CSE 202: Design and Analysis of Algorithms

Lecture 2

Instructor: Kamalika Chaudhuri
Announcement

• Reminder: Email me the name of your group member by Thursday Jan 19

• Pick up calibration quizzes after class
Greedy Algorithms

- Minimum Spanning Trees
- The Union/Find Data Structure
A Network Design Problem

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.
A Network Design Problem

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

Node = Computer
Edge = Pair of computers
Edge Cost(u,v) = Distance(u,v)
A Network Design Problem

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

- Node = Computer
- Edge = Pair of computers
- Edge Cost(u,v) = Distance(u,v)

Find a subset of edges $T$ such that the cost of $T$ is minimum and all nodes are connected in $(V,T)$.
A Network Design Problem

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

- **Node** = Computer
- **Edge** = Pair of computers
- **Edge Cost**\((u,v)\) = Distance\((u,v)\)

Find a subset of edges \(T\) such that the cost of \(T\) is minimum and all nodes are connected in \((V,T)\)

Can \(U\) contain a cycle?
**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

Node = Computer  
Edge = Pair of computers  
Edge Cost(u,v) = Distance(u,v)

Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T)

Can U contain a cycle?  
Solution is connected and acyclic, so a tree
A connected, undirected and acyclic graph is called a **tree**.
Trees

A connected, undirected and acyclic graph is called a **tree**

Property 1. A tree on \( n \) nodes has exactly \( n - 1 \) edges
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

**Proof.** By induction.
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

**Proof.** By induction.

**Base Case:**
$n$ nodes, no edges,
n connected components
Trees

A connected, undirected and acyclic graph is called a tree

**Property 1.** A tree on \( n \) nodes has exactly \( n - 1 \) edges

**Proof.** By induction.

**Base Case:**
\( n \) nodes, no edges,
\( n \) connected components

**Inductive Case:**
Add edge between two connected components
No cycle created
\#components decreases by 1
Trees

A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

**Proof.** By induction.

**Base Case:**
$n$ nodes, no edges, $n$ connected components

**Inductive Case:**
Add edge between two connected components
No cycle created
#components decreases by 1

**At the end:** 1 component
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on \( n \) nodes has exactly \( n - 1 \) edges

**Proof.** By induction.

**Base Case:**
\( n \) nodes, no edges, 
\( n \) connected components

**Inductive Case:**
Add edge between two connected components
No cycle created
\#components decreases by 1

**At the end:** 1 component
How many edges were added?
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

Is any graph on $n$ nodes and $n - 1$ edges a tree?
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on n nodes has exactly $n - 1$ edges

Is any graph on n nodes and $n - 1$ edges a tree?
Trees

A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on \( n \) nodes has exactly \( n - 1 \) edges

Is any graph on \( n \) nodes and \( n - 1 \) edges a tree?

![Diagram of a tree](image)

**Property 2.** Any **connected**, undirected graph on \( n \) nodes and \( n - 1 \) edges is a tree
A connected, undirected and acyclic graph is called a tree

**Property 1.** A tree on n nodes has exactly n - 1 edges

**Property 2.** Any connected, undirected graph on n nodes and n - 1 edges is a tree

**Proof:** Suppose G is connected, undirected, has some cycles.
A connected, undirected and acyclic graph is called a tree.

**Property 1.** A tree on \( n \) nodes has exactly \( n - 1 \) edges.

**Property 2.** Any connected, undirected graph on \( n \) nodes and \( n - 1 \) edges is a tree.

**Proof:** Suppose \( G \) is connected, undirected, has some cycles. While \( G \) has a cycle, remove an edge from this cycle.
A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on \( n \) nodes has exactly \( n - 1 \) edges

**Property 2.** Any **connected**, undirected graph on \( n \) nodes and \( n - 1 \) edges is a tree

**Proof:** Suppose \( G \) is connected, undirected, has some cycles. While \( G \) has a cycle, remove an edge from this cycle.
Result: \( G' = (V, E') \) where \( E' \) is a tree. So \( |E'| = n - 1 \)
Trees

A connected, undirected and acyclic graph is called a **tree**

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

**Property 2.** Any **connected**, undirected graph on $n$ nodes and $n - 1$ edges is a tree

**Proof:** Suppose $G$ is connected, undirected, has some cycles. While $G$ has a cycle, remove an edge from this cycle. Result: $G' = (V, E')$ where $E'$ is a tree. So $|E'| = n - 1$

Thus, $E = E'$, and $G$ is a tree
Minimum Spanning Trees (MST)

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

- Node = Computer
- Edge = Pair of computers
- Edge Cost(u,v) = Distance(u,v)

Find a subset of edges $T$ such that the cost of $T$ is minimum and all nodes are connected in $(V,T)$

**Goal:** Find a spanning tree $T$ of the graph $G$ with minimum total cost

We’ll see a greedy algorithm to construct $T$
Properties of MSTs

For a cut \((S, V\setminus S)\), the lightest edge in the cut is the minimum cost edge that has one end in \(S\) and the other in \(V\setminus S\).

**Property 1.** A lightest edge in any cut always belongs to an MST
Properties of MSTs

For a cut \((S, V\setminus S)\), the lightest edge in the cut is the minimum cost edge that has one end in \(S\) and the other in \(V\setminus S\).

**Property 1.** A lightest edge in any cut always belongs to an MST

**Proof.** Suppose not.

Let \(e = \) lightest edge in \((S, V\setminus S)\), \(T = MST\), \(e\) is not in \(T\)
Properties of MSTs

For a cut \((S, V\setminus S)\), the lightest edge in the cut is the minimum cost edge that has one end in \(S\) and the other in \(V\setminus S\).

**Property 1.** A lightest edge in any cut always belongs to an MST

**Proof.** Suppose not.

Let \(e = \text{lightest edge in } (S, V\setminus S), T = \text{MST}, e \text{ is not in } T\)

\\(T \cup \{e\}\) has a cycle with edge \(e'\) across \((S, V\setminus S)\)
Properties of MSTs

For a cut $(S, V\setminus S)$, the lightest edge in the cut is the minimum cost edge that has one end in $S$ and the other in $V\setminus S$.

**Property 1.** A lightest edge in any cut always belongs to an MST

**Proof.** Suppose not.

Let $e =$ lightest edge in $(S, V\setminus S), T =$ MST, $e$ is not in $T$

$T \cup \{e\}$ has a cycle with edge $e'$ across $(S, V\setminus S)$

Let $T' = T \setminus \{e'\} \cup \{e\}$
Properties of MSTs

For a cut \((S, V\setminus S)\), the lightest edge in the cut is the minimum cost edge that has one end in \(S\) and the other in \(V\setminus S\).

Property 1. A lightest edge in any cut always belongs to an MST

Proof. Suppose not.

Let \(e = \) lightest edge in \((S, V\setminus S)\), \(T = \) MST, \(e\) is not in \(T\)

\(T \cup \{e\}\) has a cycle with edge \(e'\) across \((S, V\setminus S)\)

Let \(T' = T \setminus \{e'\} \cup \{e\}\)

\[
\text{cost}(T') = \text{cost}(T) + \text{cost}(e) - \text{cost}(e') < \text{cost}(T)
\]
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST.
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle

**Property 2.** The heaviest edge in a cycle never belongs to an MST

**Proof.** Suppose not. Let $T = \text{MST}$, $e =$ heaviest edge in some cycle, $e \in T$

![Diagram](image)
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle

**Property 2.** The heaviest edge in a cycle never belongs to an MST

**Proof.** Suppose not. Let \( T = \text{MST} \), \( e = \text{heaviest edge in some cycle, } e \in T \)
Delete \( e \) from \( T \) to get subtrees \( T_1 \) and \( T_2 \)
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle

Property 2. The heaviest edge in a cycle never belongs to an MST

Proof. Suppose not. Let $T = \text{MST}$, $e = \text{heaviest edge in some cycle}$, $e \in T$
Delete $e$ from $T$ to get subtrees $T_1$ and $T_2$
Let $e' = \text{lightest edge in the cut } (T_1, V \setminus T_1)$
Then, $\text{cost}(e') < \text{cost}(e)$
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle

**Property 2.** The heaviest edge in a cycle never belongs to an MST

**Proof.** Suppose not. Let $T =$ MST, $e =$ heaviest edge in some cycle, $e$ in $T$
Delete $e$ from $T$ to get subtrees $T_1$ and $T_2$
Let $e' =$ lightest edge in the cut $(T_1, V \setminus T_1)$
Then, $\text{cost}(e') < \text{cost}(e)$
Let $T' = T \setminus \{e\} + \{e'\}$
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle

**Property 2.** The heaviest edge in a cycle never belongs to an MST

**Proof.** Suppose not. Let \( T = \text{MST}, e = \text{heaviest edge in some cycle}, e \in T \)
Delete \( e \) from \( T \) to get subtrees \( T_1 \) and \( T_2 \)
Let \( e' = \text{lightest edge in the cut } (T_1, V \setminus T_1) \)
Then, \( \text{cost}(e') < \text{cost}(e) \)
Let \( T' = T \setminus \{e\} + \{e'\} \)
\( \text{cost}(T') = \text{cost}(T) + \text{cost}(e) - \text{cost}(e') < \text{cost}(T) \)
A Generic MST Algorithm

\[ X = \{ \} \]

While there is a cut \((S, V\setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V\setminus S) \]

Does this output a tree?
A Generic MST Algorithm

\[
X = \{ \}
\]
While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[
X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S)
\]

Does this output a tree?
- At each step, no cycle is created
- Continues while there are disconnected components

Why does this produce a MST?
A Generic MST Algorithm

\[ X = \{ \} \]

While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

**Proof of correctness by induction.**

**Base Case:** At \(t=0\), \(X\) is in some MST \(T\)
A Generic MST Algorithm

X = {}  
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S)

Proof of correctness by induction.
Base Case: At t=0, X is in some MST T
Induction: Assume at t=k, X is in some MST T
A Generic MST Algorithm

\[ X = \{ \} \]
While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

Proof of correctness by induction.

**Base Case:** At \(t=0\), \(X\) is in some MST \(T\)

**Induction:** Assume at \(t=k\), \(X\) is in some MST \(T\)
Suppose we add \(e\) to \(X\) at \(t=k+1\)
A Generic MST Algorithm

\[ X = \{ \} \]
While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

Proof of correctness by induction.

Base Case: At \(t=0\), \(X\) is in some MST \(T\)

Induction: Assume at \(t=k\), \(X\) is in some MST \(T\)
Suppose we add \(e\) to \(X\) at \(t=k+1\)
Suppose \(e\) is not in \(T\). Adding \(e\) to \(T\) forms a cycle \(C\)
A Generic MST Algorithm

\[ X = \{ \} \]
While there is a cut \((S, \mathcal{V} \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, \mathcal{V} \setminus S) \]

**Proof of correctness by induction.**

**Base Case:** At \(t=0\), \(X\) is in some MST \(T\)

**Induction:** Assume at \(t=k\), \(X\) is in some MST \(T\)
Suppose we add \(e\) to \(X\) at \(t=k+1\)
Suppose \(e\) is not in \(T\). Adding \(e\) to \(T\) forms a cycle \(C\)
Let \(e' = \) another edge in \(C\) across \((S, \mathcal{V} \setminus S)\), \(T' = T \setminus \{e'\} \cup \{e\} \)
A Generic MST Algorithm

\[ X = \{ \} \]

While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

**Proof of correctness by induction.**

**Base Case:** At \(t=0\), \(X\) is in some MST \(T\)

**Induction:** Assume at \(t=k\), \(X\) is in some MST \(T\)
Suppose we add \(e\) to \(X\) at \(t=k+1\)
Suppose \(e\) is not in \(T\). Adding \(e\) to \(T\) forms a cycle \(C\)
Let \(e' = \) another edge in \(C\) across \((S, V \setminus S)\), \(T' = T \setminus \{e'\} \cup \{e\}\)
Cost \((T') = \text{cost}(T) + \text{cost}(e') - \text{cost}(e) \leq \text{cost}(T)\)
A Generic MST Algorithm

\[ X = \{ \} \]
While there is a cut \( (S, V \setminus S) \) s.t. \( X \) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

**Proof of correctness by induction.**

**Base Case:** At \( t=0 \), \( X \) is in some MST \( T \)

**Induction:** Assume at \( t=k \), \( X \) is in some MST \( T \)
Suppose we add \( e \) to \( X \) at \( t=k+1 \)
Suppose \( e \) is not in \( T \). Adding \( e \) to \( T \) forms a cycle \( C \)
Let \( e' = \) another edge in \( C \) across \( (S, V \setminus S) \), \( T' = T \setminus \{e'\} \cup \{e\} \)
\[ \text{cost}(T') = \text{cost}(T) + \text{cost}(e') - \text{cost}(e) \leq \text{cost}(T) \]
Thus, \( T' \) is a MST that contains \( X \)
Kruskal’s Algorithm

\[ X = \{ \} \]
For each edge \( e \) in \textit{increasing order} of weight:
   If the end-points of \( e \) lie in different components in \( X \),
   Add \( e \) to \( X \)

Why does this work \texttt{correctly}?
Kruskal’s Algorithm

\[ X = \{ \} \]
For each edge \( e \) in \textit{increasing order} of weight:
   If the end-points of \( e \) lie in different components in \( X \),
   Add \( e \) to \( X \)

Why does this work \textit{correctly}?

**Efficient Implementation:** Need a data structure with properties:
   - Maintain disjoint sets of nodes
   - Merge sets of nodes (union)
   - Find if two nodes are in the same set (find)

The Union-Find data structure
The Union-Find Data Structure

**procedure** makeset(x)

p[x] = x
rank[x] = 0

**procedure** find(x)

if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]

**procedure** union(x,y)

rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
The Union-Find Data Structure

**procedure** makeset(x)
\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure** find(x)
if \( x \neq p[x] \):
\[ p[x] = \text{find}(p[x]) \]
return \( p[x] \)

**procedure** union(x,y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
\[ p[\text{rooty}] = \text{rootx} \]
else:
\[ p[\text{rootx}] = \text{rooty} \]
if rank[rootx] = rank[rooty]:
\[ \text{rank}[\text{rooty}]++ \]
The Union-Find Data Structure

```plaintext
procedure makeset(x)
p[x] = x
rank[x] = 0

procedure find(x)
if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]

procedure union(x,y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
```

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h),
union(f, g), union(b, c), union(h, d), find(e)
The Union-Find Data Structure

**procedure** makeset(x)
\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure** find(x)
\[ \text{if } x \neq p[x]: \]
\[ p[x] = \text{find}(p[x]) \]
\[ \text{return } p[x] \]

**procedure** union(x,y)
\[ \text{root}_x = \text{find}(x) \]
\[ \text{root}_y = \text{find}(y) \]
\[ \text{if } \text{root}_x = \text{root}_y: \text{ return } \]
\[ \text{if } \text{rank}[\text{root}_x] > \text{rank}[\text{root}_y]: \]
\[ p[\text{root}_y] = \text{root}_x \]
\[ \text{else:} \]
\[ p[\text{root}_x] = \text{root}_y \]
\[ \text{if } \text{rank}[\text{root}_x] = \text{rank}[\text{root}_y]: \]
\[ \text{rank}[\text{root}_y]++ \]
The Union-Find Data Structure

**procedure** makeset(x)
\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure** find(x)
if \( x \neq p[x] \):
\[ p[x] = \text{find}(p[x]) \]
return \( p[x] \)

**procedure** union(x, y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
\[ p[rooty] = rootx \]
else:
\[ p[rootx] = rooty \]
if rank[rootx] = rank[rooty]:
\[ \text{rank}[rooty]++ \]
The Union-Find Data Structure

**procedure** makeset(x)

\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure** find(x)

\[ \text{if } x \neq p[x]: \]
\[ p[x] = \text{find}(p[x]) \]
\[ \text{return } p[x] \]

**procedure** union(x, y)

\[ \text{rootx} = \text{find}(x) \]
\[ \text{rooty} = \text{find}(y) \]
\[ \text{if rootx = rooty: return} \]
\[ \text{if rank[rootx] > rank[rooty]:} \]
\[ p[\text{rooty}] = \text{rootx} \]
\[ \text{else:} \]
\[ p[\text{rootx}] = \text{rooty} \]
\[ \text{if rank[rootx] = rank[rooty]:} \]
\[ \text{rank[rooty]}++ \]

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h), union(f, g), union(b, c), union(h, d), find(e)
The Union-Find Data Structure

**procedure makeset(x)**

```plaintext
p[x] = x
rank[x] = 0
```

**procedure find(x)**

```plaintext
if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]
```

**procedure union(x, y)**

```plaintext
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
    rank[rooty]++
```
The Union-Find Data Structure

**procedure makeset(x)**

```
p[x] = x
rank[x] = 0
```

**procedure find(x)**

```
if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]
```

**procedure union(x,y)**

```
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
```
The Union-Find Data Structure

procedure makeset(x)
  p[x] = x
  rank[x] = 0

procedure find(x)
  if x ≠ p[x]:
    p[x] = find(p[x])
  return p[x]

procedure union(x,y)
  rootx = find(x)
  rooty = find(y)
  if rootx = rooty: return
  if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
  else:
    p[rootx] = rooty
  if rank[rootx] = rank[rooty]:
    rank[rooty]++

**Fact 1:** Total time for m find operations = O((m+n) log*n)

**Fact 2:** Time for each union operation = O(1) + Time(find)

**Fact 3:** Total time for m find and n union ops = O((m+n)log* n)
The Union-Find Data Structure

**Property 1:** If x is not a root, then rank[p[x]] > rank[x]

**Proof:** By property of union

```plaintext
procedure makeset(x)
p[x] = x
rank[x] = 0

procedure find(x)
if x \neq p[x]:
    p[x] = find(p[x])
return p[x]

procedure union(x,y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
```
The Union-Find Data Structure

**Property 1:** If \( x \) is not a root, then \( \text{rank}[p[x]] > \text{rank}[x] \)

**Proof:** By property of union

**Property 2:** For root \( x \), if \( \text{rank}[x] = k \), then subtree at \( x \) has size \( \geq 2^k \)

**Proof:** By induction

---

**Procedure makeset**\( (x) \)

\[
\begin{align*}
p[x] &= x \\
\text{rank}[x] &= 0
\end{align*}
\]

**Procedure find**\( (x) \)

\[
\begin{align*}
\text{if } x \neq p[x]: \\
p[x] &= \text{find}(p[x]) \\
\text{return } p[x]
\end{align*}
\]

**Procedure union**\( (x,y) \)

\[
\begin{align*}
\text{root}_x &= \text{find}(x) \\
\text{root}_y &= \text{find}(y) \\
\text{if } \text{root}_x = \text{root}_y: & \text{ return } \\
\text{if } \text{rank}[	ext{root}_x] > \text{rank}[	ext{root}_y]: \\
p[\text{root}_y] &= \text{root}_x \\
\text{else: } \\
p[\text{root}_x] &= \text{root}_y \\
\text{if } \text{rank}[	ext{root}_x] = \text{rank}[	ext{root}_y]: \\
\text{rank}[	ext{root}_y] &= +
\end{align*}
\]
The Union-Find Data Structure

Property 1: If x is not a root, then rank[p[x]] > rank[x]

Proof: By property of union

Property 2: For root x, if rank[x] = k, then subtree at x has size >= 2^k

Proof: By induction

Property 3: There are at most n/2^k nodes of rank k

Proof: Combining properties 1 and 2

procedure makeset(x)
  p[x] = x
  rank[x] = 0

procedure find(x)
  if x ≠ p[x]:
    p[x] = find(p[x])
  return p[x]

procedure union(x,y)
  rootx = find(x)
  rooty = find(y)
  if rootx = rooty: return
  if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
  else:
    p[rootx] = rooty
  if rank[rootx] = rank[rooty]:
    rank[rooty]++