# CS 170
# Efficient Algorithms and Intractable Problems

## Lecture 10 (updated)
### Huffman Codes and Minimum Spanning Trees

Nika Haghtalab and John Wright

EECS, UC Berkeley

# Announcements

Midterm 1 next week 10/3 (be on the lookout for the Midterm Logistics post)

→ Feel free to post about past exams (linked under the Ed central index post), we will also set up past exam mega threads

→ Scope: Including today's material!

→ Midterm 1 Review Sessions: 11 - 2 Saturday, Sunday @ Woz - Soda 411

HW 5 is optional. Posted with solutions, so review the solutions!

→ Feel free to ask exam questions in OH/HWP

HW 2 grades released, regrades due Wed 9/27

Nika's OH combined with Tuesday's after QA.

→ 2-3 in Cory Courtyard. We'll walk together to Cory.

# Last Lecture and Today: Greedy Algorithms

Algorithms that build up a solution

piece by piece, always choosing the next piece

that offers the most obvious and immediate benefit!

We saw:
- Scheduling
- Satisfiability
- Started on optimal coding

Today:
- More on optimal coding
- Minimum Spanning Trees (1 alg next time)

# Recap: A Pattern in Greedy Algorithm and Analyses

Greedy makes a series of choices. We show that no choice rules out the optimal solution. How?

Inductive Hypothesis:

→The first $m$ choices of greedy match the first $m$ steps of some optimal solution.

→Or, after greedy makes $m$ choices, achieving optimal solution is still a possibility.

Base case: → At the beginning, achieving optimal is still possible!

Inductive step: **Use problem-specific structure**

If the first $m$ choices match, we can change OPT's $m + 1^{st}$ choice to that of greedy's, and still have a valid solution that no worst than OPT.

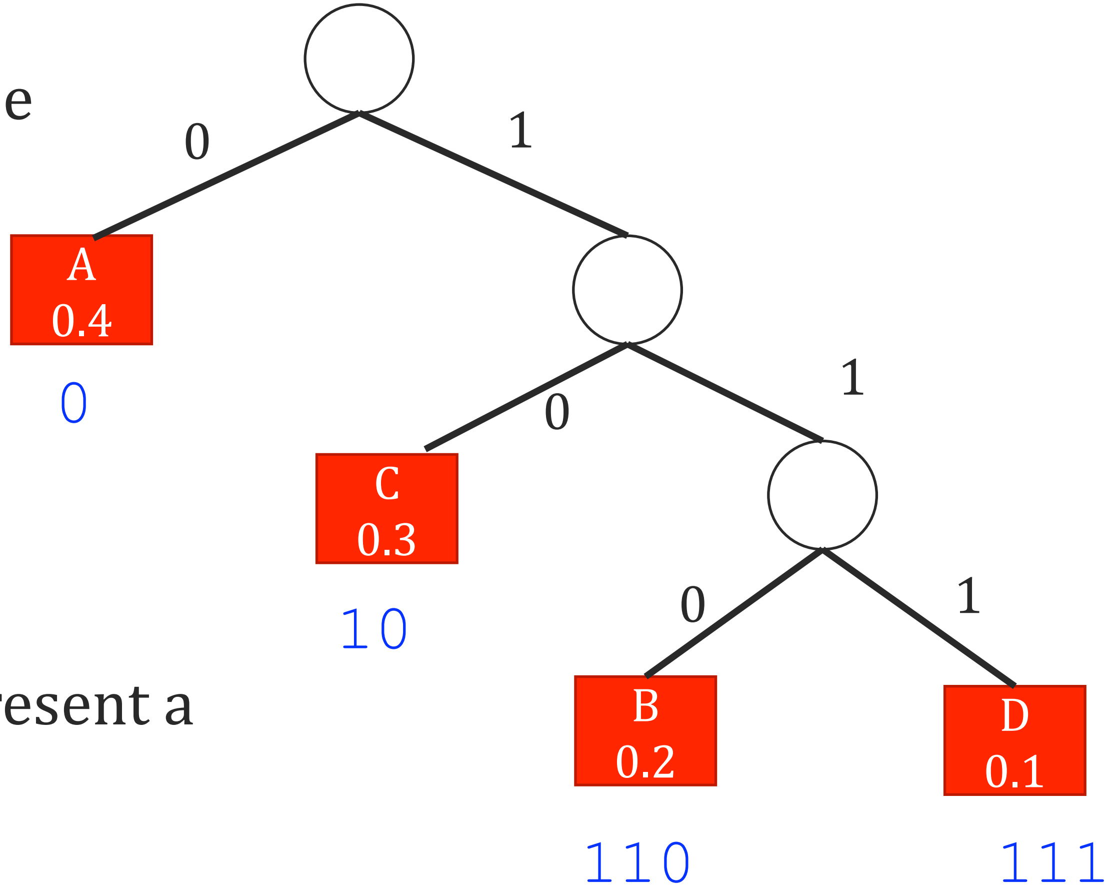
**Conclusion:** The greedy algorithm outputs an optimal solution.

# Recap: Prefix codes and Trees


means "A" has freq. 0.4.

Any prefix-free code can be represented as a binary tree with $k$ leaves.

- Leaves indicate the coded letter
- The code is the "address" of a letter in the tree




Any tree with the letters at the leaves, also represent a prefix-free code.

# Recap: Tree and Code Size

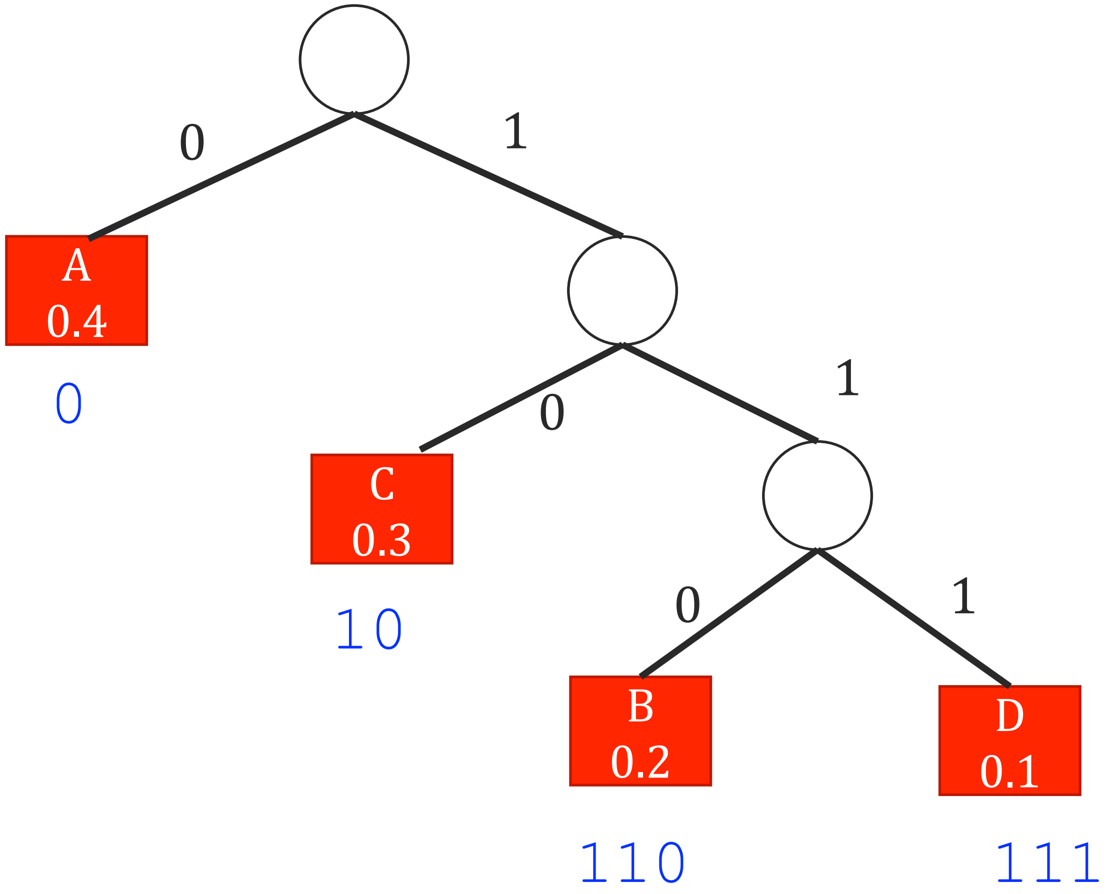
A
0.4

means “A” has freq. 0.4.

Imagine we are encoding a length N text:
→ that is written in $n$ letters with frequencies $f_1, f_2, \ldots, f_n$.

How long is the encoded message?

$$\text{length of encoding} = \sum_{i=1}^{n} N \cdot f_i \cdot \text{len}(\textit{encoding } i)$$




**Definition:** Cost of a prefix-code/tree is

$$\text{Cost(tree)} = \sum_{i=1}^{n} f_i \cdot \text{depth}(\textit{leaf } i)$$

# Recap: Optimal Prefix-free Codes

**Input:** $n$ symbols with frequencies $f_1, \dots, f_n$

**Output:** A tree (prefix-free code) encoding.

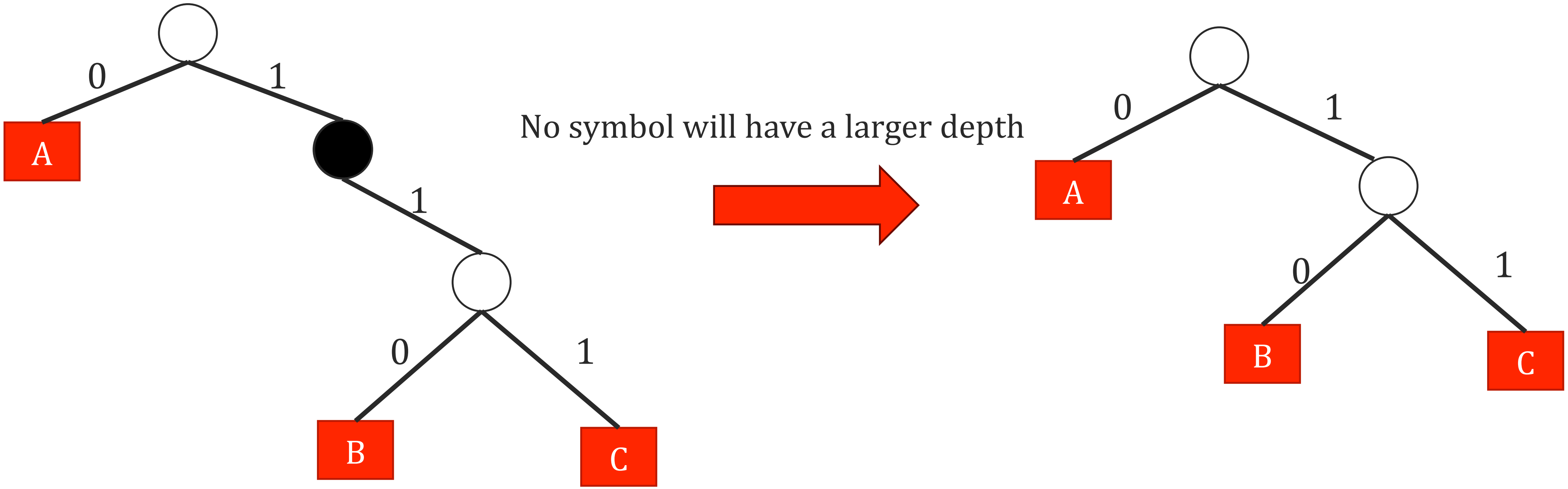
**Goal:** We want to output the tree/code with the smallest cost

Smallest Cost

$$\boxed{\text{Cost(tree)} = \sum_{i=1}^{n} f_i \cdot \text{depth}(leaf\ i)}$$

# What do optimal subtrees look like?

Even without looking at the frequencies, could this tree be optimal?



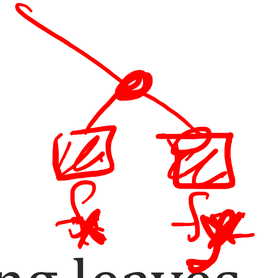

**Claim:** There is a "full binary tree" that is an optimal coding.

Proof: we just argued above!

Means that every non-leaf node has two children.

# What do optimal subtrees look like?

$f_1 \le f_2 \le \cdots f_n$



**Claim:** There is an optimal tree where the two lowest freq. symbols are sibling leaves.

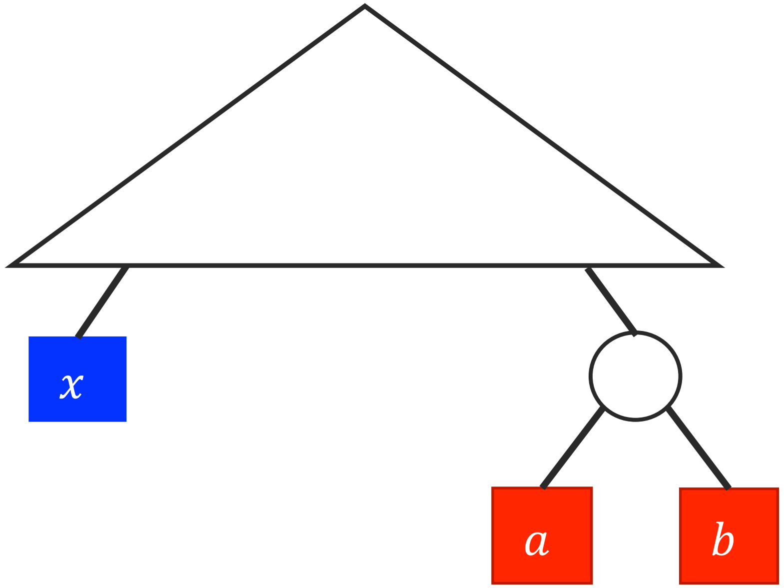
**Proof:** By contradiction. Let $x, y$ be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols $a, b$ be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of $a, b$ is neither $x$ or $y$. Let's say $x \neq a$.

What happens if we swap $x$ and $a$?



# What do optimal subtrees look like?

**Claim:** There is an optimal tree where the two lowest freq. symbols are sibling leaves.

**Proof:** By contradiction. Let $x, y$ be symbols with lowest frequencies and assume they aren't siblings.

$\sum_i f_i \cdot dep(node\ i)$

$f_x depth(x) + f_a depth(a)$

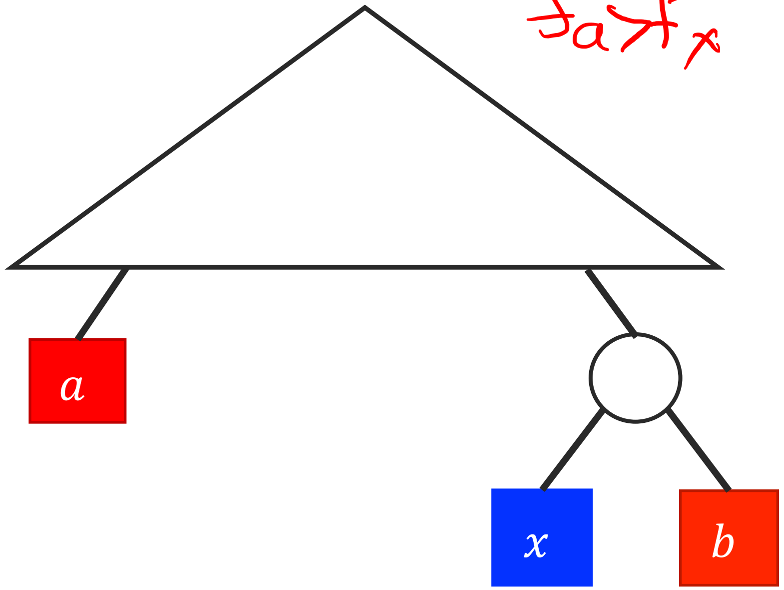
$f_a > f_x$

- Let symbols $a, b$ be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of $a, b$ is neither $x$ or $y$. Let's say $x \neq a$.

What happens if we swap $x$ and $a$?
→ **The cost of tree can't increase,** because $f_a \geq f_x$ and we just switch the length of $a$'s code and $x$'s code.



# What do optimal subtrees look like?

**Claim:** There is an optimal tree where the two lowest freq. symbols are sibling leaves.

**Proof:** By contradiction. Let $x, y$ be symbols with lowest frequencies and assume they aren't siblings.

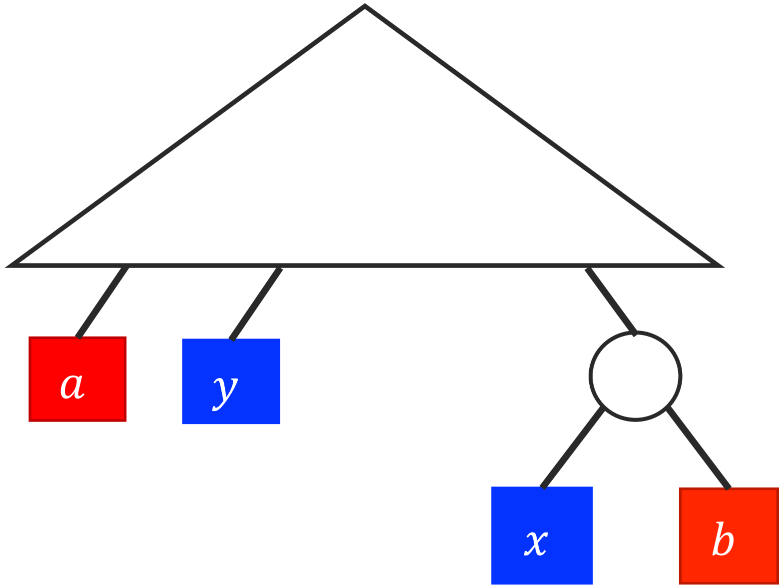
- Let symbols $a, b$ be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of $a, b$ is neither $x$ or $y$. Let's say $x \neq a$.

What happens if we swap $x$ and $a$?
→ **The cost of tree can't increase,** because $f_a \geq f_x$
and we just switch the length of $a$'s code and $x$ 's code.

Repeat this swap and logic if $y \neq b$ either.



# What do optimal subtrees look like?

**Claim:** There is an optimal tree where the two lowest freq. symbols are sibling leaves.

**Proof:** By contradiction. Let $x, y$ be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols $a, b$ be the deepest pair of siblings.

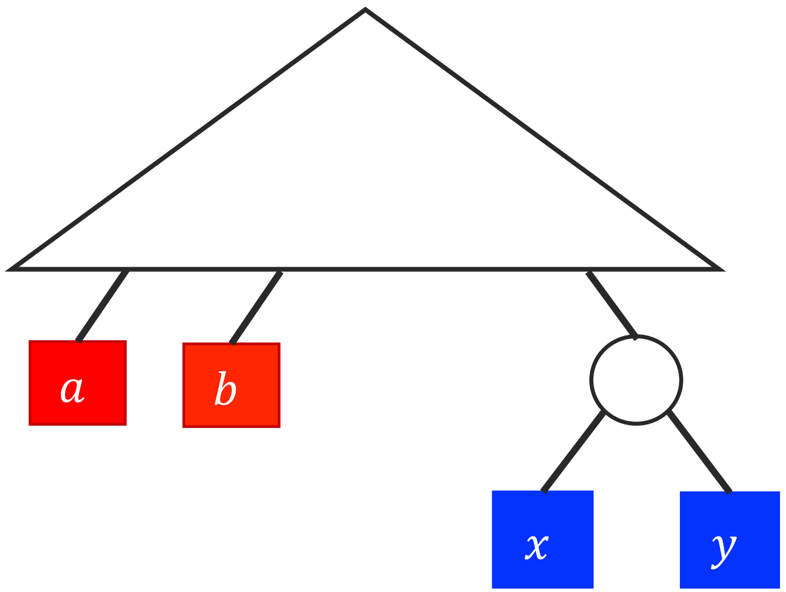
→A lowest sibling pair exists because we have a full binary tree.

→At least one of $a, b$ is neither $x$ or $y$. Let's say $x \neq a$.

What happens if we swap $x$ and $a$?
→ **The cost of tree can't increase,** because $f_a \geq f_x$ and we just switch the length of $a$'s code and $x$ 's code.

Repeat this swap and logic if $y \neq b$ either.

We found a cheaper tree, where $x, y$ are siblings!



# Greedy algorithm

**Idea:** Since the lowest frequency letters are sibling leaves in some optimal tree, we will greedily build subtrees from the lowest frequency letters.

This is called Huffman Coding.

Node $a$ object with

a.freq = $f_a$

a.left = left child

a.right = right child

Huffman-code($f_1, \dots, f_n$)

    For all $a = 1, \dots, n$,

        create node $a$ with $a.\text{freq} = f_a$ and no children

        Insert the node in a priority queue $Q$ use key $f_a$

    While $\text{len}(Q) > 1$

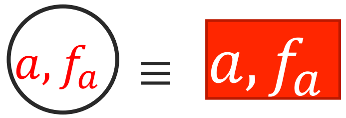
        $x$ and $y \leftarrow$ the nodes in $Q$ with lowest keys

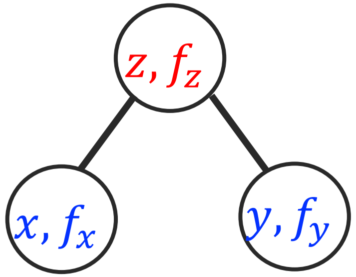
        create a node $z$, with $z.\text{freq} = x.\text{freq} + y.\text{freq}$

        Let $z.left = x$ and $z.right = y$.

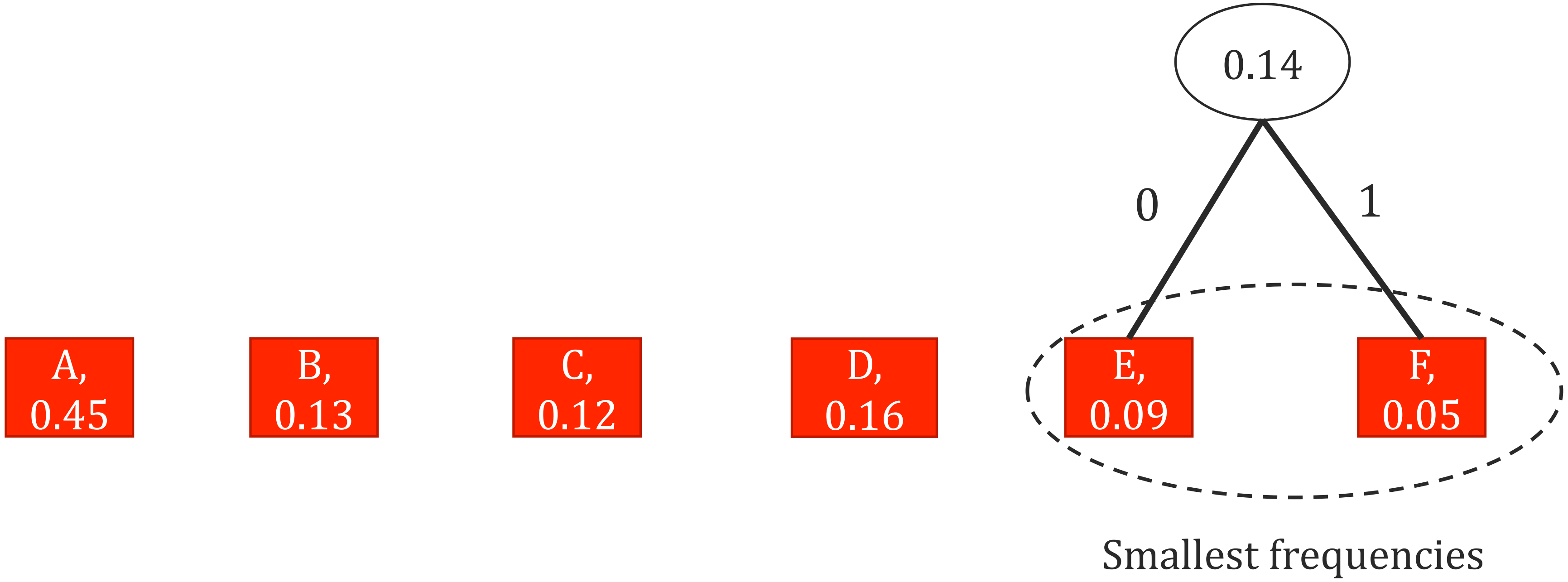
        Insert $z$ with key $f_z$ into $Q$ and remove $x, y$.

    Return the only node left in $Q$.

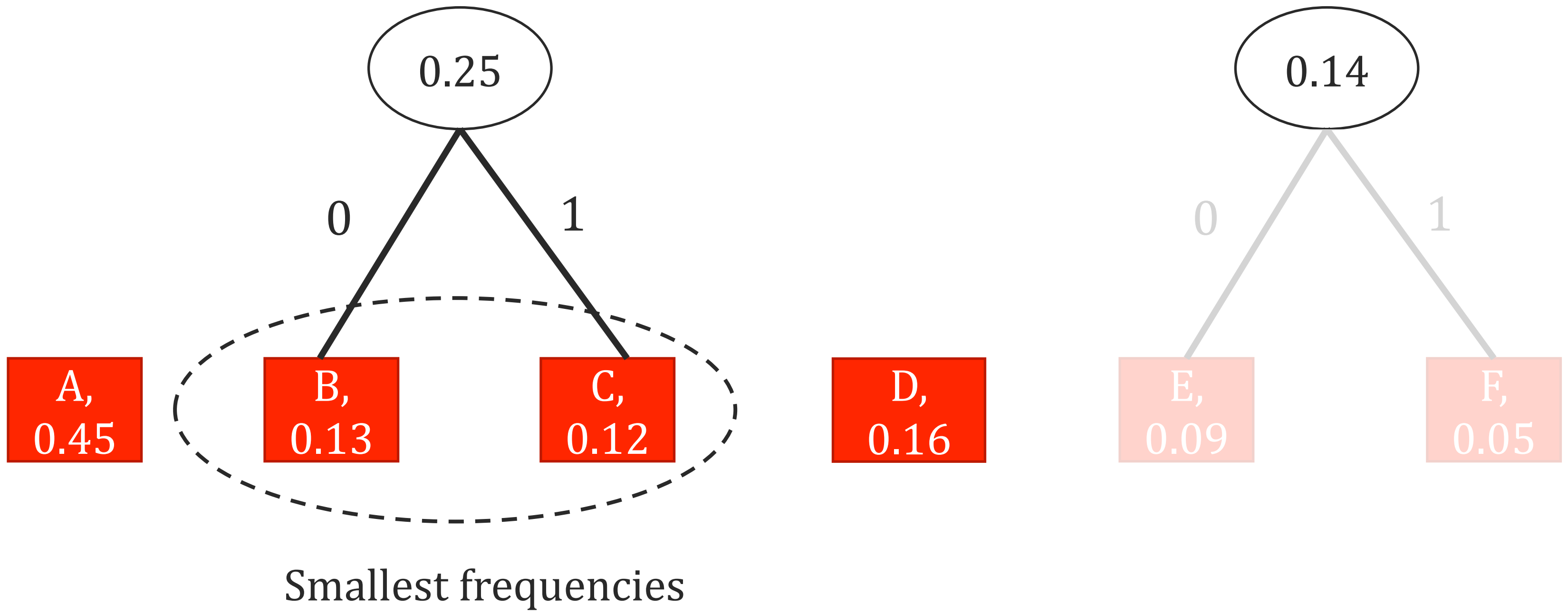


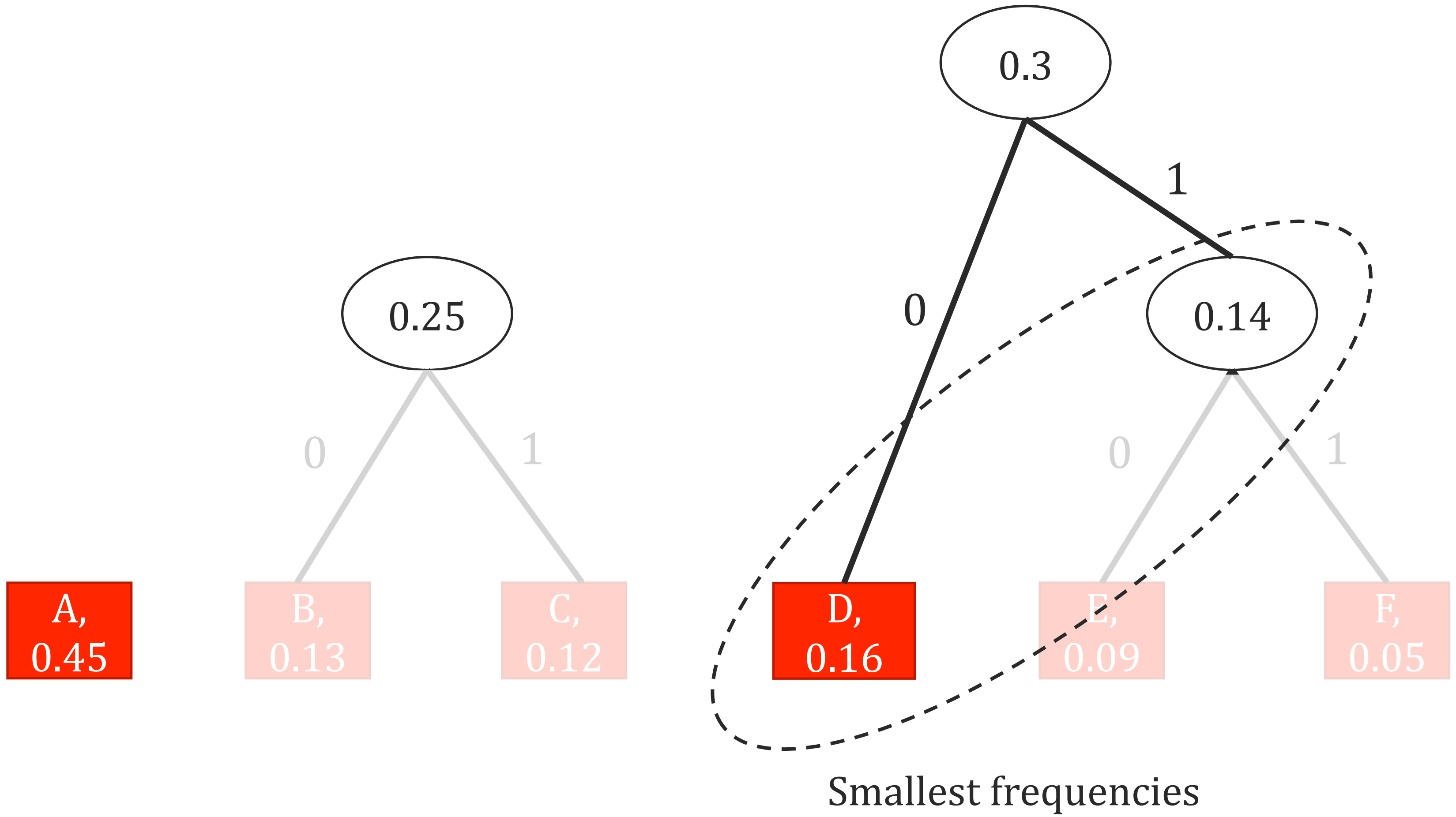
# Example of Huffman Code



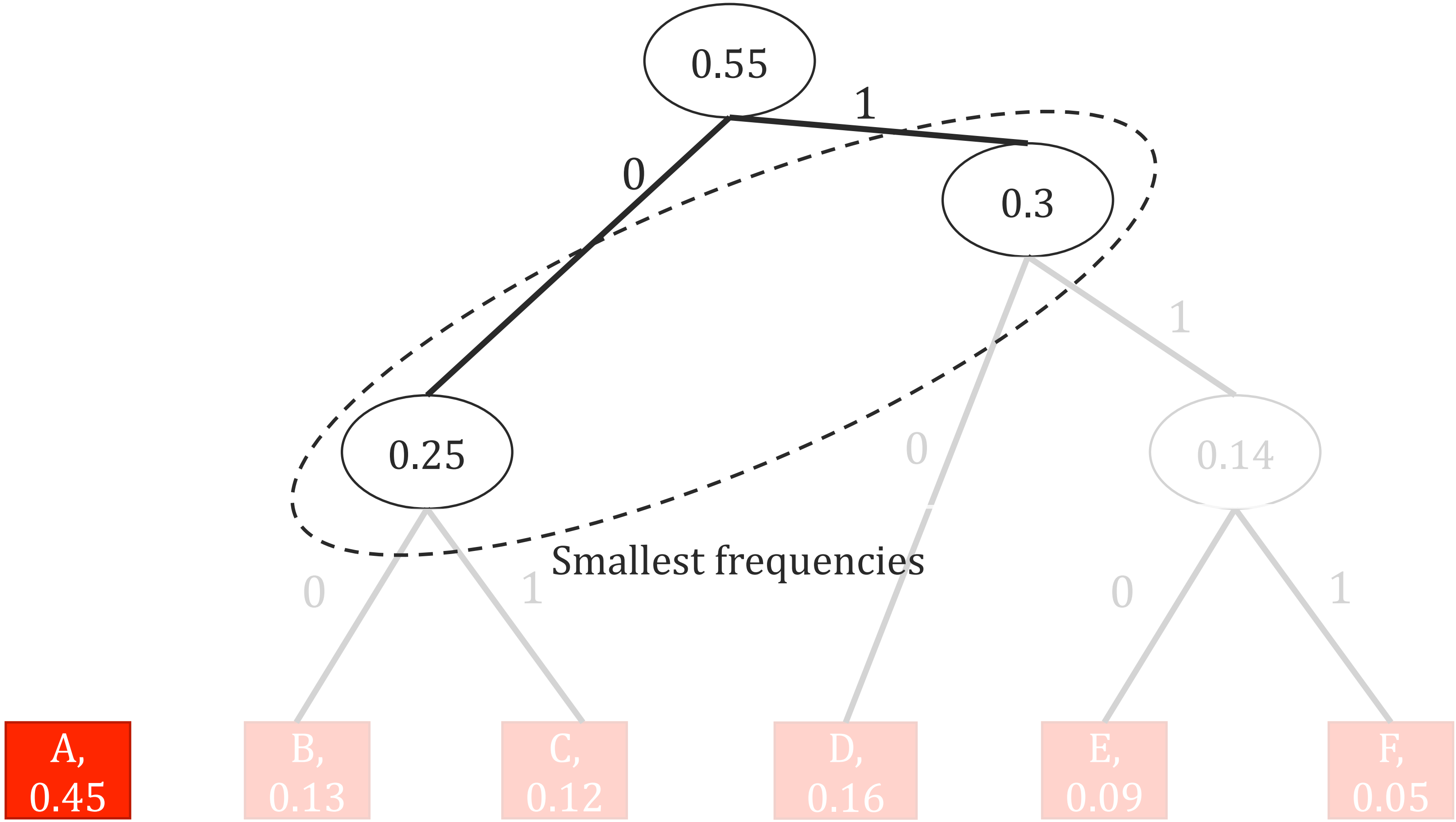
# Example of Huffman Code



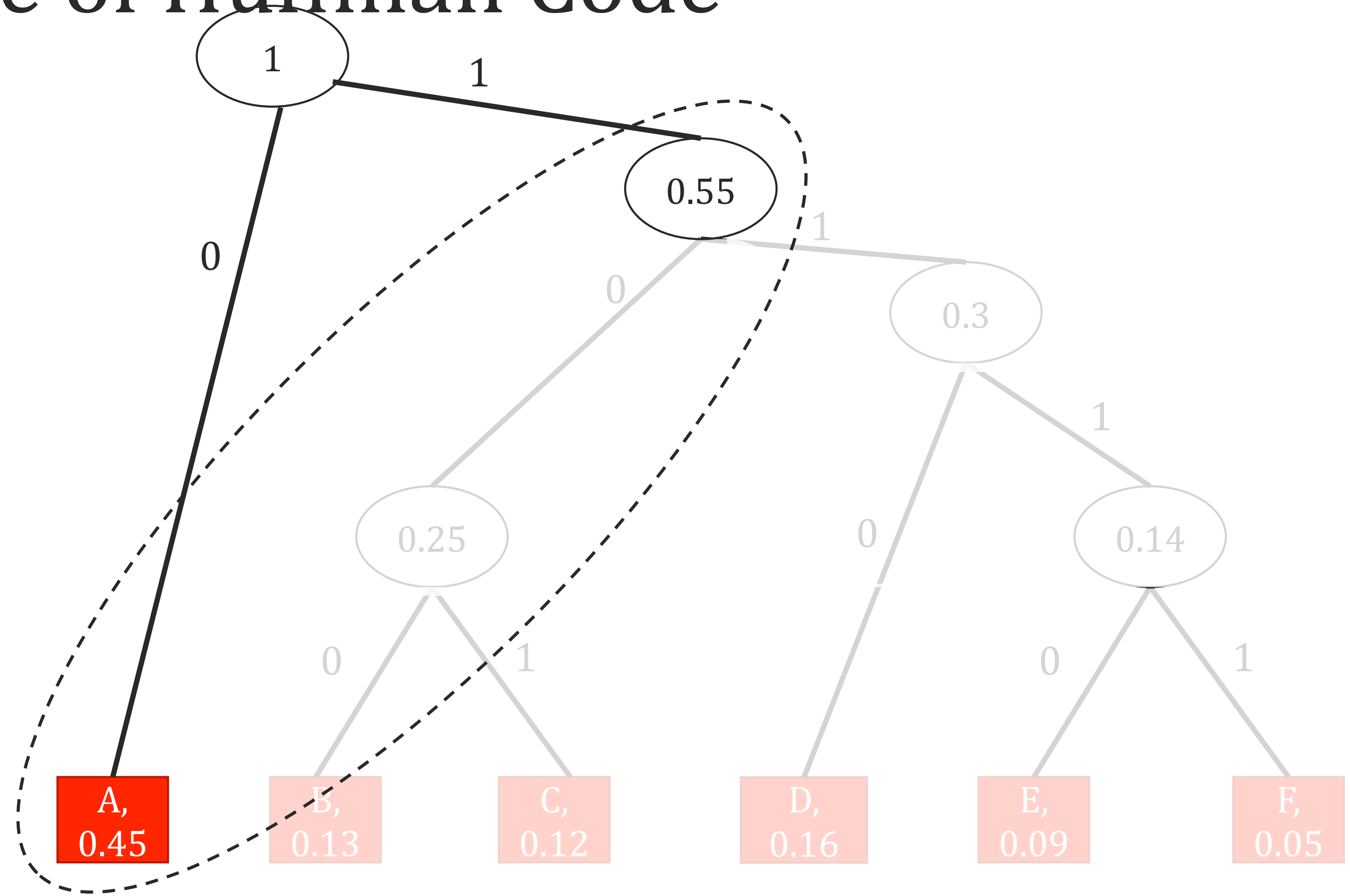
# Example of Huffman Code



# Example of Huffman Code

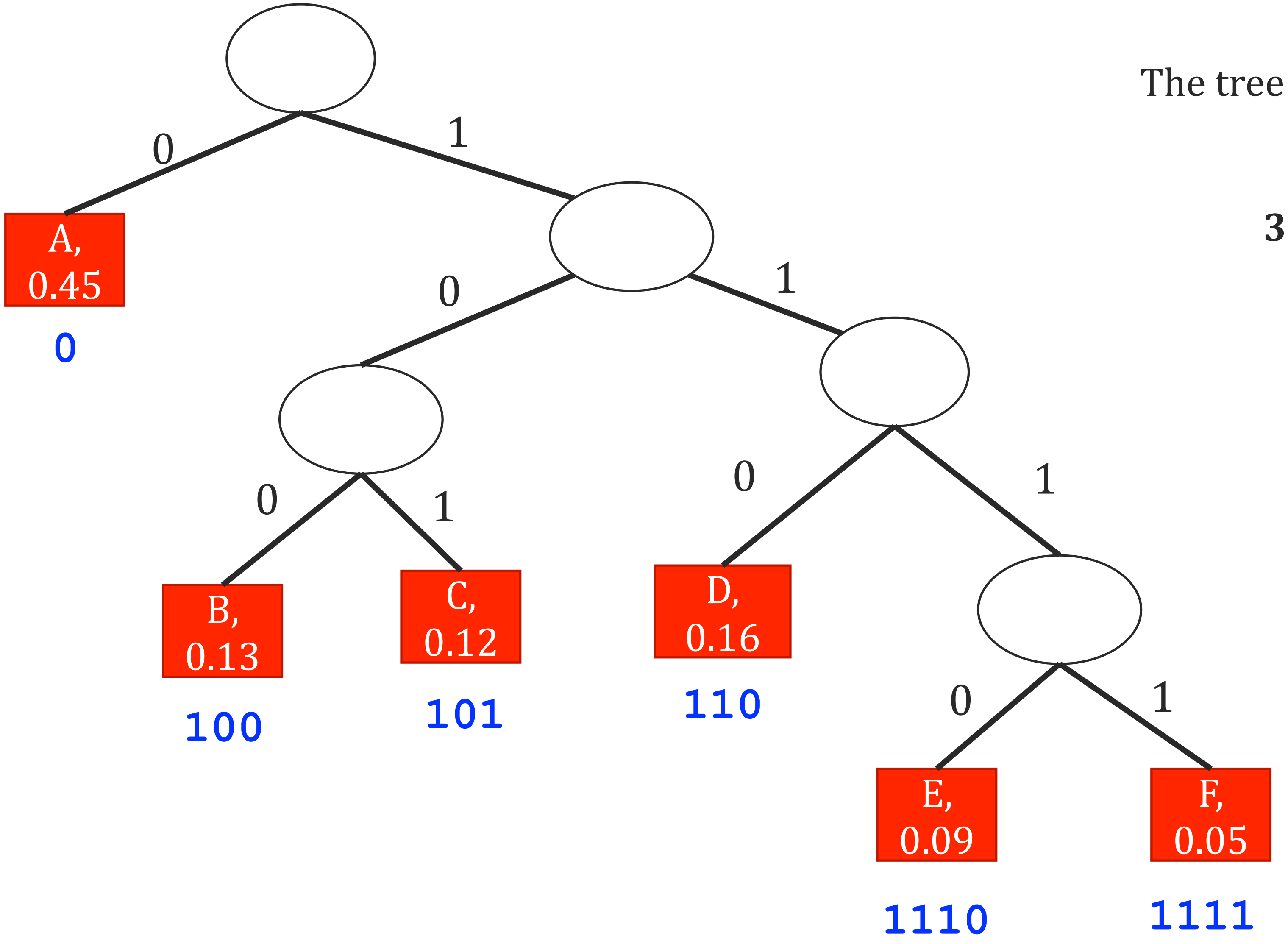


# Example of Huffman Code




Smallest frequencies

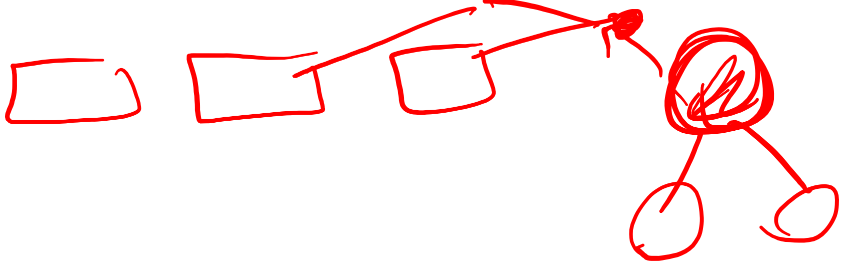
# The corresponding code




The tree cost:

$$1 \cdot 0.45 + 3 \cdot (0.13 + 0.12 + 0.16) + 4 \cdot (0.09 + 0.05) = 2.24$$

# Runtime of Huffman Coding



Priority queue operation (Lec. 8): Binary heap takes $O(\log(n))$ to Insert and DeleteMin.

Huffman-code$(f_1, \dots, f_n)$

$n$ Inserts = $O(n\log(n))$ $\longrightarrow$ For all $a = 1, \dots, n$,

create node $a$ with $a.\mathrm{freq} = f_a$ and no children

Insert the node in a priority queue $Q$ use key $f_a$

$n$ iterations, total of $O(n\log(n))$

While $\mathrm{len}(Q) > 1$

$x$ and $y \leftarrow$ the nodes in $Q$ with lowest keys $\longleftarrow$ 2 DeleteMin

create a node $z$, with $z.\mathrm{freq} = x.\mathrm{freq} + y.\mathrm{freq}$

Let $z.left = x$ and $z.right = y$.

Insert $z$ with key $f_z$ into $Q$ and remove $x, y$. $\longleftarrow$ 1 Insert

Return the only node left in $Q$.

Total runtime of Huffman coding: $O(n\log(n))$

# Optimality of Huffman Coding

**Claim:** Huffman coding is an optimal prefix-free tree.

Recall we use induction to show that greedy choices don't rule out optimality.

We use induction on the number of letters $n$.

**Base case:** $n = 2$. The optimal code is to assign one letter to 0 and the other 1.
Huffman does the same.

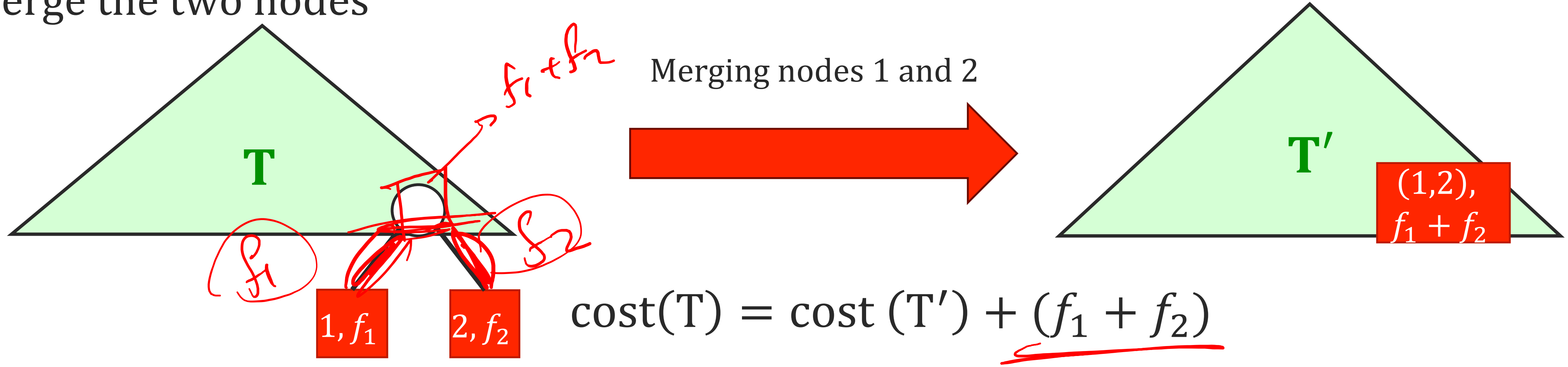
**Induction Hypothesis:** For $n - 1$ letters, Huffman coding is an optimal pre-fix tree.

# Optimality of Huffman Coding

**Claim:** Huffman coding is an optimal prefix-free tree.

**Induction step:** Let T below be the optimal prefix-free tree for frequencies $f_1, \ldots, f_n$ and WLOG $f_1 \le f_2 \le \cdots \le f_n$ .

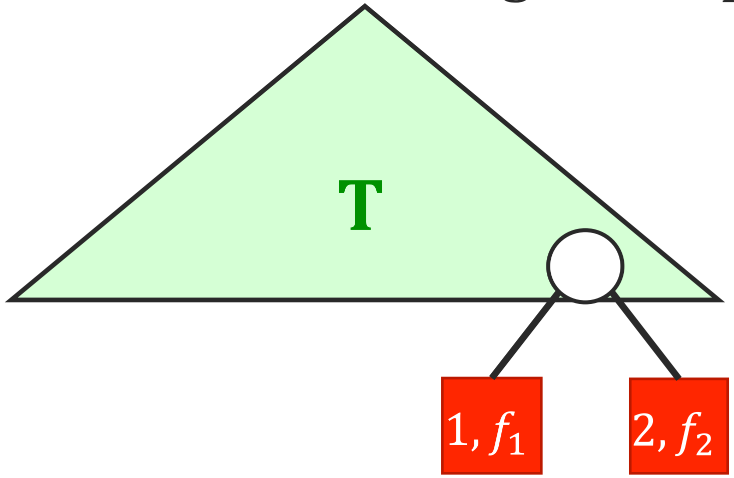
- WLOG, assume that the two lowest frequency nodes are siblings.
  → Because, we proved earlier that that's what optimal trees look like!
- Merge the two nodes



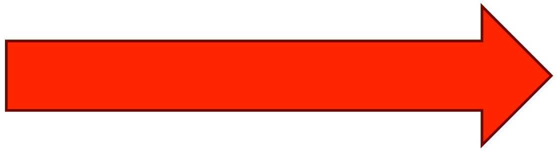

$$\text{cost}(\text{T}) = \text{cost}\ (\text{T}') + (f_1 + f_2)$$

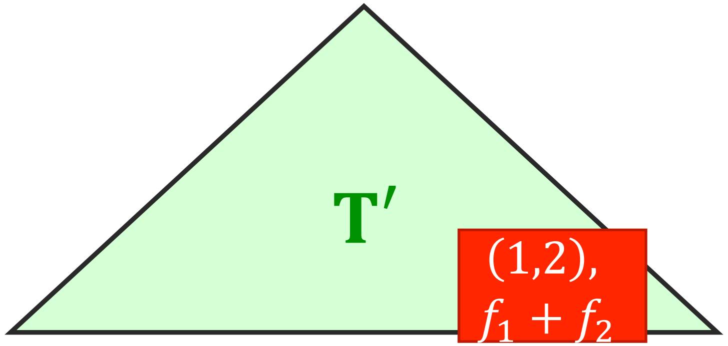
# Optimality of Huffman Coding

**Claim:** Huffman coding is an optimal prefix-free tree.



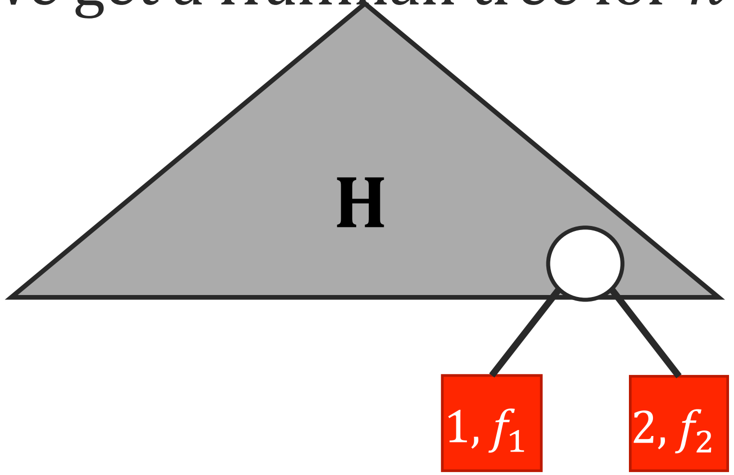

Merging nodes 1 and 2



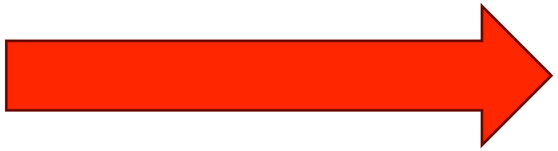


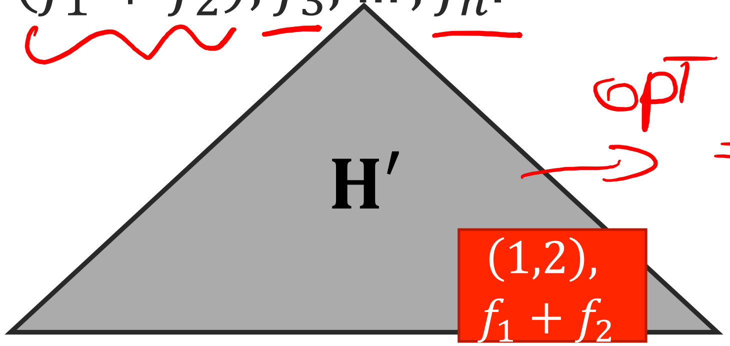

$$\text{cost(T)} = \text{cost (T}') + (f_1 + f_2)$$

By construction of Huffman **tree *H***, $f_1$ and $f_2$ are lowest siblings. Merge them here too.
→ We get a Huffman tree for $n-1$ letters and frequencies $(f_1+f_2), f_3, \ldots, f_n$.




Merging nodes 1 and 2






opT n-1

$$\text{cost(H)} = \text{cost (H}') + (f_1 + f_2)$$

# Optimality of Huffman Coding

**Claim:** Huffman coding is an optimal prefix-free tree.

We showed that for tree T that is optimal for $n$ letters, $\text{Cost}(\text{T}) = \text{cost}\,(\text{T}') + (f_1 + f_2)$.

And for Huffman coding tree H for $n$ letters, $\text{Cost}(\text{H}) = \text{cost}\,(\text{H}') + (f_1 + f_2)$.

Putting everything together.

$$\overbrace{\text{cost}\,(\text{H}') \leq \text{cost}(T')}^{n-1 \text{ letters}} \Rightarrow \overbrace{\text{cost}\,(H) \leq \text{cost}(T)}^{n \text{ letters}} \Rightarrow \text{Huffman is optimal for } n \text{ letters}$$

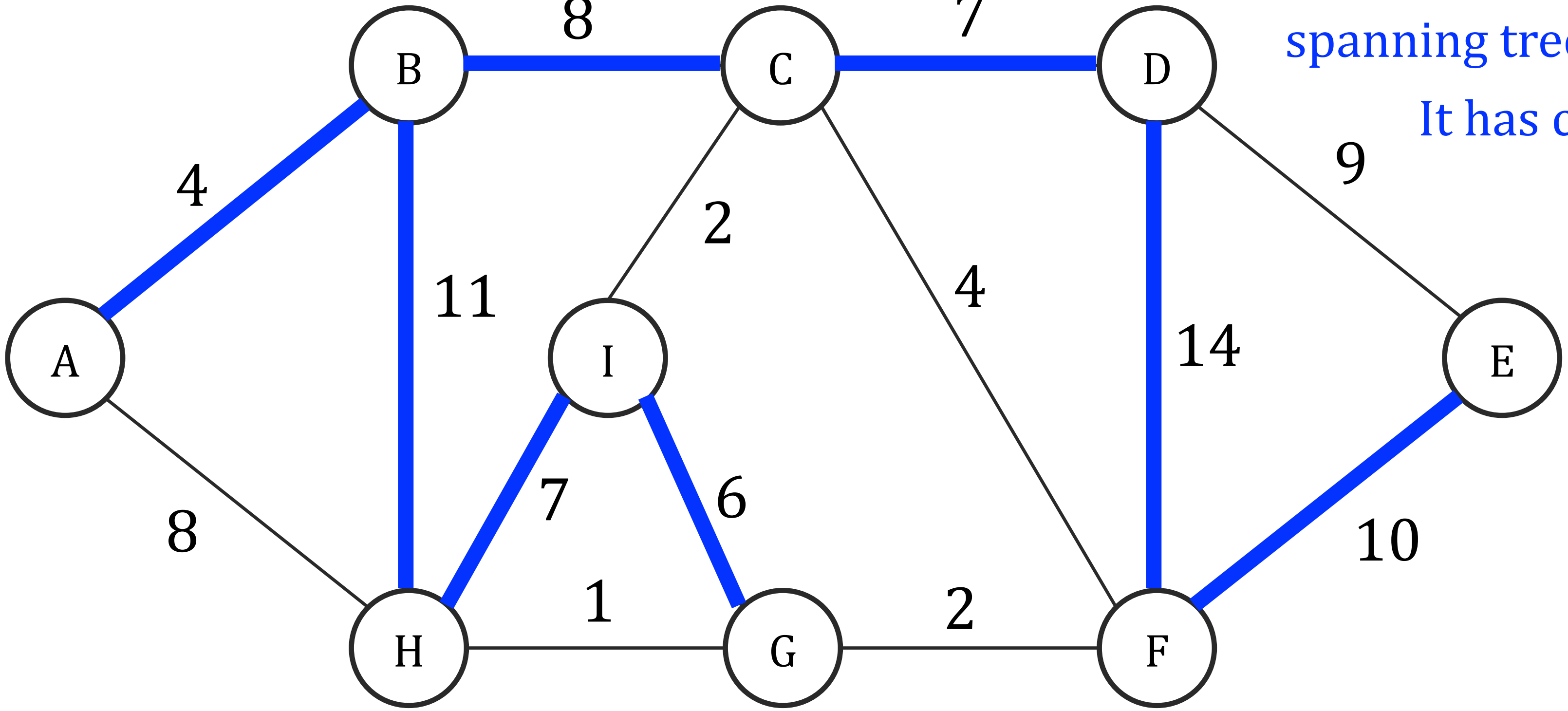
By induction hypothesis, Huffman coding for $n-1$ letters is optimal

# Minimum Spanning Trees

**Definition:** A spanning tree, is a tree that **connects all vertices** of a graph G.

**Cost of a tree**

$$\text{cost}(T) = \sum_{e \in T} w_e$$



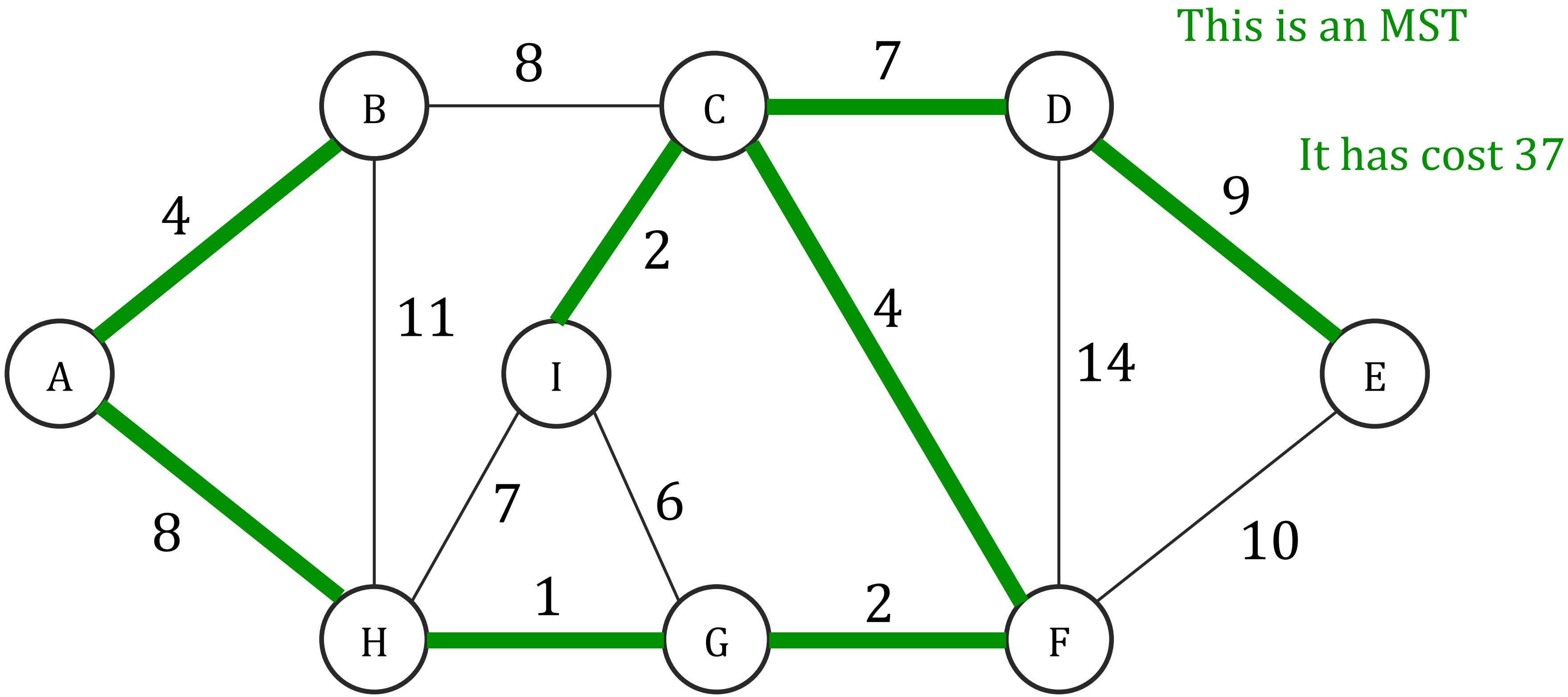

**Minimum Spanning Tree (MST) Problem:**
**Input:** a weighted graph $G = (V, E)$ with non-negative weights.
**Output:** A set of edges that connected graph and has the **smallest cost**.

# Minimum Spanning Trees

**Definition:** A spanning tree, is a tree that **connects all vertices** of a graph G.

**Cost of a tree**

$$\text{cost}(T) = \sum_{e \in T} w_e$$




**Minimum Spanning Tree (MST) Problem:**
**Input:** a weighted graph $G = (V, E)$ with non-negative weights.
**Output:** A set of edges that connected graph and has the **smallest cost**.

# MST applications and Algorithms

Biggest applications:

- Network design: Connecting cities with roads/electricity/telephone/…
- Pre-processing for other algorithms.

We will see two greedy algorithms for building Minimum Spanning Trees.

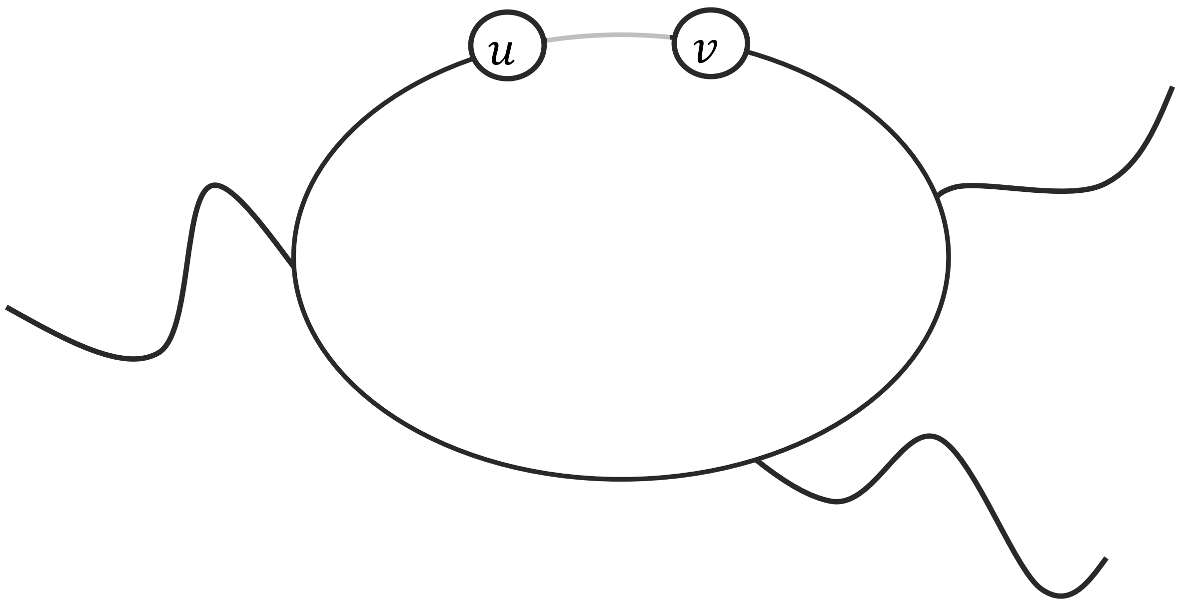
# What do MSTs look like?

Facts about **trees**: Two equivalent definition of a tree on $n$ vertices.

1. A connected acyclic graph.
2. A connected graph with $n-1$ edges.

Any **minimum weight** set of edges that **connects all vertices** is a **tree**! Why?

If a set of edges connecting all vertices has a cycle, we can remove one of its edges and still connect all vertices.
→ Removing any edge on the cycle, keeps the graph still connected.

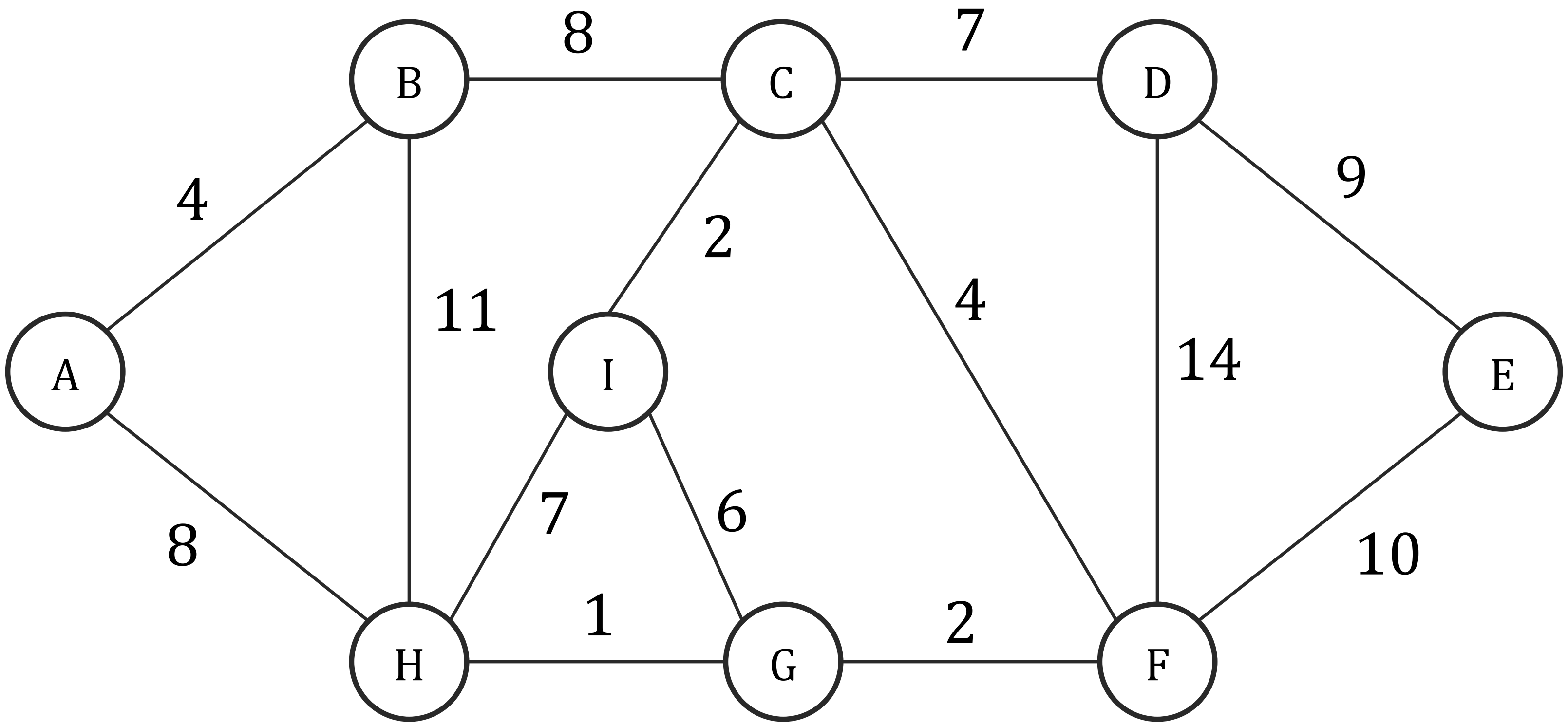


# Graph Structures and Facts

# Cuts and Graphs

**Definition:** A **cut** in a graph is a partition of vertices to two disjoint sets $S$ and $V \setminus S$.

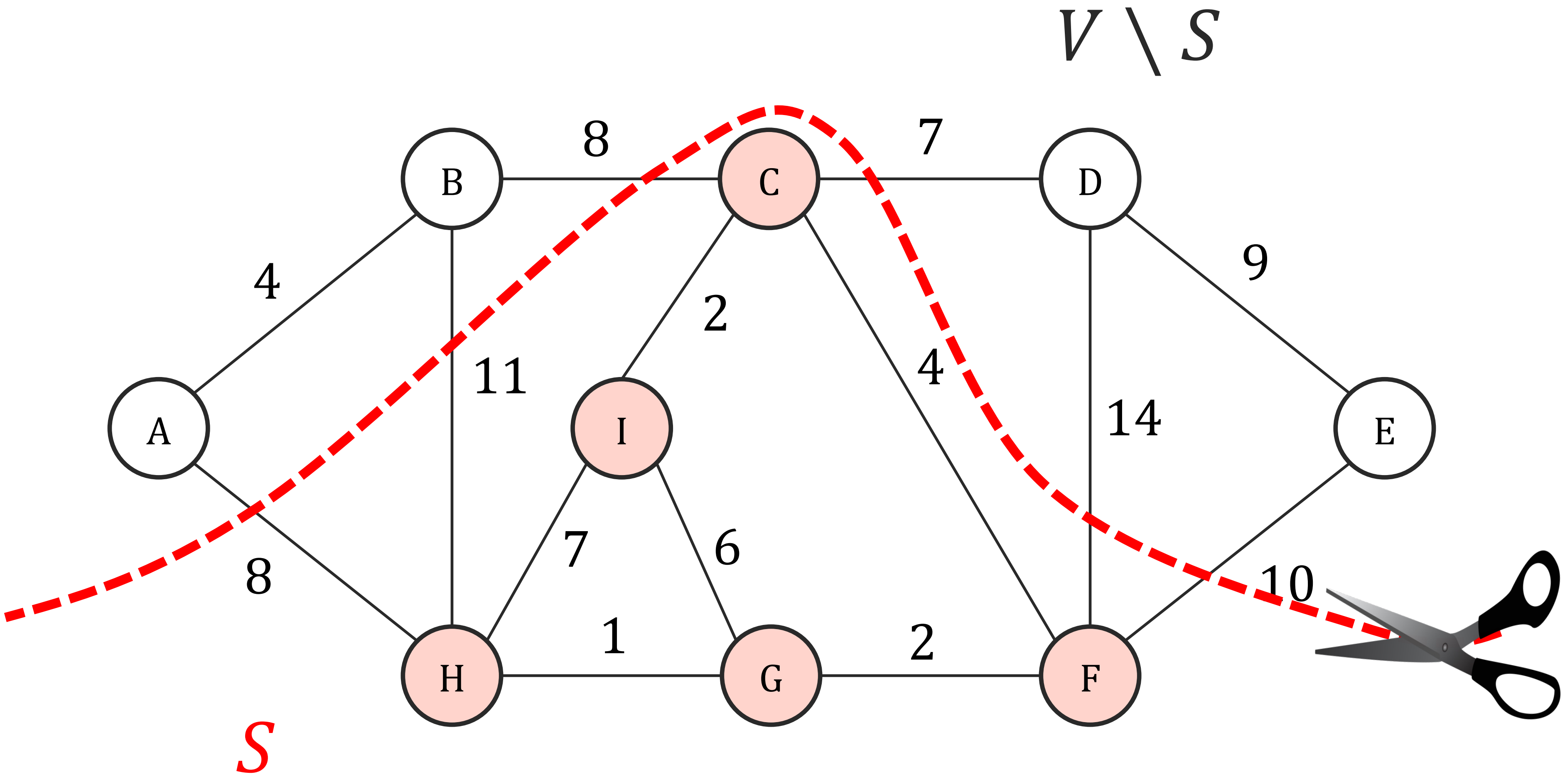
→ we'll color them differently to make the two sets clear.



# Cuts and Graphs

**Definition:** A **cut** in a graph is a partition of vertices to two disjoint sets $S$ and $V \setminus S$.

→ we'll color them differently to make the two sets clear.

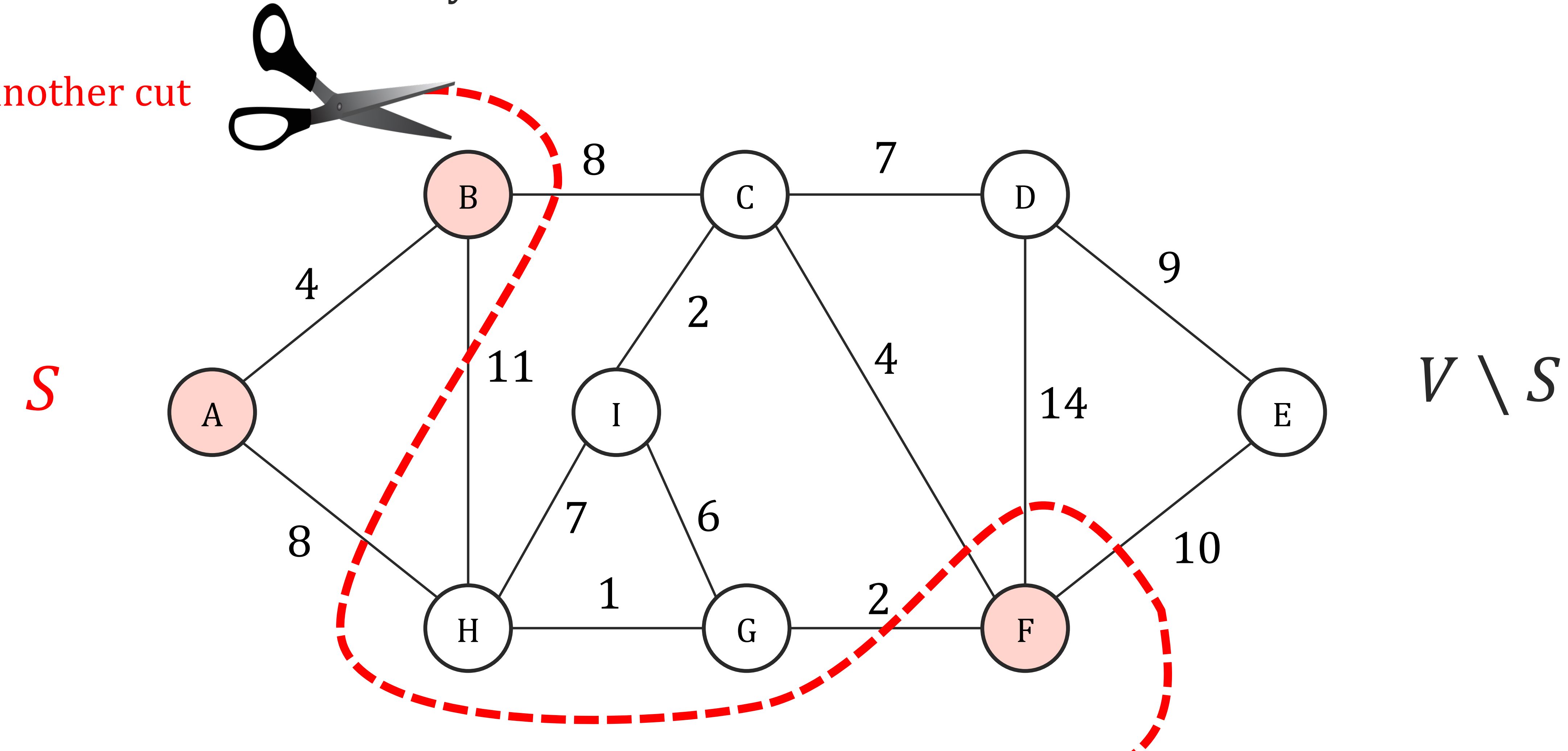


# Cuts and Graphs

**Definition:** A **cut** in a graph is a partition of vertices to two disjoint sets $S$ and $V \setminus S$.

→ we'll color them differently to make the two sets clear.

This is another cut



# Greedy Algorithms and Cuts

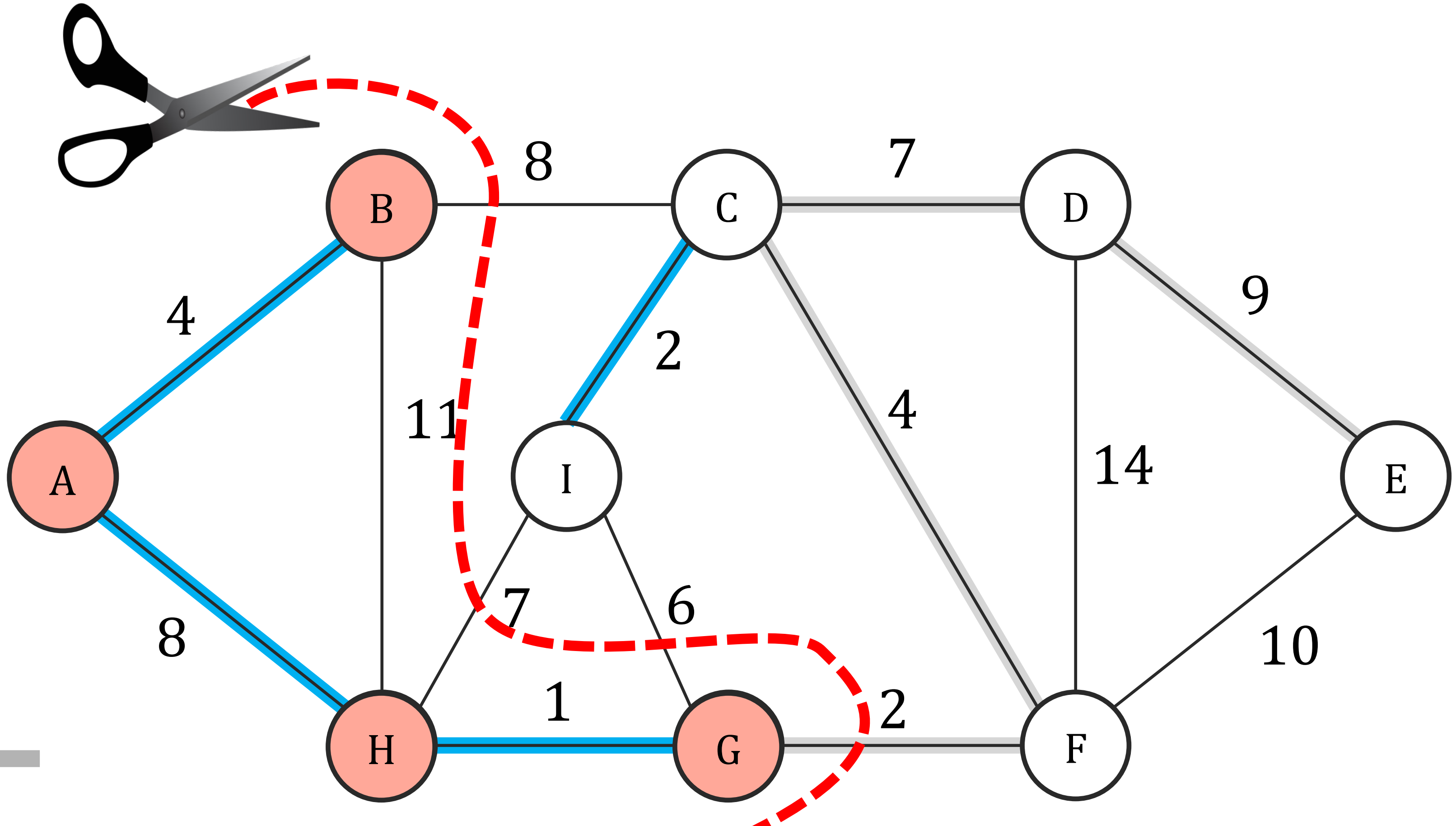
Imagine, we already discovered some of the edges $X$ of a minimum spanning tree T.

Take any **cut** where edges $X$ don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut? lightest = least $w_e$

Edges $X$:

MST T: and

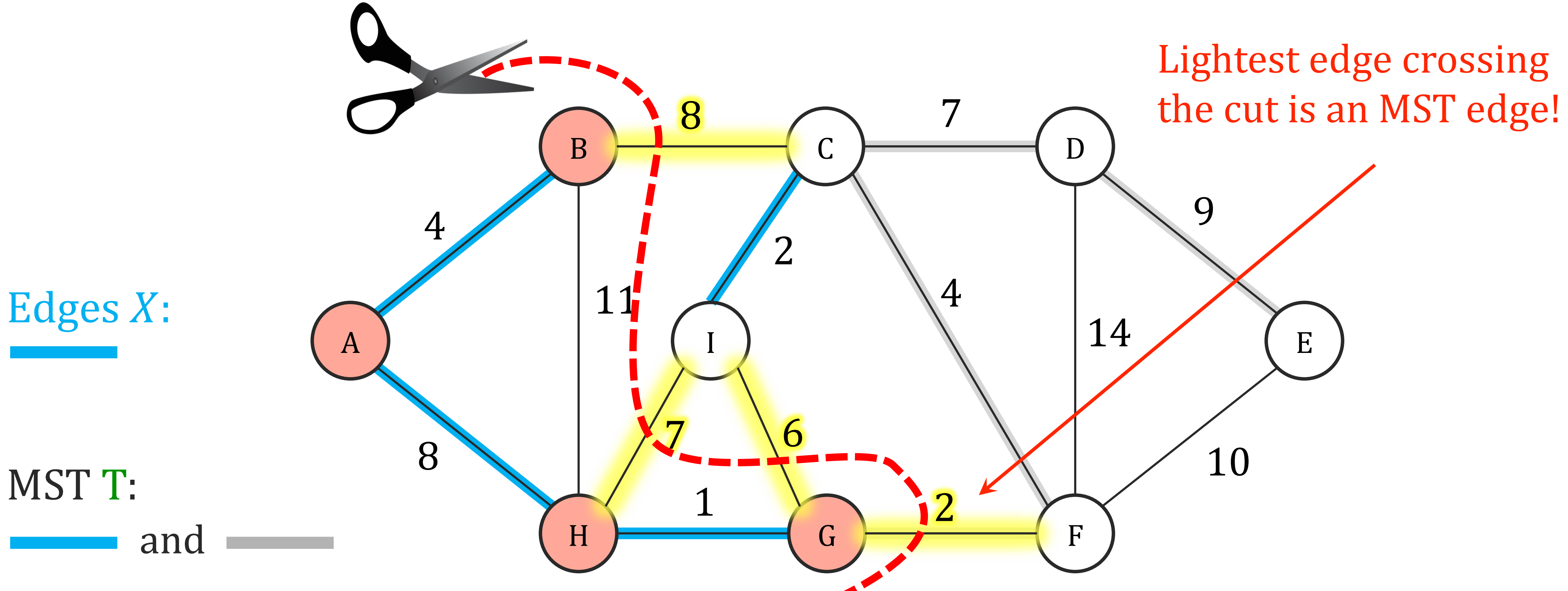


# Greedy Algorithms and Cuts

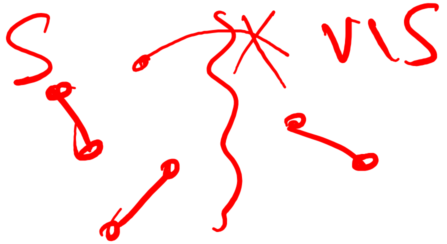
Imagine, we already discovered some of the edges $X$ of a minimum spanning tree T.

Take any **cut** where edges $X$ don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut?



# Formally: The Cut Property




**Claim:** Suppose $X \subseteq E$ is part of an MST for graph $G$. Consider a cut $S$, $V \setminus S$, such that

- $X$ has no edges from $S$ to $V \setminus S$. (no edges in X are cut).

Let $e \in E$ be the lightest weight edge from $S$ to $V \setminus S$.

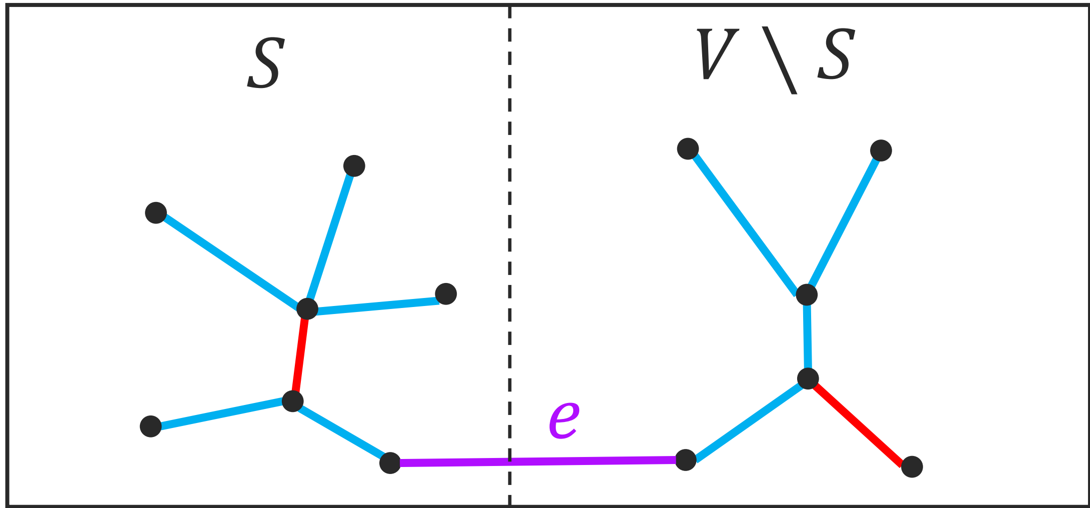
Then $X \cup \{e\}$ is **also a subset of an MST for graph** $G$.

---

**Proof:** Take the MST T that satisfies the conditions of the above claim

$X$: blue edges
T: blue and red edges.

**Case 1)** $e \in T$. Then by definition $X \cup \{e\} \in T$.



# Formally: The Cut Property

In class, we didn't specify $e'$ sufficiently. The notes are updated here to specify that $e' \in T$ is chosen from the cycle in $T \cup \{e\}$.

**Claim:** Suppose $X \subseteq E$ is part of an MST for graph $G$. Consider a cut $S$, $V \setminus S$, such that

- $X$ has no edges from $S$ to $V \setminus S$.

Let $e \in E$ be the lightest weight edge from $S$ to $V \setminus S$.

Then $X \cup \{e\}$ is **also a subset of an MST for graph** $G$.

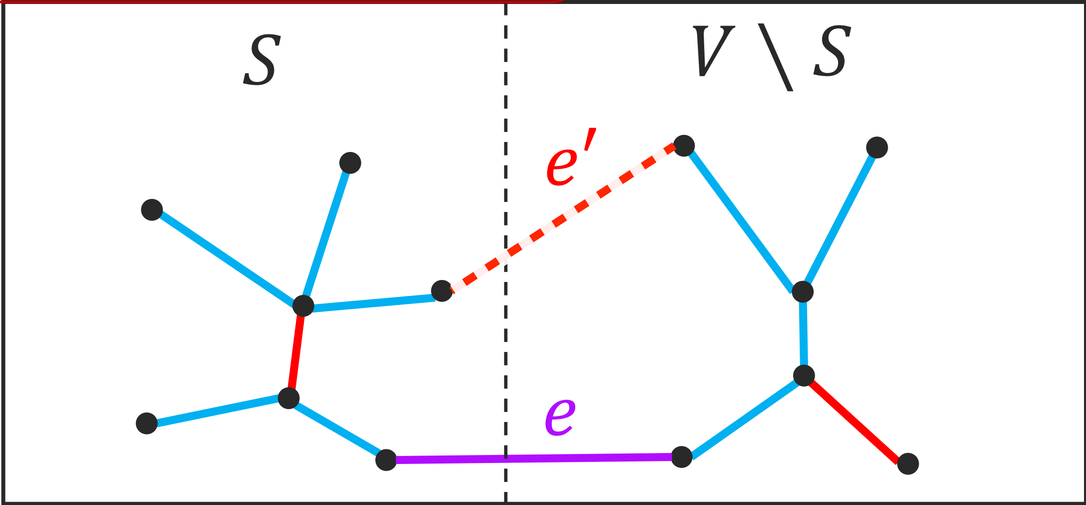
---

**Proof:** Take the MST T that satisfies the conditions of the above claim.

$X$: blue edges
T: blue and red edges.

**Case 2)** $T \cup \{e\}$ forms a cycle, since $T$ is connected already.
→ This cycle must have another edge $e' \in T$ that cross from $S$ to $V \setminus S$.

Consider $T' = T \cup \{e\} \setminus e'$:
→$T'$ also connects all vertices of the graph
→$cost(T') = cost(T) + w_e - w_{e'} \leq cost\,(T)$.
→So, $T'$ is also a minimum spanning tree!
$X \cup \{e\}$ is **also a subset of an MST for graph** $G$



# Greedy Algorithms based on the Cut Property

Any algorithm that fits the following form finds an MST.

**Meta Algorithm for MST**

$X = \{\}$
Repeat until $|X| = |V| - 1$
    Pick $S \subseteq V$, s.t. $X$ has no edges from $S$ to $V \setminus S$
    $e \leftarrow$ lightest weight edge from $S$ to $V \setminus S$
    $X \leftarrow X \cup \{e\}$

Different Algorithms pick $S$ differently

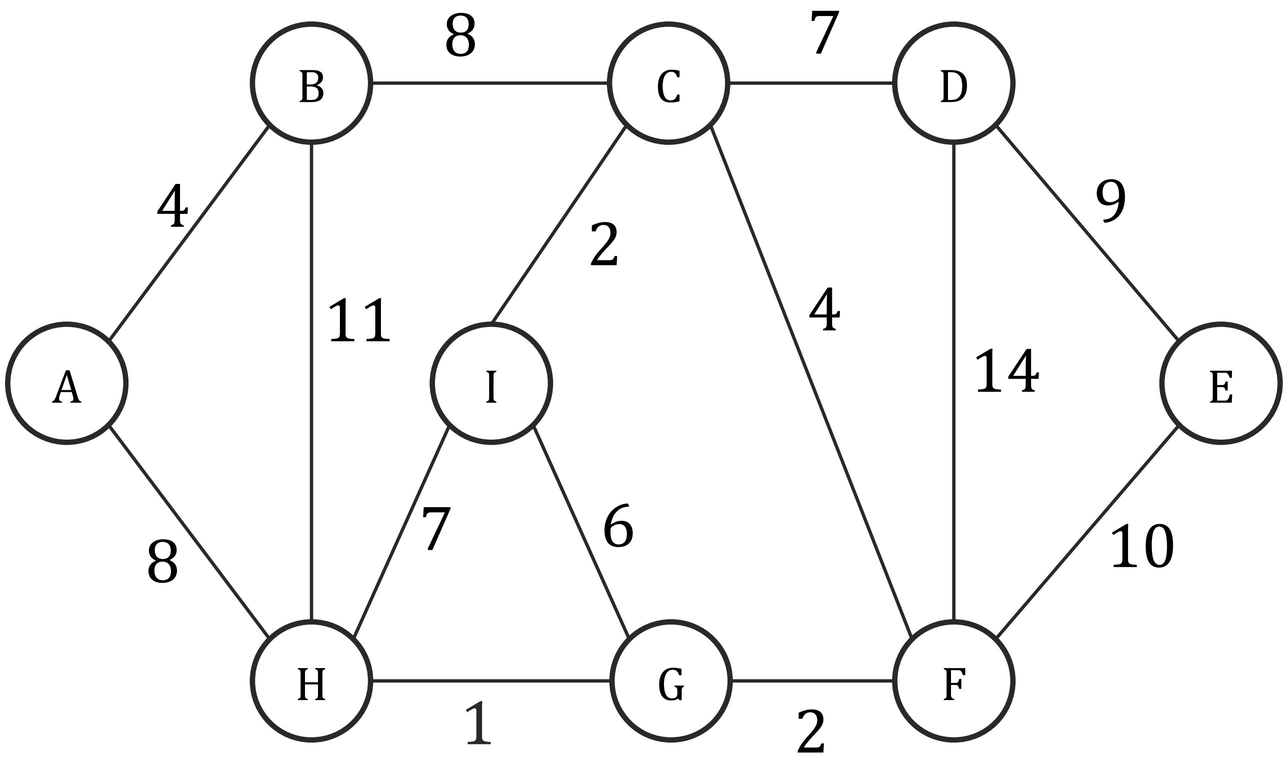
**Claim**: The meta Algorithm above returns a minimum spanning tree.
**Proof:** By induction …
Induction step:
The cut property ensures that $X \cup \{e\}$ is always a subset of an MST.

# Kruskal’s Algorithm

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

  X = {}

  **for** $e \in E$ in **increasing order of weight**

    **If** adding $e$ to $X$ doesn't create a cycle

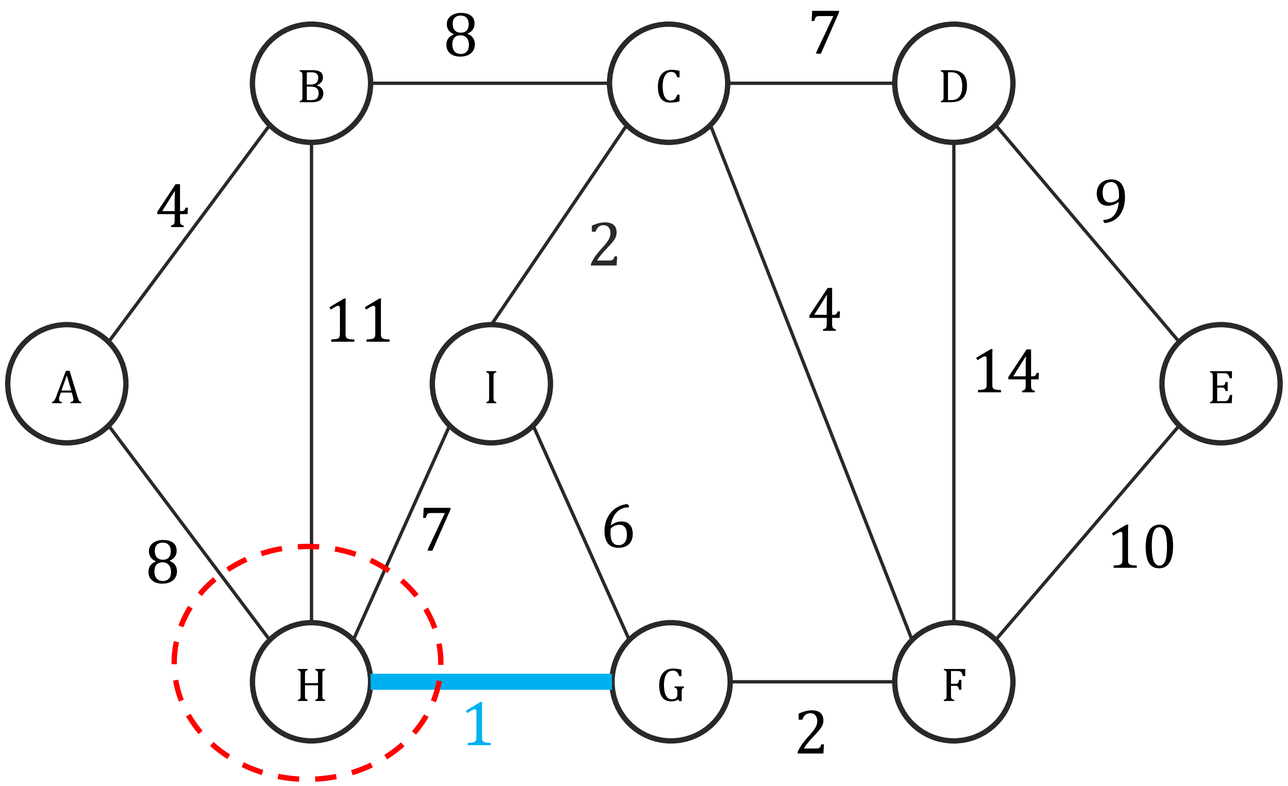
    $X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

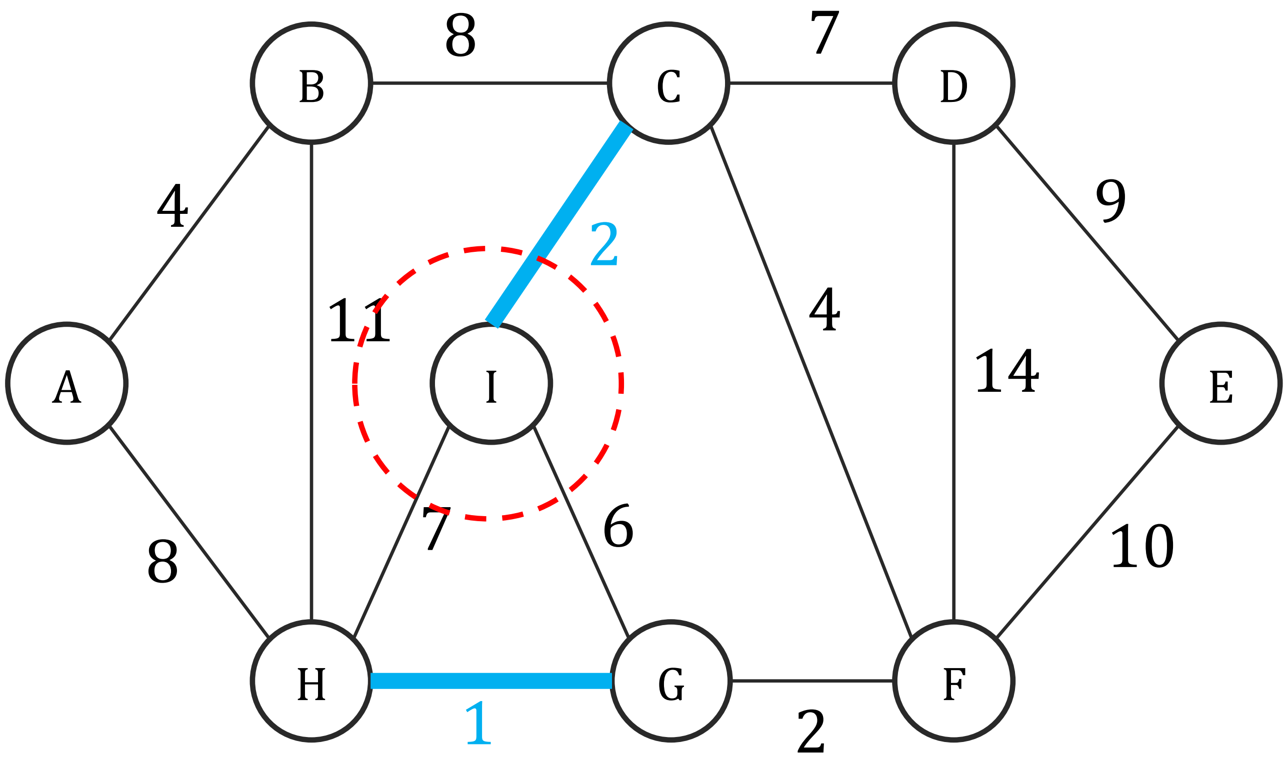
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

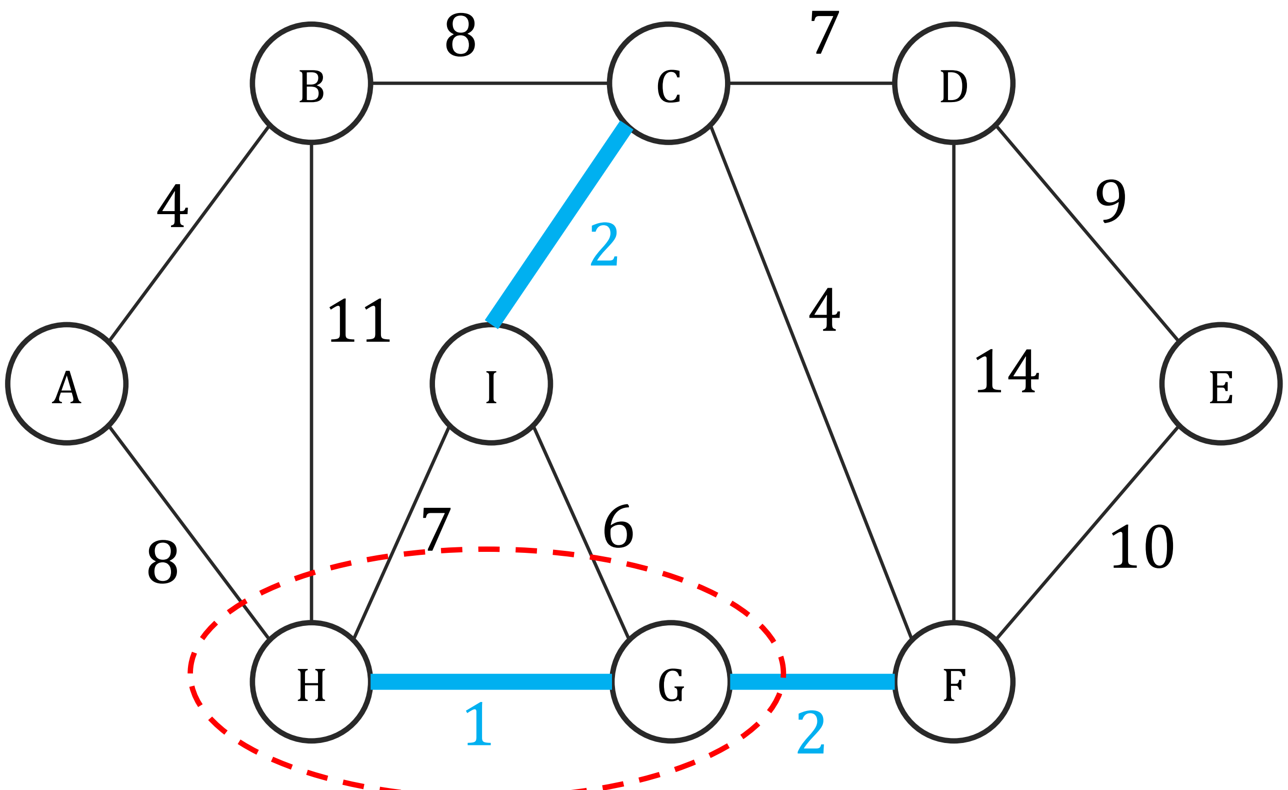
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

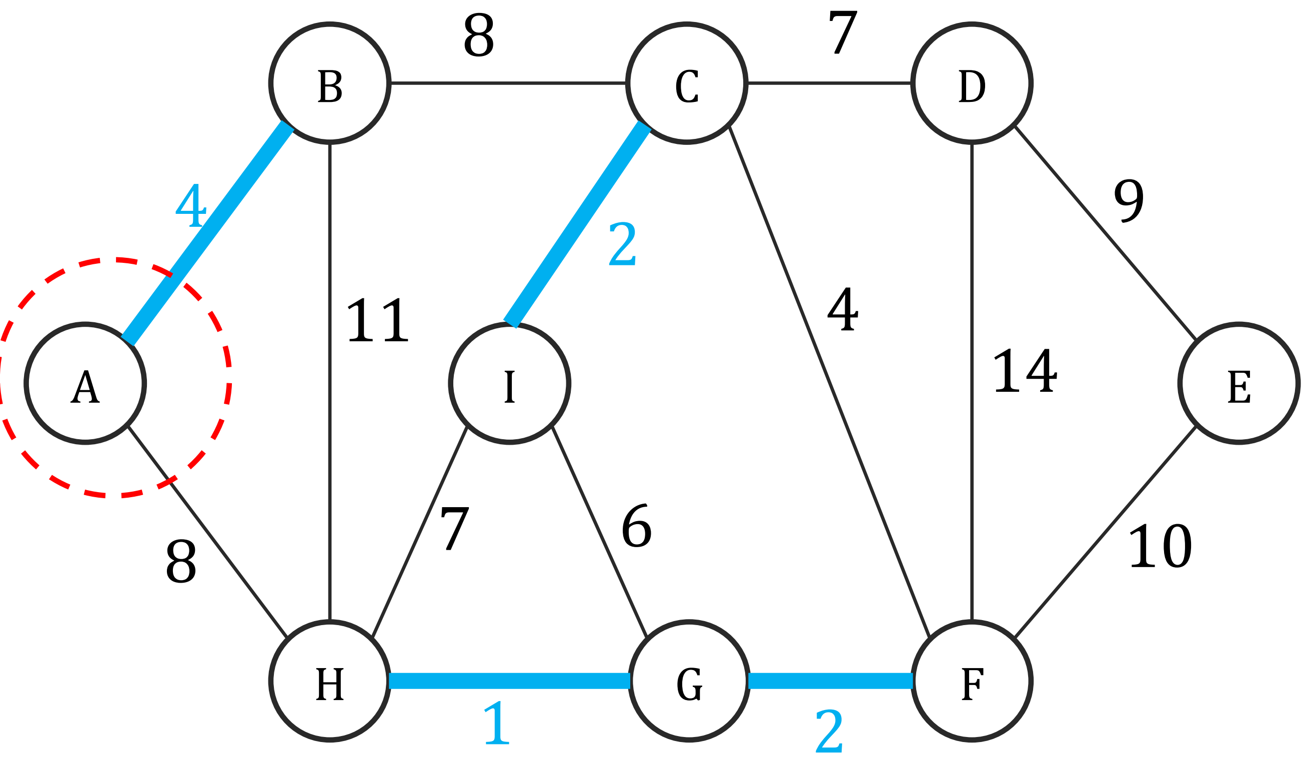
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

  X = {}

  **for** $e \in E$ in **increasing order of weight**

    **If** adding $e$ to $X$ doesn't create a cycle

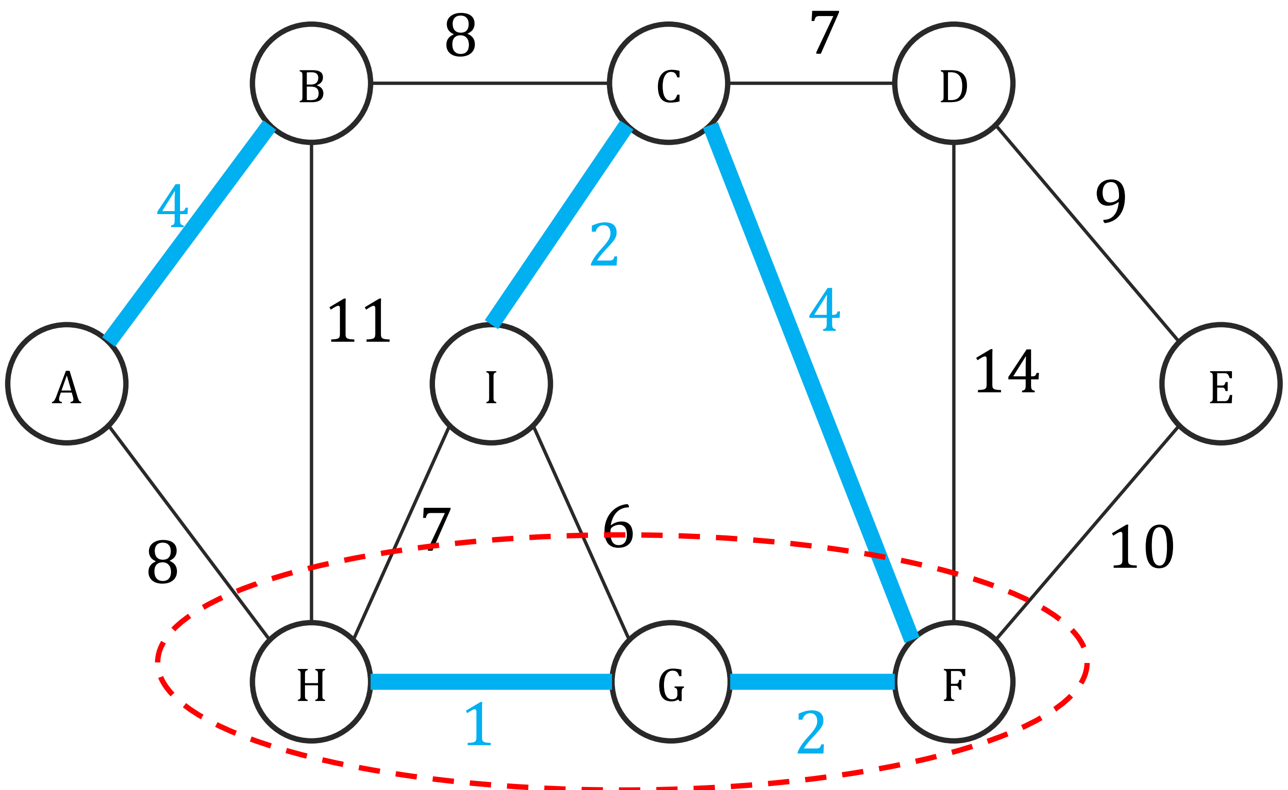
    $X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

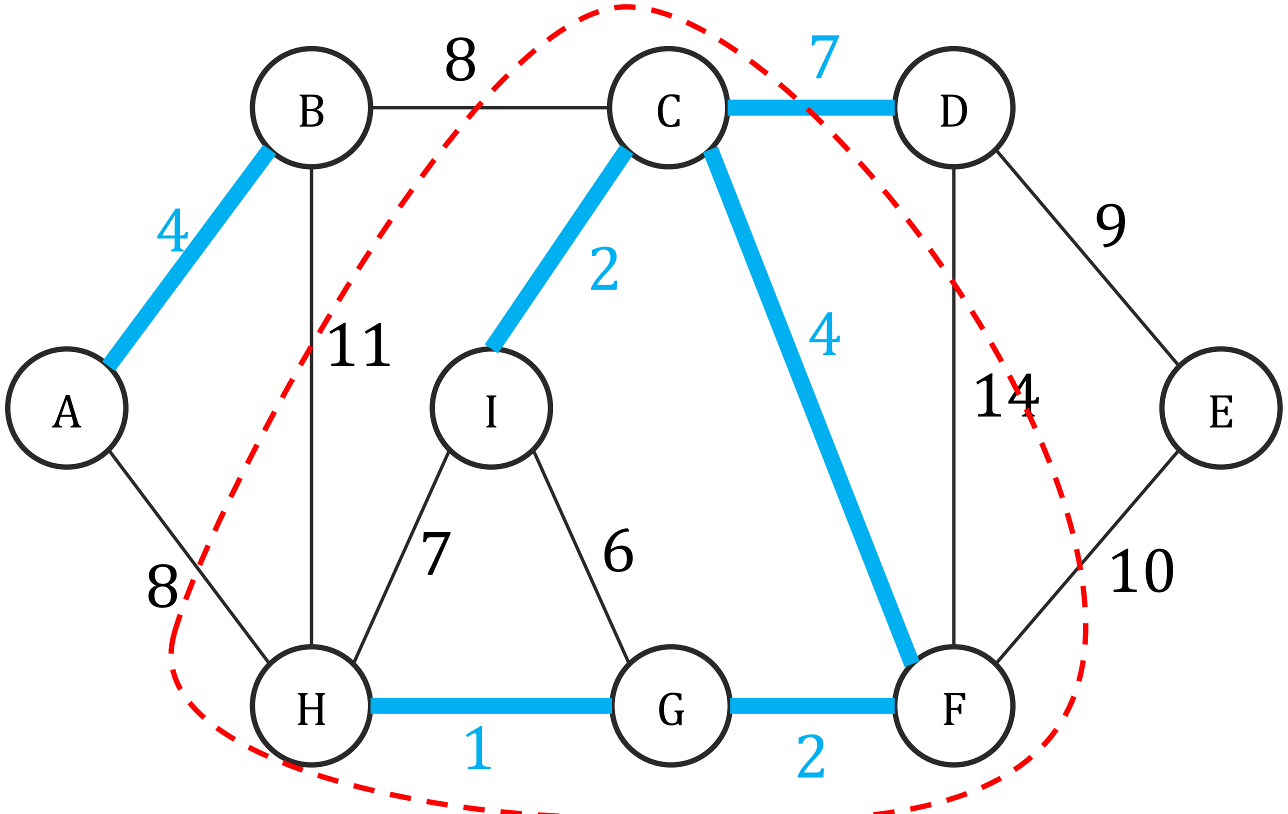
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

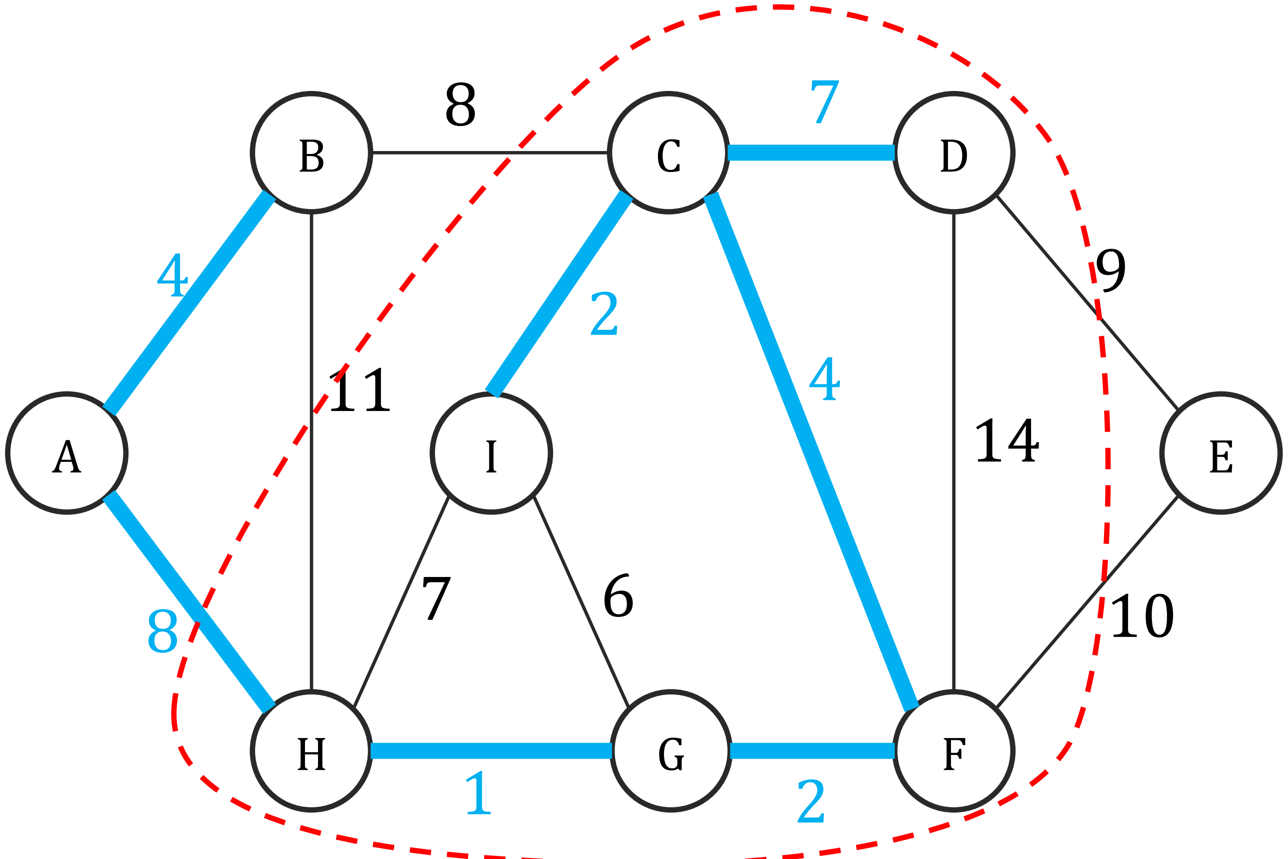
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

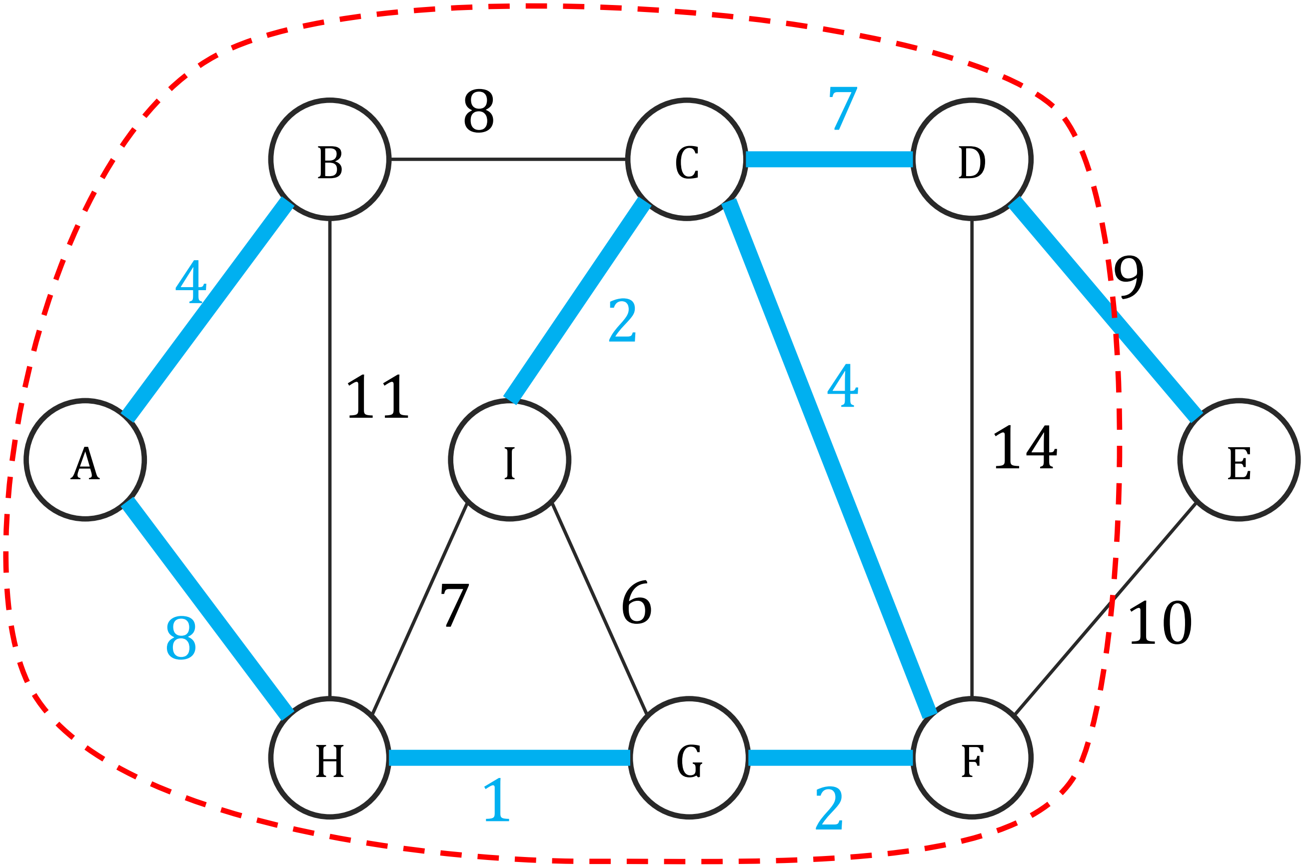
$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that $(u, v)$ is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for $u$ and $v$.




**Kruskal**(G = (V,E)):

X = {}

**for** $e \in E$ in **increasing order of weight**

**If** adding $e$ to $X$ doesn't create a cycle

$X \leftarrow X \cup \{e\}$.

**return** X

# Kruskal's Correctness

Does Kruskal return a minimum spanning tree?

- Since $X \cup \{(u, v)\}$ doesn't have a cycle, $u$ and $v$ belong to two different connected components of $X$.
- Let $S \leftarrow$ Connected component including $u$
- So $(u, v)$ is the lightest edge from $S$ to $V \setminus S$.

→Kruskal fits the meta algorithm description, so it find an MST.

# Kruskal's Runtime and Union-Find

How do we quickly check if $X \cup \{(u, v)\}$ has a cycle?

→ We need to check if $u$'s connected component in $X$ = $v$'s connected component in $X$

$n = \#$ elements.

**Union-FIND:** A data-structure for **disjoint sets**

- makeSet($u$): create a set from element $u$. Takes $O(1)$
- find($u$): return the set that includes element $u$. Takes $O(\log(n))$
- union($u, v$): Merge two sets containing $u$ and $v$. Takes $O(\log(n))$

---

**Fast-Kruskal**(G = (V,E)):

  **for** $v \in V$, makeSet($v$)

  **for** edges $(u, v)$ ∈ E in increasing order of weight

    **If** find($v$) $\neq$ find($u$)

      $X \leftarrow X \cup \{(u, v)\}$

      union($u, v$)

**return** X

# Runtime of Kruskal's Algorithm

**Sorting $m$ edges:** $O(m\log(m)) = O(m\log(n))$. Since $m \le n^2$.

**Everything else:**

$O(1)$

- $n$ calls to makeSet $\Longrightarrow n \times O(1)$
- $2m$ calls to find: 2 calls per edge to find its endpoints. $m\log(n)$
- $n-1$ calls to union: A tree has $n-1$ edges. $n\log(n)$

If a graph is connected
$m \geq n-1$.

Total: $O((m+n)\log(n))$. For connected graphs $= O(m\log(n))$.

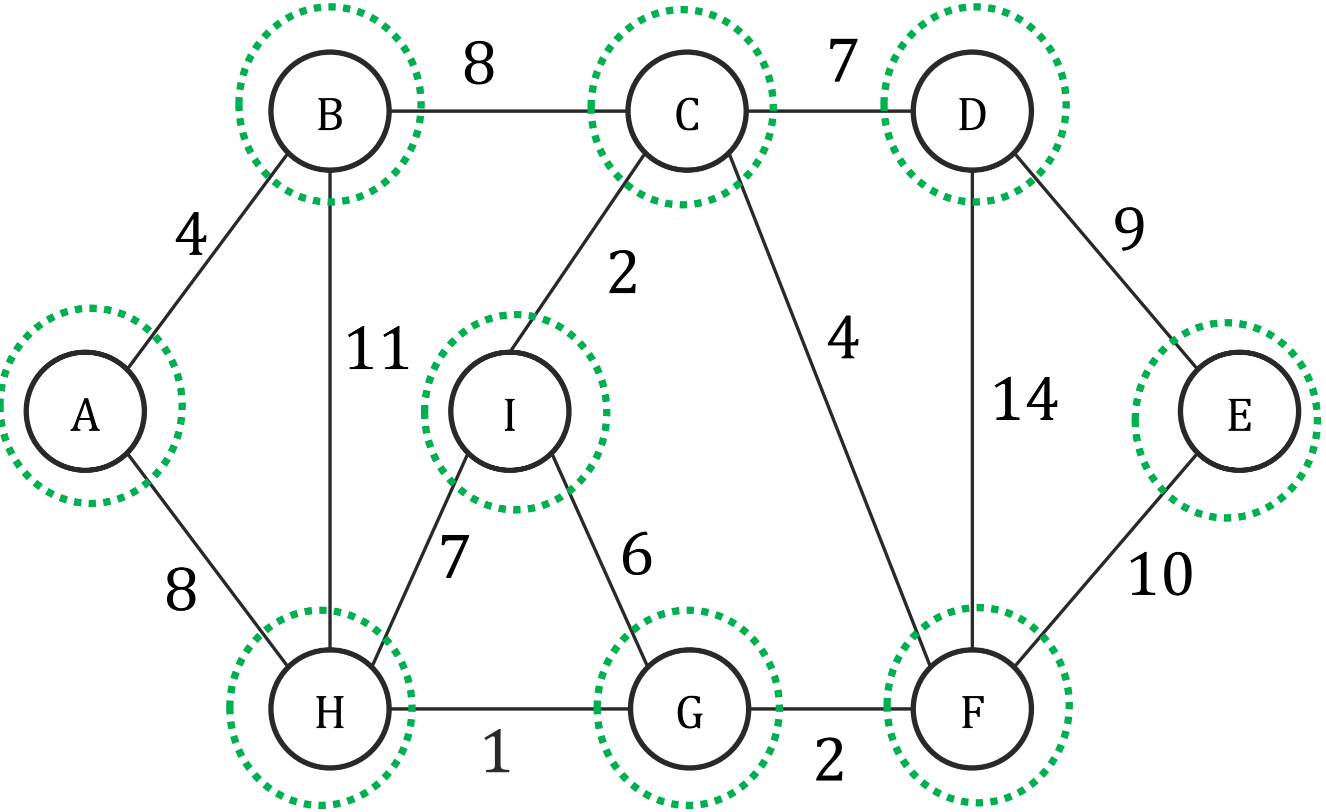
---

**Fast-Kruskal**(G = (V,E)):
 **for** $v \in V$, makeSet$(v)$
 **for** edges $(u, v)$ ∈ E in increasing order of weight
  **If** find$(v) \neq$ find$(u)$
   $X \leftarrow X \cup \{(u, v)\}$
   union$(u, v)$
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.




**Fast-Kruskal**(G = (V,E)):

**for** $v \in V$, makeSet($v$)

**for** edges $(u, v)$ $\in$ E in increasing order of weight

**If** find($v$) $\neq$ find($u$)

$X \leftarrow X \cup \{(u, v)\}$

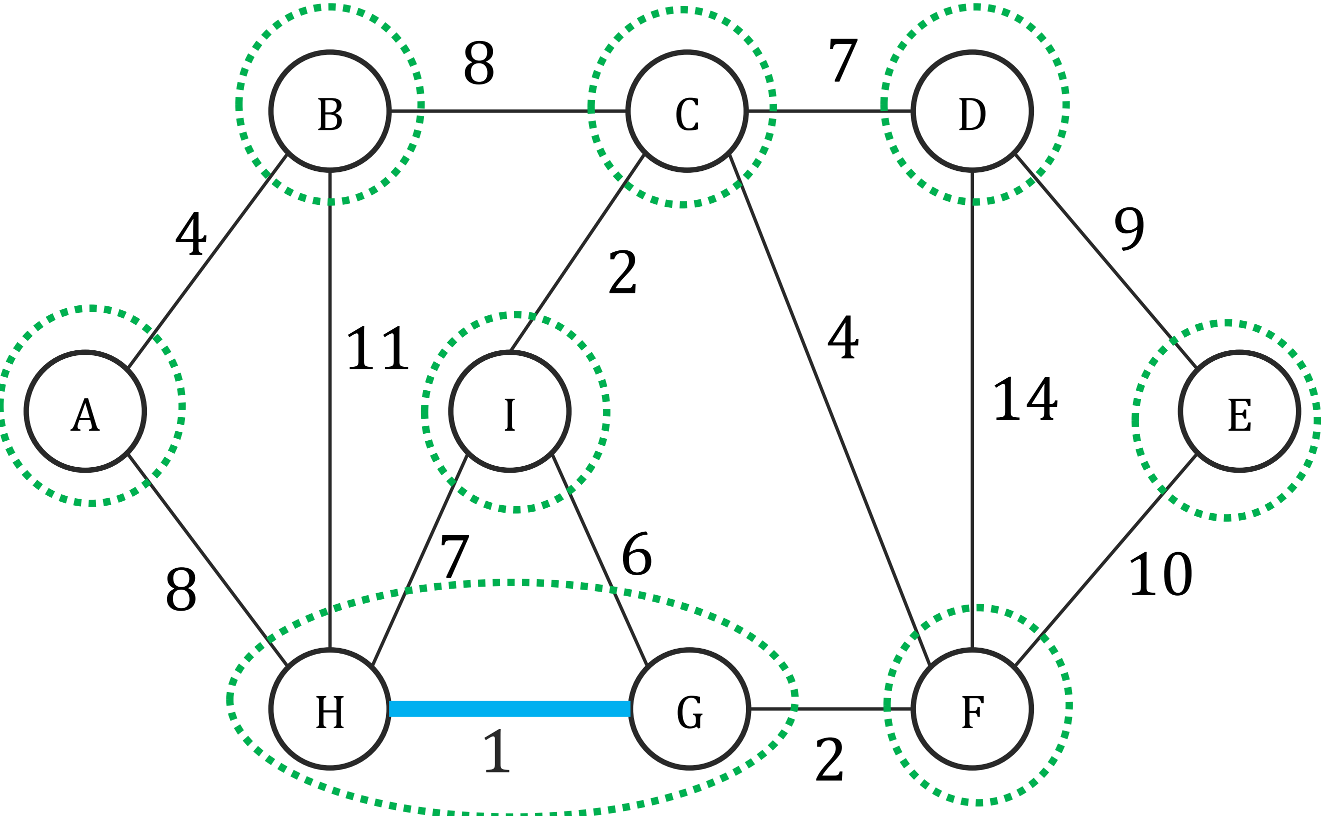
union($u, v$)

**return** X

# Kruskal’s Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

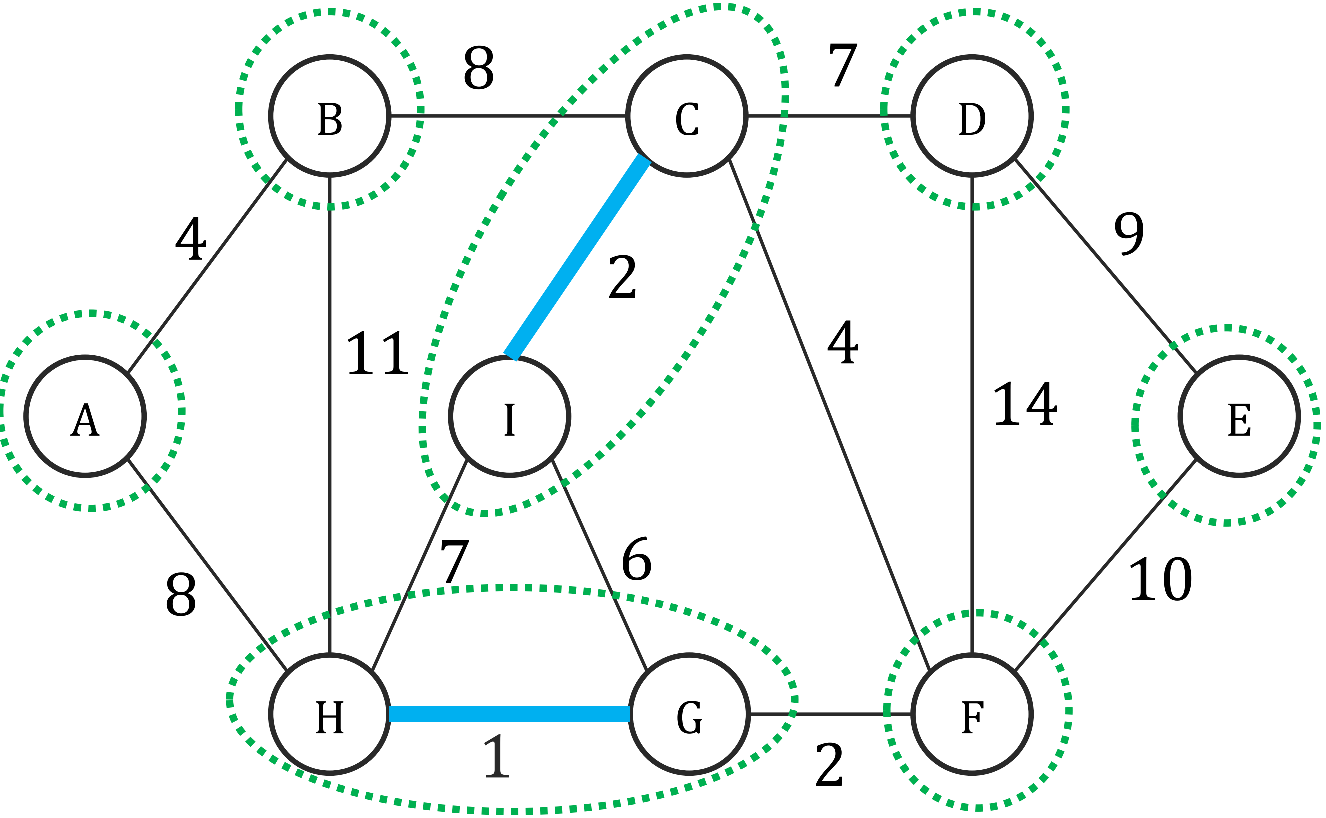



**Fast-Kruskal**(G = (V,E)):
**for** $v \in V$, makeSet($v$)
**for** edges $(u, v) \in$ E in increasing order of weight
**If** find($v$) $\neq$ find($u$)
$X \leftarrow X \cup \{(u, v)\}$
union($u, v$)
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

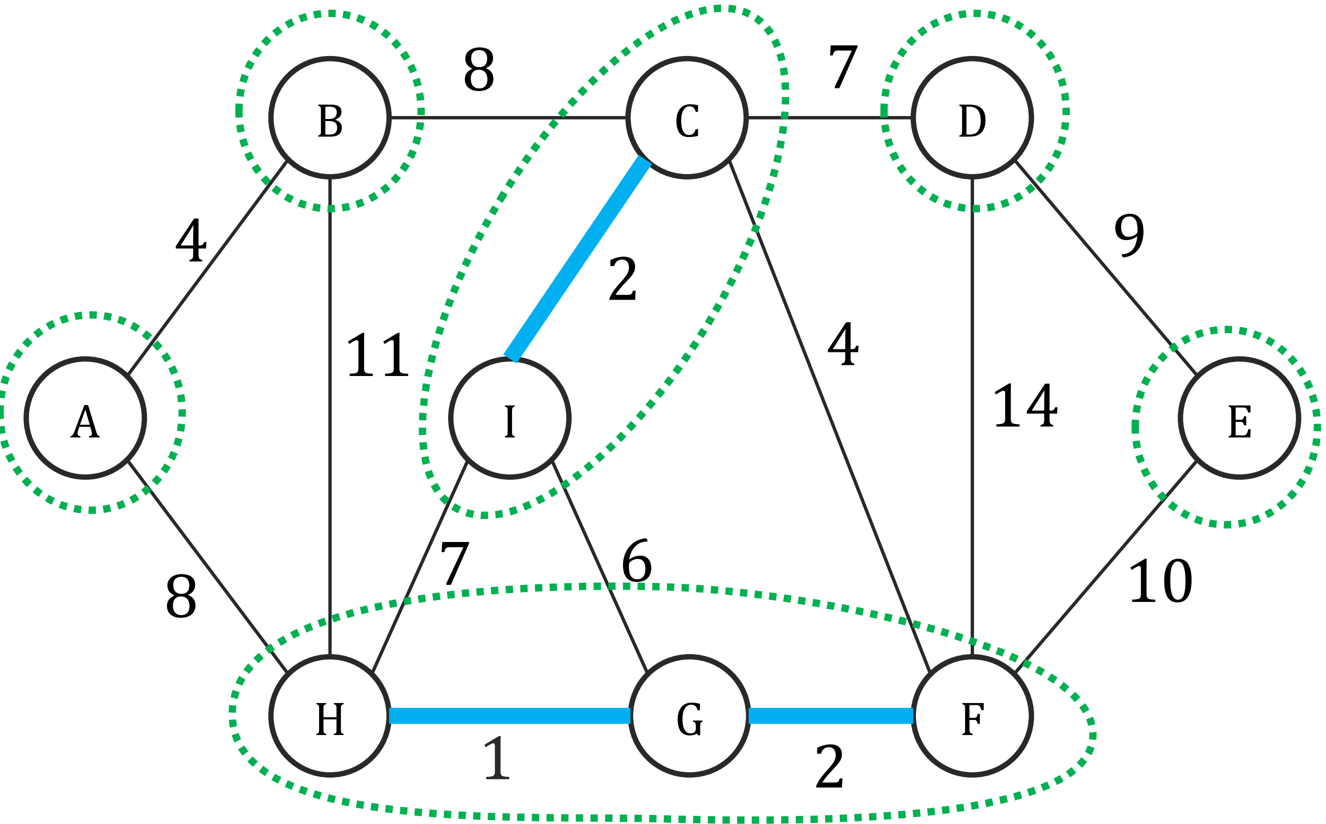



**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v) \in$ E in increasing order of weight
    **If** find($v$) $\neq$ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union($u, v$)
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

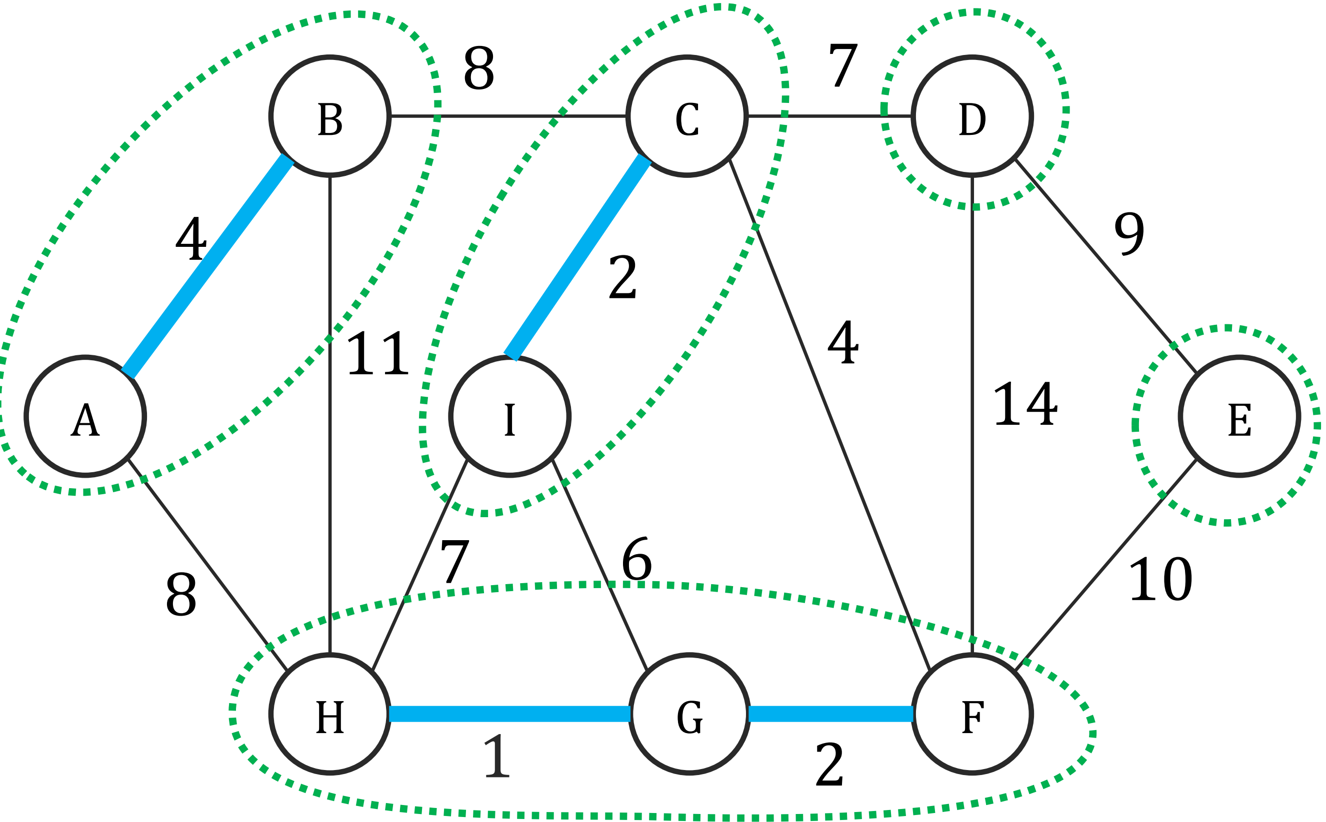



**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v)$ ∈ E in increasing order of weight
    **If** find($v$) ≠ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union($u, v$)
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.




**Fast-Kruskal**(G = (V,E)):

**for** $v \in V$, makeSet($v$)

**for** edges $(u, v)$ $\in$ E in increasing order of weight

**If** find($v$) $\neq$ find($u$)

$X \leftarrow X \cup \{(u, v)\}$

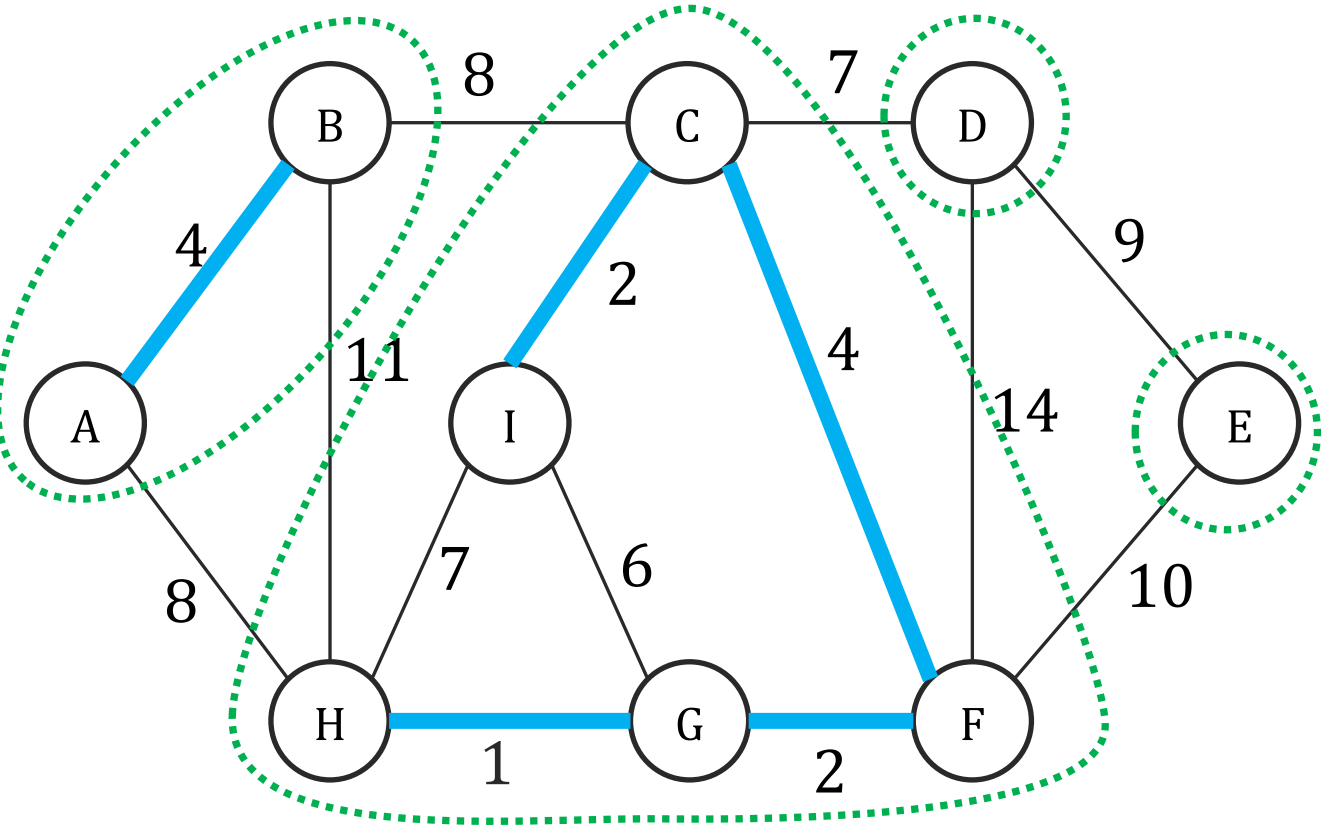
union($u, v$)

**return** X

# Kruskal’s Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

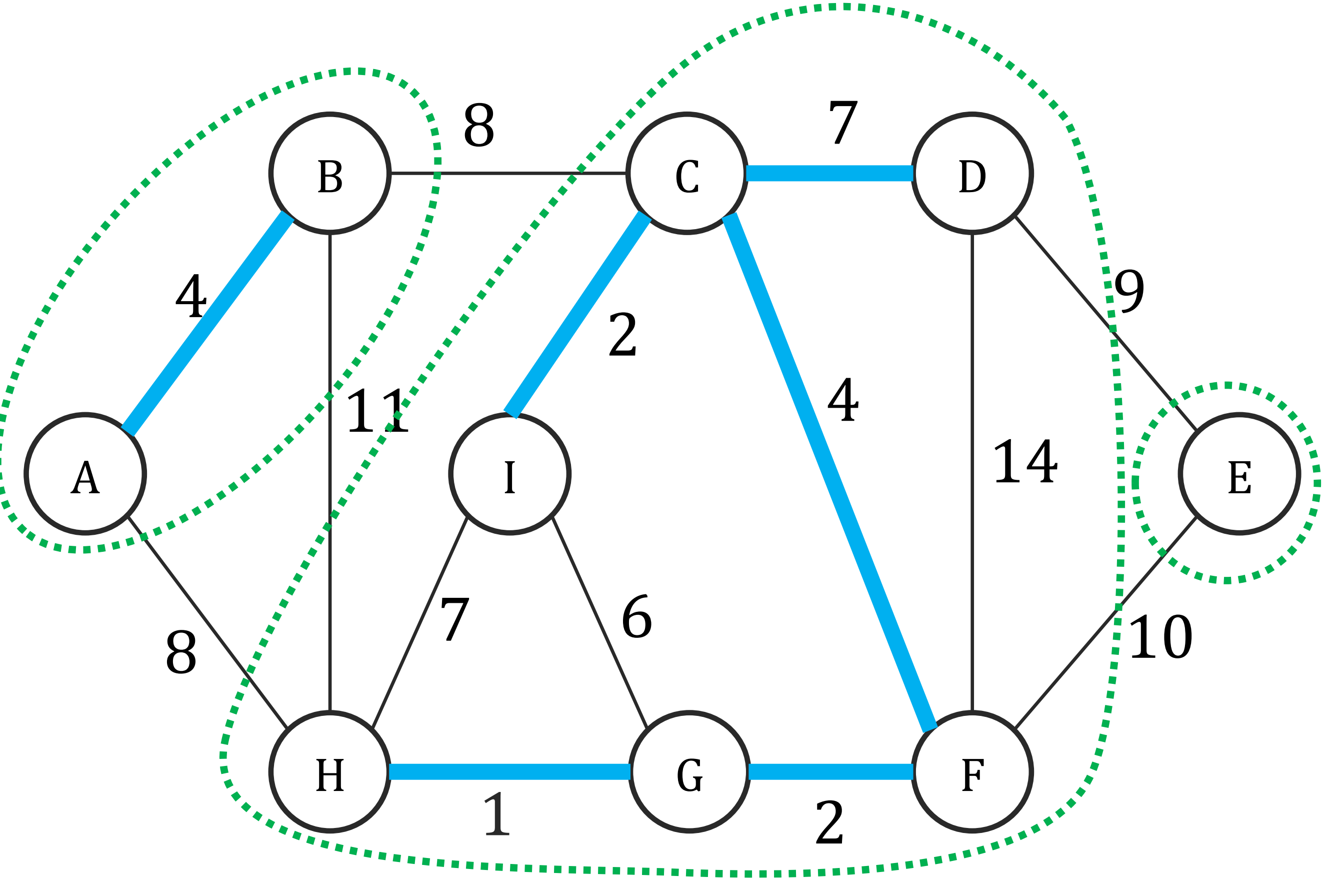



**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v)$ $\in$ E in increasing order of weight
    **If** find($v$) $\neq$ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union$(u, v)$
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

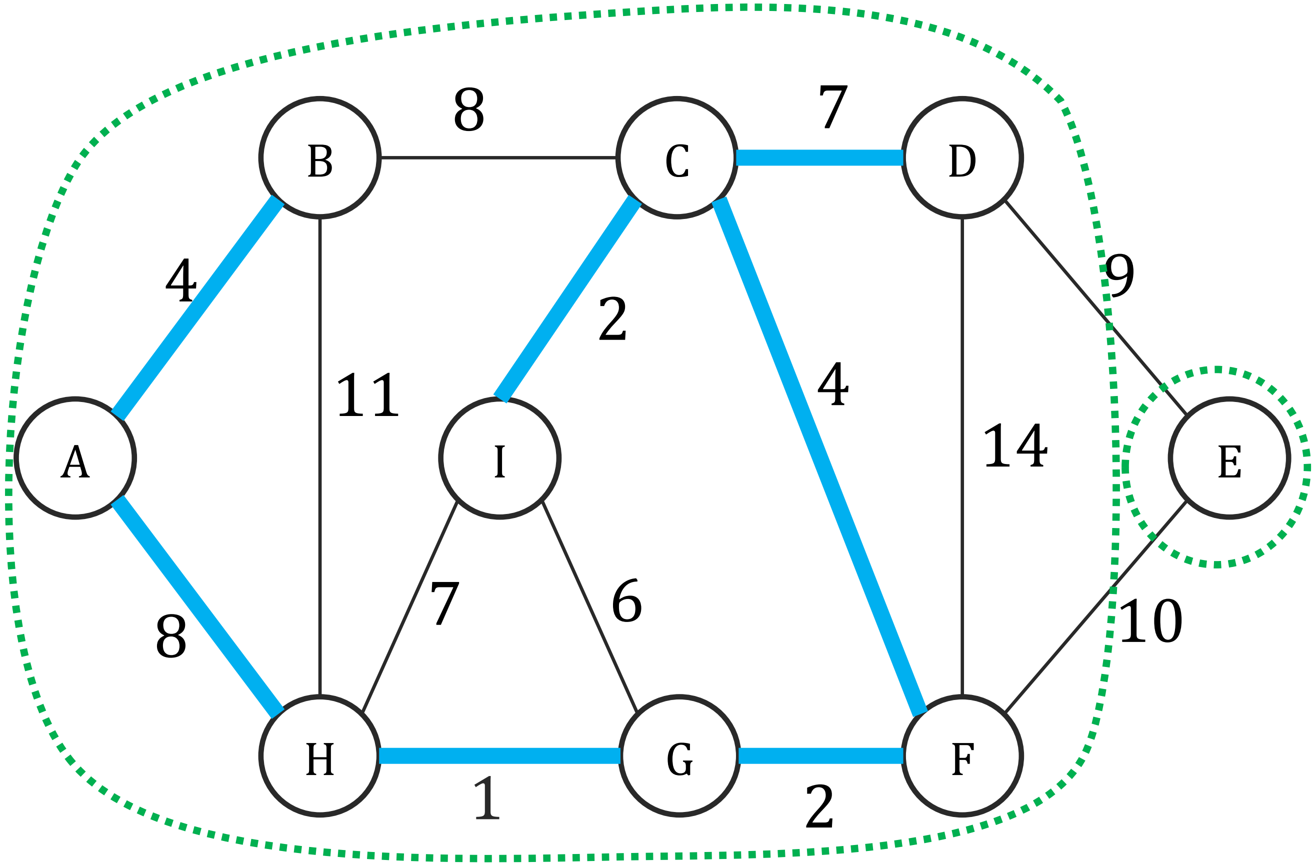



**Fast-Kruskal**$(G = (V,E))$:
  **for** $v \in V$, makeSet$(v)$
  **for** edges $(u, v) \in E$ in increasing order of weight
    **If** find$(v) \neq$ find$(u)$
      $X \leftarrow X \cup \{(u, v)\}$
      union$(u, v)$
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

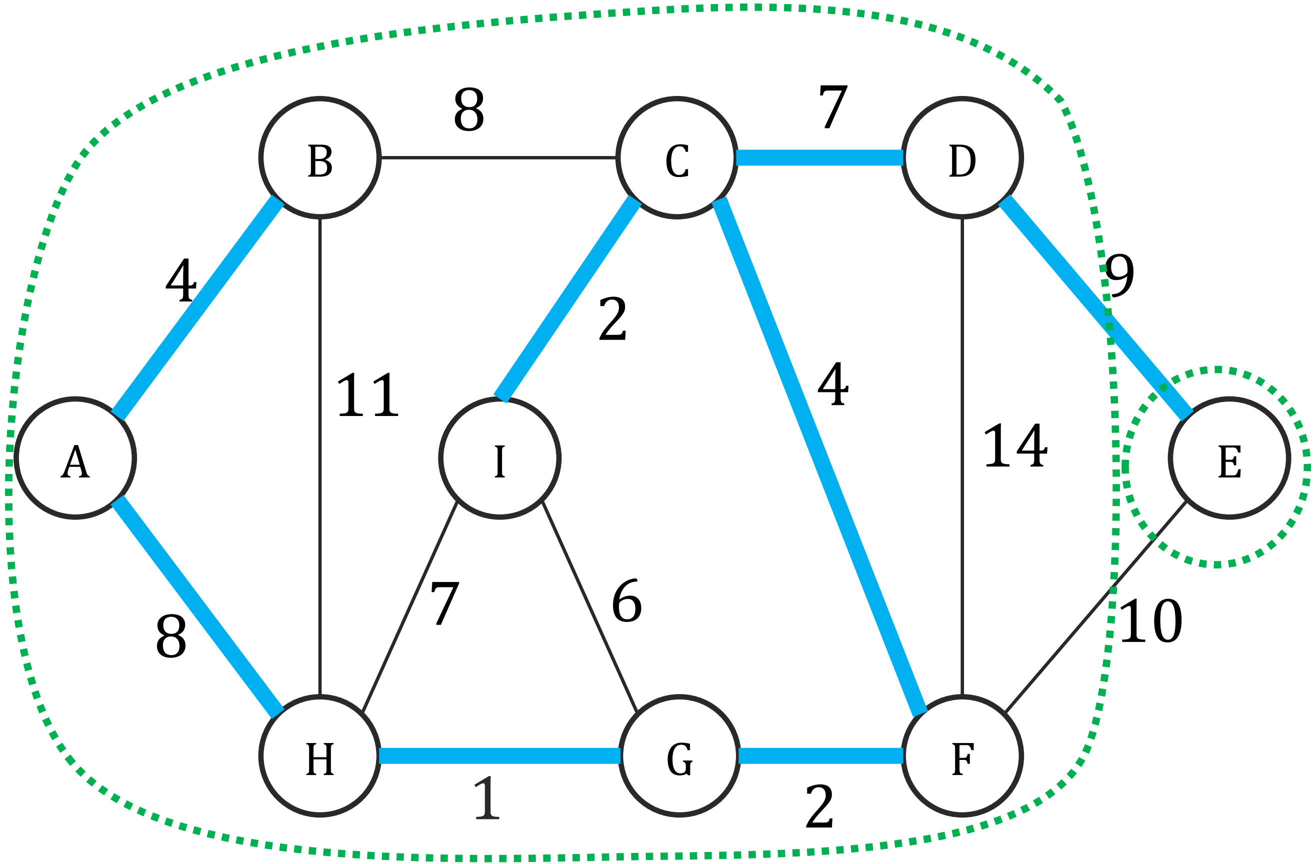



**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v)$ $\in$ E in increasing order of weight
    **If** find($v$) $\neq$ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union($u, v$)
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

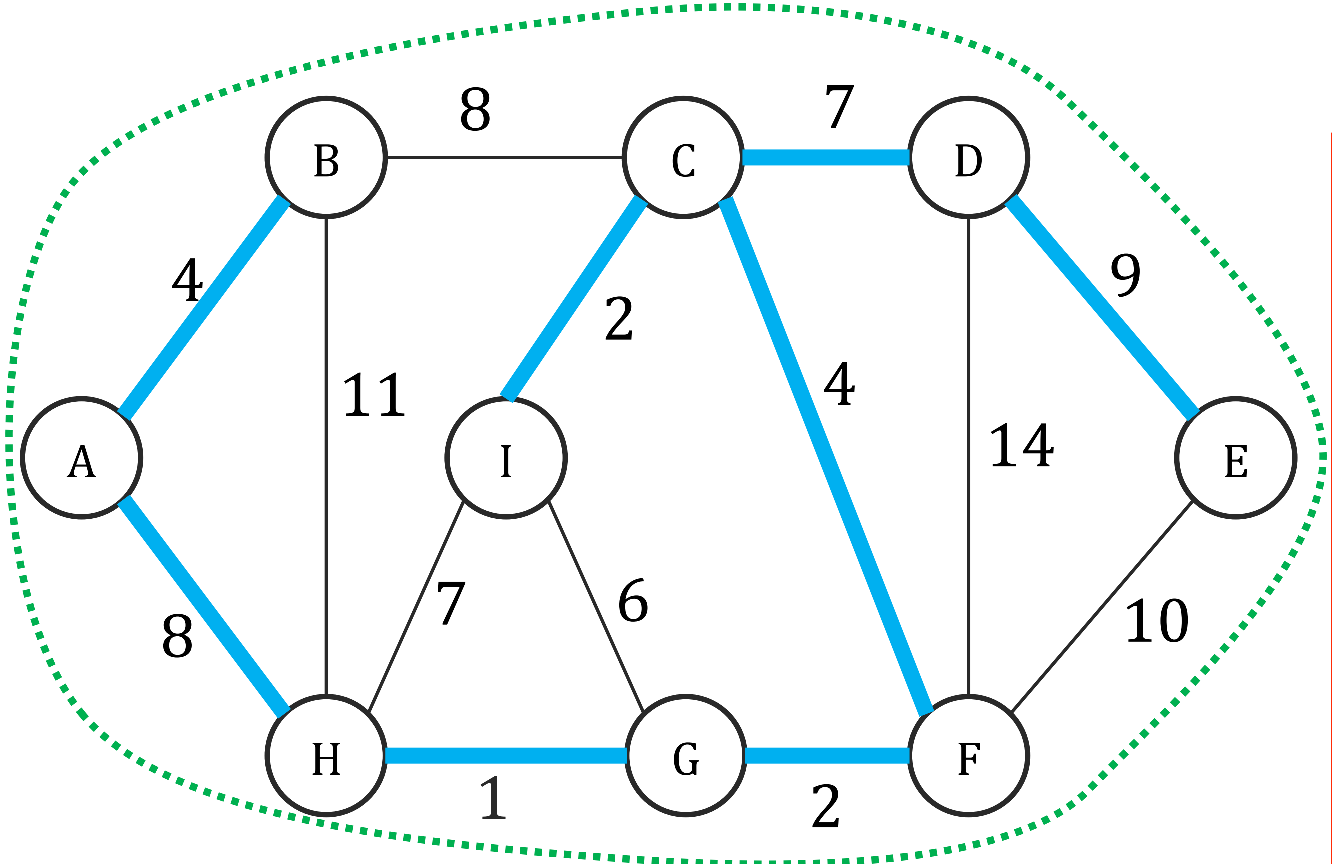



**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v)$ $\in$ E in increasing order of weight
    **If** find($v$) $\neq$ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union($u, v$)
**return** X

# Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.




**Fast-Kruskal**(G = (V,E)):
  **for** $v \in V$, makeSet($v$)
  **for** edges $(u, v)$ $\in$ E in increasing order of weight
    **If** find($v$) $\neq$ find($u$)
      $X \leftarrow X \cup \{(u, v)\}$
      union($u, v$)
**return** X

# Wrap up

We saw a meta algorithm for MSTs

→ One variant: Kruskal's Algorithm
    → Greedily add the lightest edge that doesn't create a cycle

→ Union-Find: Useful data structure for keeping track of sets and trees.

**Next time**

- Another algorithm for MSTs
- Dynamic Programming