Consider an undirected connected graph $G = (V, E)$ where each edge $e$ has weight $w(e)$.

- Given a subset of edges $A \subseteq E$, define $w(A) = \sum_{e \in A} w(e)$ to be the total weight of the edges in $A$.
- A spanning tree of $G$ is a tree $T$ that contains all nodes in $G$.

**Problem:** Can we efficiently find the minimum spanning tree (MST), i.e., spanning tree with minimum total weight?

- For simplicity, we will assume all edges have distinct weights.

### Greedy Approaches

- Consider the following greedy approaches:
  
  - Sort the edges by increasing weight.
  
  - Add next edge that doesn’t complete a cycle.

  - Sort the edges by increasing weight.
    
    - Let $S = \{s\}$.
    
    - Add next edge $(u, v)$ where $u \in S, v \not\in S$. Add $v$ to $S$.

  - Sort the edges by decreasing weight. Remove the next edge that doesn’t disconnect the graph.

  - Which approach constructs a minimum spanning tree? All of them! We’ll prove correctness for the first two.

### Important Lemma: Finding edges in MST

- **Cut Lemma:** Let $S \subseteq V$ and let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \not\in S$. The MST contains edge $e$.

- Suppose $T$ is a spanning tree that doesn’t include $e$. We’ll construct a different spanning tree $T’$ such that $w(T’) < w(T)$ and hence $T’$ can’t be the MST.

- Since $T$ is a spanning tree, there’s a $u \leadsto v$ path $P$ in $T$. Since the path starts in $S$ and ends up outside $S$, there must be an edge $e’ = (u’, v’)$ on this path where $u’ \in S, v’ \not\in S$.

- Let $T’ = T - \{e’\} + \{e\}$. This is a still spanning tree, since any path in $T’$ that needed $e’$ can be routed via $e$ instead. But since $e$ was the lightest edge between $S$ and $V \setminus S$,

  $$w(T’) = w(T) - w(e’) + w(e) \leq w(T) - w(e’) + w(e’) = w(T)$$

### Prim’s Algorithm

- **Prim’s Algorithm:** Sort the edges by increasing weight.

  - Let $S = \{s\}$.

  - While $S \neq V$: Add next edge $(u, v)$ where $u \in S, v \not\in S$ and add $v$ to $S$.

- **Proof of Correctness:**

  - Let $S$ be the set of nodes in the tree constructed so far.

  - The next edge added to the tree is the lightest edge between $S$ and $V \setminus S$. Hence, the cut lemma implies $e$ must be in the MST.

### Kruskal’s Algorithm

- **Kruskal’s Algorithm:** Sort the edges by increasing weight and keep on add the next edge that doesn’t complete a cycle.

- **Proof of Correctness:**

  - Suppose $e = (u, v)$ is the next edge added.

  - Let $S$ be the set of nodes that can be reached from $u$ before $e$ was added. Note that $v \not\in S$ since otherwise adding $e$ would have completed a cycle.

  - No other edge between $S$ and $V \setminus S$ can have been encountered before since if it had it would have been added since it doesn’t complete a cycle. Hence $e$ is the lightest edge between $S$ and $V \setminus S$. Therefore, the cut lemma implies $e$ must be in the MST.
**Kruskal Implementation: Union-Find**

**Idea:** use clever data structure to maintain connected components of growing spanning tree. Should support the following operation:

- **Find(v):** return name of set containing v
- **Union(A, B):** merge two sets

where A and B will correspond to connected components of the edges that have been added so far.

```plaintext
for each edge e do
    Let u and v be endpoints of e
    if find(u) != find(v) then
        T = T ∪{e}
        Union(find(u), find(v))
    end if
end for
```

**Union-Find Method**

- **Make-Set(v):** Takes $O(1)$ time to add a single node.
- **Find(v):** Takes $O(1)$ time to follow pointer to label.
- **Union-Set(u, v):** $O(size\ of\ smaller\ set)$.
  - Update "next" pointer at end of longer list to point to start of shorter list
  - Update "label" pointers of shorter list to point to label of other list
  - Update auxiliary pointers and size information

**Union-Find Analysis**

**Theorem:** Consider a sequence of $m$ operations including $n$ Make-Set operations. Total running time is $O(m + n \log n)$.

- Total time from Find and Make-Set: $O(m)$
- Total time from Union: $O(n \log n)$
  - Updating next pointers: $O(n)$
  - Updating label pointers: $O(n \log n)$ because the label pointer for a node can be updated at most $\log_2 n$ times.

Hence, Kruskal’s algorithm can be implemented in time

$$O(m \log m) + O(m + n \log n) = O(m \log m)$$