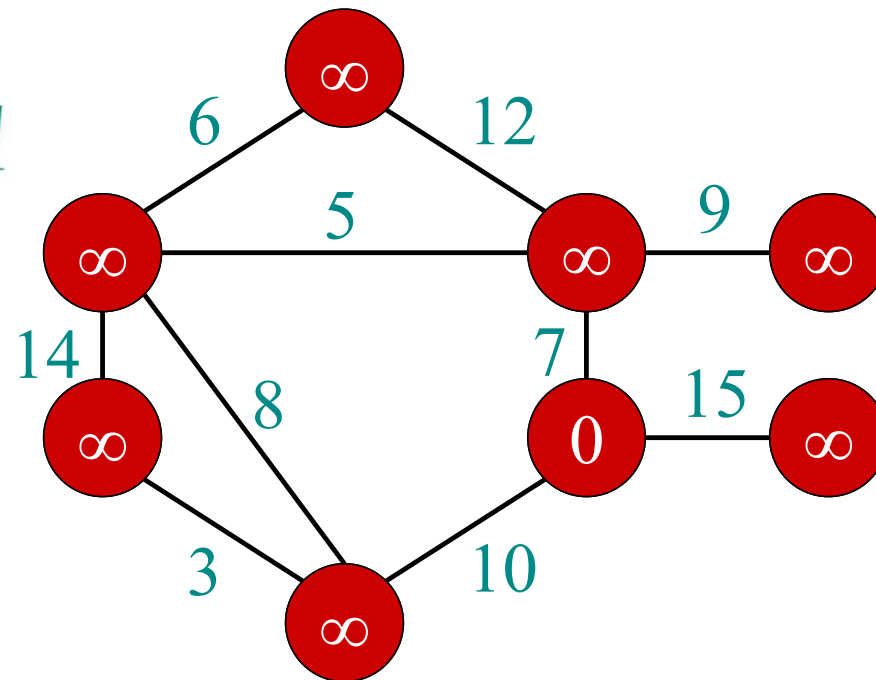
# Example of Prim's algorithm



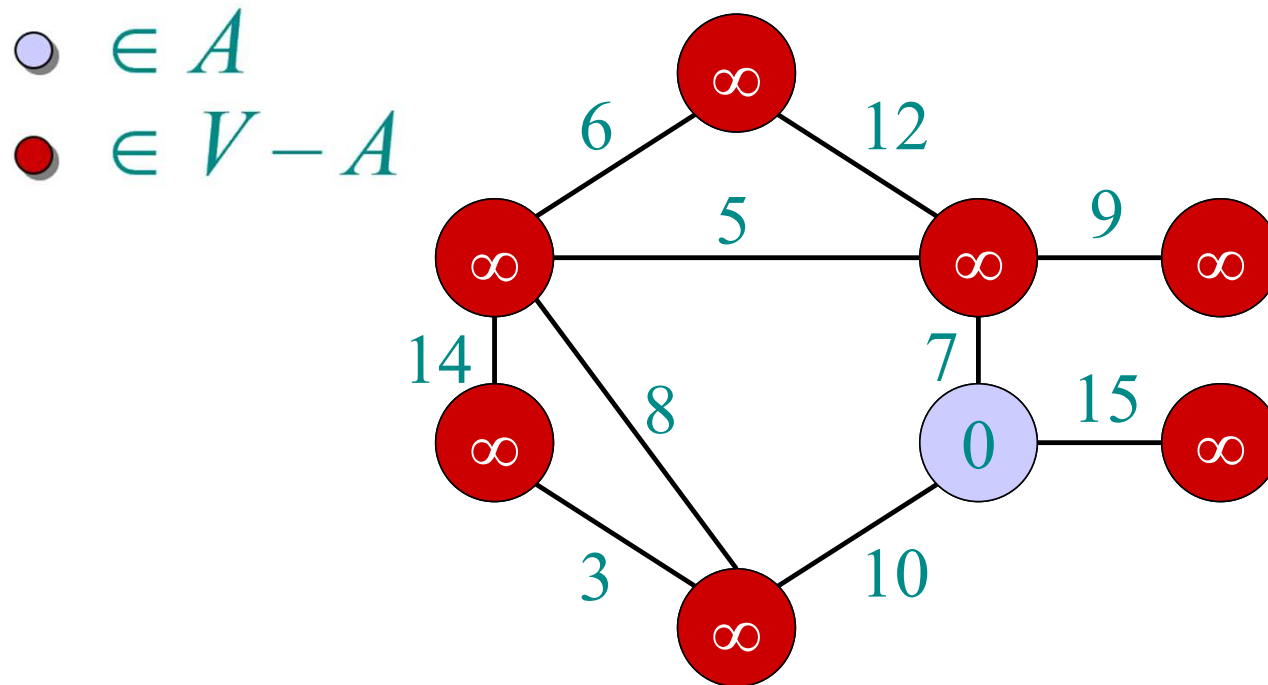
$\in A$



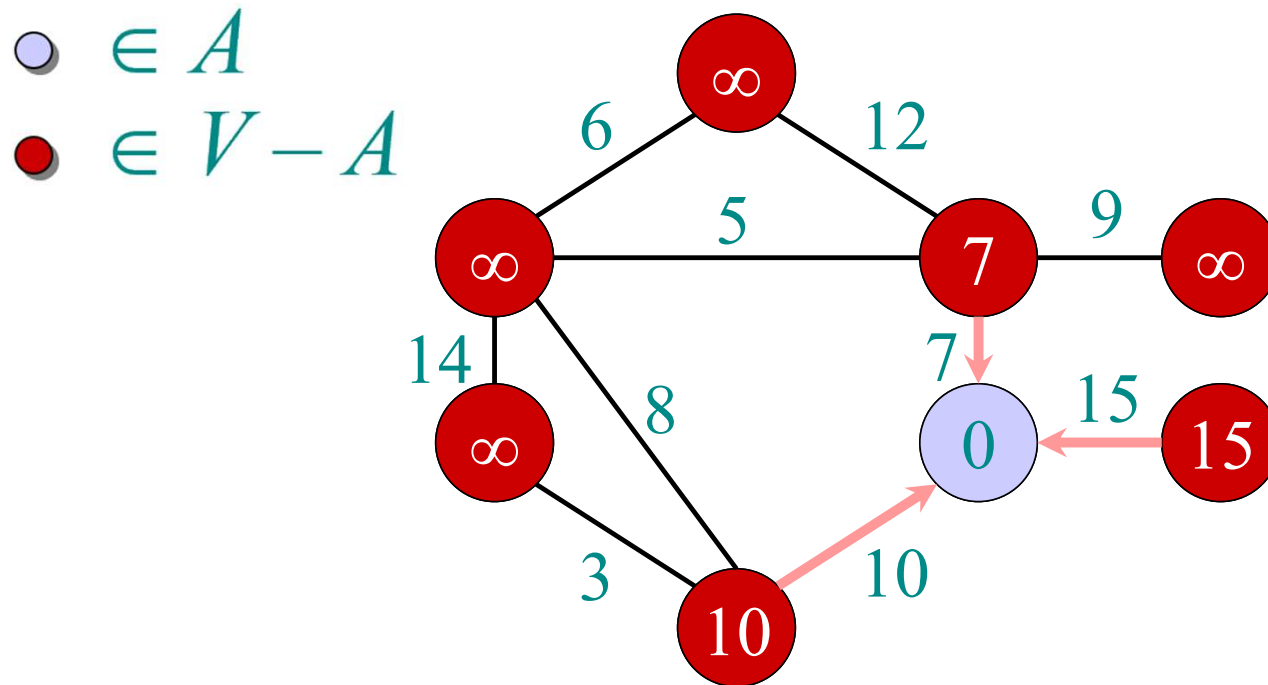
$\in V - A$



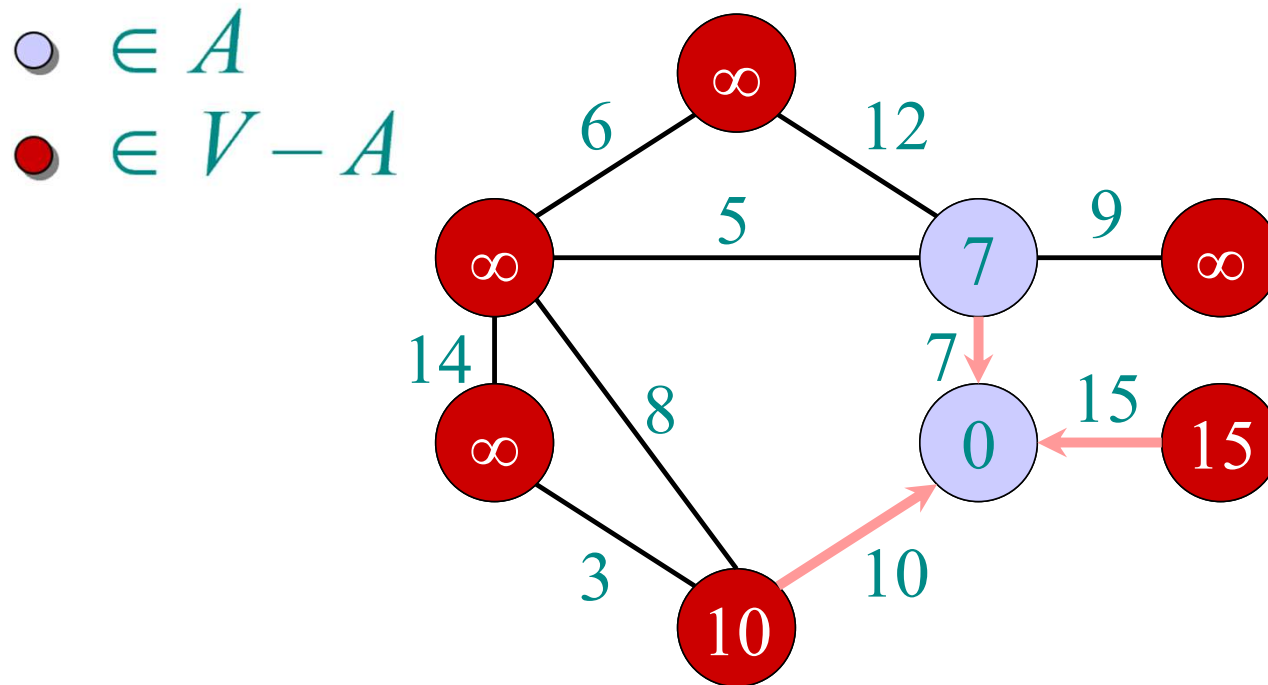
# Example of Prim's algorithm



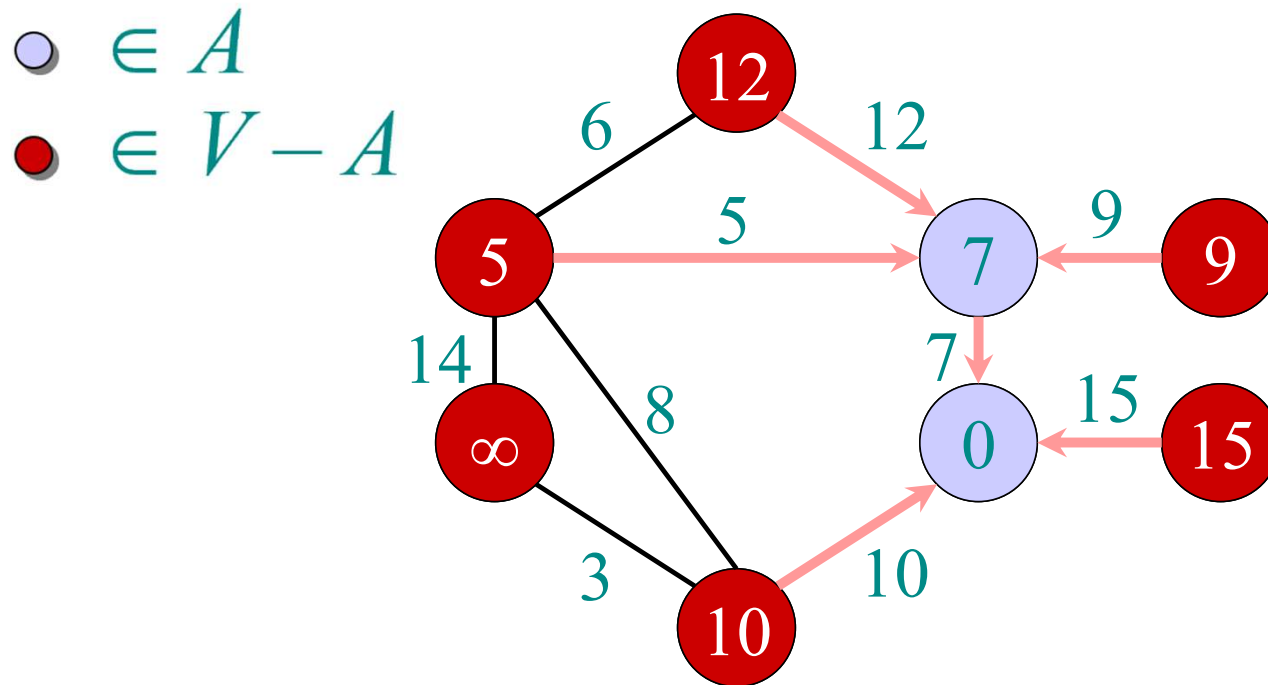
# Example of Prim's algorithm



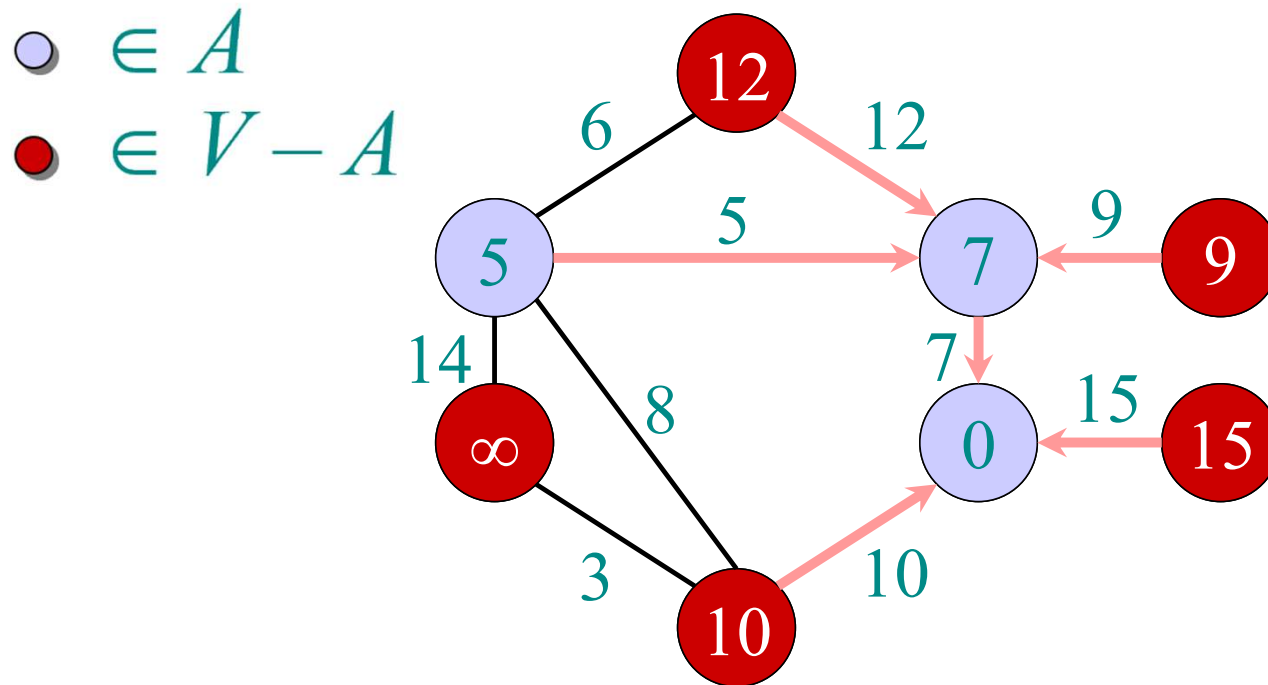
# Example of Prim's algorithm



# Example of Prim's algorithm

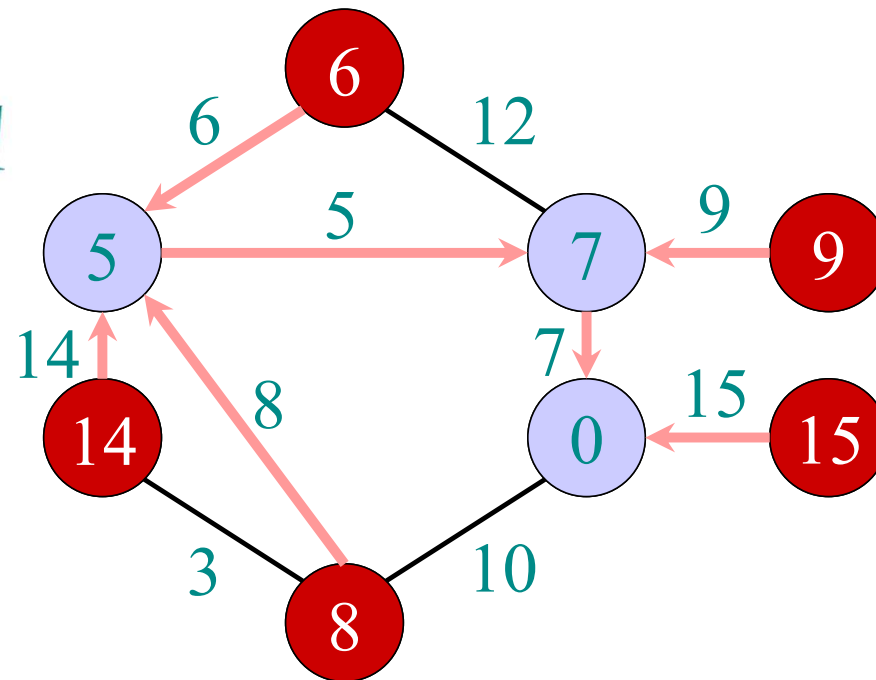


# Example of Prim's algorithm



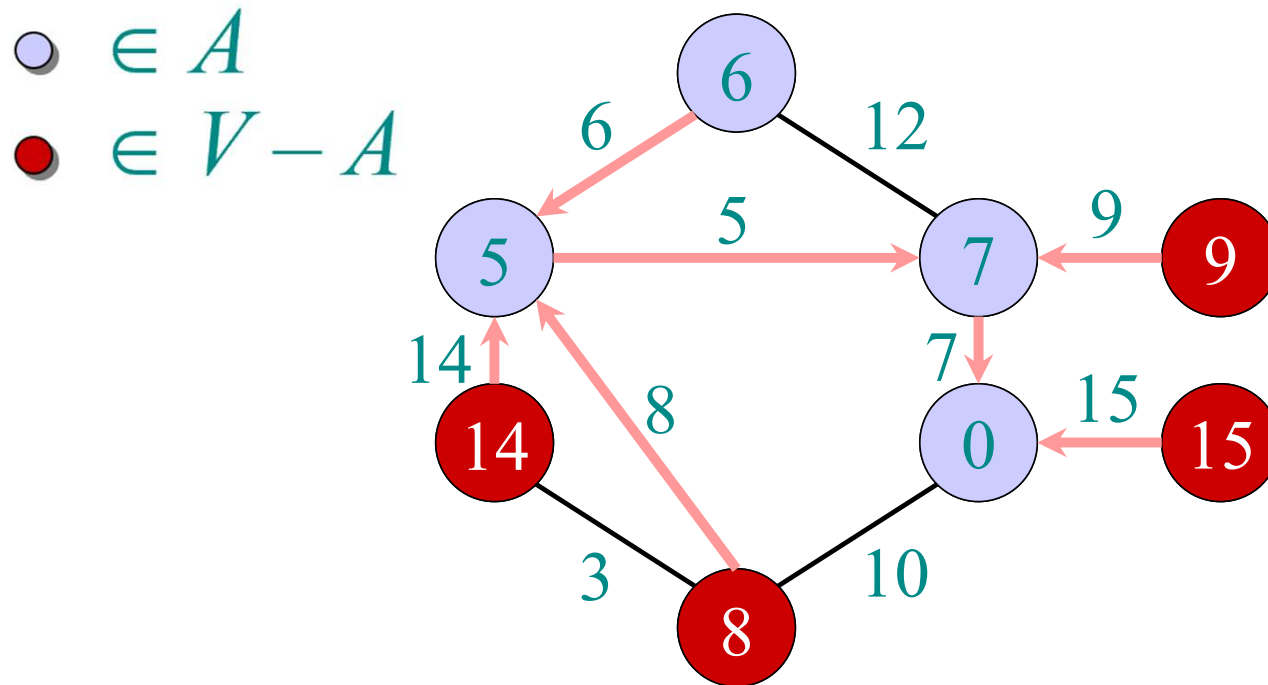
# Example of Prim's algorithm

●  $\in A$   
●  $\in V - A$



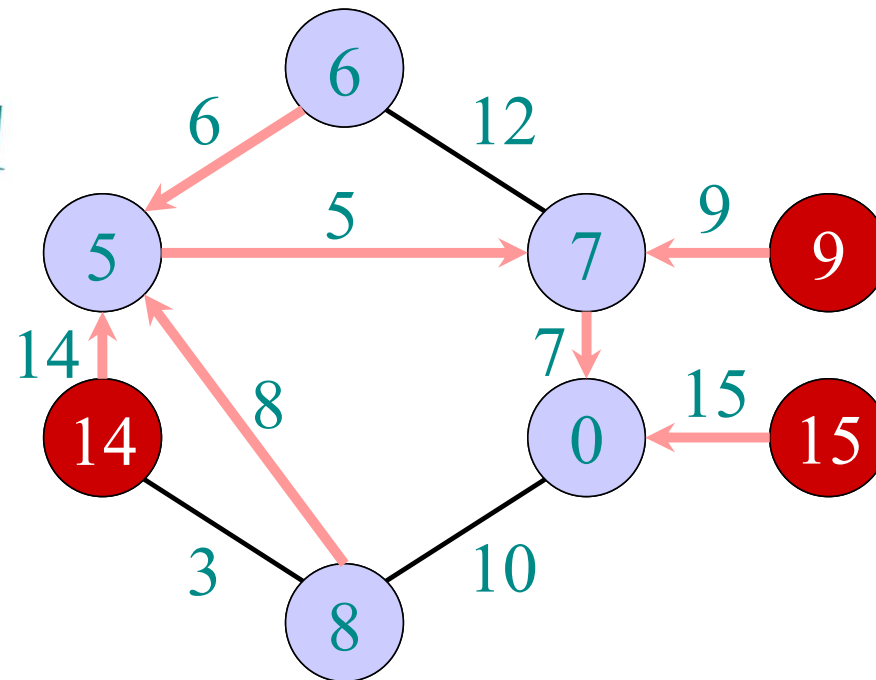


# Example of Prim's algorithm



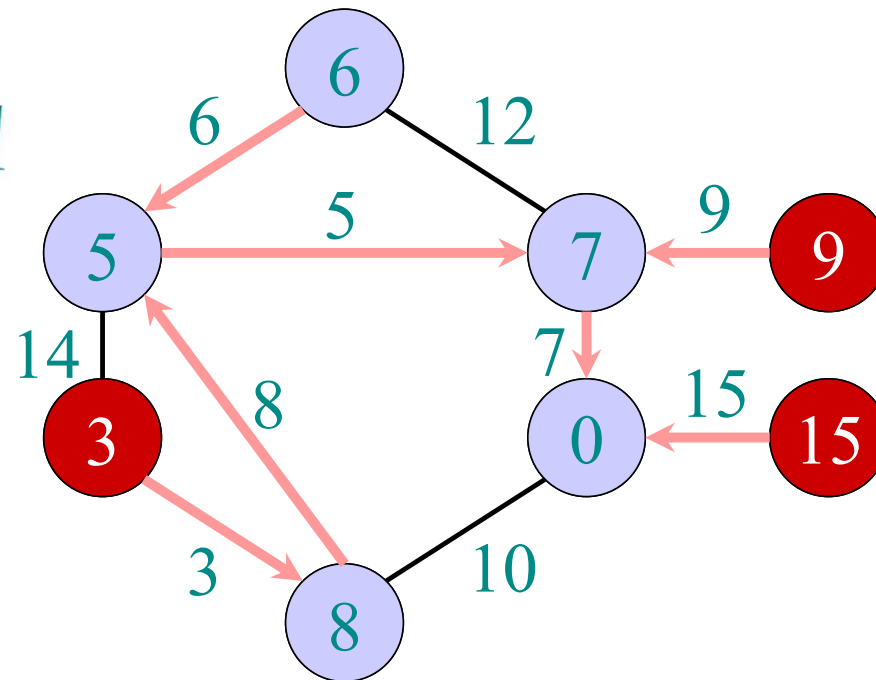
# Example of Prim's algorithm

●  $\in A$   
●  $\in V - A$

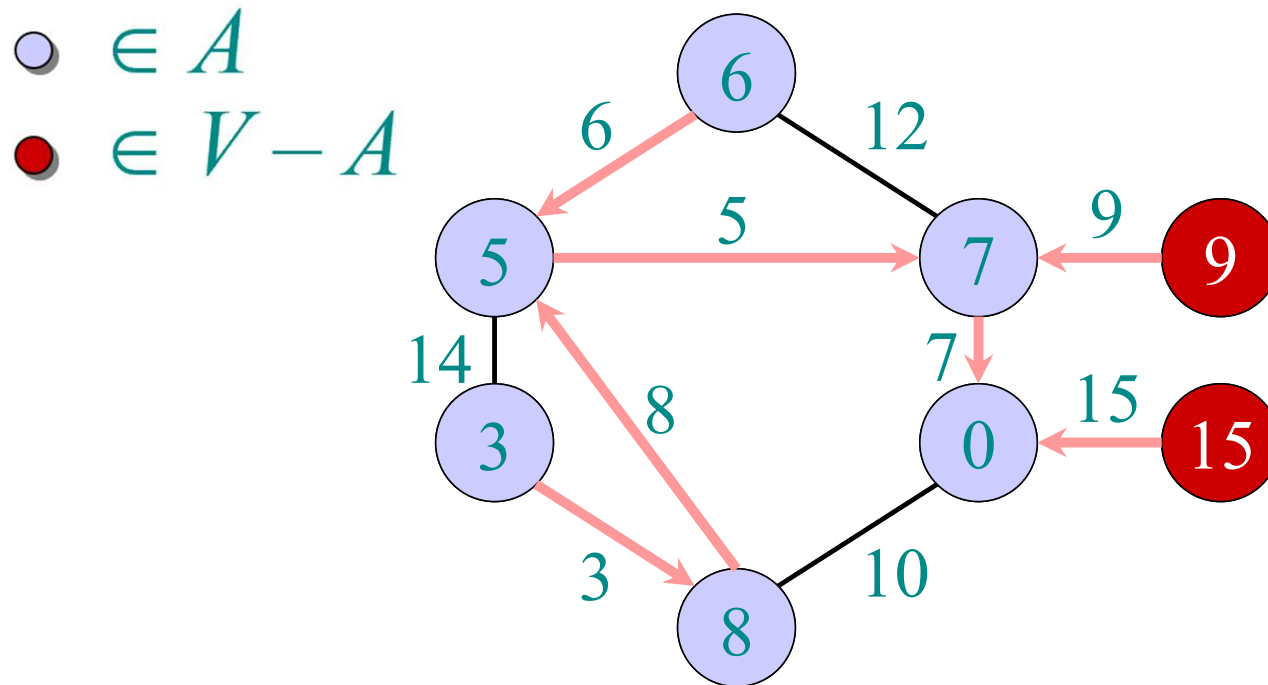


# Example of Prim's algorithm

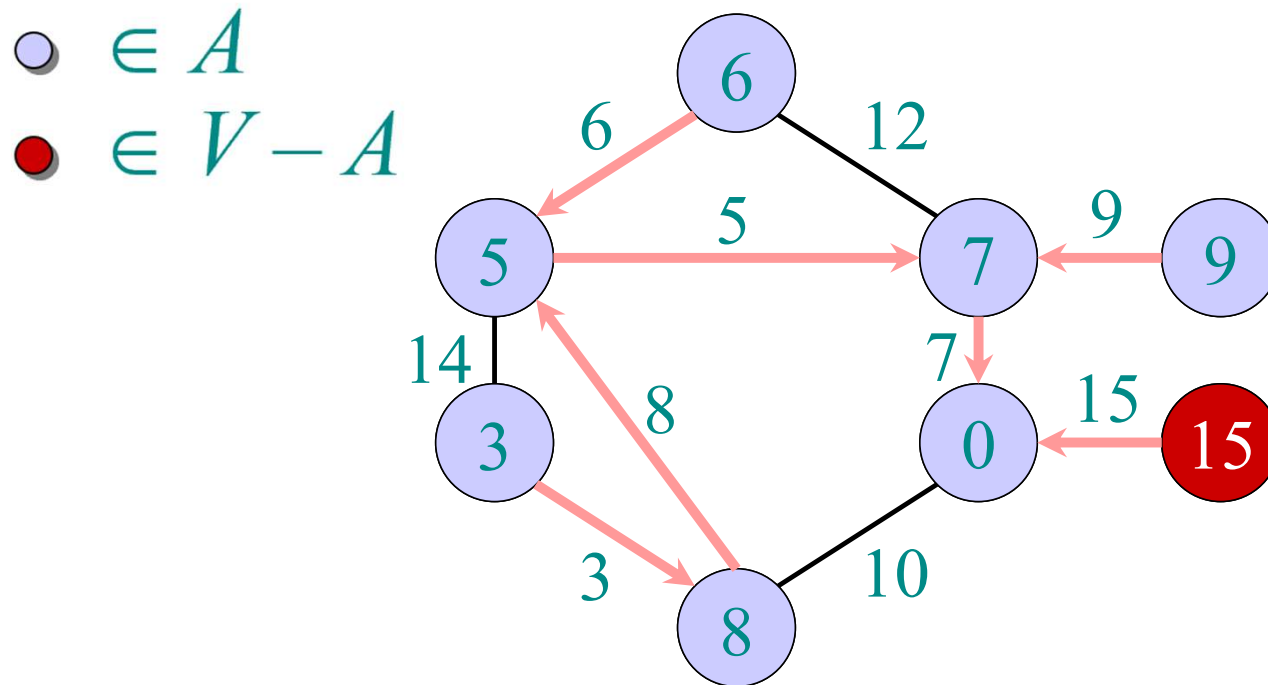
●  $\in A$   
●  $\in V - A$



# Example of Prim's algorithm

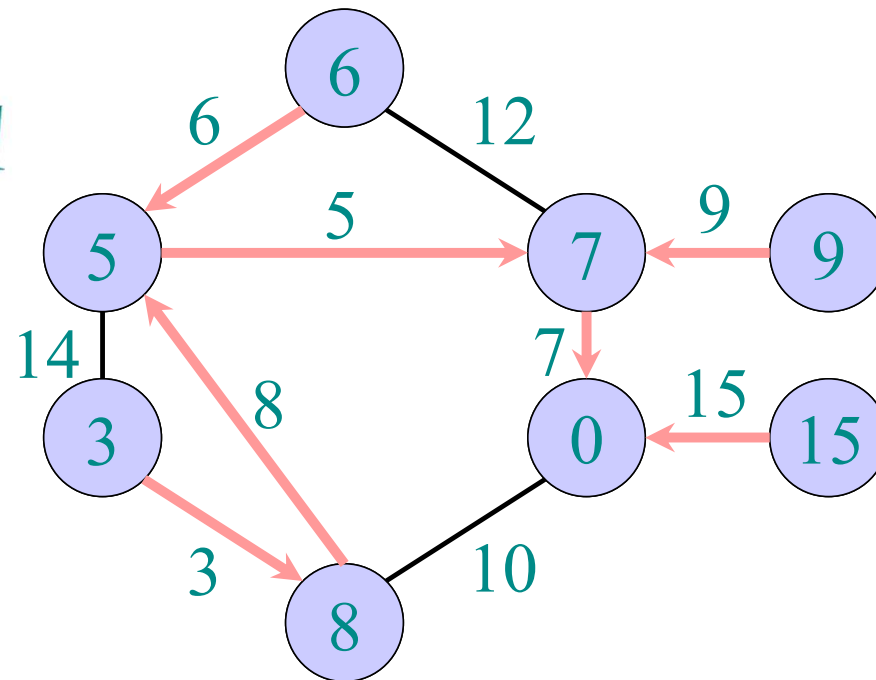


# Example of Prim's algorithm



# Example of Prim's algorithm

●  $\in A$   
●  $\in V - A$



# MST algorithms

Kruskal's algorithm (see CLRS Section 23.2):

- Also a greedy algorithm
- At each step, add a least-weight edge that does **not** cause a cycle to form

# Huffman Code

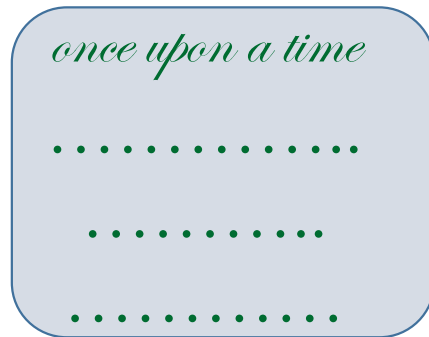
Applications in data compression, ...



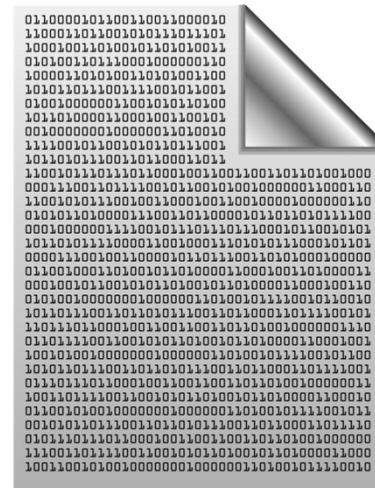
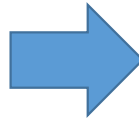
# Binary coding

Alphabet set  $A$  :  $\{a_1, a_2, \dots, a_n\}$

**A text file:** a sequence of alphabets



**A text file F**



**Binary coding of F**

**Question:** How many bits needed to encode a text file with  $m$  characters?

**Answer:**  $m \cdot \lceil \log_2 n \rceil$  bits.

# Fixed length coding

Alphabet set  $A : \{a_1, a_2, \dots, a_n\}$

**Question:** What is a binary coding of  $A$ ?

**Answer:**  $\gamma: A \rightarrow \text{binary strings}$

**Question:** What is a **fixed length** coding of  $A$ ?

**Answer:** each alphabet  $\leftarrow$  a unique binary string of length  $\lceil \log_2 n \rceil$ .

**Question:** How to decode a **fixed length binary** coding?

**Answer:** Easy 😊

0100|1010|0000|1011|...

# Fixed length coding

Alphabet set  $A$  :  $\{a_1, a_2, \dots, a_n\}$

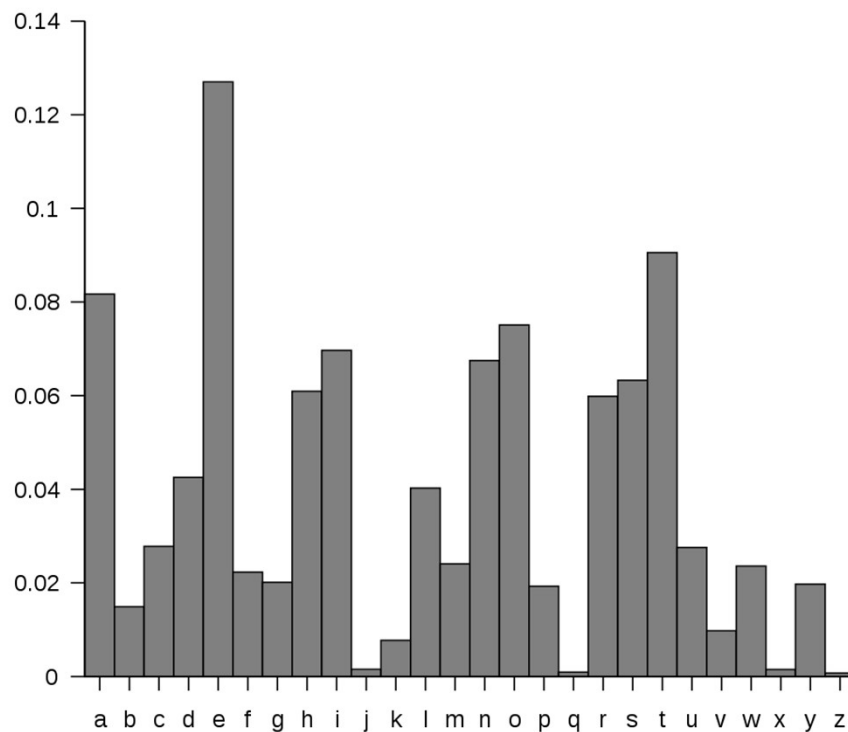
**Question:** Can we use fewer bits to store alphabet set  $A$ ?

**Answer:** No.

**Question:** Can we use fewer bits to store a file?

**Answer:** Yes

# Huge variation in the frequency of alphabets in a text



ENGLISH LETTER FREQUENCIES			
Per One-Thousand Letters			
Sorted By Letter		Sorted By Frequency	
A	73	E	130
B	9	T	93
C	30	N	78
D	44	R	77
E	130	I	74
F	28	O	74
G	16	A	73
H	35	S	63
I	74	D	44
J	2	H	35
K	3	L	35
L	35	C	30
M	25	F	28
N	78	P	27
O	74	U	27
P	27	M	25
Q	3	Y	19
R	77	G	16
S	63	W	16
T	93	V	13
U	27	B	9
V	13	X	5
W	16	K	3
X	5	Q	3
Y	19	J	2
Z	1	Z	1

[http://en.wikipedia.org/wiki/Letter\\_frequency](http://en.wikipedia.org/wiki/Letter_frequency)

**Huge variation** in the **frequency** of alphabets in a text

**Question:** How to exploit variation in the frequencies of alphabets?

**Answer:**

**More frequent** alphabets ← coding with **shorter bit string**

**Less frequent** alphabets ← coding with **longer bit string**

# Variable length encoding

Average **bit length** per symbol using  $\gamma$ :

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$

$$= 0.45 \times 1 + 0.18 \times 2 + (0.15 + 0.12 + 0.10) \times 3$$

$$= 1.92$$

Alphabets	Frequency $f$	Encoding $\gamma$
<i>a</i>	0.45	0
<i>b</i>	0.18	10
<i>c</i>	0.15	110
<i>d</i>	0.12	101
<i>e</i>	0.10	111

There is a serious problem with the  $\gamma$  encoding. Can you see?

# Variable length encoding

Can you  
fix it?

Symbols	Frequency $f$	Encoding $\gamma$
$a$	0.45	0
$b$	0.18	10
$c$	0.15	110
$d$	0.12	101
$e$	0.10	111

Average bit length per symbol using  $\gamma$ :

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$

$$= 0.45 \times 1 + 0.18 \times 2 + (0.15 + 0.12 + 0.10) \times 3$$

$$= 1.92$$

**Question:** How will you decode 01010111?

**Answer:** *abbe* or *adae* ☹

**Question:** What is the source of this ambiguity?

**Answer:**  $\gamma(b)$  is a prefix of  $\gamma(d)$ .

# Variable length Coding

Alphabets	Frequency $f$	Encoding $\gamma$
$a$	0.45	0
$b$	0.18	100
$c$	0.15	110
$d$	0.12	101
$e$	0.10	111

Average bit length per symbol using  $\gamma$ :

$$\begin{aligned} \text{ABL}(\gamma) &= \sum_{x \in A} f(x) \cdot |\gamma(x)| \\ &= 0.45 \times 1 + 0.18 \times 3 + (0.15 + 0.12 + 0.10) \times 3 \\ &= 2.1 \end{aligned}$$



# Prefix Coding

## Definition:

A coding  $\gamma(A)$  is called **prefix coding** if there do not exist  $x, y \in A$  such that

$$\gamma(x) \text{ is prefix of } \gamma(y)$$

**Algorithmic Problem:** Given a set  $A$  of  $n$  alphabets and their frequencies, compute coding  $\gamma$  such that

- $\gamma$  is prefix coding
- $ABL(\gamma)$  is **minimum**.

# The challenge of the problem

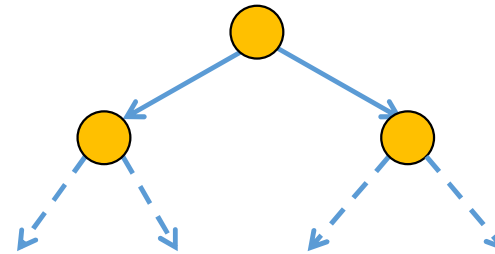
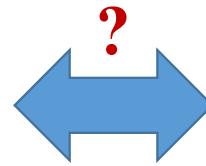


Among all possible binary coding of **A**, how to find the **optimal prefix coding**?

# The novel idea of Huffman

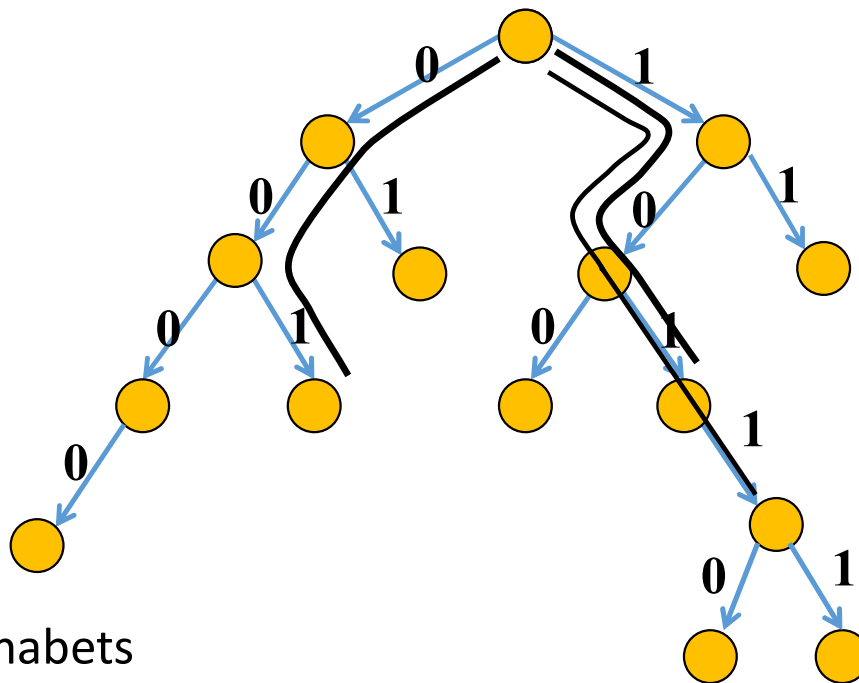


Binary coding



Binary tree

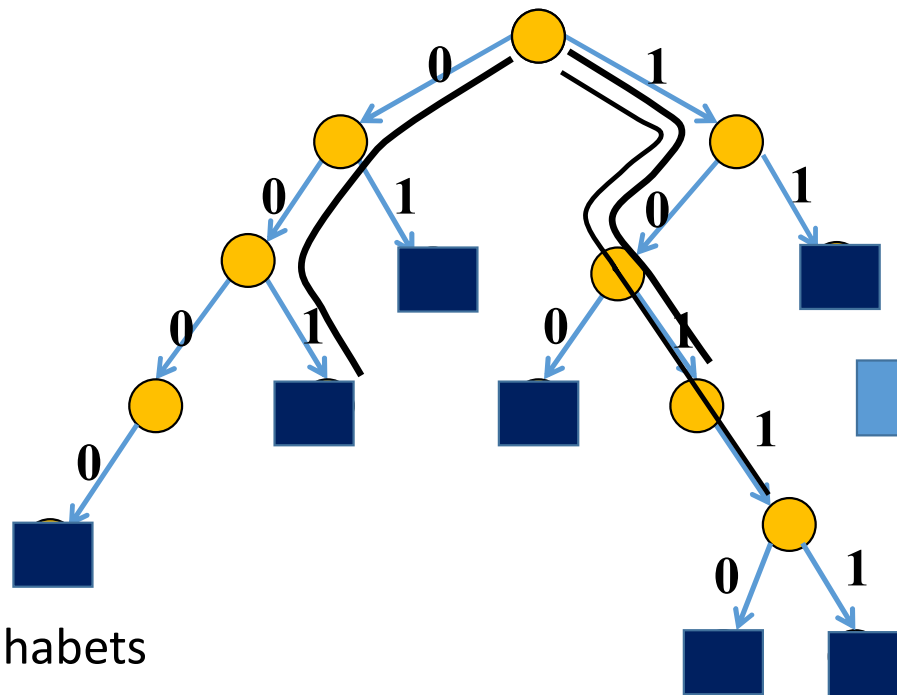
# A **labeled** binary tree



**Leaf** nodes → alphabets

Code of an alphabet = **Label of path from root**

# A **labeled** binary tree



**Leaf** nodes → alphabets

Code of an alphabet = **Label of path from root**

01,  
001,  
0000,  
11,  
100,  
10110,  
10111

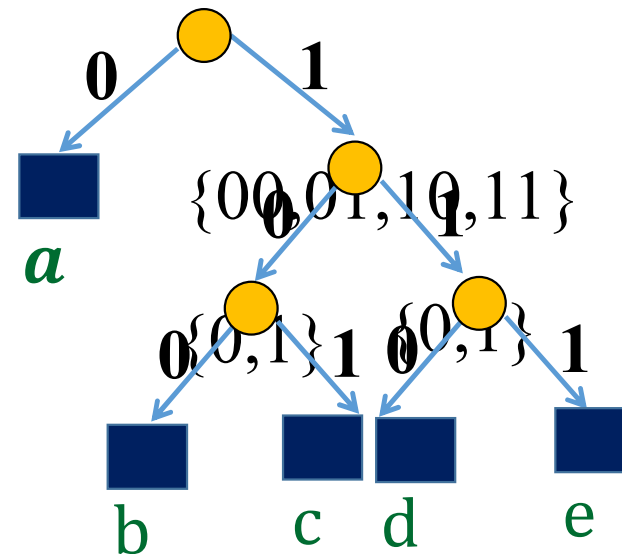
# Variable length Coding

Alphabets	Frequency $f$	Encoding $\gamma$
$a$	0.45	0
$b$	0.18	100
$c$	0.15	110
$d$	0.12	101
$e$	0.10	111

## Question:

How to build the labeled tree for a prefix code ?

$\{0, 100, 101, 110, 111\}$



# Prefix code and Labelled Binary tree

## Theorem:

For each prefix code of a set  $A$  of  $n$  alphabets, there exists a binary tree  $T$  on  $n$  leaves such that

- There is a bijjective mapping between the **alphabets** and the **leaves**.
- The label of a path from root to a leaf node corresponds to the **prefix code** of the corresponding alphabet.

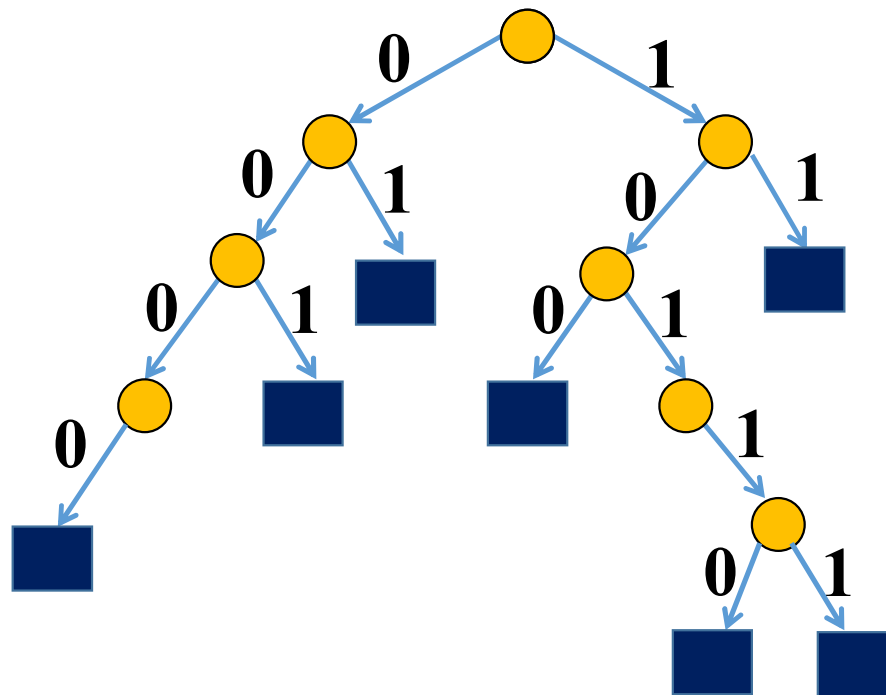
**Question:** Can you express **Average bit length** of  $\gamma$  in terms of its binary tree  $T$  ?

$$\begin{aligned} \text{ABL}(\gamma) &= \sum_{x \in A} f(x) \cdot |\gamma(x)| \\ &= \sum_{x \in A} f(x) \cdot |\text{depth}_T(x)| \end{aligned}$$

Finding the labeled binary tree for an optimal  
prefix codes



Is the following prefix coding **optimal** ?



NO

# Observations on the binary tree of an optimal prefix code

## Lemma:

The binary tree corresponding to optimal prefix coding must be a **full binary tree**:

Every internal node has degree exactly 2.

**Question:** What next?

We need to see the influence of frequencies on an optimal binary tree.

Let  $a_1, a_2, \dots, a_n$  be the alphabets of  $A$  in non-decreasing order of their frequencies.

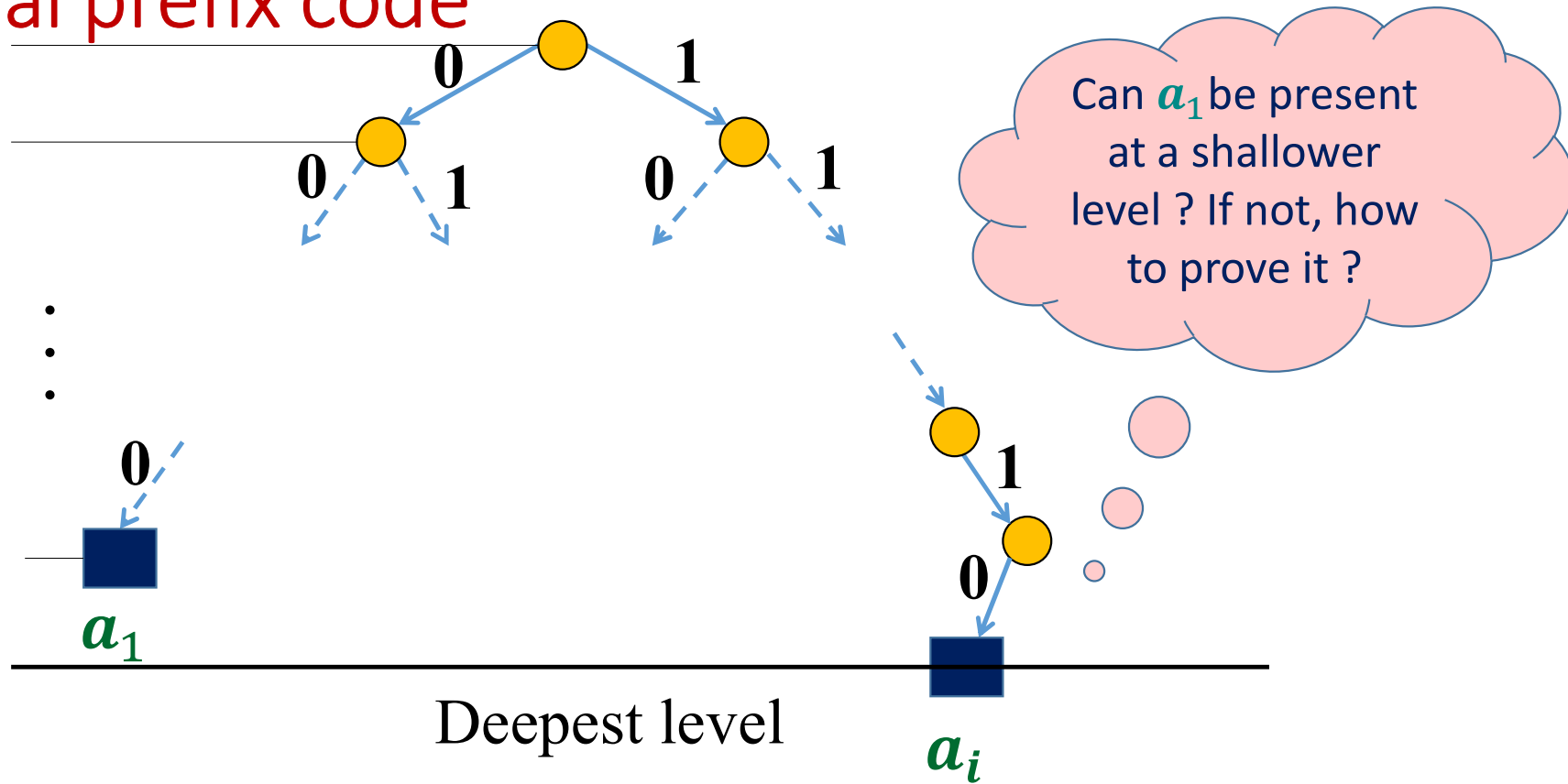
# Observations on the binary tree of an optimal prefix code

Intuitively, **more frequent** alphabets should be **closer to the root** and **less frequent** alphabets should be **farther from the root**.

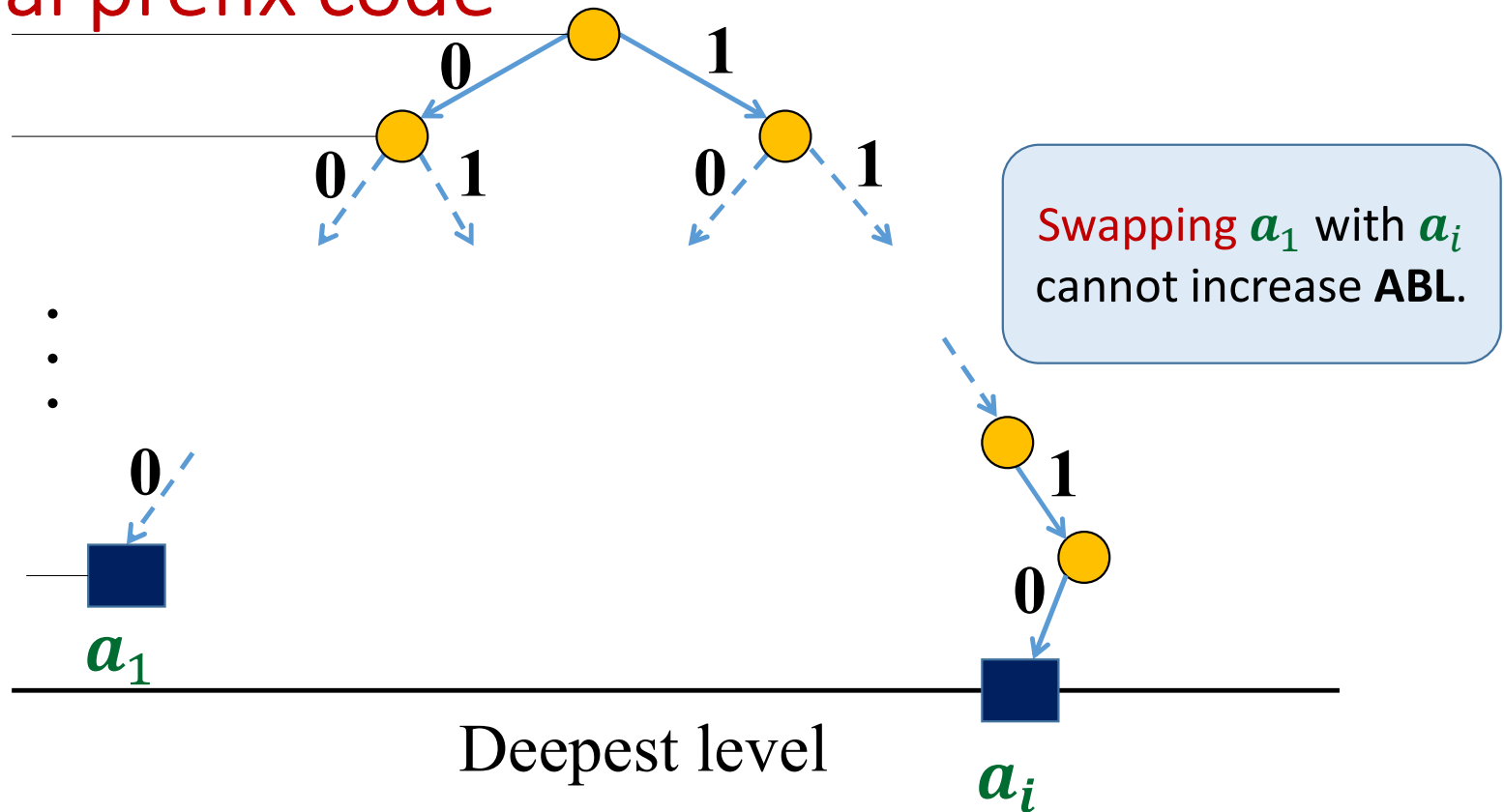
**But how to organize them to achieve optimal prefix code ?**

- We shall now make some simple observations about the structure of the binary tree corresponding to the optimal prefix codes.
- These observations will be about some local structure in the tree.
- Nevertheless, these observations will play a crucial role in the design of a binary tree with optimal prefix code for given **A**.

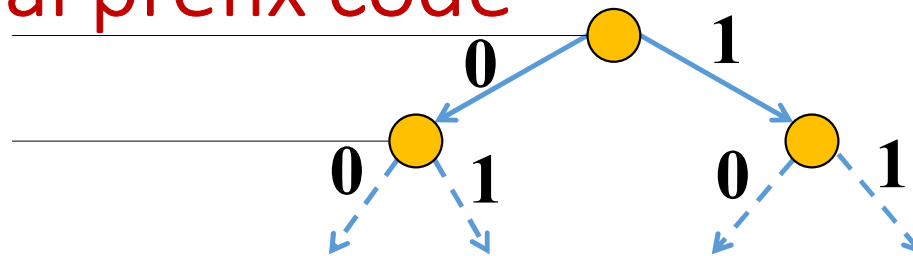
# Observations on the binary tree of an optimal prefix code



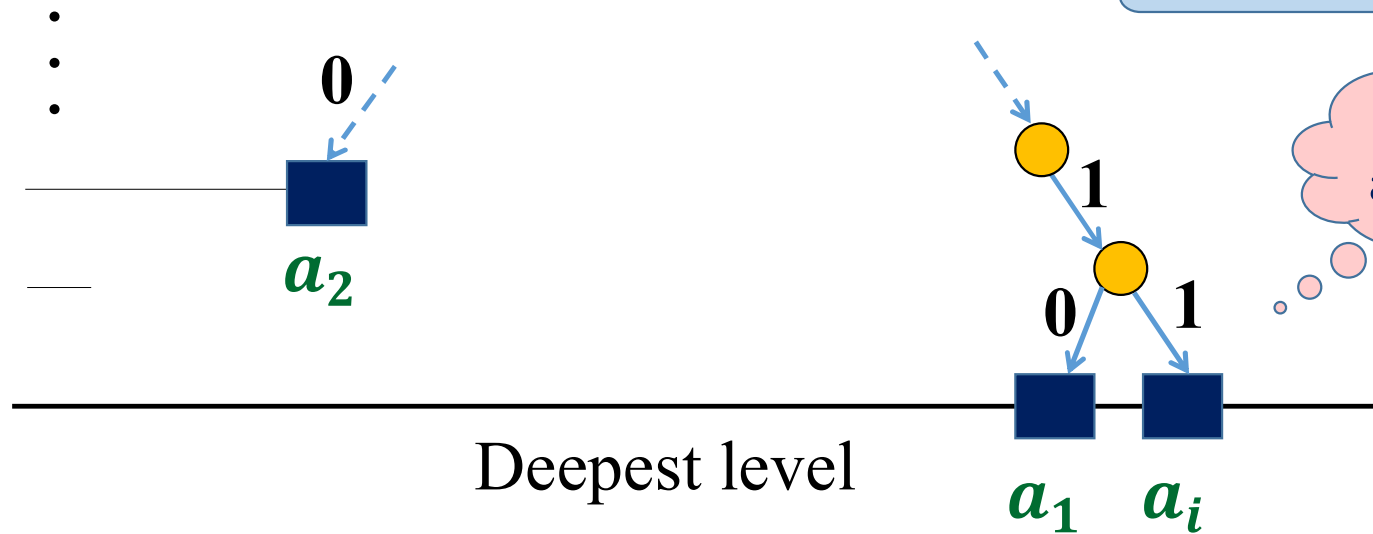
# Observations on the binary tree of an optimal prefix code



# Observations on the binary tree of an optimal prefix code

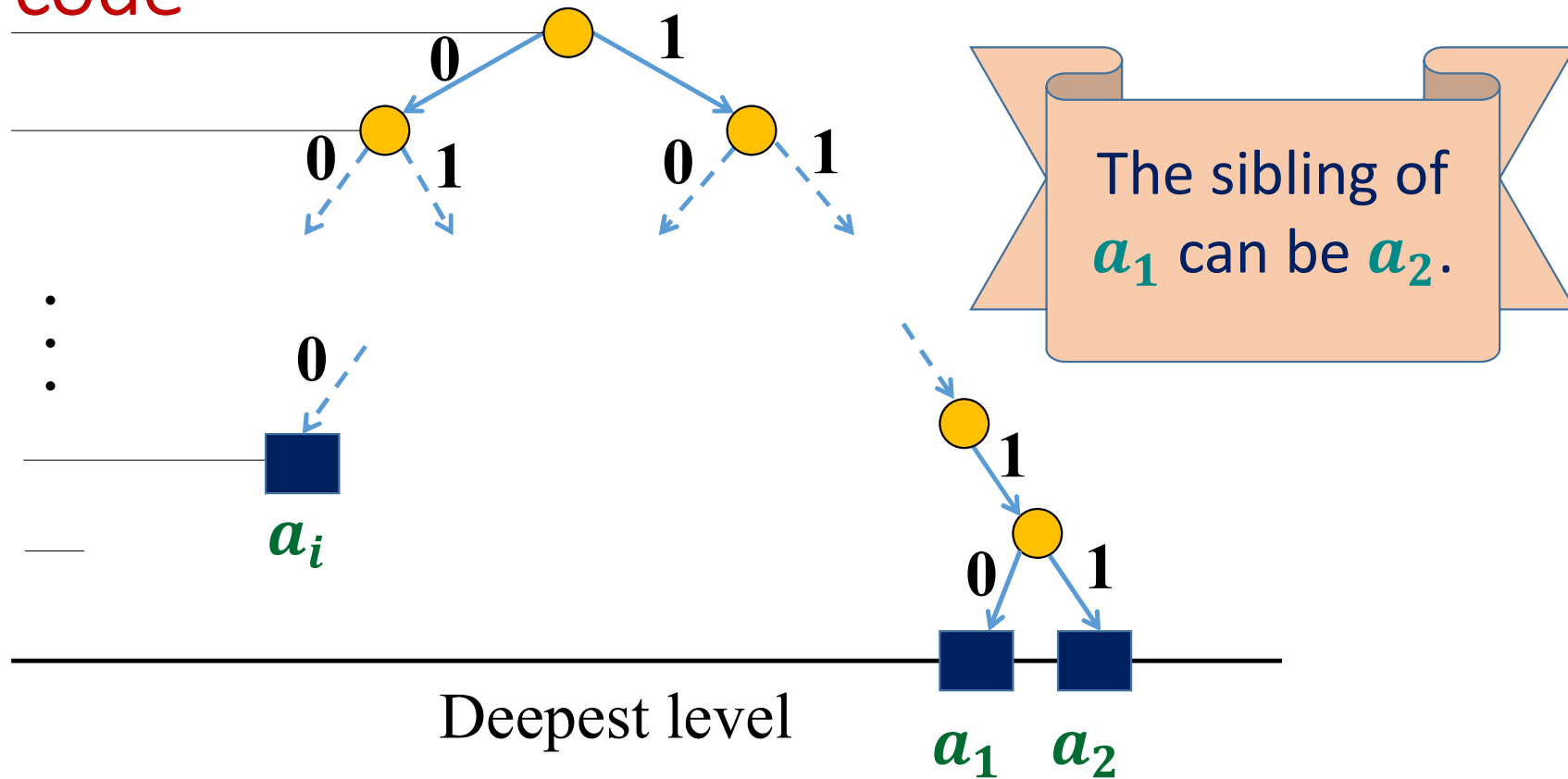


It must be a leaf node. Otherwise  $a_1$  is not at the deepest level.



What about  $a_2$ ?

Observations on the binary tree of an optimal prefix code



# An important observation

**Lemma:** There exists an optimal prefix coding in which  $a_1$  and  $a_2$  appear as siblings in the corresponding labeled binary tree.

**Important note:** It is **inaccurate** to claim that “In every optimal prefix coding,  $a_1$  and  $a_2$  appear as siblings in the labeled binary string.”

But algorithmic implication of the Lemma mentioned above is quite important:

→ We just need to focus on that binary tree of optimal prefix coding in which  $a_1$  and  $a_2$  appear as siblings.

This lemma is a powerful hint to the design of optimal prefix code.

See CLRS Section 16.3 for full details



# Acknowledgement

- The slides are modified from
  - The slides from Prof. Kevin Wayne
  - The slides from Prof. Surender Baswana
  - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - The slides from Prof. Diptarka Chakraborty
  - The slides from Prof. Arnab Bhattacharyya