

# Greedy Algorithms

Desain & Analisis Algoritma

Fakultas Ilmu Komputer

Universitas Indonesia

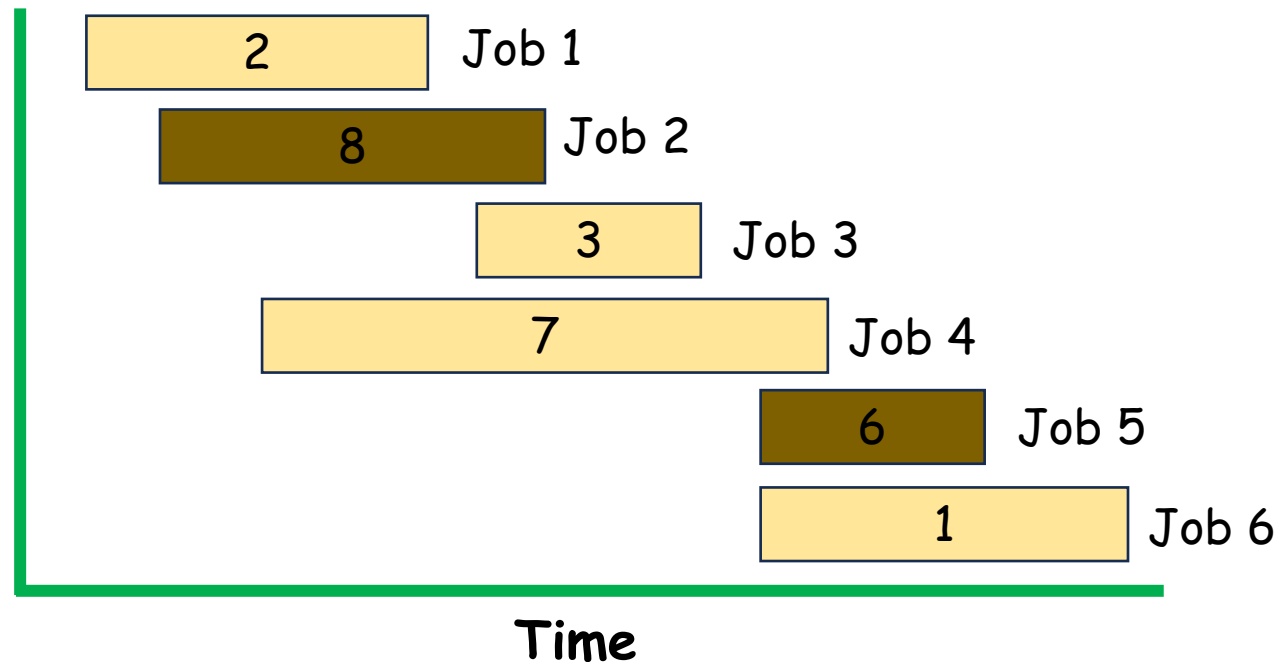
Compiled by **Alfan F. Wicaksono** from multiple sources

# Credits

- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- Dynamic Programming: Weighted Interval Scheduling, CMSC 451: Lecture 10, by Dave Mount

## Review: Weighted Interval Scheduling

- **Goal**: Find a set of compatible jobs such that sum of weights is maximum;
- Jobs  $\rightarrow$  **non-decreasing order** of finish time  $f_j$ ; so that  $f_1 \leq f_2 \leq \dots \leq f_n$ ;
- We define  $p(j)$  : largest index  $i < j$  such that job  $i$  is compatible with  $j$ .



$j$	$p(j)$
0	-
1	0
2	0
3	1
4	0
5	3
6	3

# Review: Weighted Interval Scheduling

## Top-down Dynamic Programming

- Two cases: include job  $j$ ; or do not include job  $j$

**MEMOIZED-OPT(OPT, j):**

```
if j == 0 then  
    return 0
```

```
else
```

```
    if j not in OPT then
```

```
        OPT[j] = max(  $w_j + \text{MEMOIZED-OPT}(\text{OPT}, p(j))$ ,  $\text{MEMOIZED-OPT}(\text{OPT}, j - 1)$  )
```

```
    else
```

```
        return OPT[j]
```

Price for MEMOIZED-OPT:

- Running time =  $\Theta(n)$
- Space =  $\Theta(n)$

**WIS (n):**

```
sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ 
```

```
compute p(1), p(2), ..., p(n)
```

```
create a dictionary (hashtable) OPT
```

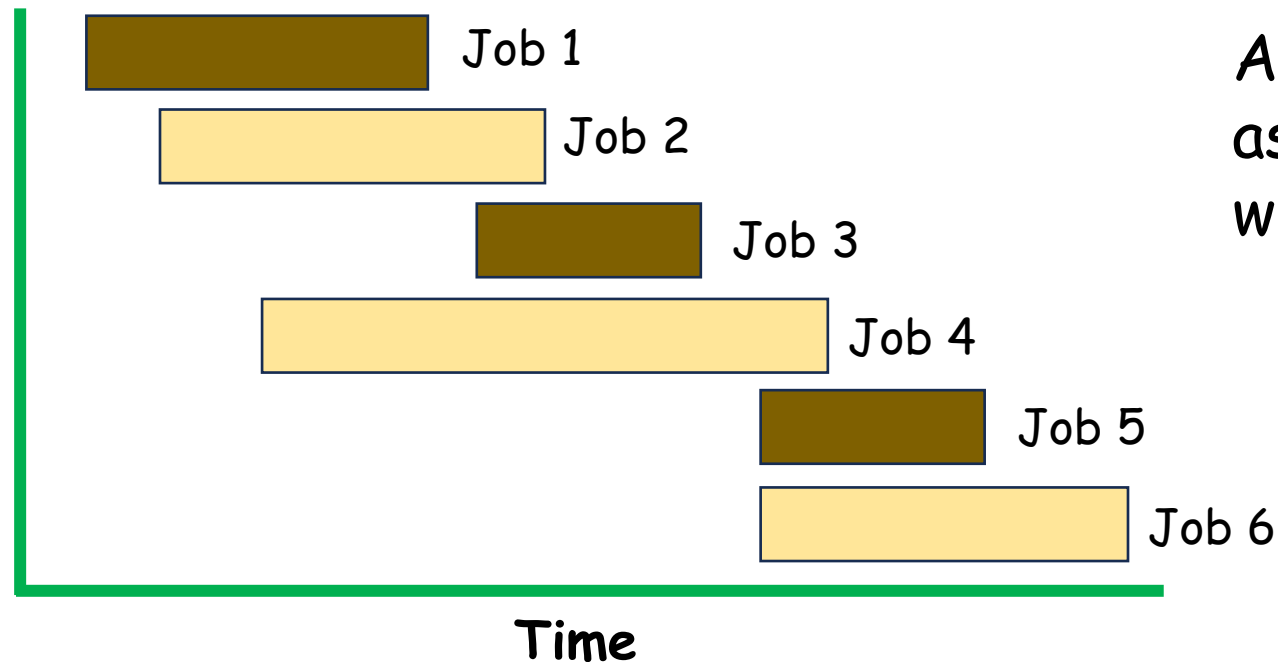
```
return MEMOIZED-OPT(OPT, n)
```

OPT[j] is value of optimal solution  
for jobs 1 to j

## (Unweighted) Interval Scheduling (Activity Selection)

- **Goal**: selecting a **maximum-size set** of mutually compatible jobs;
- Jobs  $\rightarrow$  **non-decreasing order** of finish time  $f_j$ ; so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

In this example,  $\{1, 3, 5\}$  is a largest subset of mutually compatible jobs.

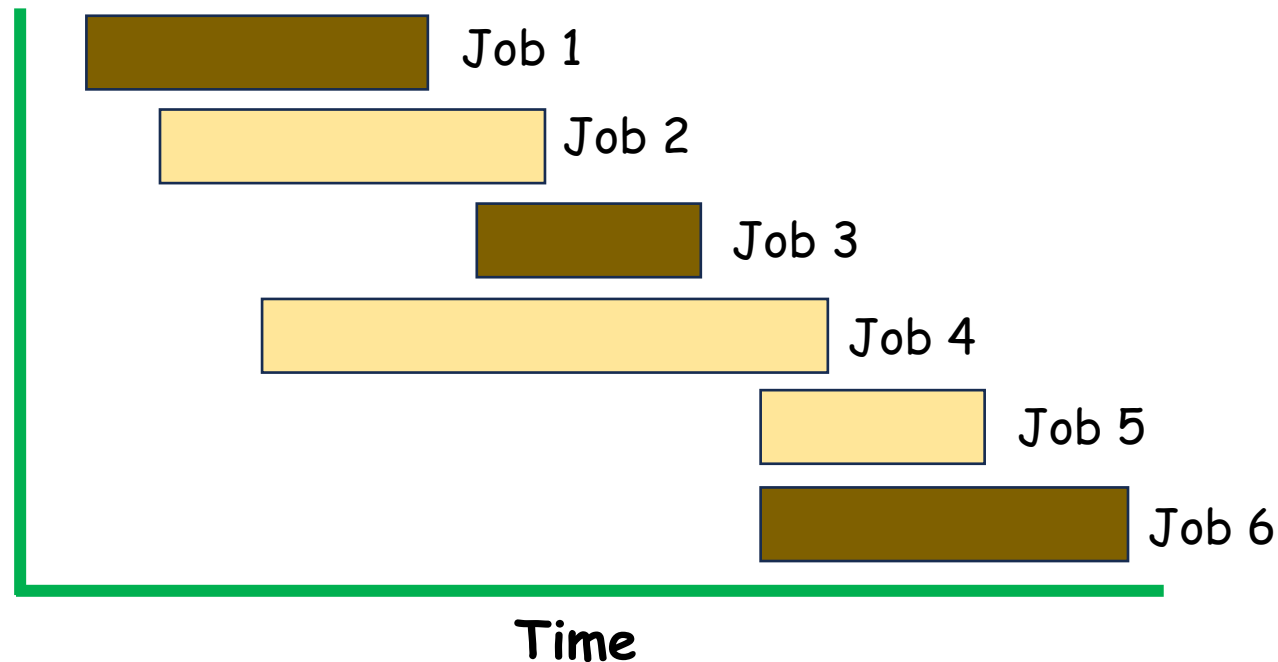


A Job is not associated with weight!

## (Unweighted) Interval Scheduling (Activity Selection)

- **Goal**: selecting a **maximum-size set** of mutually compatible jobs;
- Jobs  $\rightarrow$  **non-decreasing order** of finish time  $f_j$ ; so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

In this example, {1, 3, 6} is another largest subset.



# Shall we utilize DP?

- Algorithms for 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!**
- Simpler/more efficient algorithms will do, such as **Greedy algorithm**.
- **Greedy Solution**: what if you could choose a job to add to an optimal solution without having to first solve all the subproblems?

# Greedy Algorithm

- A greedy algorithm always makes the **choice that looks best at the moment**;
- This makes a locally optimal choice in the hope that this choice leads to a globally optimal solution.
  - Greedy algorithms do not always yield optimal solutions, but for many problems they do.
- This topic explores optimization problems for which greedy algorithms provide optimal solutions.



# Greedy = "Rakus"



Imagine that you're really hungry!

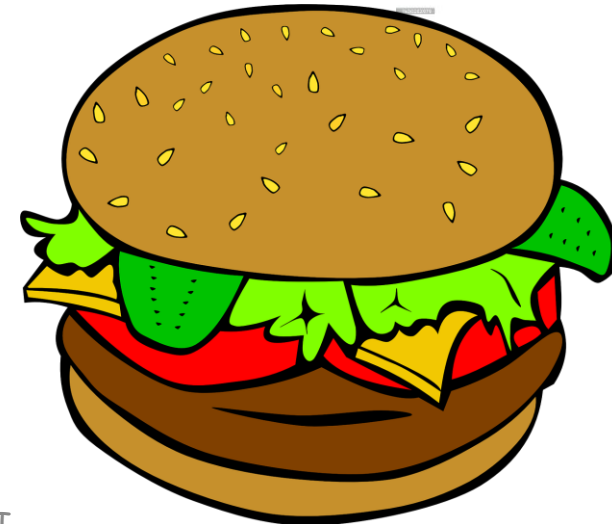
You are provided with three foods:

1. A small cookie
2. A Cherrie
3. A big burger

Which one do you select first to fulfill your desire?



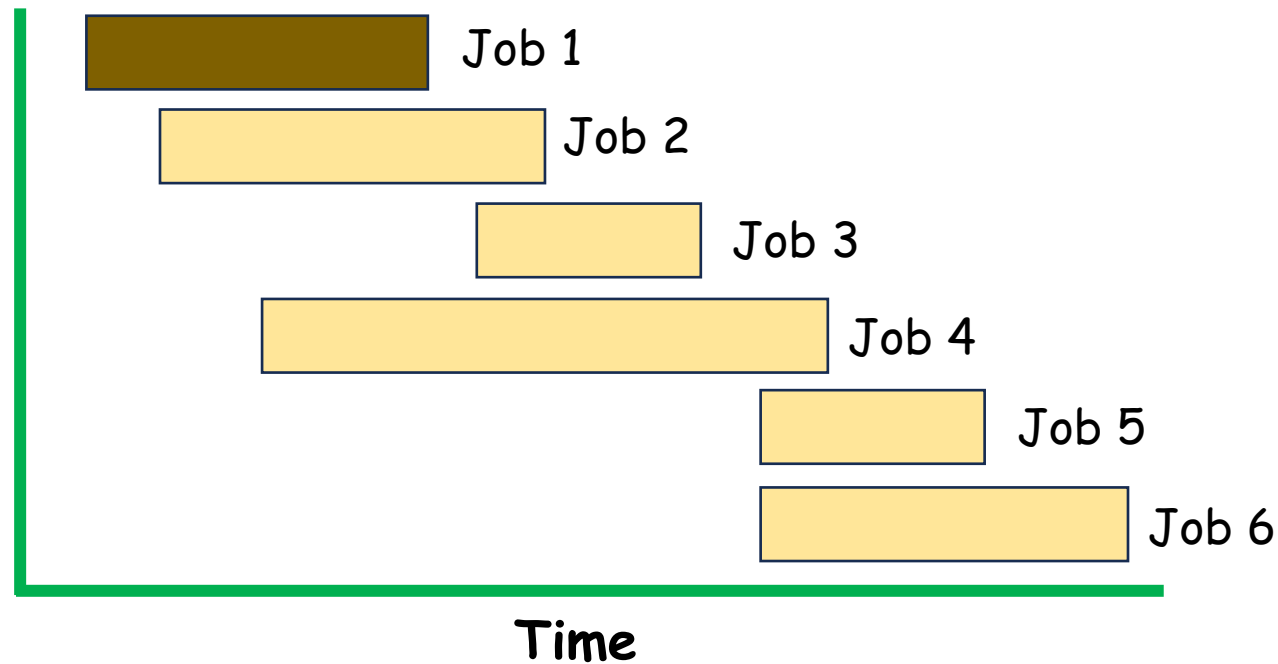
Pixabay



\*All images are free from Pixabay and iStock.

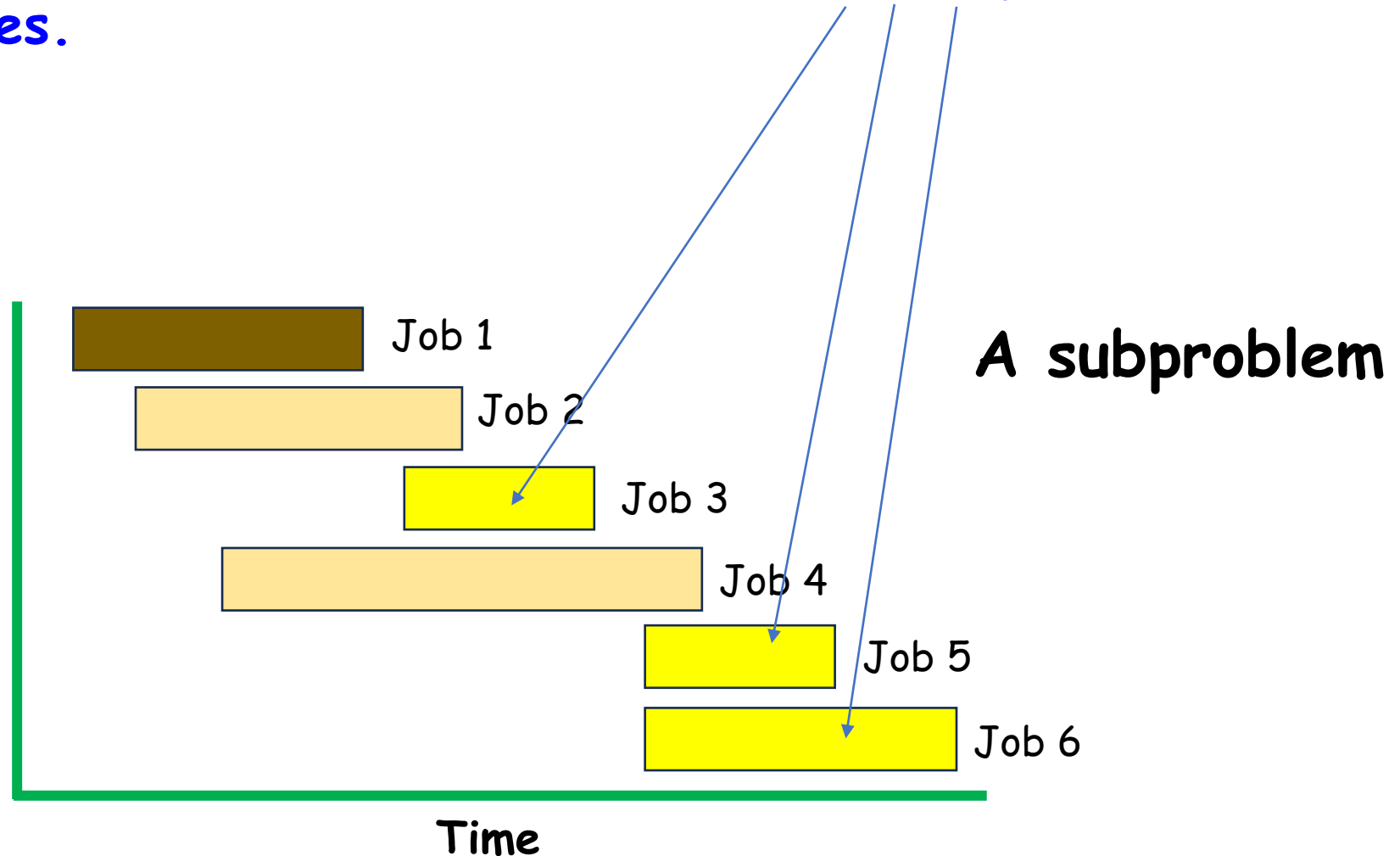
## (Unweighted) Interval Scheduling (Activity Selection)

The first greedy choice is **Job 1**.



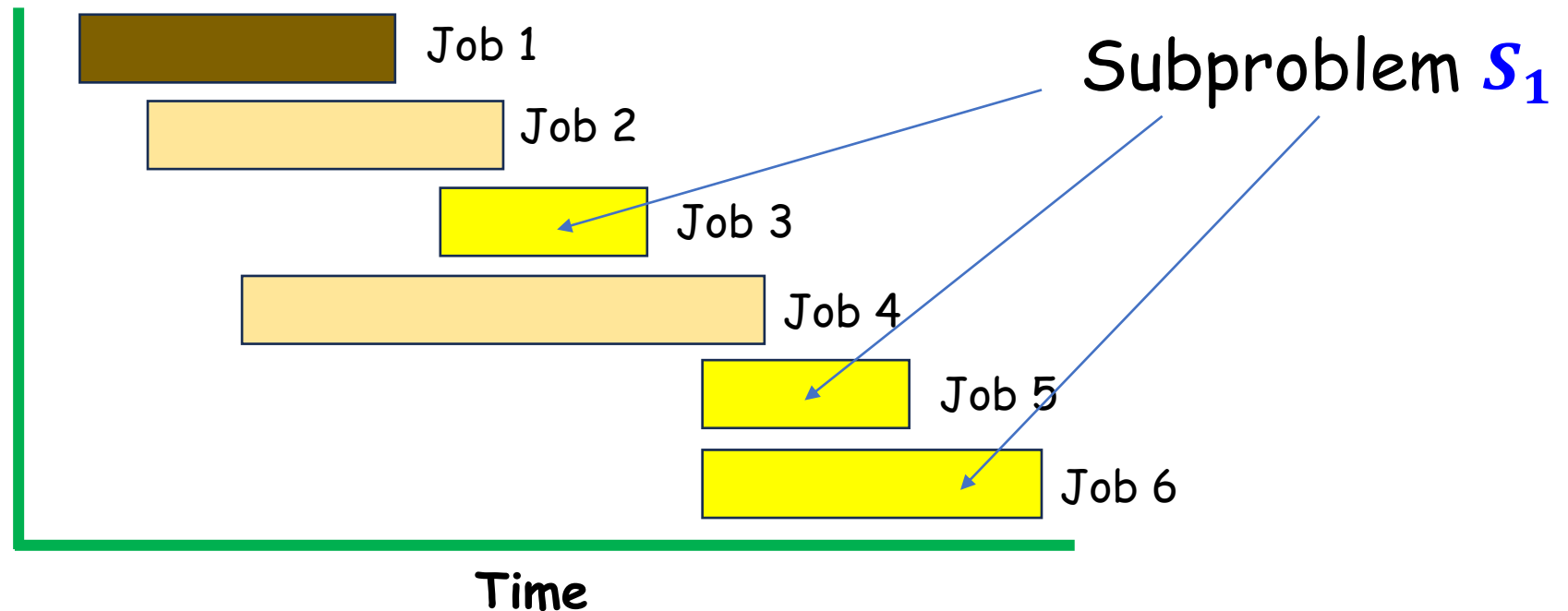
## (Unweighted) Interval Scheduling (Activity Selection)

The first greedy choice is **Job 1**. Once you make the greedy choice, you have only one **remaining subproblem** to solve: **finding activities/jobs that start after Job 1 finishes**.



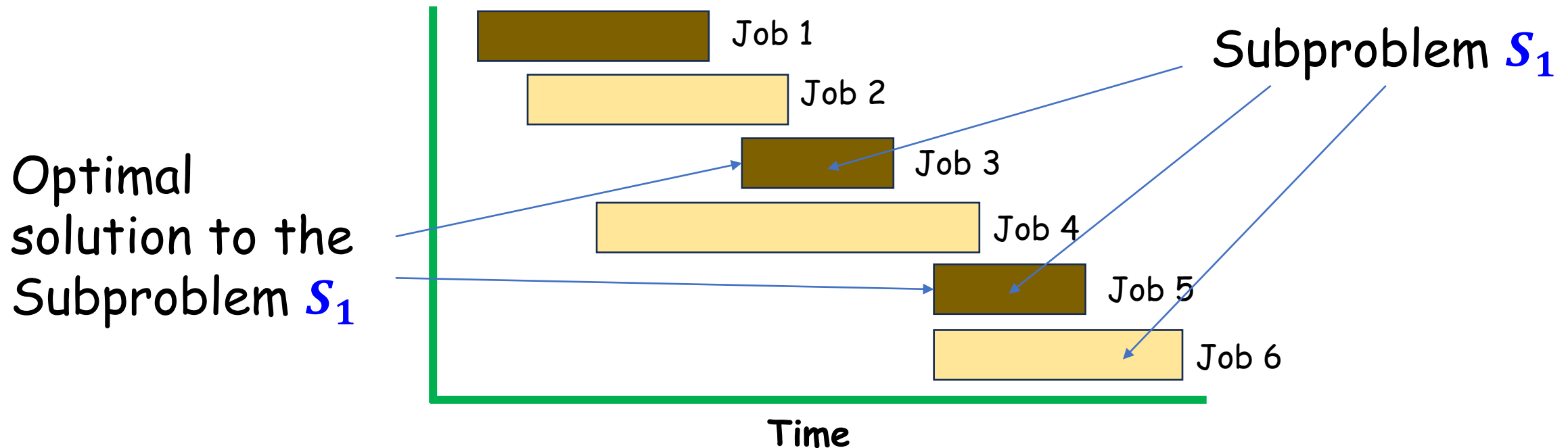
## (Unweighted) Interval Scheduling (Activity Selection)

Let  $S_k = \{j_i \in S : s_i \geq f_k\}$  be the set of jobs that start after  $j_k$  (job  $k$ ) finishes. Optimal substructure says that if  $j_1$  belongs to an optimal solution, ...



## (Unweighted) Interval Scheduling (Activity Selection)

Let  $S_k = \{j_i \in S : s_i \geq f_k\}$  be the set of jobs that start after  $j_k$  (job  $k$ ) finishes. Optimal substructure says that if  $j_1$  belongs to an optimal solution, the solution of the original problem consists of  $j_1$  and all the jobs in an optimal solution to the subproblem  $S_1$ .



## (Unweighted) Interval Scheduling (Activity Selection)

**GREEDY-JOB-SELECTOR( $s_1, \dots, s_n, f_1, \dots, f_n, n$ ):**

$A = \{j_1\}$

$k = 1$

**for**  $m = 2$  **to**  $n$ :

**if**  $s_m \geq f_k$  **then**      *// is  $a_m$  is  $S_k$ ?*

$A = A + \{j_m\}$

$k = m$

**return**  $A$

## (Unweighted) Interval Scheduling (Activity Selection)

### Theorem:

Consider any nonempty subproblem  $S_k$ , and let  $j_m$  be a job in  $S_k$  with the **earliest finish time**. Then,  $j_m$  is included in some maximum-size subset of mutually compatible jobs of  $S_k$ .

## (Unweighted) Interval Scheduling (Activity Selection)

### Proof:

Let  $A_k$  be a maximum-size subset of mutually compatible jobs in  $S_k$ , and let  $j_n$  be the job in  $A_k$  with the earliest finish time.

If  $j_n = j_m$ , we have shown that  $j_m$  belongs to  $A_k$ .

If  $j_n \neq j_m$ , let  $A'_k = (A_k - \{j_n\}) \cup \{j_m\}$  be  $A_k$  but substituting  $j_m$  for  $j_n$ . The jobs in  $A'_k$  are compatible, which follows because the jobs in  $A_k$  are compatible,  $j_n$  is the first job in  $A_k$  to finish, and  $f_m \leq f_n$ . Since  $|A_k| = |A'_k|$ , we conclude that  $A'_k$  is a maximum-size subset of mutually compatible jobs of  $S_k$ , and it includes  $j_m$ .



# Exercise!

We can have another greedy strategy for activity-selection problem:

"Select the activity of **least duration** from among those that are compatible with previously selected activities"

Do you think this greedy strategy will lead to an optimal solution? If not, show an example that the above approach does not work!

# Designing a "Greedy" solution

- Cast the optimization problem as one in which you make a choice and are **left with one subproblem** to solve;
- **Prove** that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is **always safe**;
- Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if you combine an optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.

# Property #1: Greedy-Choice Property

- When you are considering which choice to make, you make the choice that **looks best** in the current problem, **without considering results from subproblems**.
- A greedy solution usually progresses in a top-down fashion, making one greedy choice after another, reducing a problem to a smaller one.
- In dynamic programming, you make a choice at each step, but **the choice usually depends on the solutions to subproblems** -> that's why we solve DP problems in a bottom-up manner, progressing from smaller to larger subproblems.

## Property #2: Optimal Substructure

- A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- An example (from unweighted job selection problem):  
If an optimal solution  $A$  to the problem  $S$  begins with job  $j_1$ , then the set of jobs  $A' = A - \{j_1\}$  is an optimal solution to the subproblem  $S_1 = \{j_i \in S : s_i \geq f_1\}$ .

# DP or Greedy?

- Note that optimal substructure is exploited by both DP and Greedy strategies.
- You might be tempted to generate a DP solution to a problem when a greedy solution suffices **or, conversely,**
- You might mistakenly think that a greedy solution works when in fact a DP solution is required.

Example:

**Unweighted job selection problem & fractional knapsack** → A greedy solution suffices.

**0-1 knapsack problem** → A greedy solution does not lead to an optimal solution; we need a DP solution.

# Knapsack Problem

- 0-1 Knapsack Problem (**You've seen this before!**)

You are given  $n$  items and a "knapsack". Item  $i$  has weight  $w_i > 0$  and value  $v_i > 0$ . The knapsack has capacity of  $W$  kilograms. The goal is to fill knapsack so as to maximize total value.

- Fractional Knapsack Problem

The setup is the same as above, but we can take **fractions of items**, rather than to make a binary (0/1) choice for each item (leave it or take it).

10 KG	Item 1:	Rp. 60	value/weight = 6
20 KG	Item 2:	Rp. 100	value/weight = 5
30 KG	Item 3:	Rp. 120	value/weight = 4

Our Knapsack:

Capacity = 50 KG

Fractional Knapsack Solution:



Item 1  
Rp. 60

Item 2  
Rp. 100

2/3 of Item 3  
Rp. 80

Total = Rp. 240

# Fractional Knapsack Problem

**GREEDY-FRAC-KNAPSACK(W, n-items):**

**sorted\_items** = sort items in descending order of ratio **item.value/item.weight**

**profit** = 0

**for** item in sorted\_items:

**if** item.weight <= W **then**

        W = W - item.weight

        profit = profit + item.value

**else**

        profit = item.profit \* (W / item.weight)

**break**

**return** profit



# Greedy Choice: A Proof by Contradiction

Let **item i** be the item with the maximum value to weight ratio ( $v/w$ ).

Now, we assume that there is an optimal solution where we did not take as much of **item i** as possible and we also assume that our knapsack is full.

Since **item i** has the highest value to weight ratio, there must exist an **item j** in our knapsack such that:

$$\frac{v_j}{w_j} < \frac{v_i}{w_i}$$

# Greedy Choice: A Proof by Contradiction

We can take **item j** of **weight x** from our knapsack and we can add **item i** of **weight x** to our knapsack. The change in our knapsack is:

$$x \frac{v_i}{w_i} - x \frac{v_j}{w_j} = x \left( \frac{v_i}{w_i} - \frac{v_j}{w_j} \right) > 0$$

This is a **contradiction**, and our assumption is not True. In fact, we can improve the solution by taking out some of **item j** and adding more of **item i**.

# Optimal Substructure: A Proof by Contradiction

Let  $X$  is the optimal solution with value  $V$  to problem  $S$  with knapsack capacity  $W$ .

We want to prove that  $X' = X - x_j$  is an optimal solution to the subproblem  $S' = S - \{j\}$  and the knapsack capacity  $W' = W - w_j$ .

**Proof:**

Assume that  $X'$  is not optimal to  $S'$  and we have another solution  $X''$  to  $S'$  that has a higher total value  $V'' > V$ .

# Optimal Substructure: A Proof by Contradiction

Then,  $X'' \cup \{x_j\}$  is a solution to  $S$  with:

$$V'' + v_j > V' + v_j = V$$

This is a **contradiction** because  $V$  is optimal.

## Greedy solution does not give optimal solution for 0-1 Knapsack Problem

10 KG	Item 1:	Rp. 60	value/weight = 6
20 KG	Item 2:	Rp. 100	value/weight = 5
30 KG	Item 3:	Rp. 120	value/weight = 4

Our Knapsack:

Capacity = 50 KG

0-1 Knapsack using **Greedy Solution (not optimal)**:

10 KG (60)	20 KG (100)		Total = Rp. 160
------------	-------------	--	-----------------

0-1 Knapsack using **Dynamic Programming (optimal)**:

20 KG (100)	30 KG (120)	Total = Rp. 230
-------------	-------------	-----------------

# Exercise #1

In what "input condition" does a greedy solution give an optimal value for the 0-1 Knapsack Problem?

# Exercise #2

## Minimum Number of Coins

Given a value of  $V$  and an infinite supply of each of the coins  $\{1, 5, 10, 25\}$ , The task is to find the minimum number of coins needed to make the change? Design a greedy solution!

$$V = 70$$

$$\text{Output} = 4$$

$$\text{Explanation: } 70 = 2 * 25 + 2 * 10$$

# Exercise #3

## Minimum Number of Coins

Are you happy with your solution? 😊 Does your greedy solution work for denominations {1, 5, 6, 9} and  $V = 11$  ?

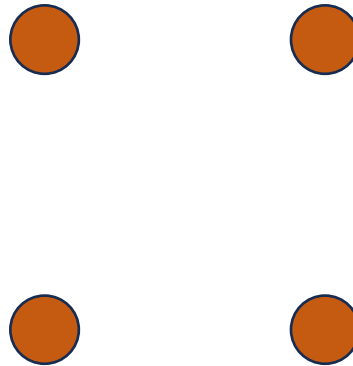
$V = 11$

Output = ?



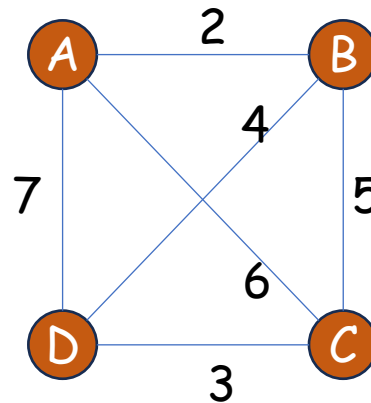
# Minimum Spanning Trees

**Application:** To interconnect a set of  $n$  pins, the designer can use an arrangement of  $n - 1$  wires, each connecting two pins. of all such arrangements, the one that uses the least amount of wire is usually the most desirable.



# Minimum Spanning Trees

This problem can be modelled using a connected, undirected graph  $G = (V, E)$ , where  $V$  is the set of pins,  $E$  is the set of possible interconnections between pairs of pins, and for each edge  $(u, v) \in E$ , a weight  $w(u, v)$  specifies the cost (amount of wired needed) to connect  $u$  and  $v$ .



$$V = \{A, B, C, D\}$$

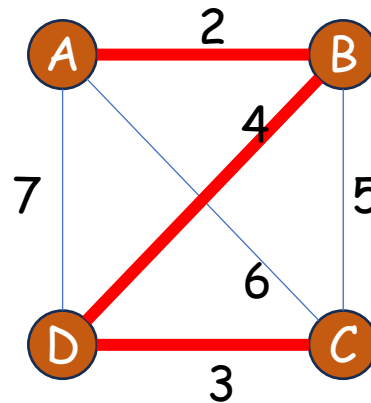
$$E = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$$

$$w(A, C) = 6, \text{ and so on ...}$$

# Minimum Spanning Trees

This problem can be modelled using a connected, undirected graph  $G = (V, E)$ , where  $V$  is the set of pins,  $E$  is the set of possible interconnections between pairs of pins, and for each edge  $(u, v) \in E$ , a weight  $w(u, v)$  specifies the cost (amount of wired needed) to connect  $u$  and  $v$ .

The goal is to find an acyclic subset  $T \subseteq E$  that spans all of the vertices and whose total weight  $\sum_{(u,v) \in T} w(u, v)$  is minimized.



$V = \{A, B, C, D\}$

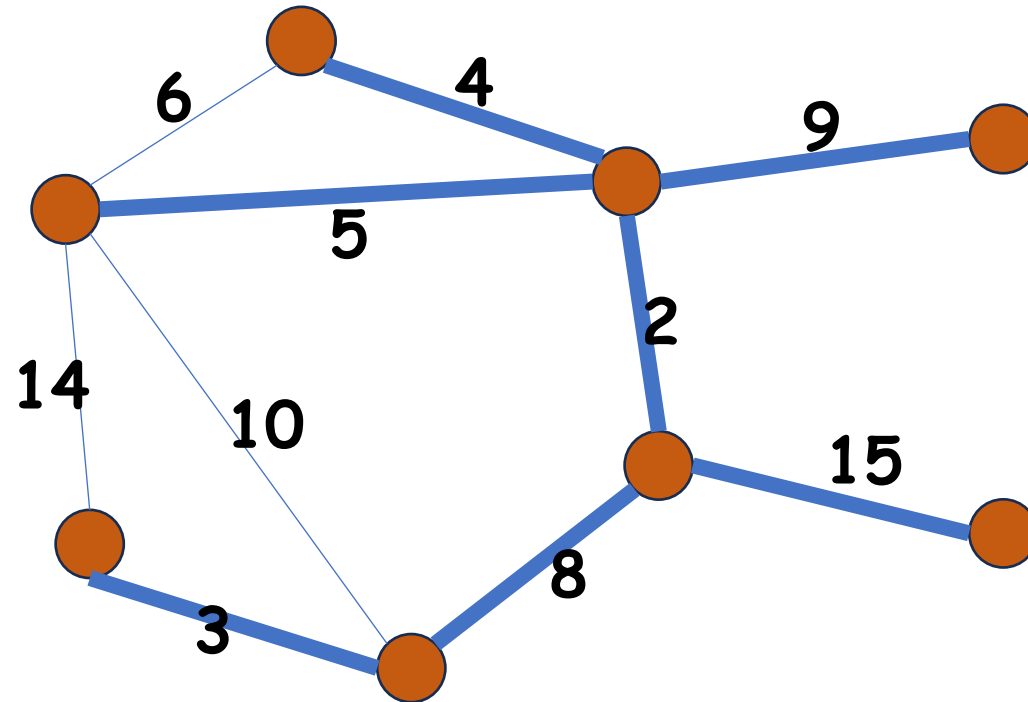
$E = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$

$w(A, C) = 6$ , and so on ...

# Minimum Spanning Trees

Yes, that's a **minimum-weight** spanning tree.

Two popular algorithms: **Prim's** and **Kruskal's** algorithms are applications of the **greedy** method.



# Minimum Spanning Trees

The input to the minimum-spanning-tree problem is a connected, undirected graph  $G = (V, E)$  with a weight function  $w: E \rightarrow \mathbb{R}$ .

Both Prim's and Kruskal's algorithms are specific cases of the general procedure **GENERIC-MST**:

```
GENERIC-MST( $G, w$ ):  
1:   $A = \{\}$   
2:  while  $A$  does not form a spanning tree:  
3:      find an edge  $(u, v)$  that is safe for  $A$   
4:       $A = A \cup \{(u, v)\}$   
5:  return  $A$ 
```

This procedure manages a set  $A$  of edges, maintaining the following **loop invariant**:

**"Prior to each iteration,  $A$  is a subset of some minimum spanning tree."**

# Minimum Spanning Trees

Each step determines a safe edge  $(u, v)$  that the procedure can add to  $A$  without violating this invariant, in the sense that  $A \cup \{(u, v)\}$  is also a subset of a minimum spanning tree.

**Initialization:** After line 1, the set  $A$  trivially satisfies the loop invariant.

**Maintenance:** The loop in lines 2-4 maintains the invariant by adding only safe edges.

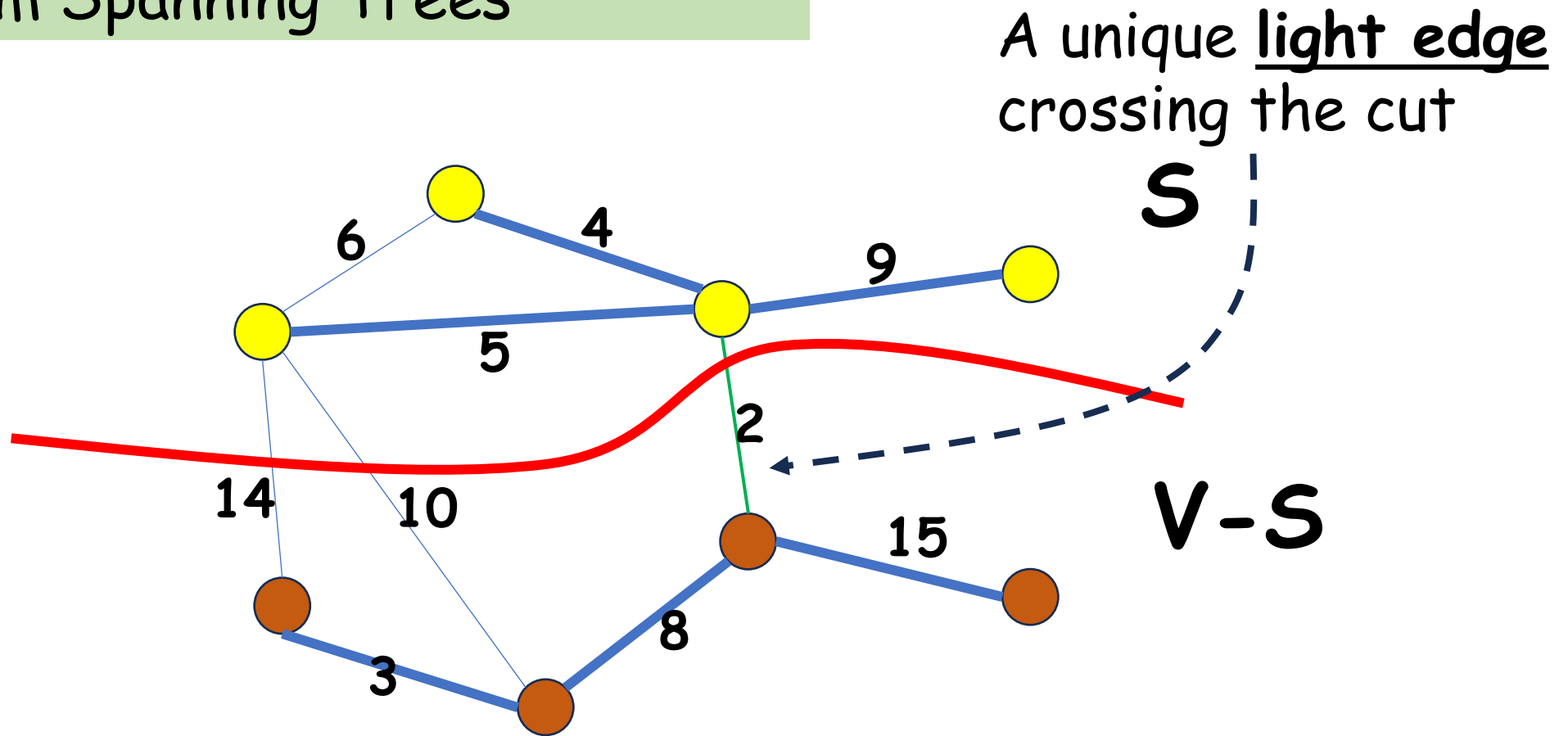
**Termination:** All edges added to  $A$  belong to a minimum spanning tree, and the loop must terminate by the time it has considered all edges. Therefore, the set  $A$  returned in line 5 must be a minimum spanning tree.

# Minimum Spanning Trees

The tricky part is finding a safe edge in line 3. We first need some definitions:

- A **cut**  $(S, V - S)$  of an undirected graph  $G = (V, E)$  is a partition of  $V$ .
- An edge  $(u, v) \in E$  **crosses** the cut  $(S, V - S)$  if one of its endpoints belongs to  $S$  and the other belongs to  $V - S$ .
- A cut **respects** a set  $A$  of edges if no edge in  $A$  crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

# Minimum Spanning Trees



Bold blue edges form a subset  $A$  of the edges. The cut  $(S, V-S)$  respects  $A$ , since no edge of  $A$  crosses the cut.



# Minimum Spanning Trees

The following theorem gives the rule for recognizing safe edges:

## Theorem

Let  $G = (V, E)$  be a connected, undirected graph with a real-valued weight function  $w$  defined on  $E$ . Let  $A$  be a subset of  $E$  that is included in some minimum spanning tree for  $G$ , let  $(S, V-S)$  be any cut of  $G$  that respects  $A$ , and let  $(u, v)$  be a light edge crossing  $(S, V-S)$ .

Then, edge  $(u, v)$  is safe for  $A$ .

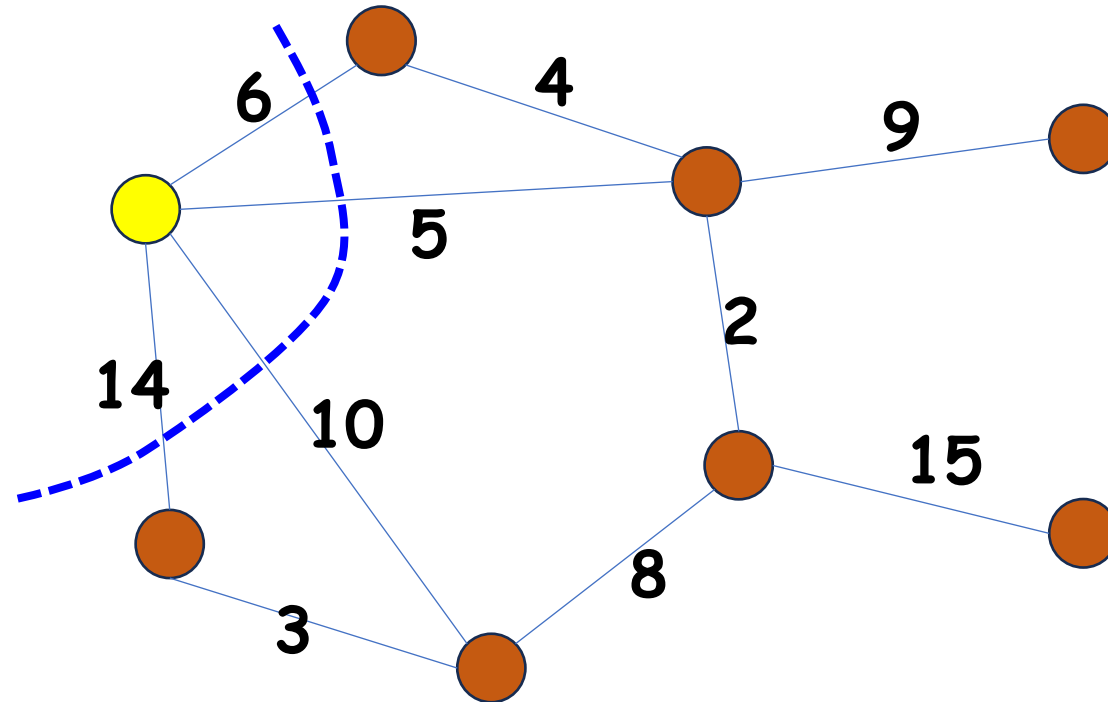
\*See your text book (Intro. To Algorithms by Cormen et al.) for a proof!

# Minimum Spanning Trees

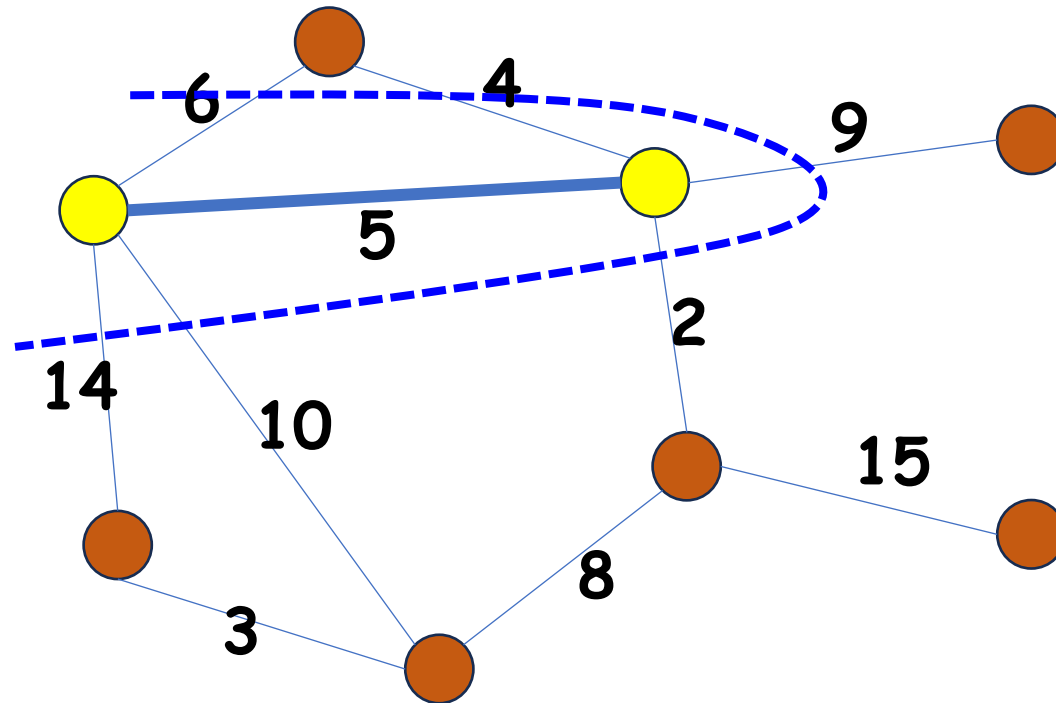
**Prim's algorithm:** a specific case of GENERIC-MST

- Prim's algorithm has the property that the edges in the set **A** always form a single tree;
- The tree starts from an arbitrary root vertex **r** and grows until it spans all the vertices in **V**;
- Each step adds to the tree **A** a **light edge** that connects **A** to an isolated vertex - one on which no edge of **A** is incident.

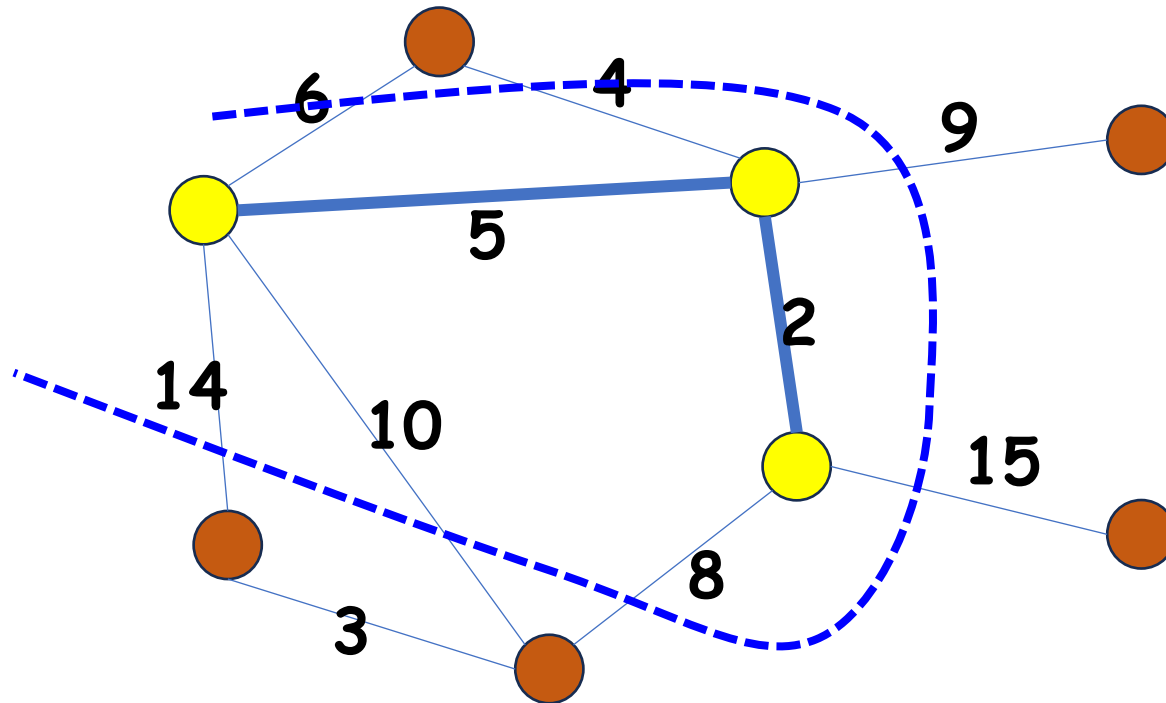
# Minimum Spanning Trees



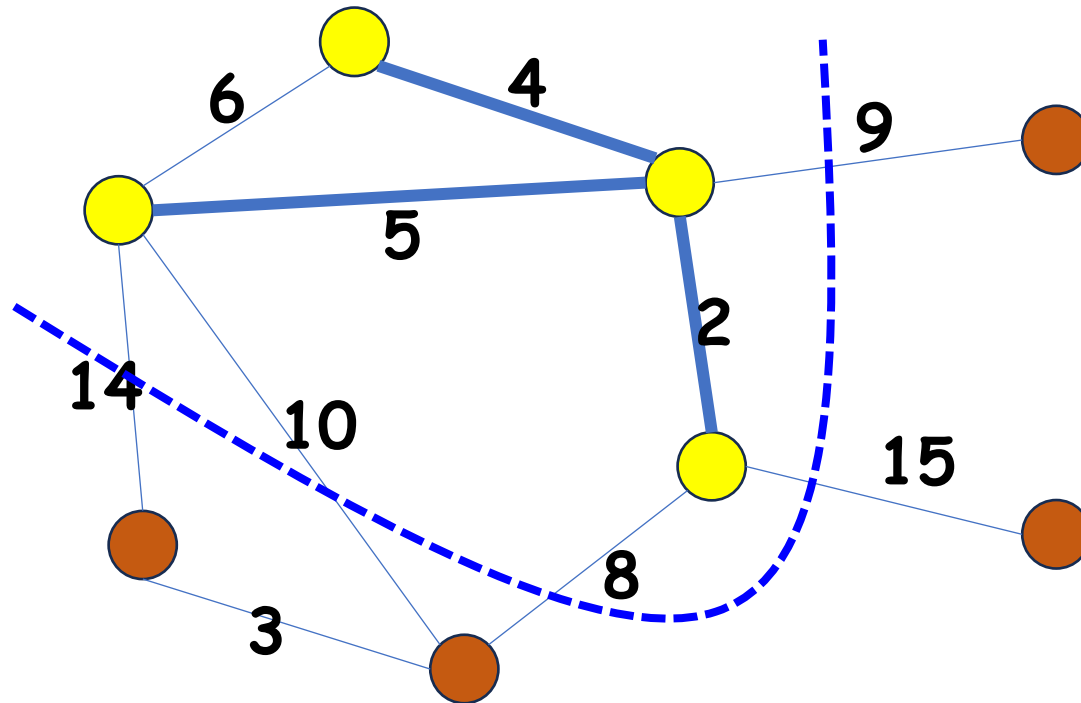
# Minimum Spanning Trees



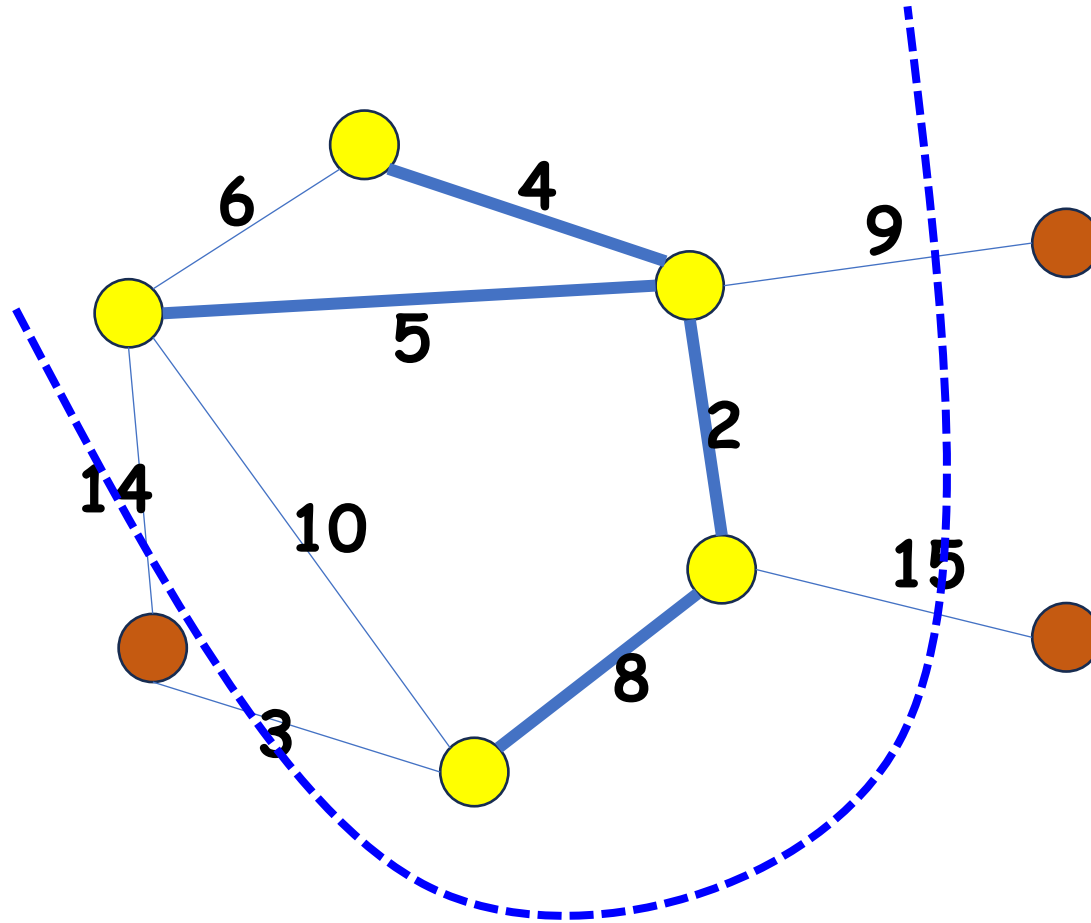
# Minimum Spanning Trees



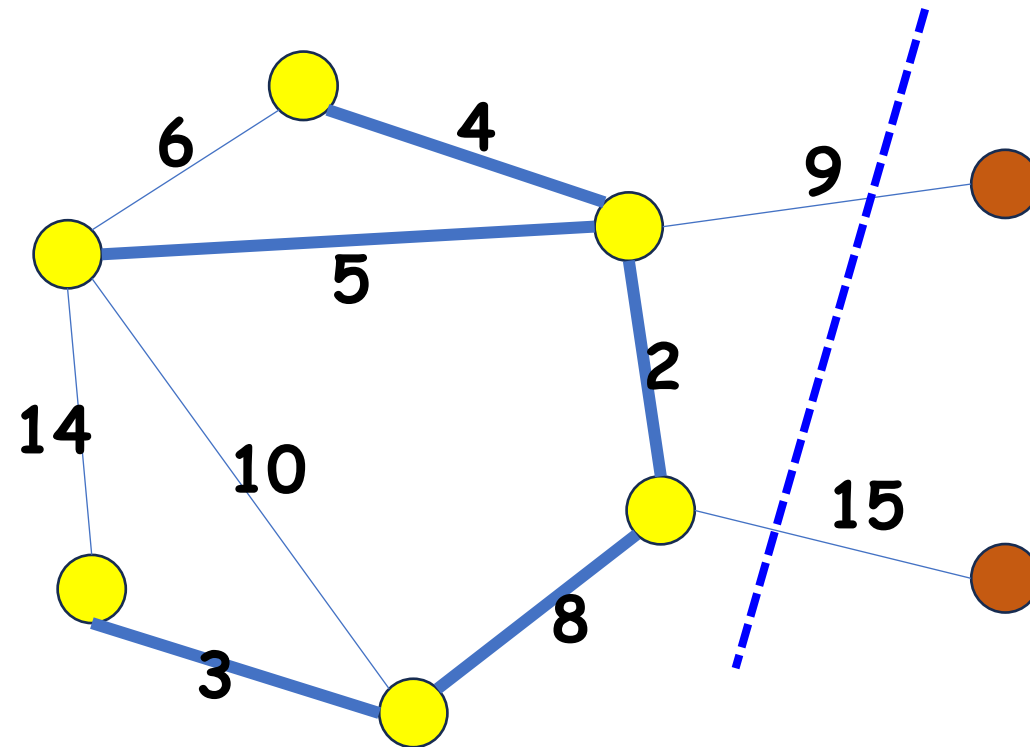
# Minimum Spanning Trees



# Minimum Spanning Trees

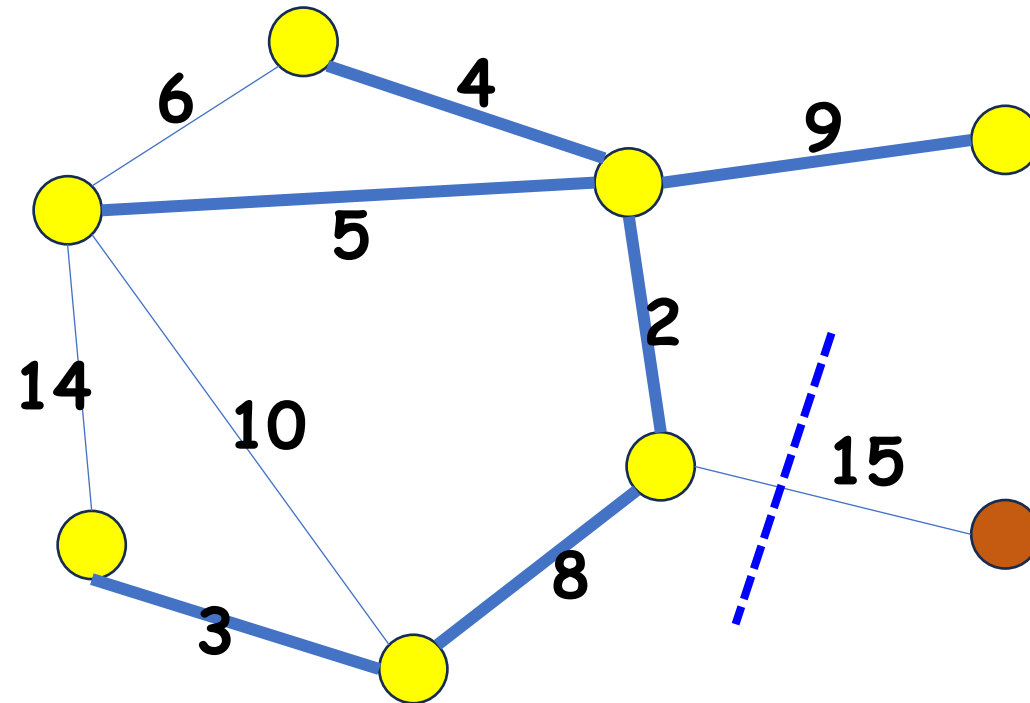


# Minimum Spanning Trees

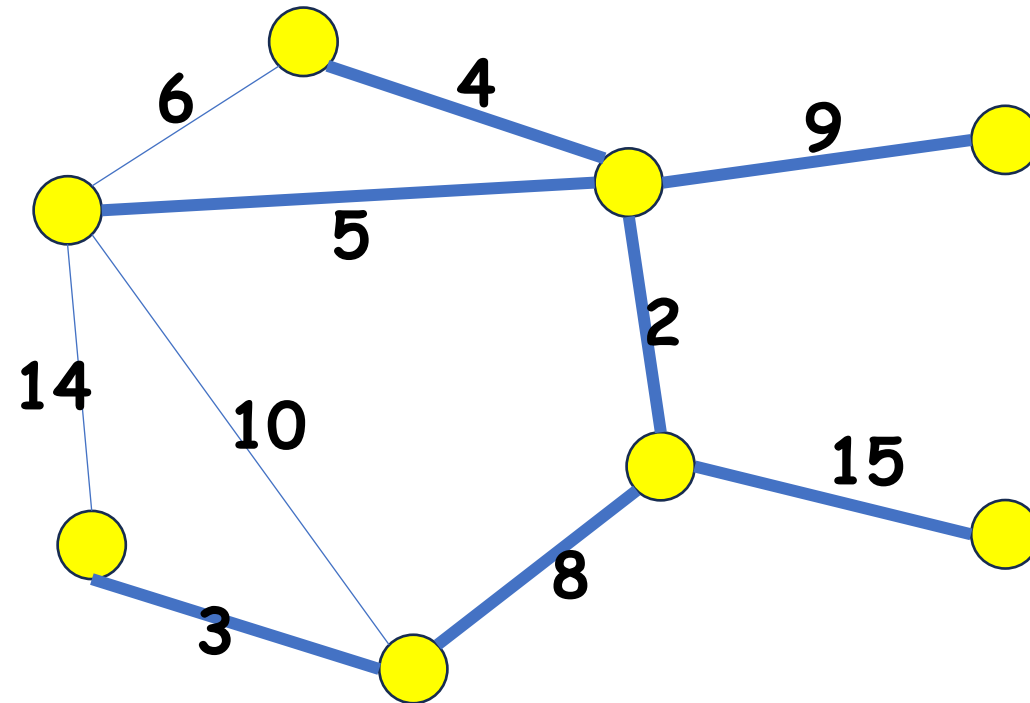




# Minimum Spanning Trees



# Minimum Spanning Trees



# Minimum Spanning Trees

**Prim's algorithm:** How to implement that idea?

All vertices not in the tree are kept in a min-priority queue  $Q$  based on a key field. For each vertex  $v$ ,  $\text{key}[v]$  is the minimum weight of any edge connecting  $v$  to a vertex in the tree. By default,  $\text{key}[v] = \infty$  if there is no such edge.

There is another field,  $\pi[v]$ , denoting the parent of  $v$  in the tree.

During the execution of the algorithm, the set  $A$  from GENERIC-MST is kept implicitly as:

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$$

When it terminates, the  $Q$  is empty. Therefore the MST  $T$  for  $G$  is

$$T = A = \{(v, \pi[v]) : v \in V - \{r\}\}$$

# Minimum Spanning Trees

MST-PRIM( $G, w, r$ )

**for** each  $u \in V[G]$ :

$\text{key}[u] = \infty$

$\pi[u] = \text{NULL}$

$\text{key}[r] = 0$

$Q = V[G]$

**while**  $Q$  is not  $\{\}$ :

$u = \text{EXTRACT-MIN}(Q)$

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

**if**  $v \in Q$  **and**  $w(u, v) < \text{key}[v]$  **then**

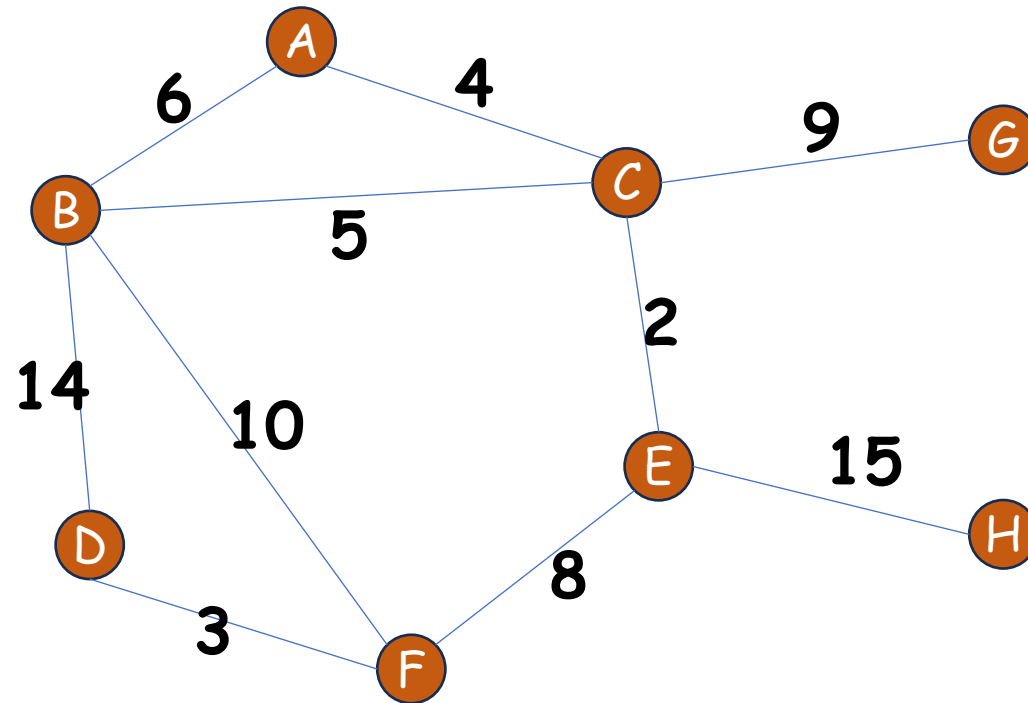
            // update  $\pi[v] = u$  and  $\text{key}[v] = w(u, v)$

$\text{UPDATE-KEY}(Q, v, u, w(u, v))$

# Minimum Spanning Trees

$Q = \{(B, 0, N), (A, \infty, N), (C, \infty, N), (D, \infty, N), (E, \infty, N), (F, \infty, N), (G, \infty, N), (H, \infty, N)\}$

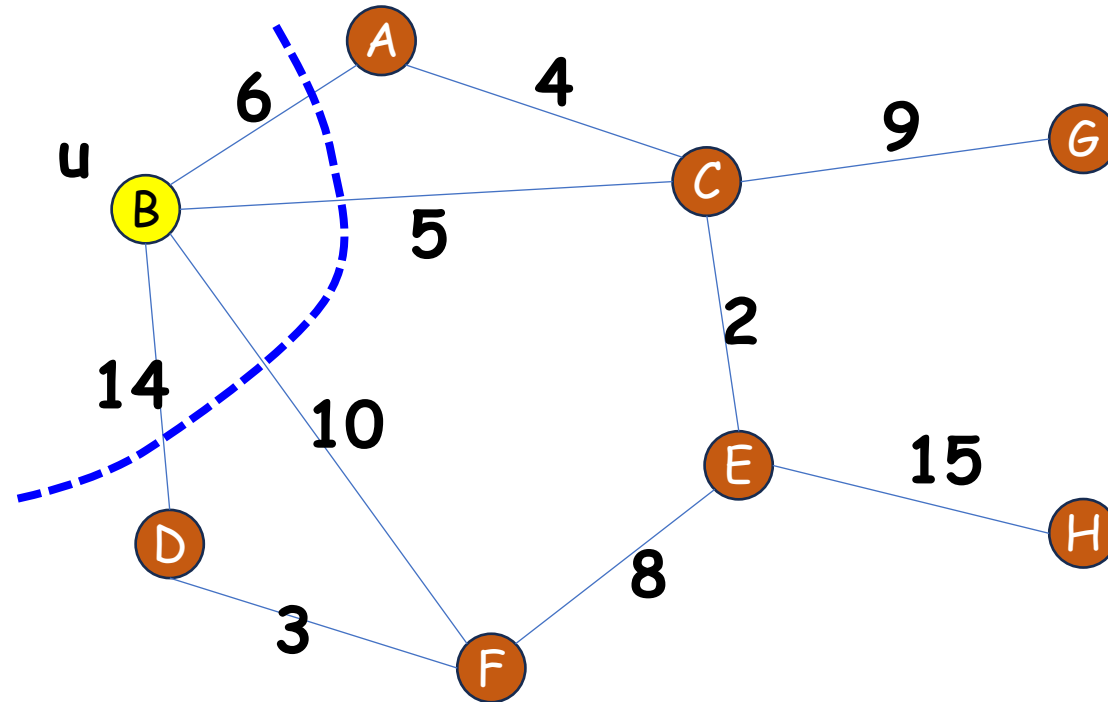
$A = \{\}$



# Minimum Spanning Trees

$Q = \{(C, 5, B), (A, 6, B), (F, 10, B), (D, 14, B), (E, \infty, N), (G, \infty, N), (H, \infty, N)\}$

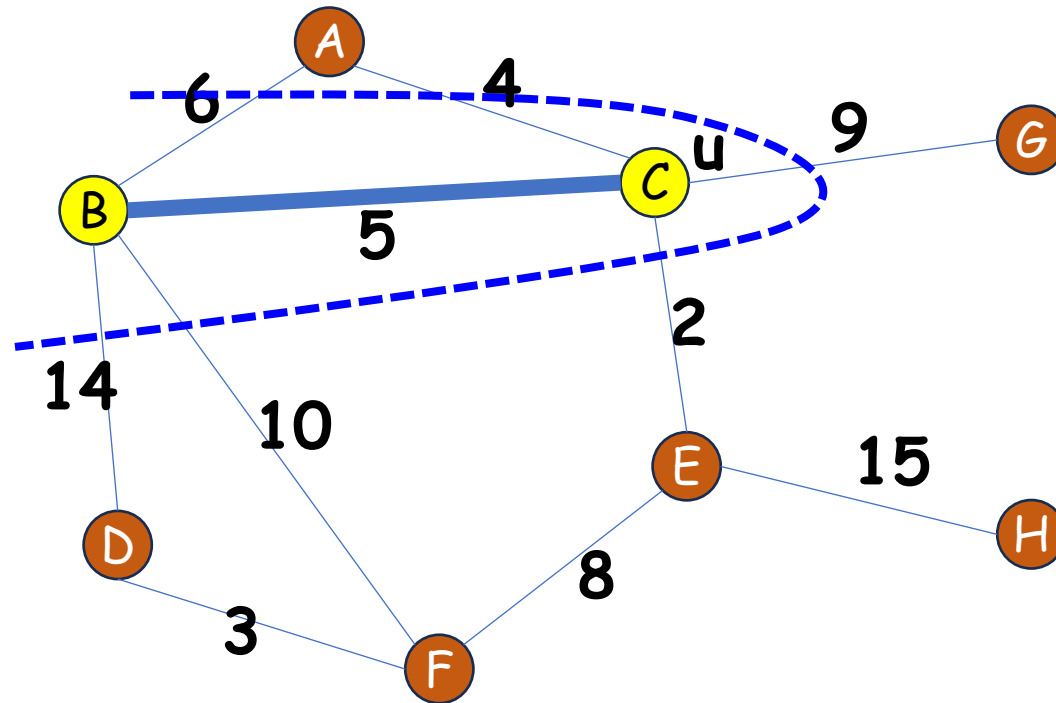
$A = \{(B, N)\}$



# Minimum Spanning Trees

$Q = \{(A,6,B), (F,10,B), (D,14,B), (E,\infty,N), (G,\infty,N), (H,\infty,N)\}$

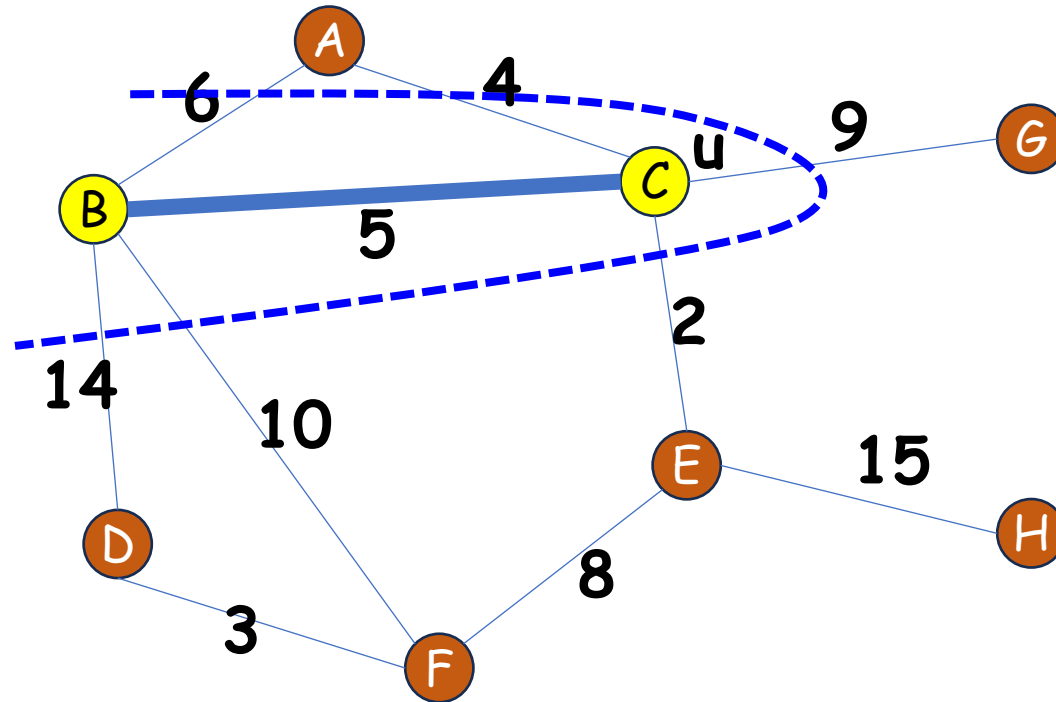
$A = \{(B,N), (C,B)\}$



# Minimum Spanning Trees

$Q = \{(E, 2, C), (A, 4, C), (G, 9, C), (F, 10, B), (D, 14, B), (H, \infty, N)\}$

$A = \{(B, N), (C, B)\}$

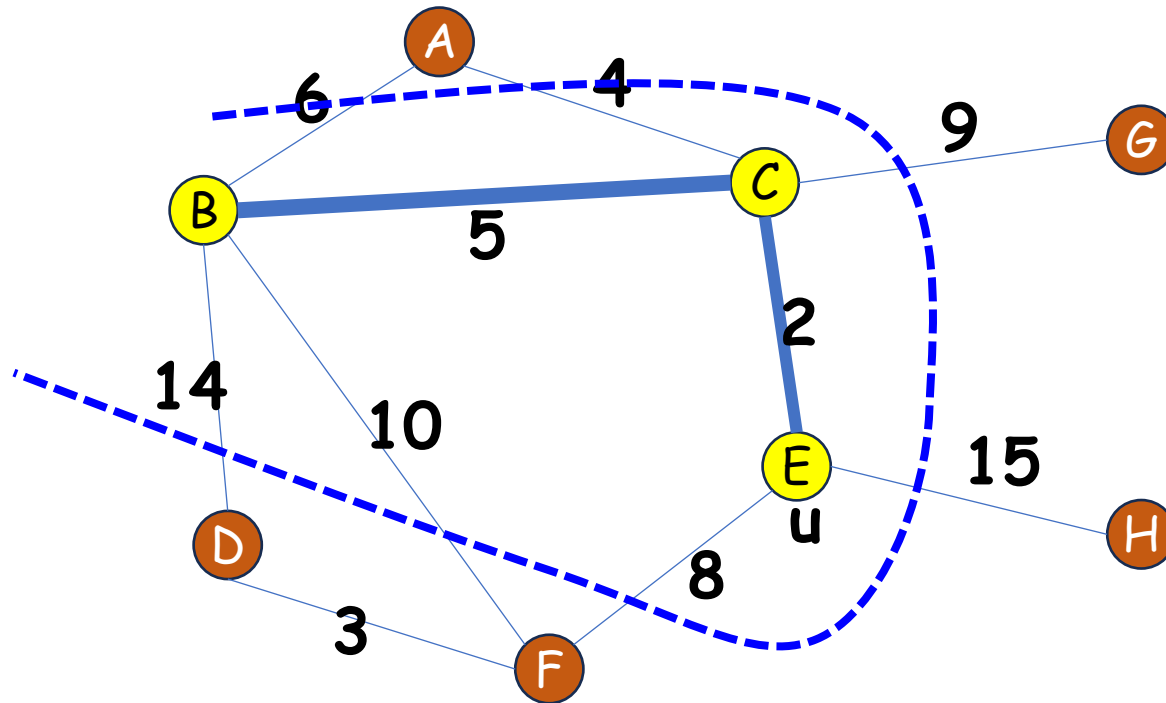




# Minimum Spanning Trees

$Q = \{(A, 4, C), (G, 9, C), (F, 10, B), (D, 14, B), (H, \infty, N)\}$

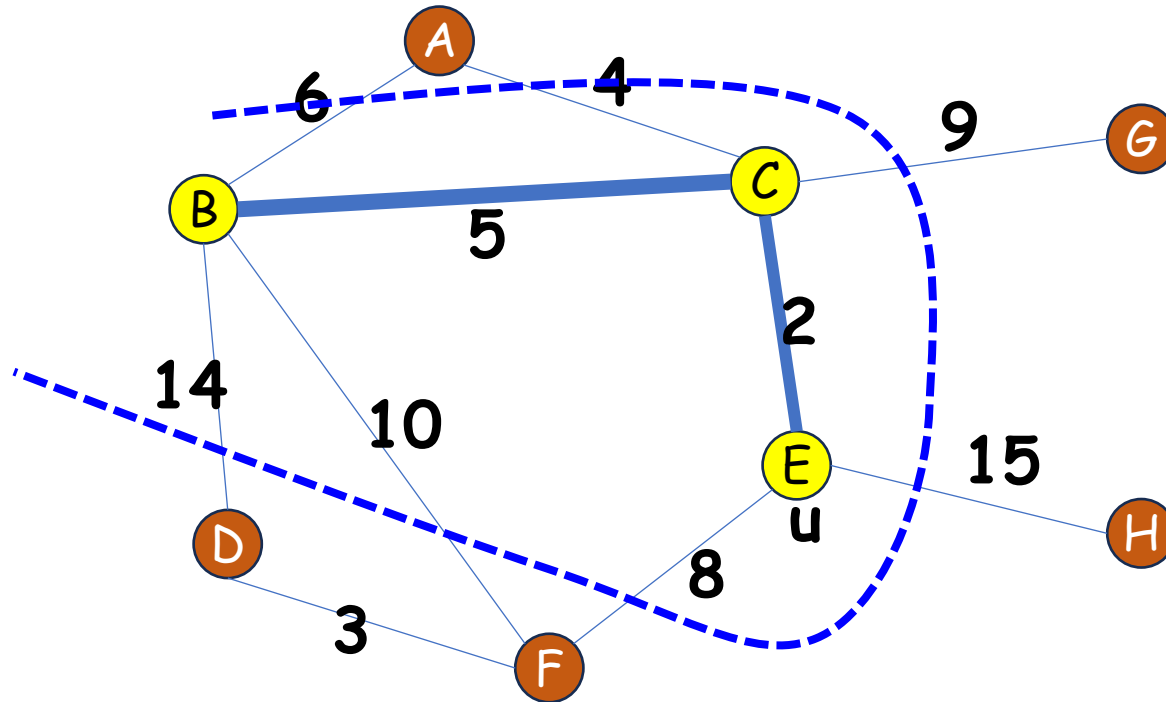
$A = \{(B, N), (C, B), (E, C)\}$



# Minimum Spanning Trees

$Q = \{(A, 4, C), (F, 8, E), (G, 9, C), (D, 14, B), (H, 15, E)\}$

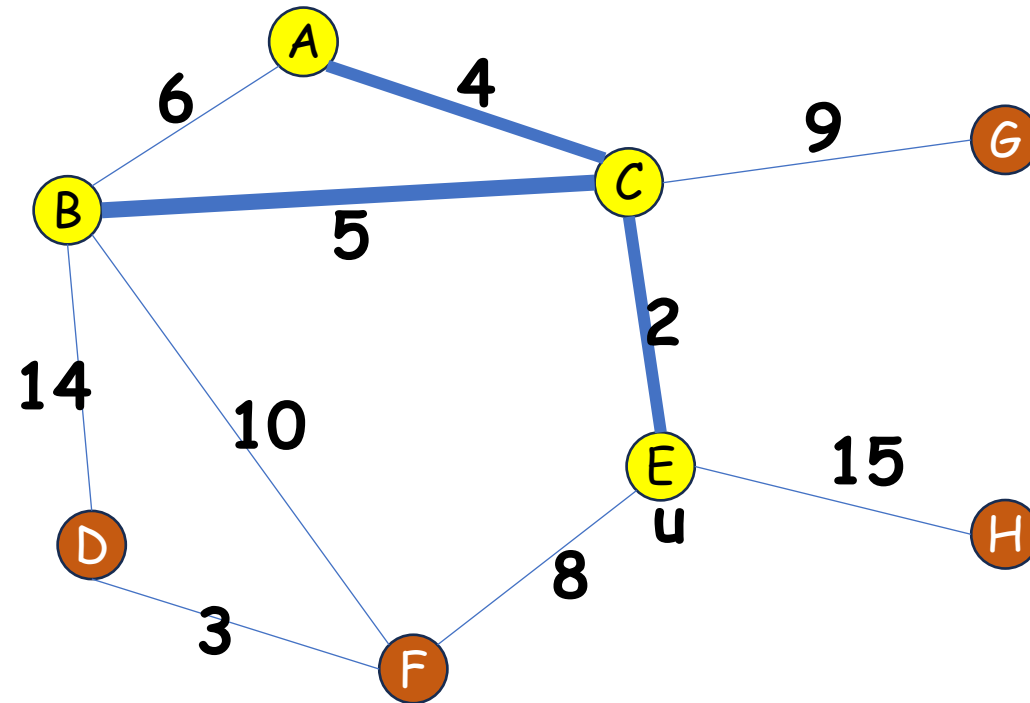
$A = \{(B, N), (C, B), (E, C)\}$



# Minimum Spanning Trees

$Q = \{(A, 4, C), (F, 8, E), (G, 9, C), (D, 14, B), (H, 15, E)\}$

$A = \{(B, N), (C, B), (E, C), (A, C)\}$



And so on, and so forth, ...

# Minimum Spanning Trees

MST-PRIM( $G, w, r$ )

**for** each  $u \in V[G]$ :

$\text{key}[u] = \infty$

$\pi[u] = \text{NULL}$

$\text{key}[r] = 0$

$Q = V[G]$

**while**  $Q$  is not  $\{\}$ :

$u = \text{EXTRACT-MIN}(Q)$

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

**if**  $v \in Q$  **and**  $w(u, v) < \text{key}[v]$  **then**

            // update  $\pi[v] = u$  and  $\text{key}[v] = w(u, v)$

$\text{UPDATE-KEY}(Q, v, u, w(u, v))$

If we implement  $Q$  as a **binary min-heap**, this initialization can be done in  $O(|V|)$  time.

# Minimum Spanning Trees

MST-PRIM( $G, w, r$ )

**for** each  $u \in V[G]$ :

$\text{key}[u] = \infty$

$\pi[u] = \text{NULL}$

$\text{key}[r] = 0$

$Q = V[G]$

**while**  $Q$  is not  $\{\}$ :

$u = \text{EXTRACT-MIN}(Q)$  }

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

**if**  $v \in Q$  **and**  $w(u, v) < \text{key}[v]$  **then**

            // update  $\pi[v] = u$  and  $\text{key}[v] = w(u, v)$

$\text{UPDATE-KEY}(Q, v, u, w(u, v))$

The body of while loop is executed  $|V|$  times, and since each EXTRACT-MIN operation requires  $O(\log |V|)$ , then total running time is  $O(|V| \log |V|)$ .

# Minimum Spanning Trees

MST-PRIM( $G, w, r$ )

**for** each  $u \in V[G]$ :

key[ $u$ ] =  $\infty$

$\pi[u]$  = NULL

key[ $r$ ] = 0

$Q = V[G]$

**while**  $Q$  is not {}:

$u = \text{EXTRACT-MIN}(Q)$

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

**if**  $v \in Q$  **and**  $w(u, v) < \text{key}[v]$  **then**

// update  $\pi[v] = u$  and  $\text{key}[v] = w(u, v)$

UPDATE-KEY( $Q, v, u, w(u, v)$ )

Total running time

=  $O(|V| \log |V| + |E| \log |V|)$

=  $O(|E| \log |V|)$  **why?**

Sum of the length of all **adjacency lists** is  $2|E|$ . So, this **for** loop is executed in  $O(|E|)$  times.

The membership test can be implemented in  $O(1)$  time by keeping a bit for each vertex, telling whether or not it is in  $Q$ .

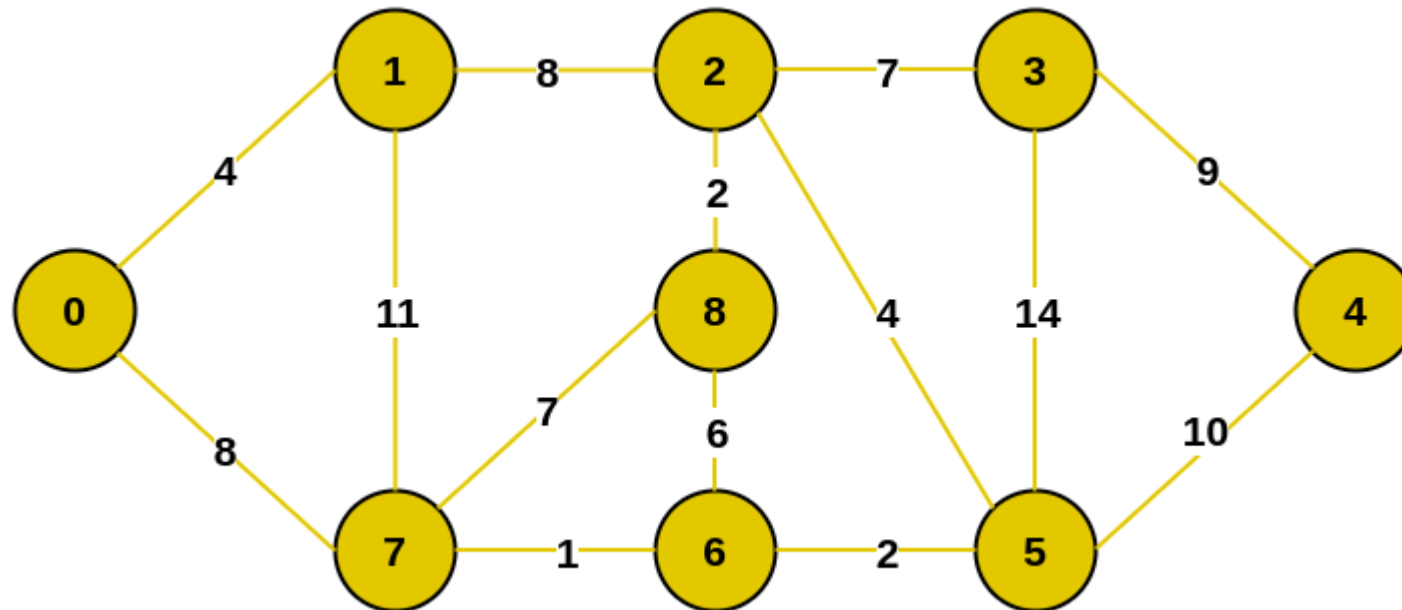
Updating a key in a binary min-heap is  $O(\log |V|)$ .

# Exercise - Easy

- For a connected & undirected graph  $G = (V, E)$ , there is not point in minimizing the number of edges in a MST  $T$ , since all spanning trees have exactly ..... Edges.

# Exercise - Easy

- Find two MSTs from the following graph. Remember that MST is not unique!





# Exercise - Easy

- Can you tell us a condition where the solution of an MST problem is unique?

# Exercise - Medium

- Give a simple implementation of Prim's algorithm that runs in  $O(|V|^2)$  time when the graph  $G = (V, E)$  is represented as an adjacency matrix.
- Hint: you need to maintain **three** 1-D arrays:
  - Parents,  $\pi[u]$
  - Keys,  $\text{key}[u]$
  - Binary indicators,  $\text{mst}$ , where  $\text{mst}[u]$  is 1 if  $u$  is in the MST, and 0 if otherwise.

# Minimum Spanning Trees

## Kruskal's Algorithm

Kruskal's algorithm finds a **safe edge** to add to the growing forest by finding, of all the edges that connect **any two trees in the forest**, an edge  **$(u, v)$**  with the lowest weight.

Let  **$C1$**  and  **$C2$**  denote the two trees that are connected by  **$(u, v)$** . Since  **$(u, v)$**  must be a **light edge** connecting  **$C1$**  to some other tree, the latest theorem implies that  **$(u, v)$**  is a **safe edge** for  **$C1$** .

# Minimum Spanning Trees

MST-KRUSKAL( $G, w$ )

$A = \{\}$

**for** each vertex  $v \in G.V$

MAKE-SET( $v$ )

create a single list of the edges in  $G.E$

sort the list of edges into monotonically increasing order by weight  $w$

**for** each edge  $(u, v)$  from the sorted list:


**if** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) **then**

$A = A \cup \{(u, v)\}$

UNION( $u, v$ )

**return**  $A$

MST-KRUSKAL on the following page uses a **disjoint-set data structure** to maintain several disjoint sets of elements.

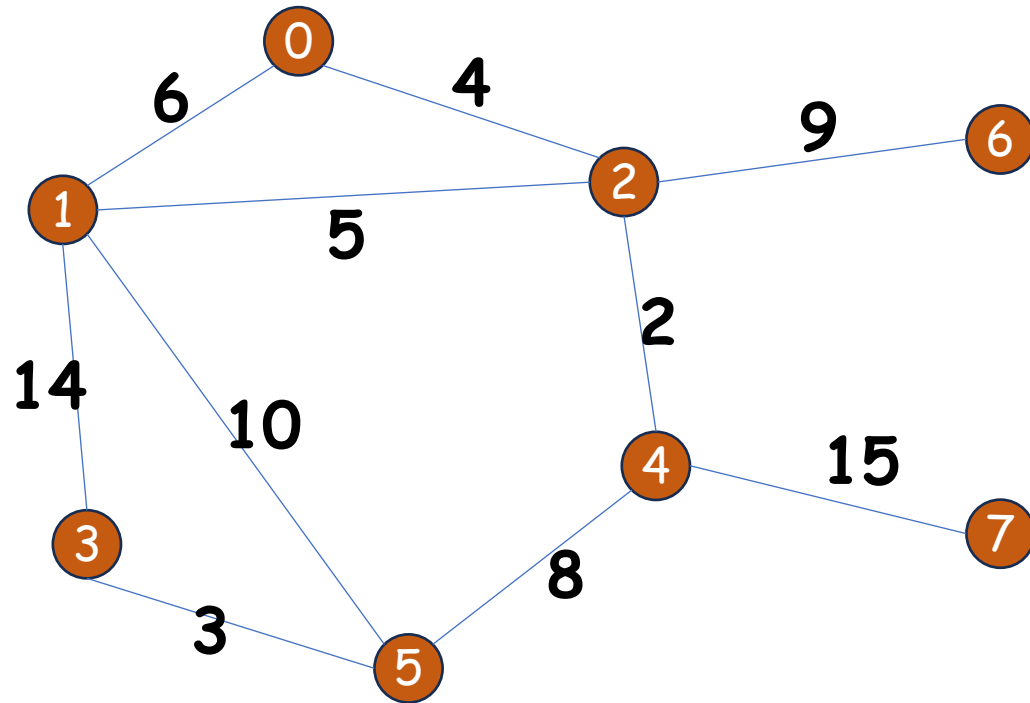


This is to determine whether two vertices  $u$  and  $v$  belong to the same tree. The operation FIND-SET( $u$ ) returns a representative element from the set that contains  $u$ .

# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

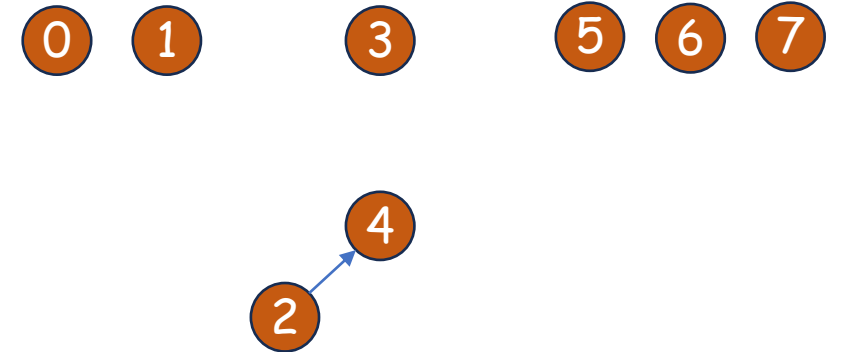
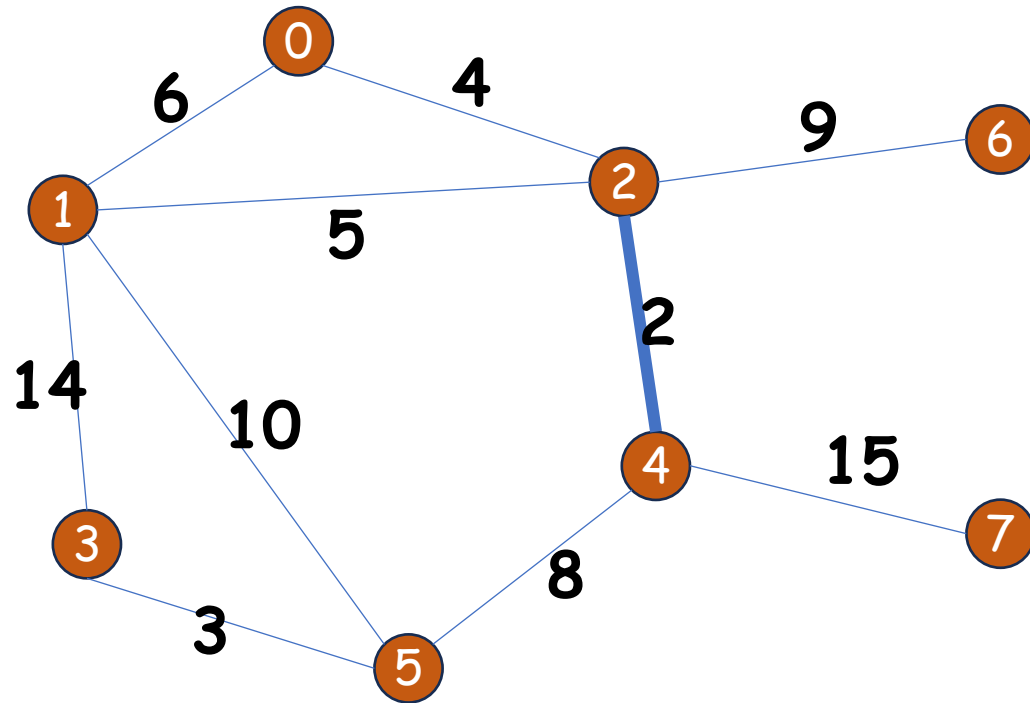
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

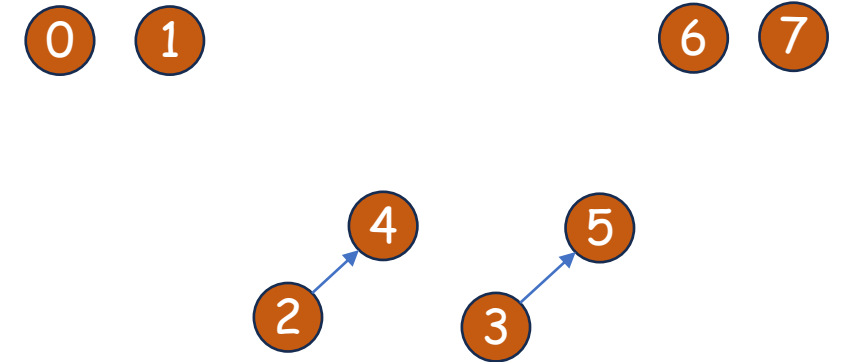
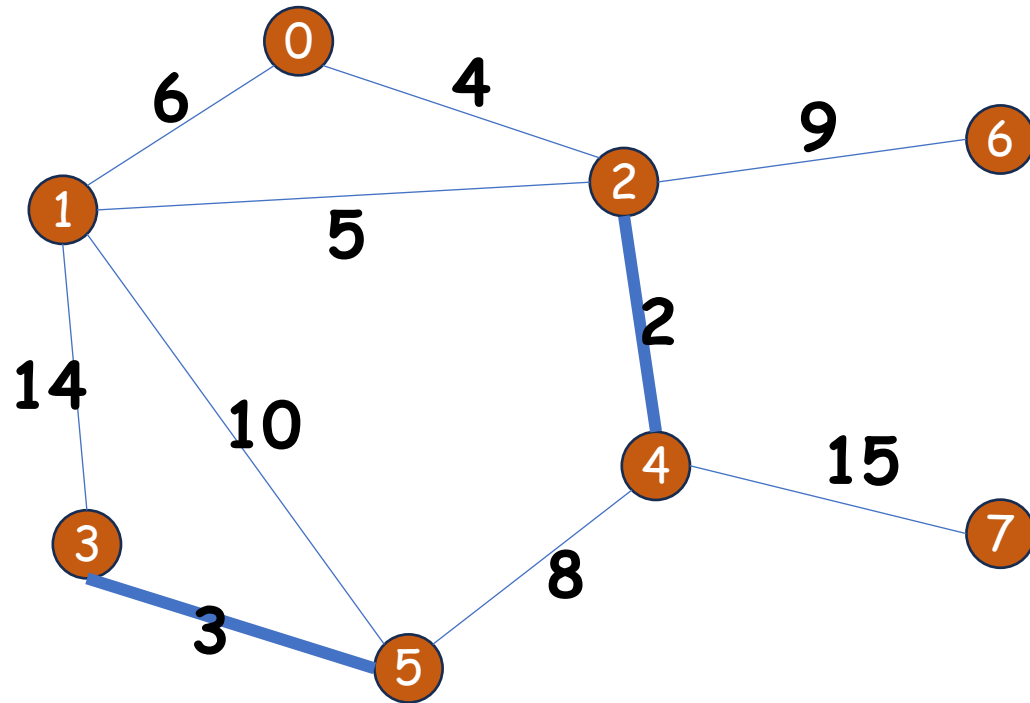
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

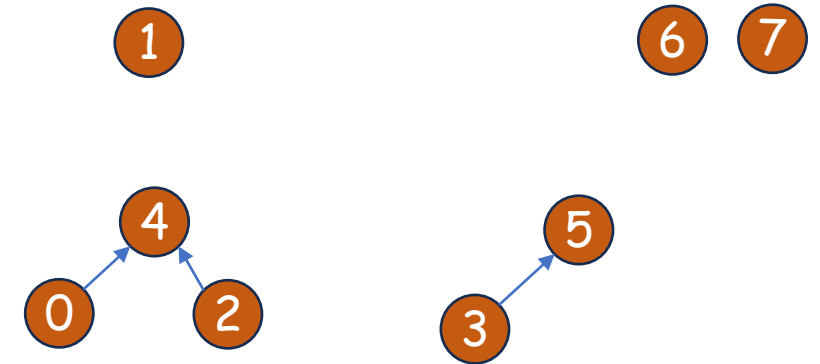
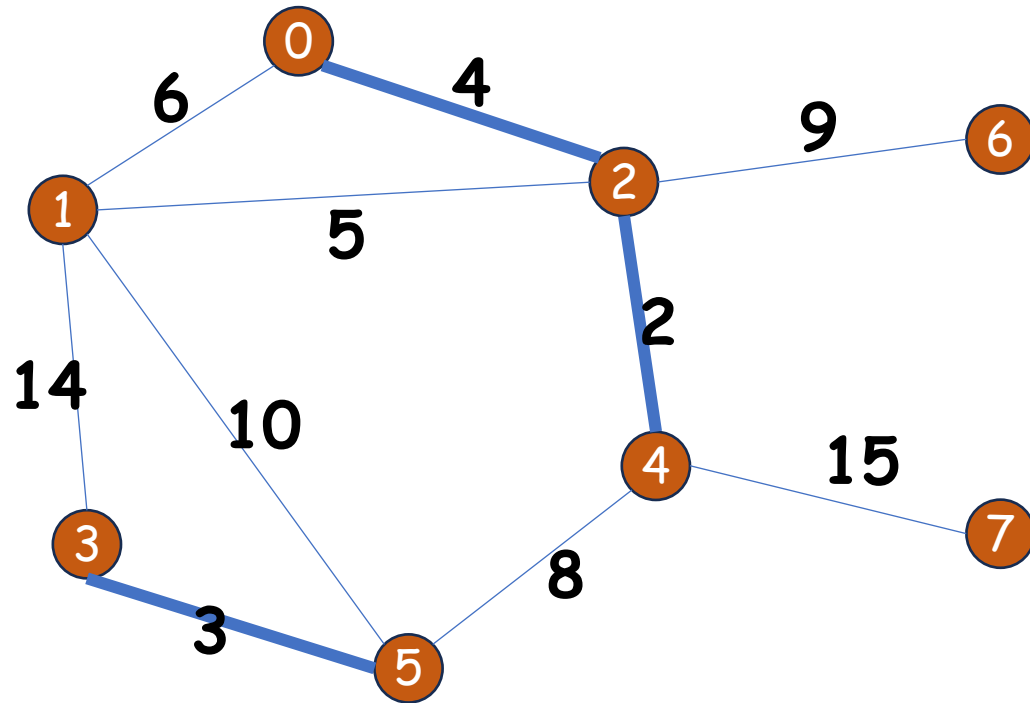
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

Disjoint Sets:

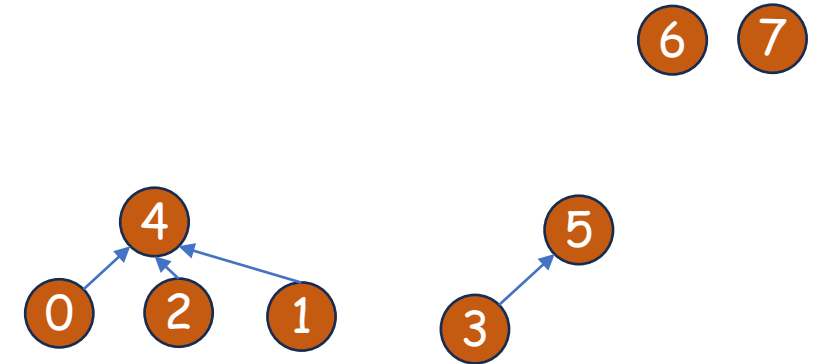
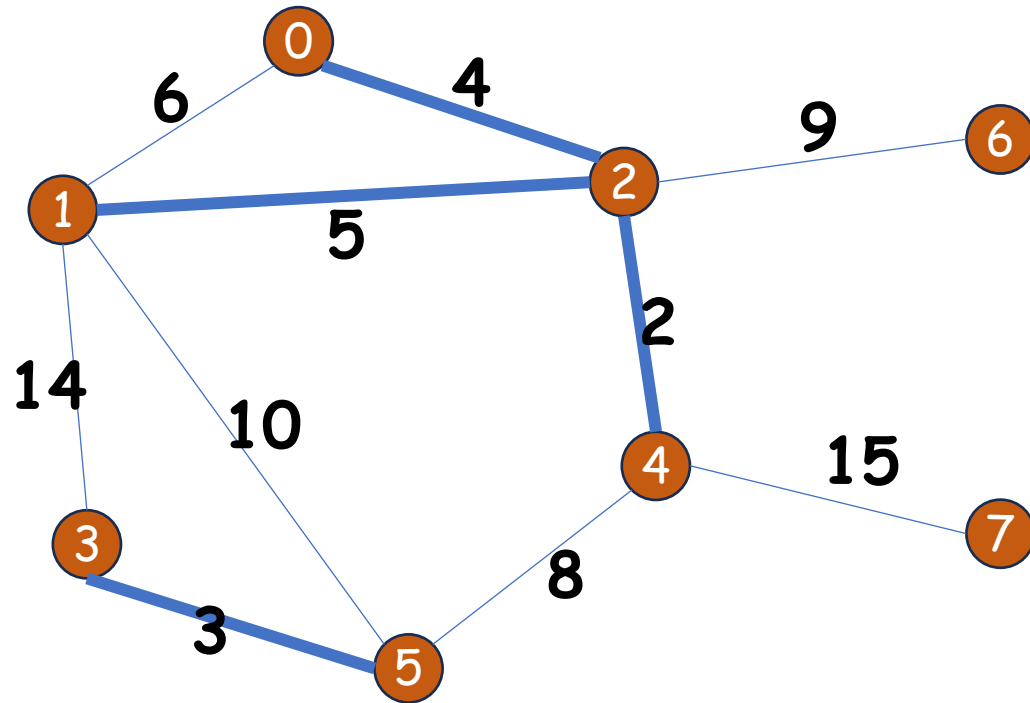




# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

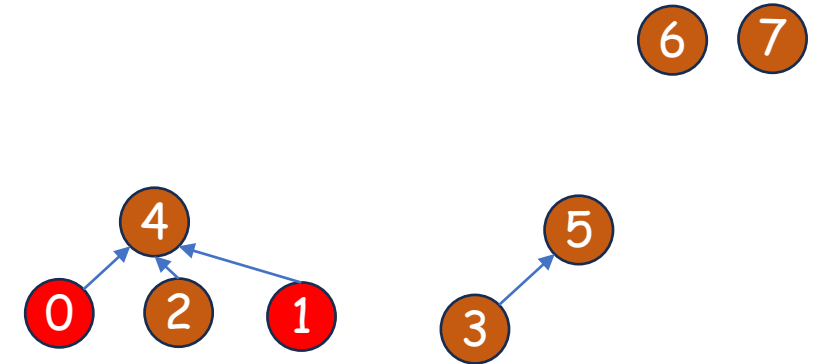
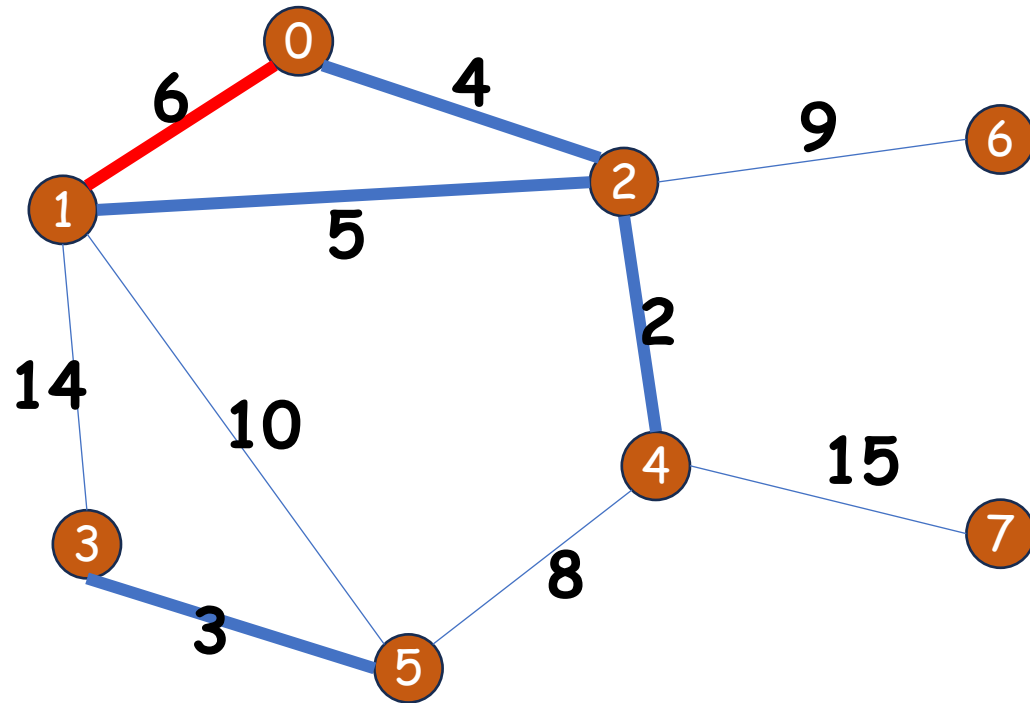
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

Disjoint Sets:

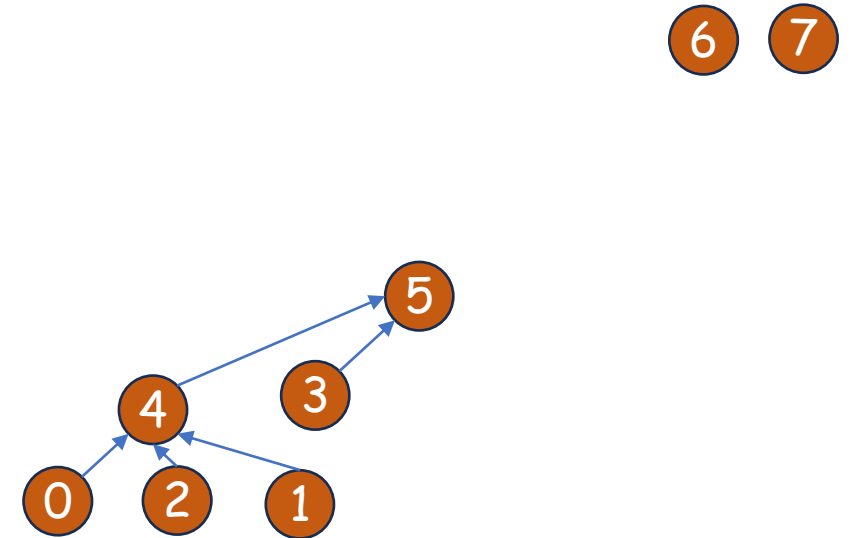
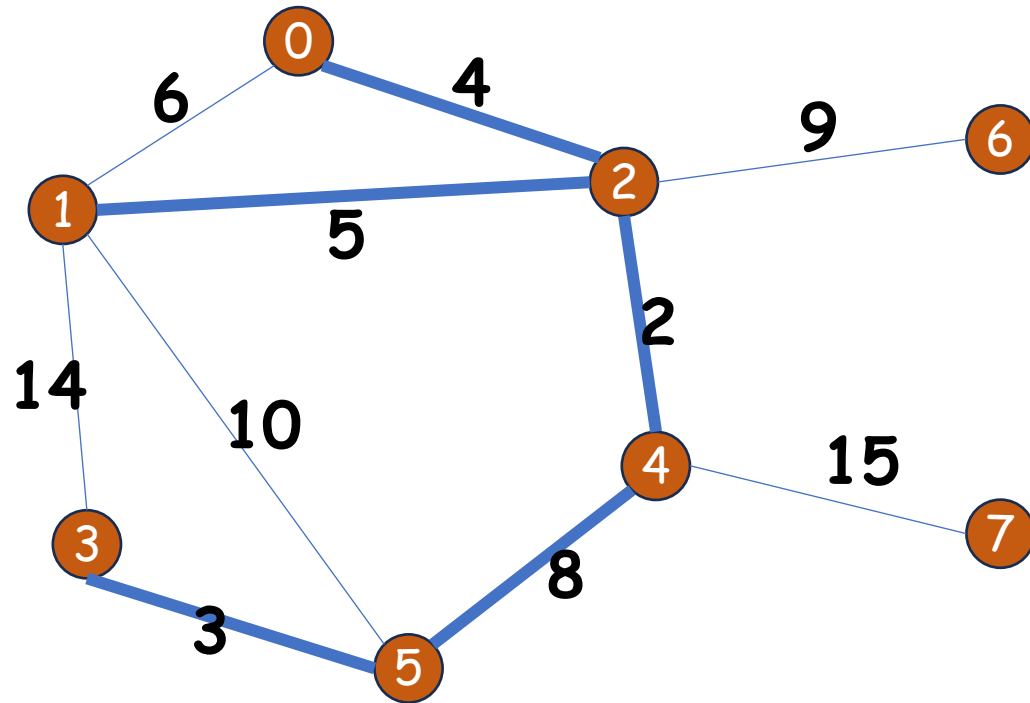


**No!** 0 and 1 are from the same set!

# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

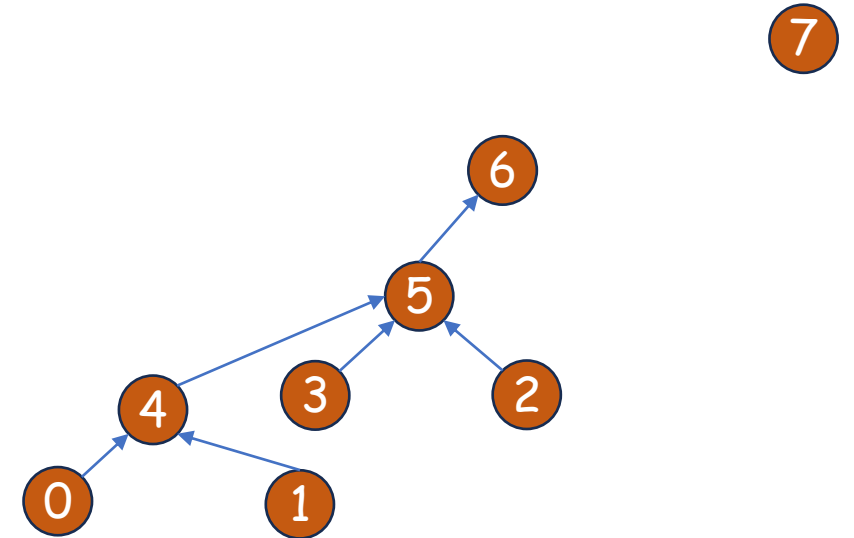
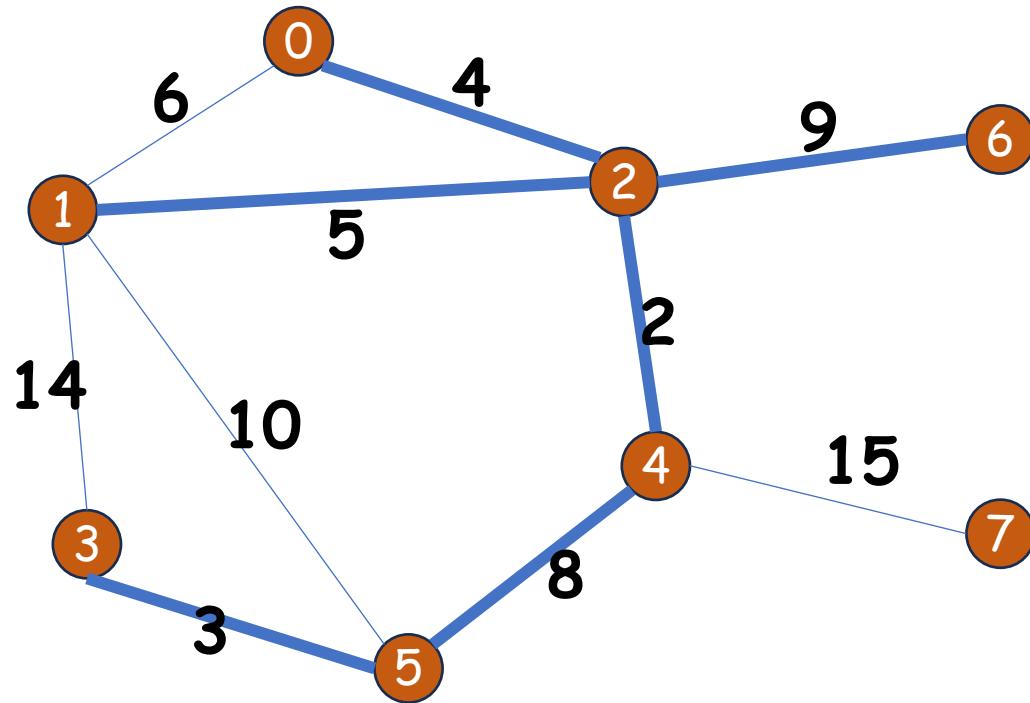
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

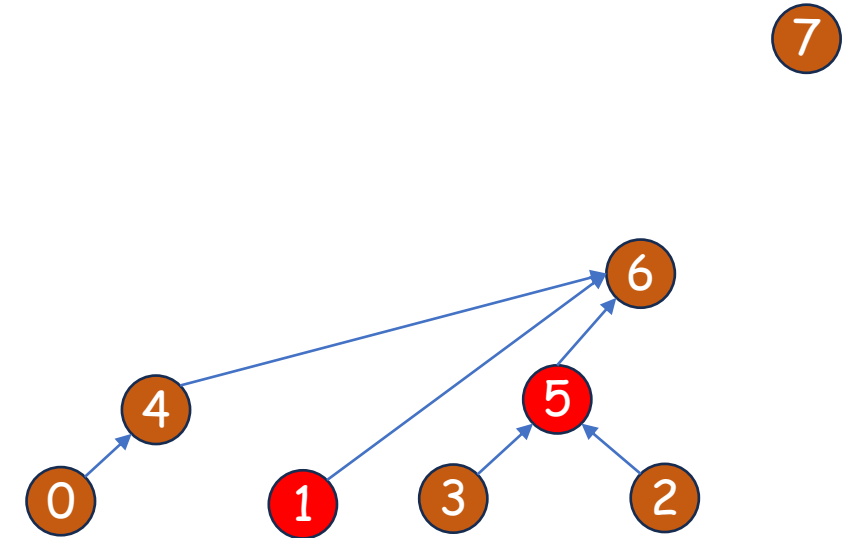
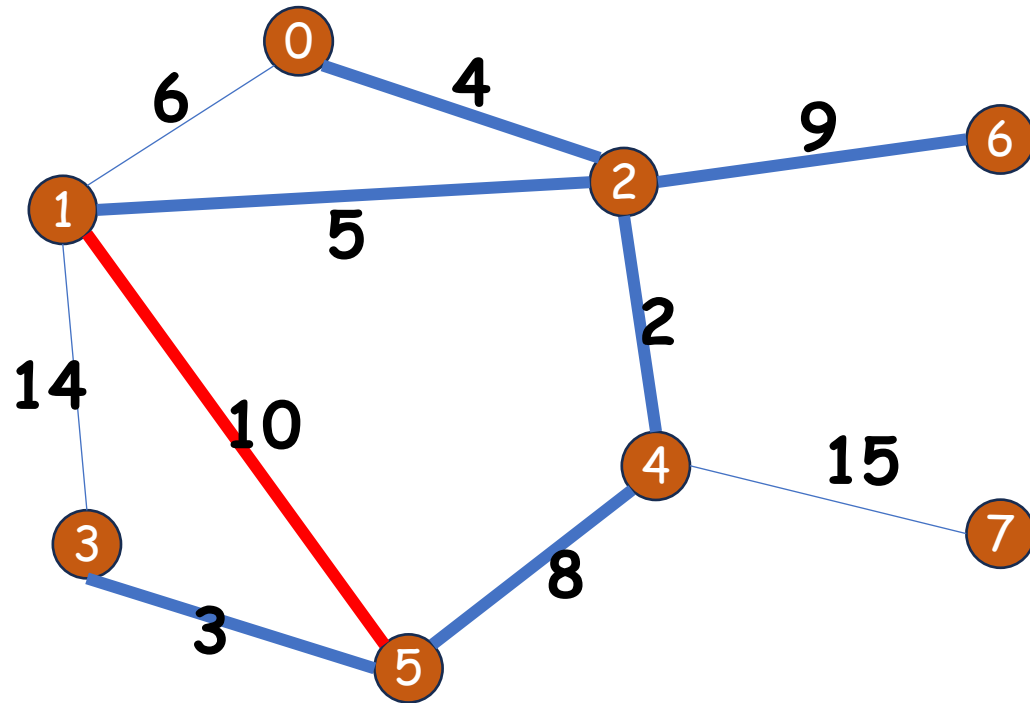
Disjoint Sets:



# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

Disjoint Sets:

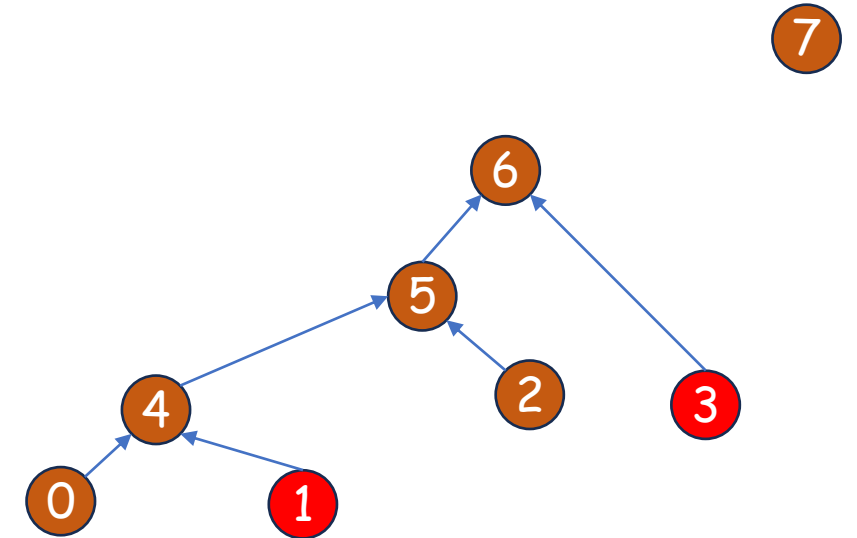
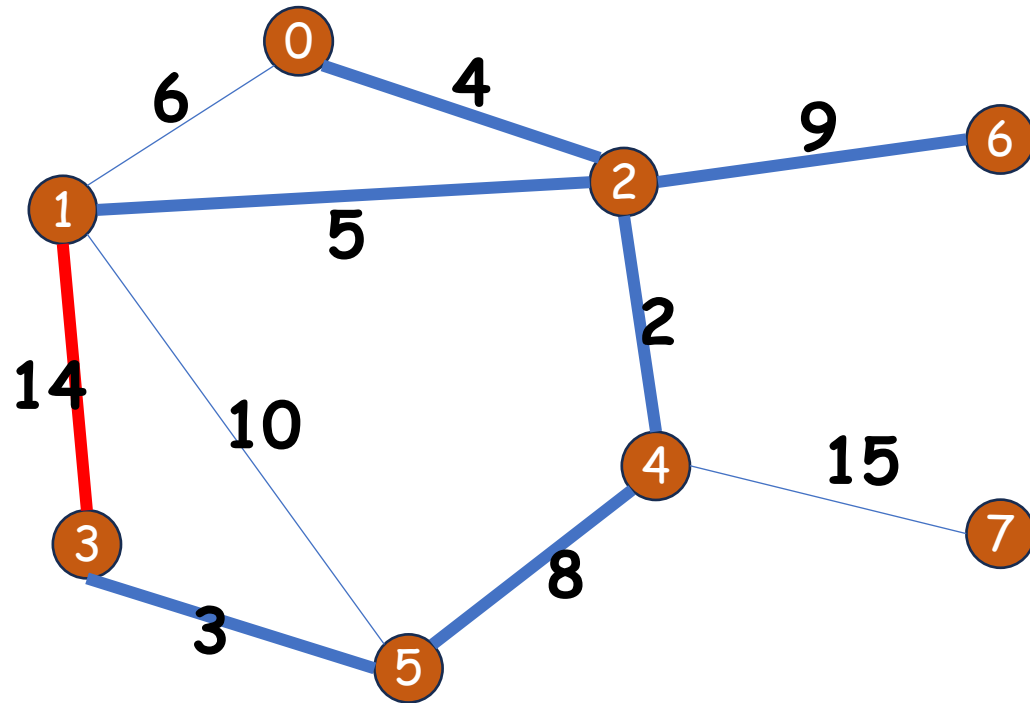


**No!** 1 and 5 are from the same set!

# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

Disjoint Sets:



**No!** 1 and 3 are from the same set!

# Minimum Spanning Trees

\* Path compression + union-by-rank  
Rank = # of nodes

Disjoint Sets:

