Greedy Algorithms

- At each step in the algorithm, one of several choices can be made.

- **Greedy Strategy**: make the choice that is the best at the moment.

- After making a choice, we are left with **one subproblem** to solve.

- The solution is created by making a sequence of **locally optimal** choices.

A greedy algorithm does not always achieve a globally optimal solution. But even when the final solution is not optimal:

"Greedy, for lack of a better solution, is good."
**Greedy Algorithms: Optimality Conditions**

**Greedy Choice property:**
A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

**Optimal Substructure:**
An optimal solution to the problem contains within it optimal solutions to subproblems.
Greedy Example: Minimum Spanning Trees

- A TV cable company wants to connect a set of \( N \) buildings such that the total amount of cable is minimized.
- Interconnect \( N \) pins in an electronic circuit using the least amount of wire.
- Create a highway infrastructure among \( N \) cities that minimizes the total length, such that every city is reachable from any other city.
- \textit{and many others} ...
Problem:

Given: a connected, undirected graph $G = (V, E)$, where each edge $(u, v)$ has a weight $w(u, v)$.

Find: a tree $T \subseteq E$ that connects all the vertices in $V$ such that it has a minimum total weight $w(T) = \sum_{(u,v) \in T} w(u, v)$.
**Trees and Forests**

**Tree:** A tree is a connected, acyclic, undirected graph.

**Forest:** If an undirected graph is acyclic, but possibly disconnected, is it a forest.

![a tree](image1.png)  ![a forest](image2.png)
If $G = (V, E)$ be an undirected graph, the following statements are equivalent:

1. $G$ is a tree;

2. Any two vertices in $G$ are connected by a unique simple path;

3. $G$ is connected, but if any edge is removed from $E$, the resulting graph is disconnected;

4. $G$ is connected, and $|E| = |V| - 1$;

5. $G$ is acyclic, and $|E| = |V| - 1$;

6. $G$ is acyclic, but if any edge is added to $E$, the resulting graph contains a cycle.
Minimum Spanning Trees

- Definition: Let $G(V, E)$ be any undirected graph, $T(V, E')$ is said to be a **spanning tree** of $G(V, E)$ if $E' \subseteq E$ and $T(V, E')$ is a tree.

- **Problem**: given a connected, undirected weighted graph, find a **spanning tree** using edges that minimize the total weight.

- The weight of a spanning tree is the sum of the edge weights.

- **Input**: An undirected graph $G(V, E)$ where each edge has a weight associated.

- **Output**: A minimum weight spanning tree of $G$. 
Which edges form the minimum spanning tree (MST) of the graph below?
Is the MST of a graph unique?
No, a graph can have more than one MSTs!
MSTs satisfy the **optimal substructure** property: an optimal (minimum spanning) tree is composed of optimal (MS) subtrees.

- Let $T$ be an MST of $G$, and an edge $(u, v) \in T$.
- Removing $(u, v)$ partitions $T$ into two trees $T_1$ and $T_2$.
- Claim: $T_1$ is an MST of $G_1 = (V_1, E_1)$ and $T_2$ is an MST of $G_2 = (V_2, E_2)$. (Do $V_1$ and $V_2$ share vertices? Why?)
- Proof (**cut and paste**): $w(T) = w(u, v) + w(T_1) + w(T_2)$
  (there cannot be a better tree than $T_1$ or $T_2$, otherwise, using **cut and paste**, we would get a spanning tree $T'$ with smaller total weight than $T$)
Idea of solving the MST problem: grow a MST

**General Idea:** Grow a minimum spanning tree – prior to each iteration, keep \( A \) as a subset of edges from a minimum spanning tree.

**Generic-MST** \((G, w)\)

\[
A := \emptyset \\
\text{while } A \text{ does not form a spanning tree} \\
\quad \text{find an edge } (u, v) \text{ that is } \mathbf{safe} \text{ for } A; \\
\quad A := A \cup (u, v)); \\
\]

return \( A \);

**safe** means \( A \cup \{(u, v)\} \) is also a subset of certain MST
What kind of edges are safe?

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 \(u \in S\) and \(v \in V - S\), or vice versa.

- 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.
Theorem 23.1

- Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$.
- Let $T$ be a MST of $G$, and let $A$ be a subset of edges s.t. $A \subseteq T$.
- Let $(S, V - S)$ be a cut of $G$ that respects $A$.
- Let $(u, v)$ be a light edge crossing the cut $(S, V - S)$.
- Then $(u, v)$ is safe for $A$ (i.e., $A \cup (u, v)$ will be a subset of a MST).
Theorem 23.1

Proof: Let $T$ be a MST that includes $A$, and assume that $T$ does not contain the min-weight edge $(u, v)$, since if it does, we are done.

1. Construct another MST $T'$ that includes $A \cup \{(u, v)\}$.
   
   $$T' = T - \{(x, y)\} \cup \{u, v\} \text{ (Figure 23.3)}$$

2. $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. But $T$ is a MST, so $w(T) \leq w(T')$; thus, $T'$ must be a MST also.

3. Since $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$; thus $A \cup \{(u, v)\} \subseteq T'$

Since $T'$ is a MST, $(u, v)$ is safe for $A$. 
Figure 23.3
Corollary 23.2

- Let $A$ be a subset of an MST.
- Let $G_A = (V, A)$ be the forest induced by $A$.
- Let $C = (V_C, E_C)$ be a tree in the forest $G_A$.
- If $(u, v)$ is a light edge connecting $C$ to some other tree in $A$, then $(u, v)$ is safe for $A$ (i.e., $A \cup (u, v)$ will be a subset of a MST).
Two algorithms for MST: Two different schemes of maintaining $A$

Based on different approaches of maintaining $A$, we have two algorithms: **Kruskal’s algorithm** and **Prim’s algorithm**:

- Kruskal’s algorithm keeps the set $A$ as a forest (a set of disjoint sets).
- Prim’s algorithm grows a single tree $A$. 
**Kruskal’s Algorithm**

**Basic idea:** To grow a sparse forest $A$ into a tree.

- At the beginning, each vertex is considered to be a different tree. $A$ is the forest containing those trees.

**Grow this forest into a tree**

- Sort the edges in nondecreasing order by weight and put them in a list $L$.
- For each edge in $L$, in order:
  
  - Remove the first edge $(u, v)$ from $L$ (i.e. the cheapest edge);
  - If $(u, v)$ connects two trees (i.e., $T_i$ and $T_j$) without introducing any cycle, then grow $T_i$ and $T_j$ into a bigger tree; otherwise discard $(u, v)$.

> $(u, v)$ is **safe** for $A$ by Corollary 23.2.
An example

original graph

MST
\( w(T) = 18 \)
An example, Cont’d
An example, Cont’d

Cycle will be introduced

Cycle will be introduced
**Kruskal’s algorithm: Implement a Forest**

**Q: How to implement a forest?**
**A:** use a disjoint-set data structure to maintain several disjoint sets of elements. Each set represents a tree.

**Q: How to check if a cycle is formed?**
**A:** Each set/tree has a set/tree ID (unique representative). When you try to connect two vertices in the same tree (with the same tree ID), a cycle will be formed.
A simple data structure for Forests (Disjoint sets)

The operations we need to support:

- Find-Set (return the set/tree ID).
- Union (combine two sets/trees into one larger set/tree).
- Make-Set (construct set/tree).

A simple solution: linked lists:

- Maintain elements in same set as a linked list with each element having a pointer to the first element of the list (unique representative).
- Each list maintains pointers head to the representative, and tail to the last object in the list.
Disjoint Sets: Implementation

Sets

\[
\begin{array}{c}
1 & 2 & 3 \\
10 & 6
\end{array}
\]

Representation

\[
\begin{array}{c}
3 \rightarrow 2 \rightarrow 1 \rightarrow 10 \rightarrow 6 \\
8 \rightarrow 5 \rightarrow 4 \rightarrow 12
\end{array}
\]

Union-Set

\[
\begin{array}{c}
3 \rightarrow 2 \rightarrow 1 \rightarrow 10 \rightarrow 6 \\
8 \rightarrow 5 \rightarrow 4 \rightarrow 12
\end{array}
\]
Disjoint Sets: Time complexity

- **Make-Set(v):** make a list with one element
  \( \Rightarrow O(1) \) time.

- **Find-Set(u):** follow pointer and return the unique representative
  \( \Rightarrow O(1) \) time.

- **Union(u, v):** point all the pointers of v’s elements to u’s unique representative
  \( \Rightarrow O(|v|) \) time.
  \( \Rightarrow |V| \) Union operations can take \( \Theta(|V|^2) \) time.
  Can do better, using the **weighted union heuristic**.
Augment the representation:

- Store the length of the list with each list.
- Always append the smaller list onto the longer list.

**Theorem 21.1**

Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of $m$ Make-Set, Union, and Find-Set operations, $n$ of which are Make-Set operations, takes $O(m + n \lg n)$ time.
Kruskal’s Algorithm

MST-Kruskal \((G, w)\)

\[
A := \emptyset; \\
\text{for each vertex } v \in V \\
\quad \text{Make-Set}(v); \quad /* \text{construct trees} */
\]

Sort the edges of \(E\) by weight ;

for each edge \((u, v) \in E\), in order

\[
\text{if Find-Set}(u) \neq \text{Find-Set}(v) \quad /* \text{Not in the same tree} */ \\
\quad A := A \cup \{(u, v)\} \\\n\quad \text{Union-Set}(u, v); \quad /* \text{combine two trees into one} */
\]
Running time for Kruskal’s algorithm

1. Sort: $O(|E| lg |E|)$.

2. $|V|$ Make-Set calls.

3. $2|E|$ Find-Set() calls.

4. $|V| - 1$ Union-Set calls.

Total: $2|E| + 2|V| - 1$ operations on the disjoint sets, $|V|$ of each are Make-Set operations

$\Rightarrow O(2|E| + 2|V| - 1 + |V| lg |V|)$ time complexity, by Theorem 21.1.

Overall, the complexity for Kruskal’s algorithm is:

$O(|E| lg |E|) = O(|E| lg |V|)$. 
**Basic idea:** To grow a single tree $A$ (from a one-node tree to a MST).

- Select an arbitrary vertex to start the tree $A$; Let $V_A$ be the vertices covered by $A$.

- **growing the tree $A$:**
  - each time select an edge $(u, v)$ of minimum weight connecting a vertex in $V_A$ and a vertex outside of $V_A$.
  - include $(u, v)$ into $A$. 
Prim’s algorithm: Implementation

We need to keep a list for the vertices not covered by $A$, and we hope that each time we can efficiently pick the closest one to include. Thus we need a Priority Queue.

**Extra variables:**

- $Q$: a min-priority queue to store the vertices which are not in $V_A$ yet.

- **key**: for each element $v$ (a vertex) in $Q$, there is a field $\text{key}$ to record the minimum weight of any edge connecting $v$ to a vertex in the tree; i.e., $\text{key}[v]$ tells the distance from $v$ to the tree $A$. If no such edge, $\text{key}[v] = \infty$. 
**Prim’s algorithm**

\( \text{MST-PRIM} \ (G(V, E), w, r) \) /* r is the arbitrarily selected starting point */

1. for each \( u \in V \)
   2. \( \text{key}[u] := \infty; \)

3. \( \text{key}[r] := 0; \) /* the first to be picked into \( V_A \) */

4. \( Q := V; \) /* put all vertices into a PQ */

5. while \( Q \) is not empty
   6. \( u := \text{Extract-Min}(Q); \) /* get the vertex which is closest to the tree \( A \), and remove it from the queue */

7. for each \( v \in \text{Adj}[u] \) /* update the dist. to \( A \) */
   8. if \( (v \in Q) \) and \( w(u, v) < \text{key}[v] \)
      9. \( \text{key}[v] := w(u, v) \)
Prim’s algorithm: Complexity

Use a **Binary Heap** to implement the min-priority queue.

**MST-PRIM** \((G(V, E), w, r)\)

1. for each \(u \in V\)
2. \(\text{key}[u] := \infty;\)
3. \(\text{key}[r] := 0;\)
4. \(Q := V;\) — **Build-Min-Heap**: \(O(|V|)\)
5. while \(Q\) is not empty — **Totally execute** \(|V|\) times
6. \(u := \text{Extract-Min}(Q);\) — **Extract-Min**: \(O(lg|V|)\)
7. for each \(v \in \text{Adj}[u]\) — **What about this part?**
8. if \((v \in Q) \text{ and } w(u, v) < \text{key}[v]\)
9. \(\text{key}[v] := w(u, v)\)
Using Binary Heaps

If we use a Heap to implement the min-priority queue:

- Build-Min-Heap (line 4) takes $O(|V|)$.
- **while** loop (line 5) will execute $|V|$ times.
- Extract-Min (line 6) takes $O(lg|V|)$.
- The **for** loop in lines 7 - 9 is executed $O(|E|)$ times altogether, because the sum of the lengths of all adjacency lists is $2|E|$.
- line 8: $O(1)$
- line 9: It’s actually an operation of **Decrease-Key**. With Heap: $O(lg|V|)$.

Overall the complexity for Prim’s algorithm:
$O(|V| + |V|lg|V| + |E|lg|V|) = O(|E|lg|V|)$.