# CS 600.226: Data Structures

Michael Schatz

Dec 5, 2018

Lecture 38: Union-Find

# Assignment 10: The Streets of Baltimore

Out on: November 30, 2018
Due by: December 7, 2018 before 10:00 pm
Collaboration: None
Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

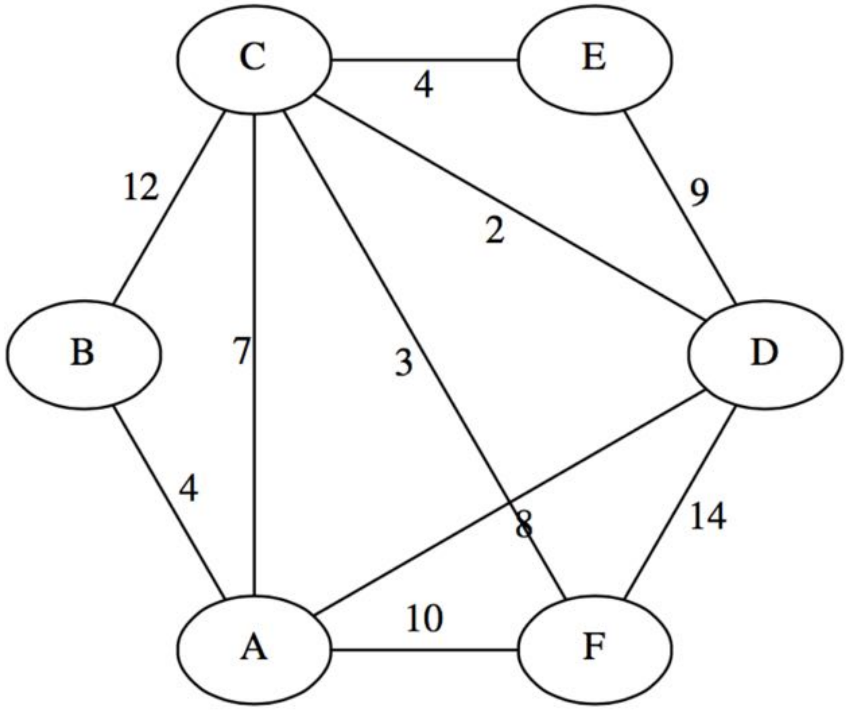
## Overview

The tenth assignment returns to our study of graphs, although this time we are using a weighted graph rather than the unweighted movie/movie-star graph. Specifically, you will be touring the streets of Baltimore to find the shortest route from the JHU campus to other destinations around Baltimore.

***Remember: javac –Xlint:all & checkstyle *.java & Junit***
***(No JayBee)***

# Part 1: Minimum Spanning Trees

# Long Distance Calling



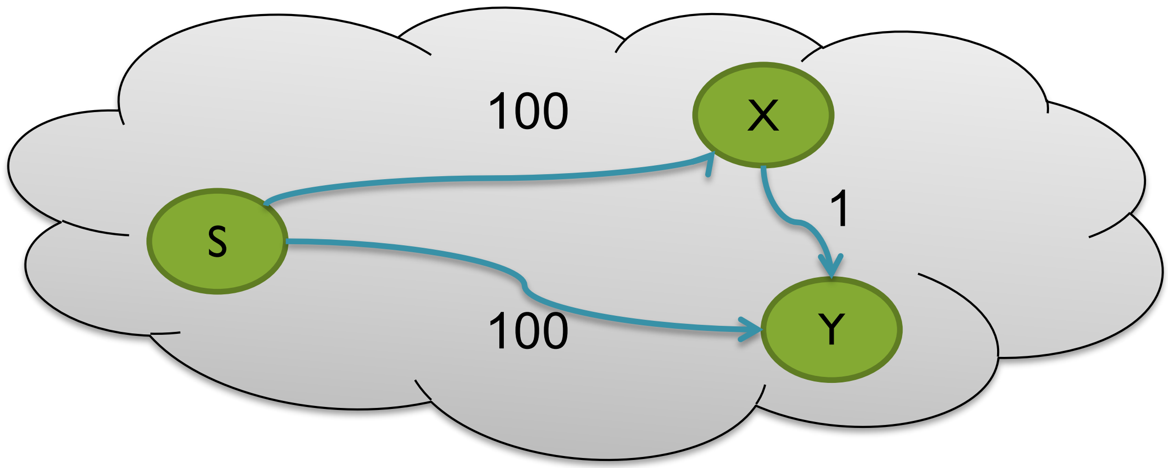

Supposes it costs different amounts of money to send data between cities A through F. Find the least expensive set of connections so that anyone can send data to anyone else.

Given an undirected graph with weights on the edges, find the subset of edges that (a) connects all of the vertices of the graph and (b) has minimum total costs

This subset of edges is called the minimum spanning tree (MST)

Removing an edge from MST disconnects the graph, adding one forms a cycle

# Dijkstra's != MST




Dijkstra's will build the tree S->X, S->Y
(tree visits every node with shortest paths from S to every other node)

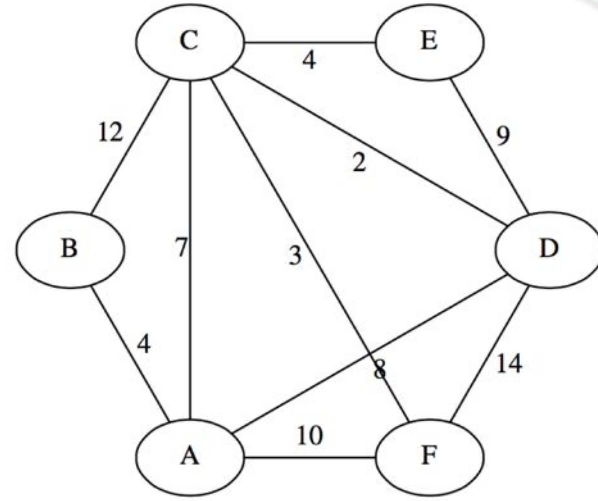
but the MST is S->X, X->Y
(tree visits every node and minimizes the sum of the edges)

# Prim’s Algorithm

1. ***Pick an arbitrary starting vertex and add it to set T. Add all of the other vertices to set R***

Every vertex will be included eventually, so doesn’t matter where you start

T={A}   R={B,C,D,E,F}

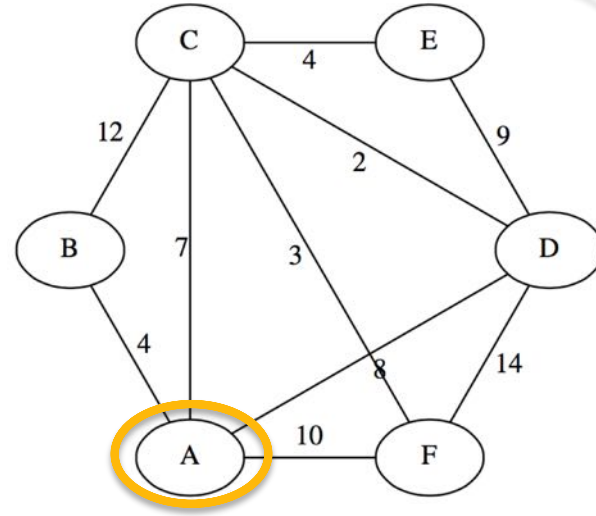


# Prim's Algorithm

1. ***Pick an arbitrary starting vertex and add it to set T. Add all of the other vertices to set R***

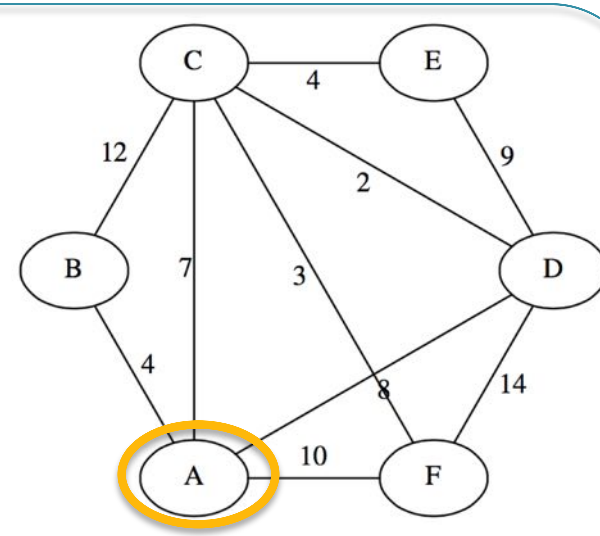
Every vertex will be included eventually, so doesn't matter where you start

T={A}   R={B,C,D,E,F}




***2. While R is not empty, pick an edge from T to R with minimum cost***

A-B: 4   <- pick me
A-C: 7
A-D: 8
A-F: 10

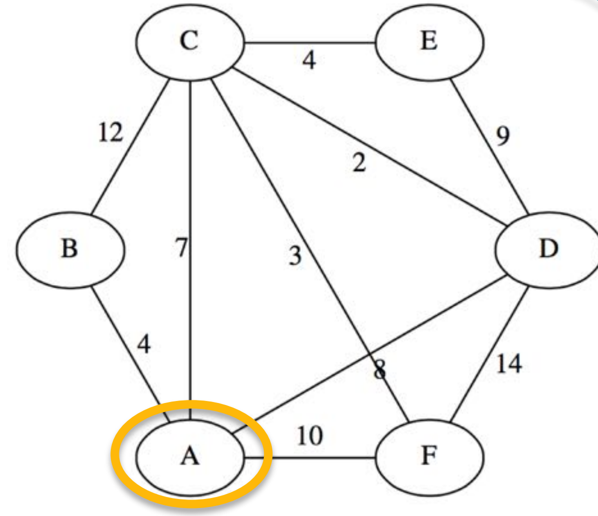


# Prim's Algorithm

1. ***Pick an arbitrary starting vertex and add it to set T. Add all of the other vertices to set R***

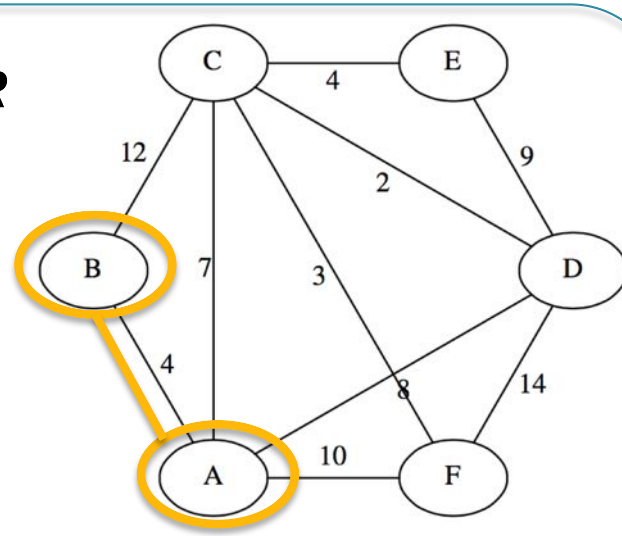
Every vertex will be included eventually, so doesn't matter where you start

T={A}   R={B,C,D,E,F}




***2. While R is not empty, pick an edge from T to R with minimum cost***

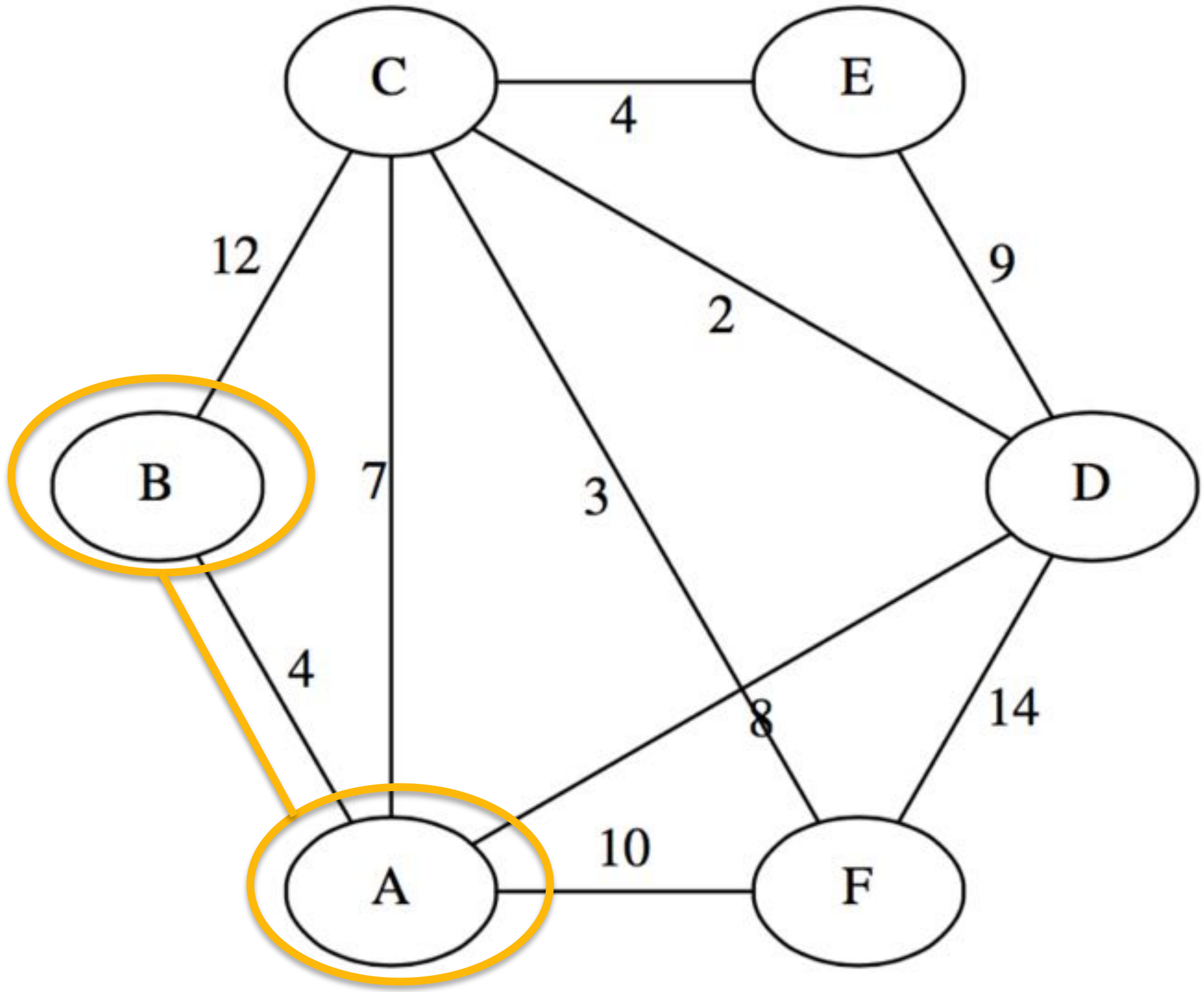
A-B: 4   <- pick me
A-C: 7
A-D: 8
A-F: 10



# Prim's Algorithm

***3. Repeat!***

A-B

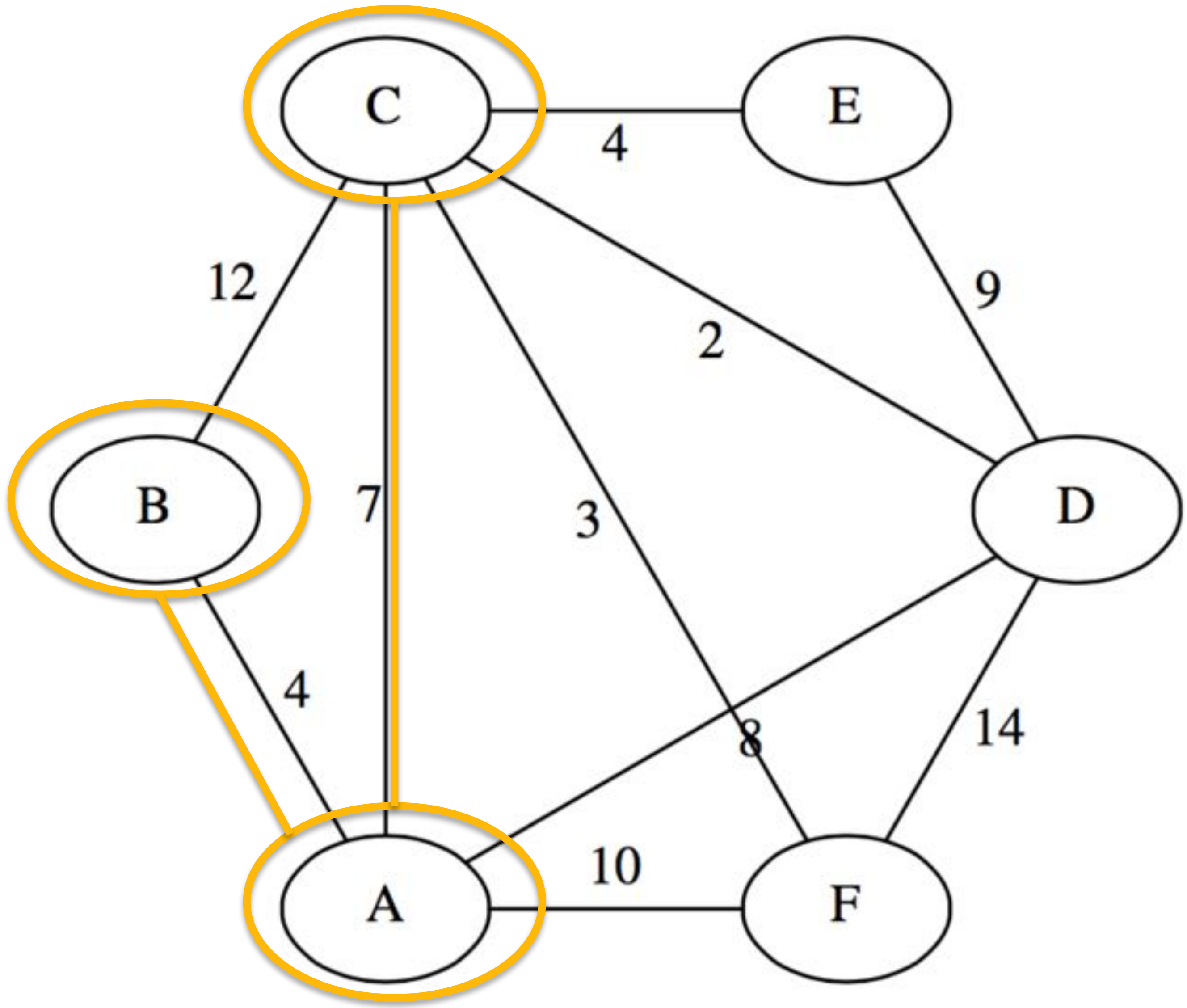


# Prim's Algorithm

***3. Repeat!***

A-B

A-C



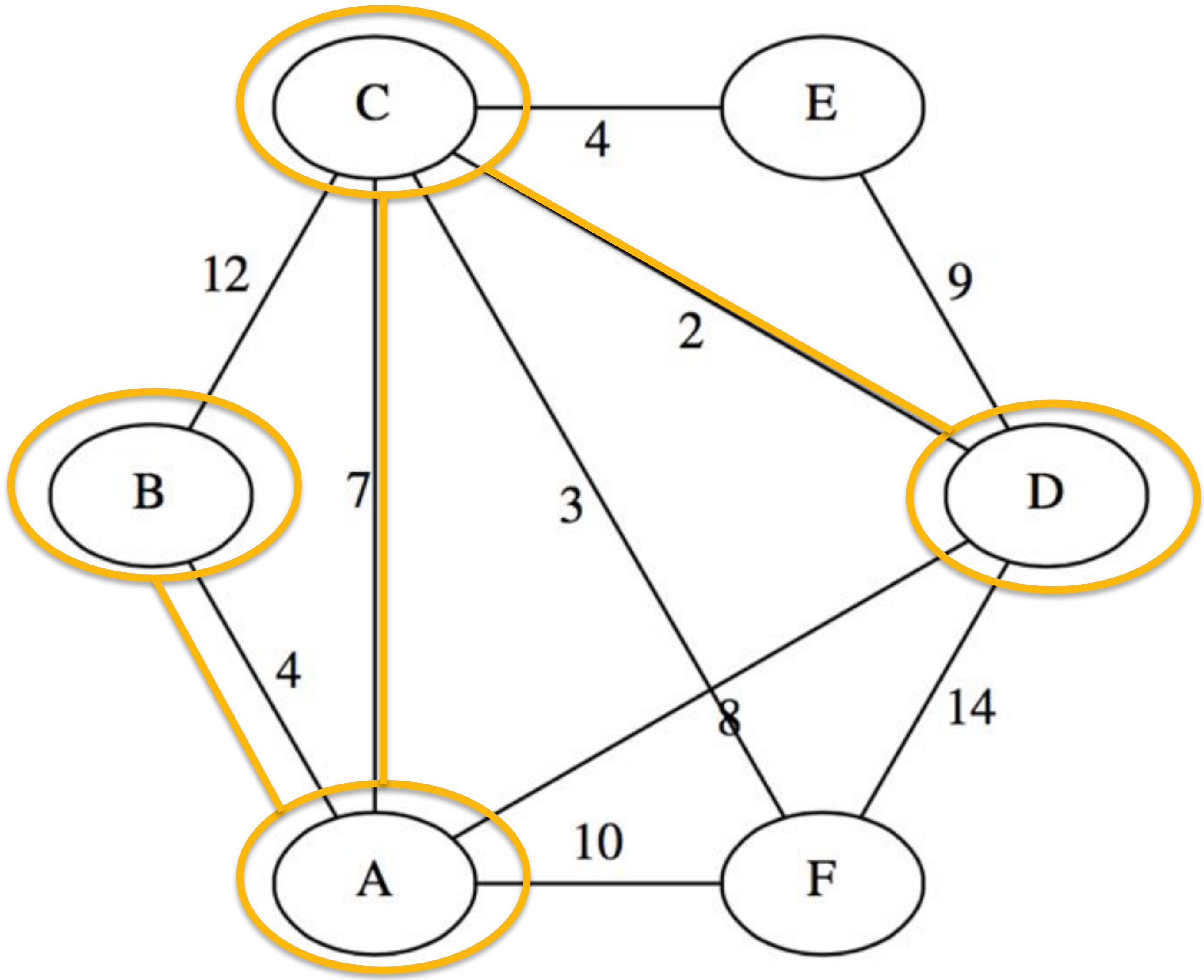
# Prim's Algorithm

***3. Repeat!***

A-B

A-C

C-D



# Prim's Algorithm

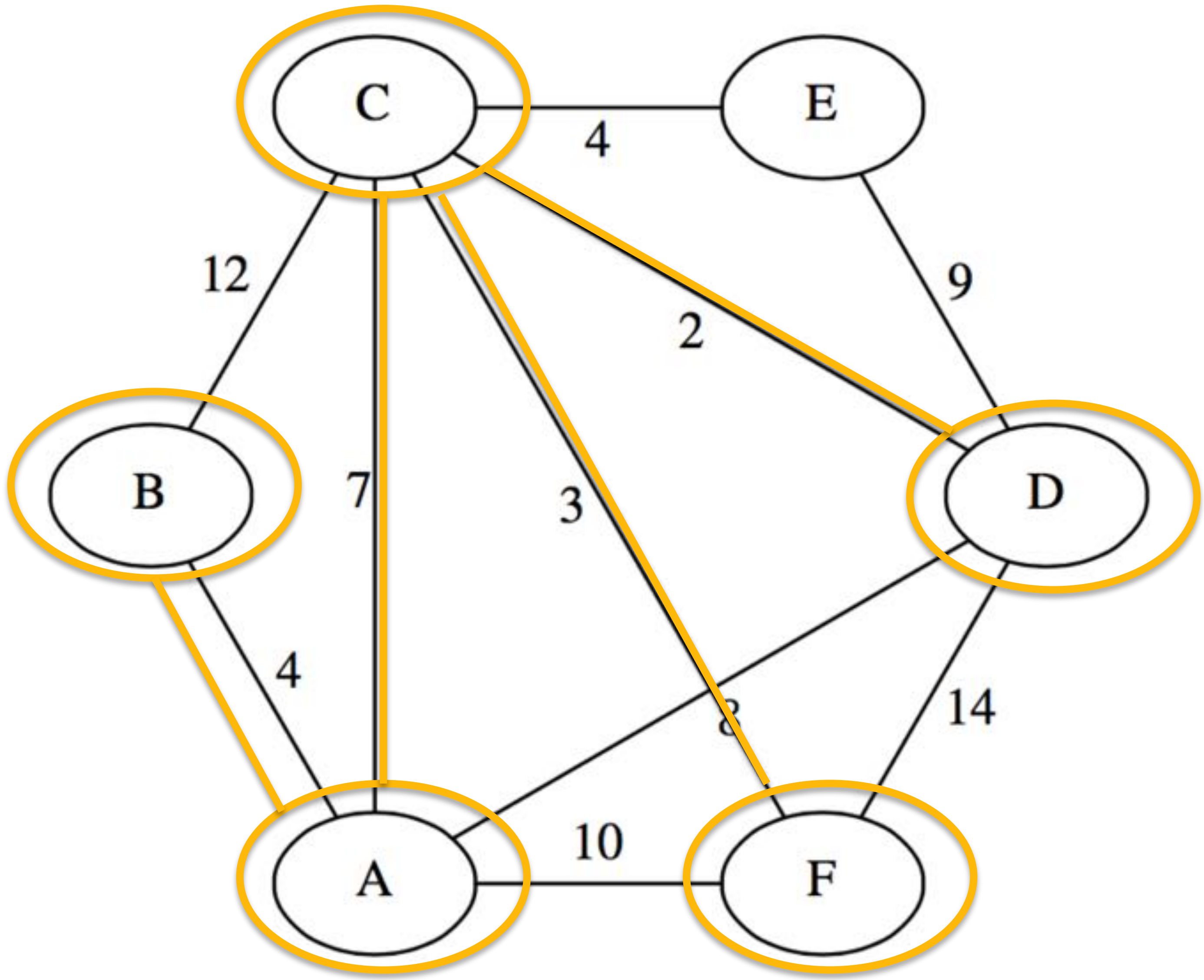
***3. Repeat!***

A-B

A-C

C-D

C-F



# Prim's Algorithm

***3. Repeat!***

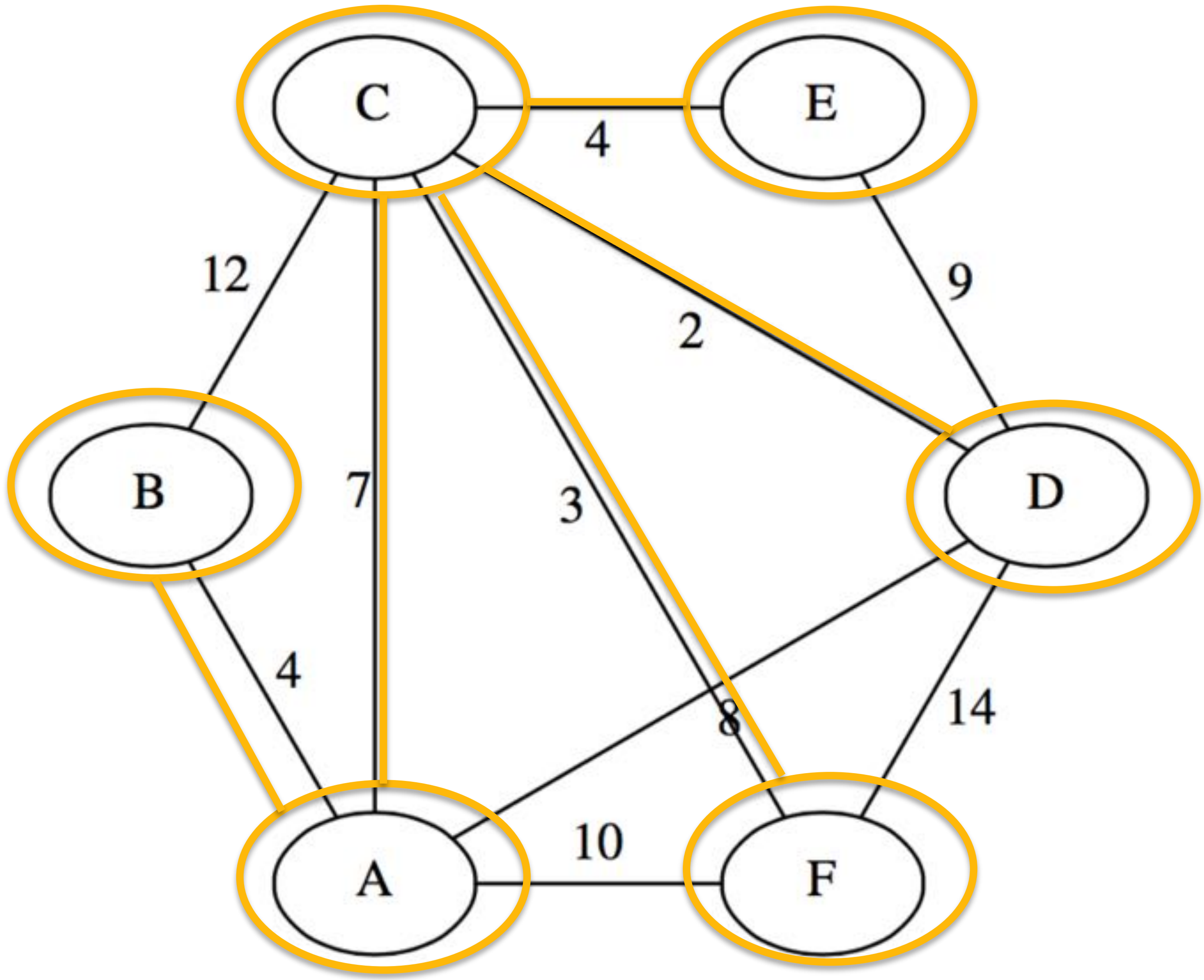
A-B

A-C

C-D

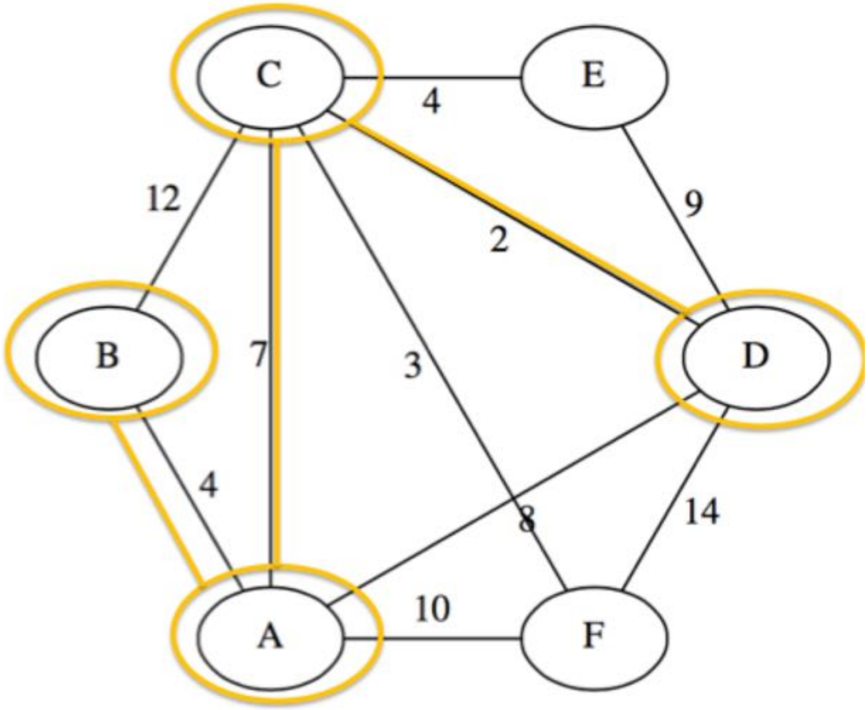
C-F

C-E




***By making a set of simple local choices, it finds the overall best solution***
***The greedy algorithm is optimal ☺***

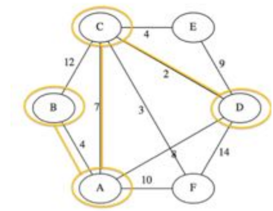
# Prim's Algorithm




***Prim's Algorithm Sketch***

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge:
   - Of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and add it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

# Prim’s Algorithm



***Prim’s Pseudo-code***

1. For each vertex v,
   - Compute C[v] (the cheapest cost of a connection to v from the MST) and an edge E[v] (the edge providing that cheapest connection).
   - Initialize C[v] to ∞ and E[v] to null indicating that there is no edge connecting v to the MST
2. Initialize an empty forest F (set of trees) and a set Q of vertices that have not yet been included in F (initially, all vertices).
3. **Repeat the following steps until Q is empty:**
   1. **Find and remove a vertex v from Q with the minimum value of C[v]**
   2. Add v to F and, if E[v] is not null, also add E[v] to F
   3. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has smaller weight than C[w], perform the following steps:
      1. Set C[w] to the cost of edge vw
      2. Set E[w] to point to edge vw.
4. Return F

How fast is the naïve version? $O(|V|^2)$

What data structures do we need to make it fast? HEAP

# Prim’s Algorithm

***Faster Prim’s Pseudo-code***

1. Add the cost of ***all of the edges*** to a heap
2. Repeatedly pick the next smallest edge (u,v)
    - If u or v is not already in the MST, add the edge uv to the MST

How fast is this version

O(|E| lg |E|)

***Fastest Prim’s Pseudo-code***

1. Add ***all of the vertices*** to a min-priority-queue prioritized by the min edge cost to be added to the MST. Initialize (key=v, dist=∞, edge=<>)
2. Repeatedly pick the next closest vertex v from the MPQ
    1. Add v to the MST using the recorded edge
    2. For each edge (v, u)
        - If cost(v,u) < MPQ(u)
            - MPQ.decreasePriority(u, cost(v,u), (v,u))

How fast is this version

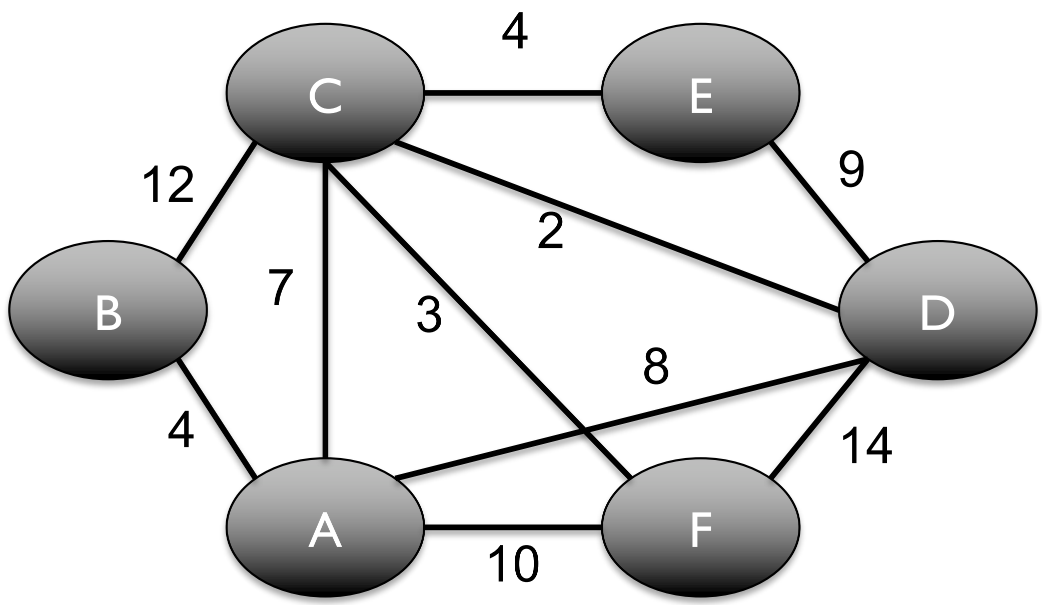
O(|E| lg |V|)

Using Fibonacci Heap

O(|E| + |V| lg |V|)

# Part 2: Kruskal's Algorithm and Union Find

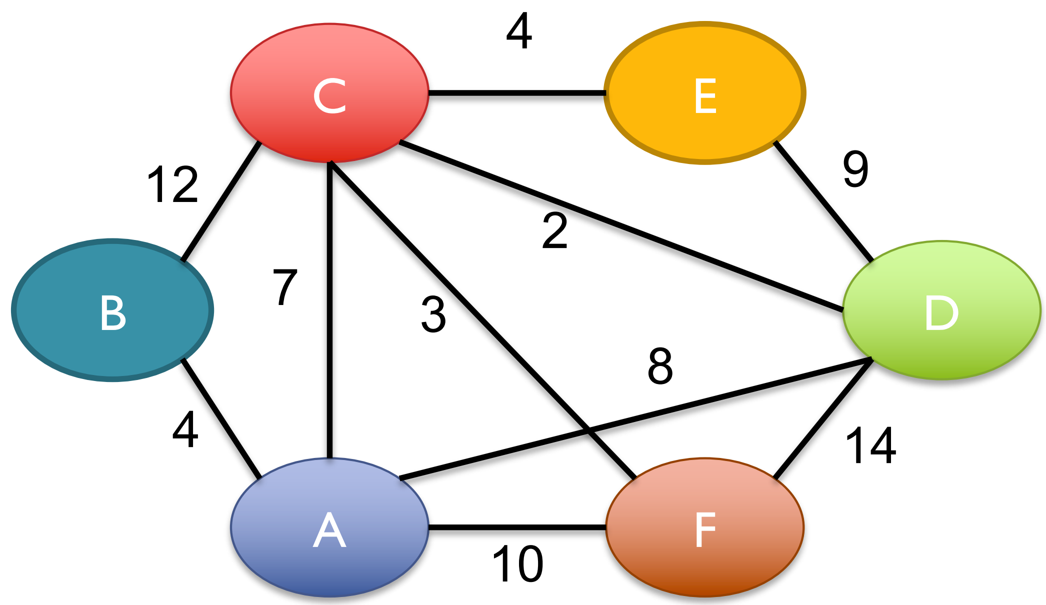
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
   1. remove an edge with minimum weight from S
   2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

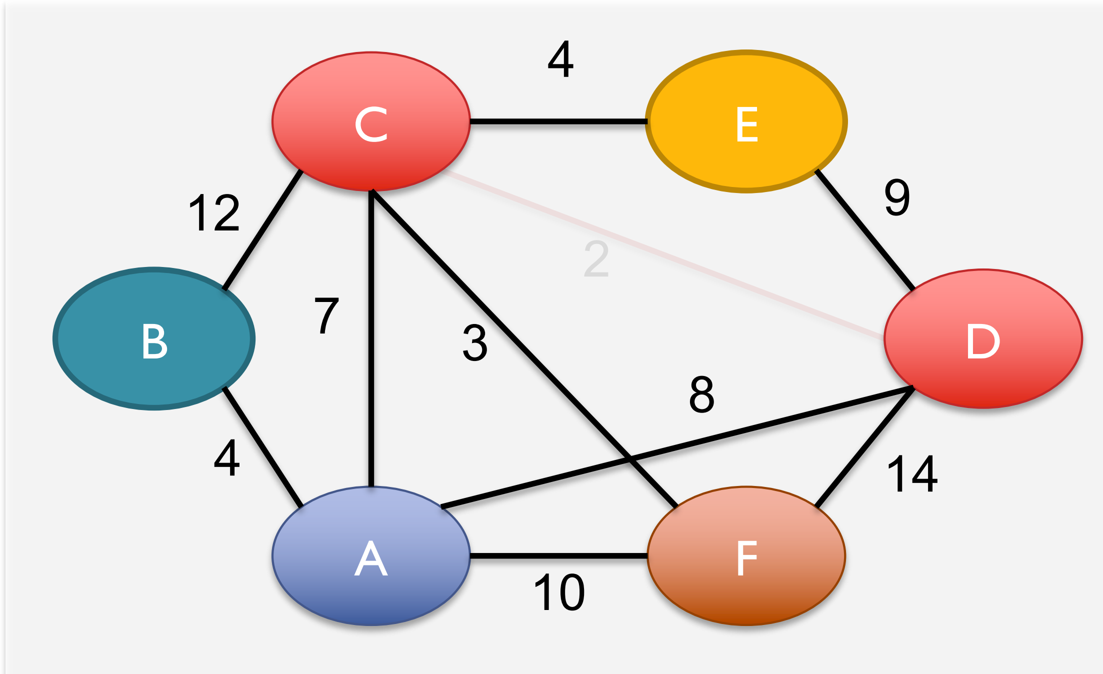
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

# Kruskal's Algorithm

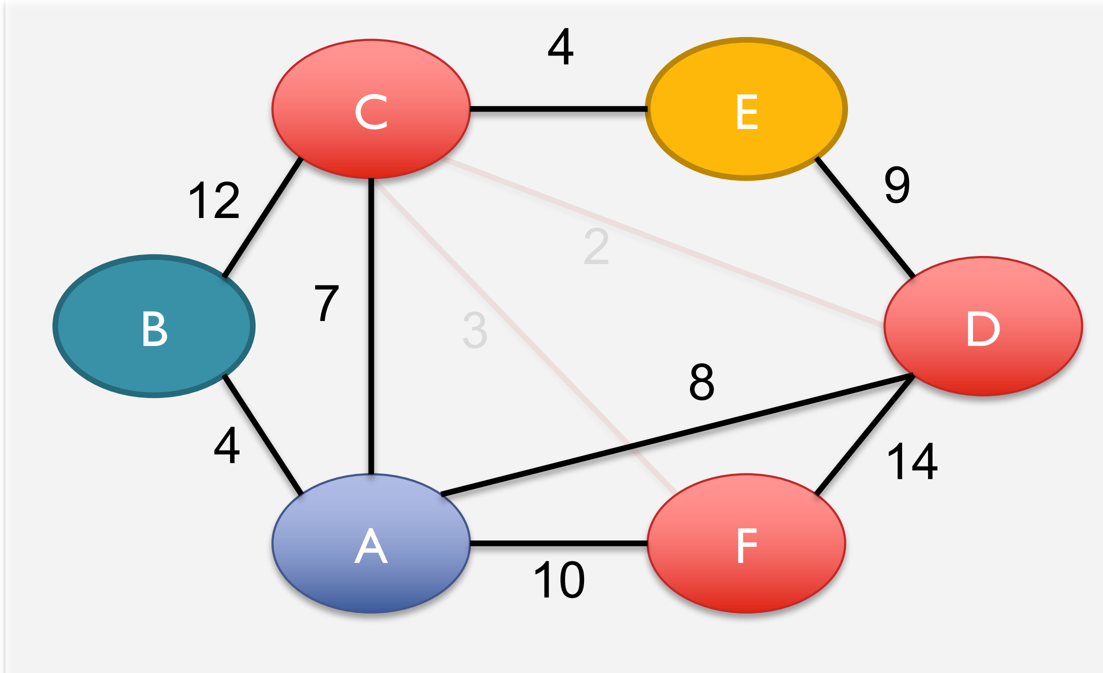



Tinting is just to make it easier to look at

***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
   1. remove an edge with minimum weight from S
   2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

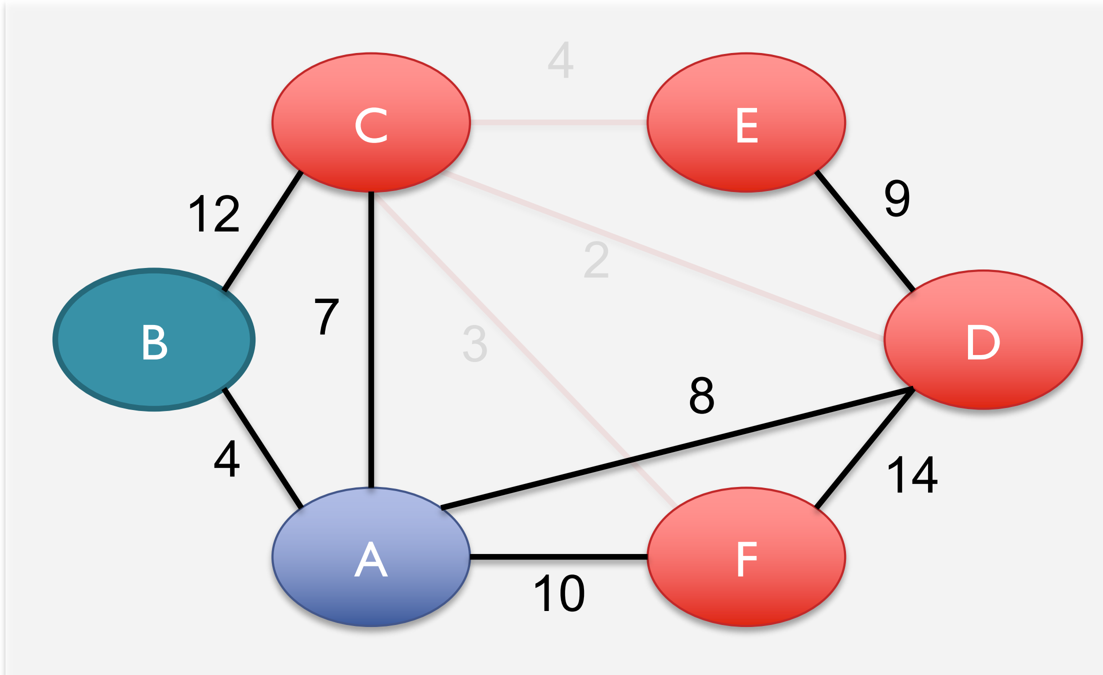
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
   1. remove an edge with minimum weight from S
   2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

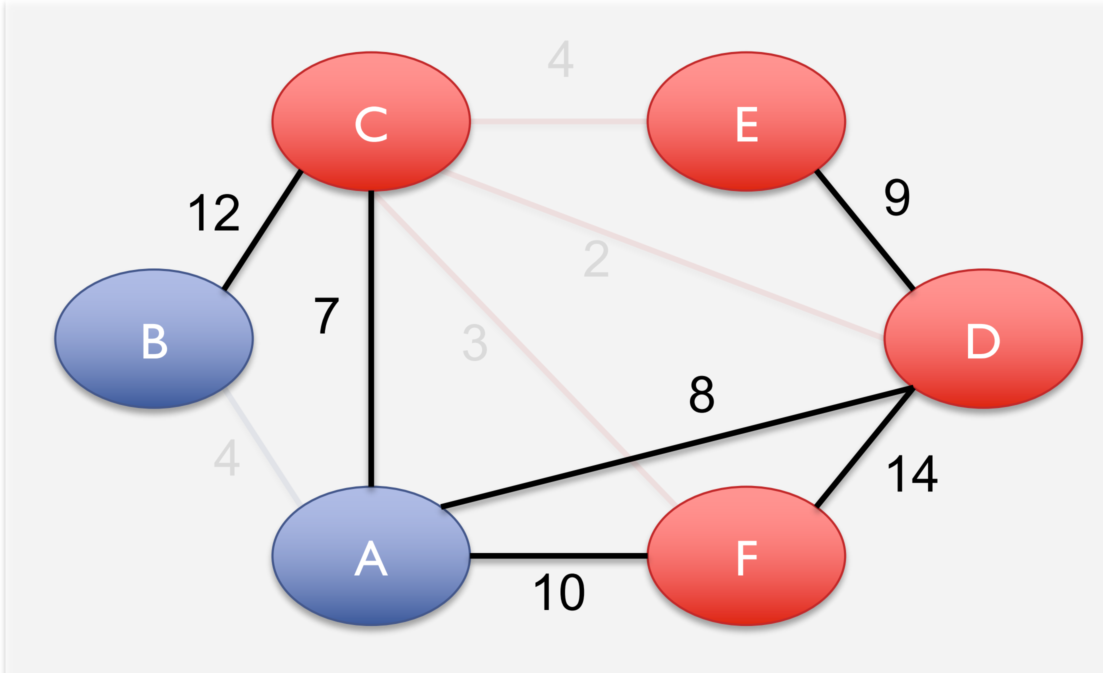
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

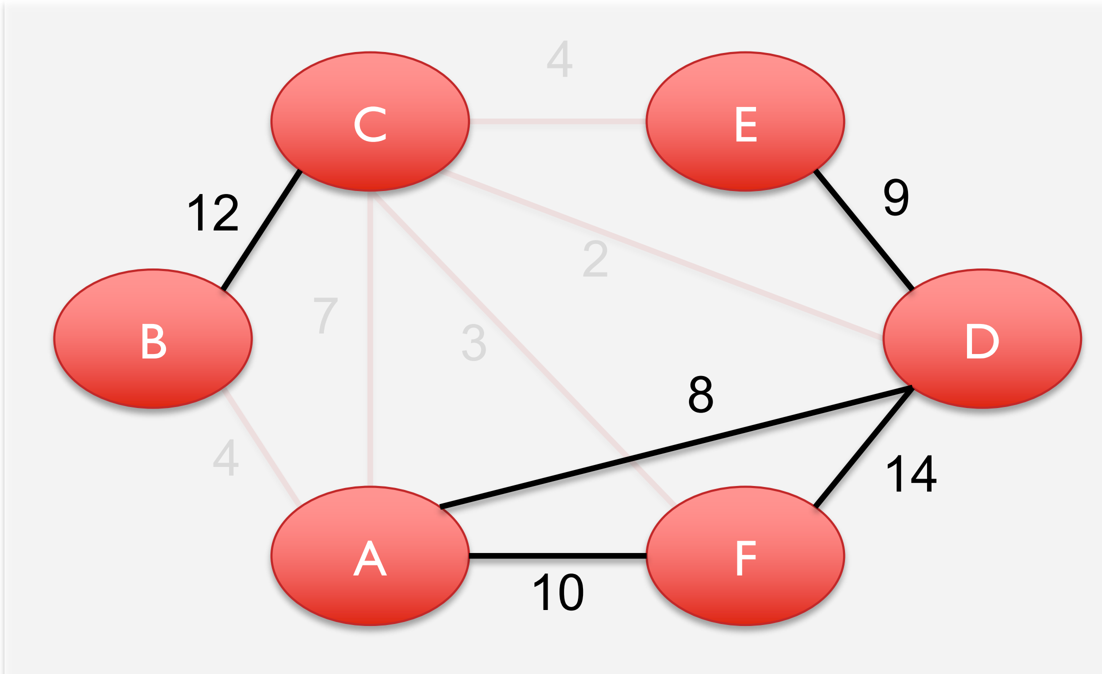
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
   1. remove an edge with minimum weight from S
   2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

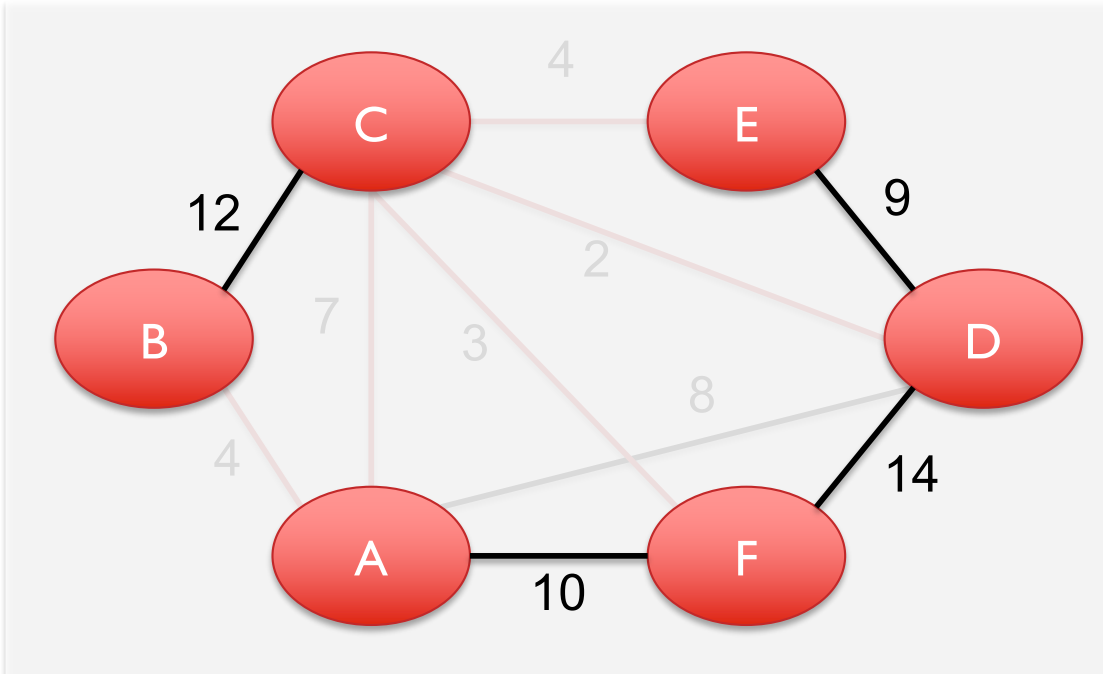
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

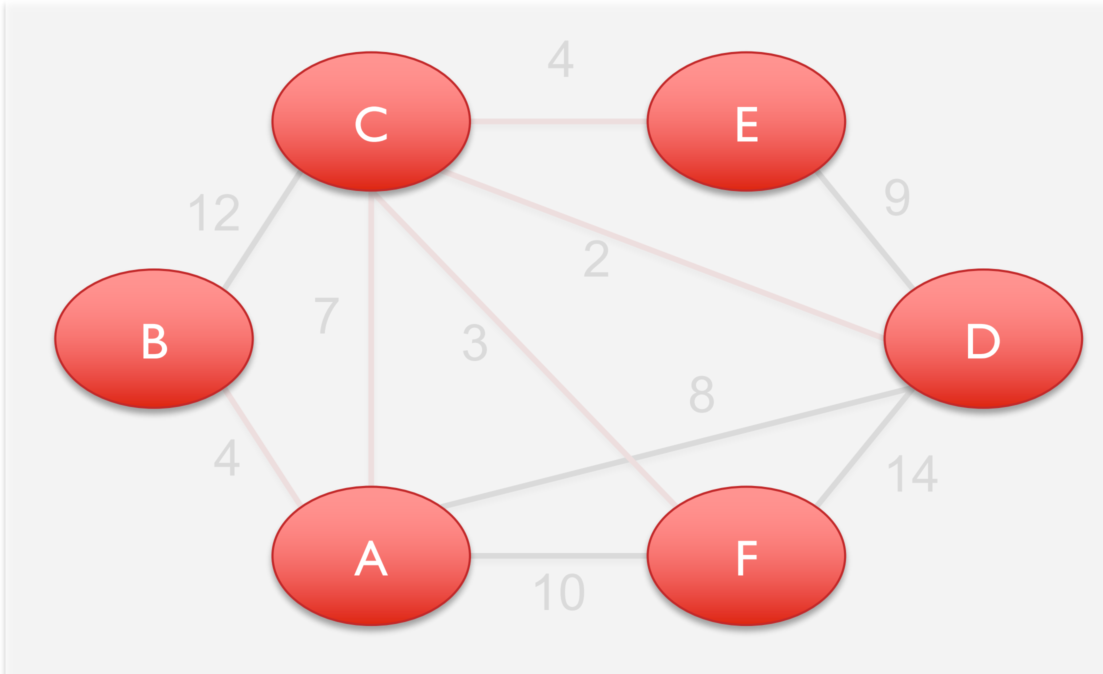
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
   1. remove an edge with minimum weight from S
   2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

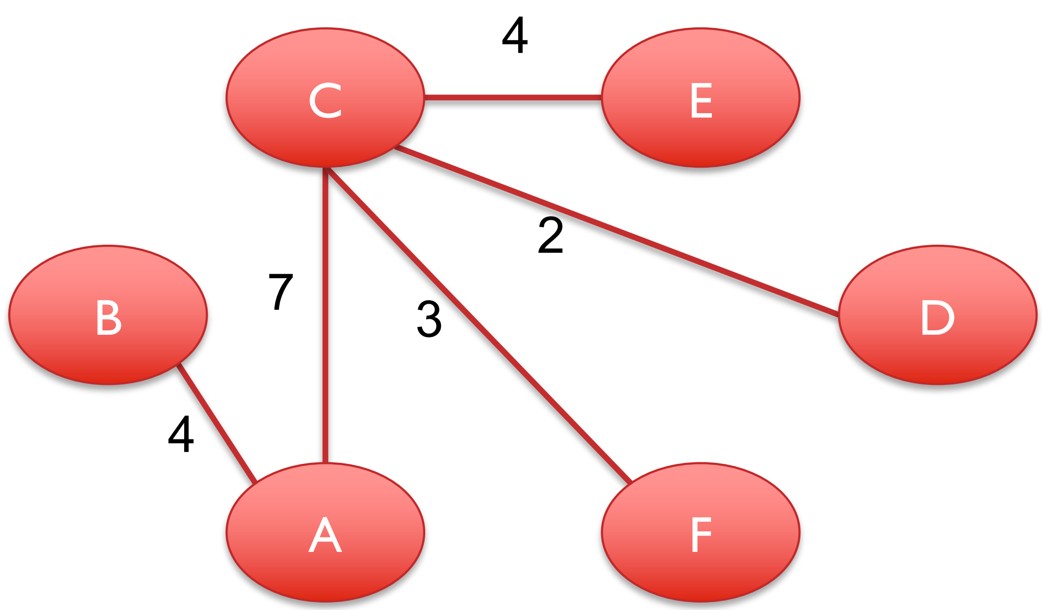
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

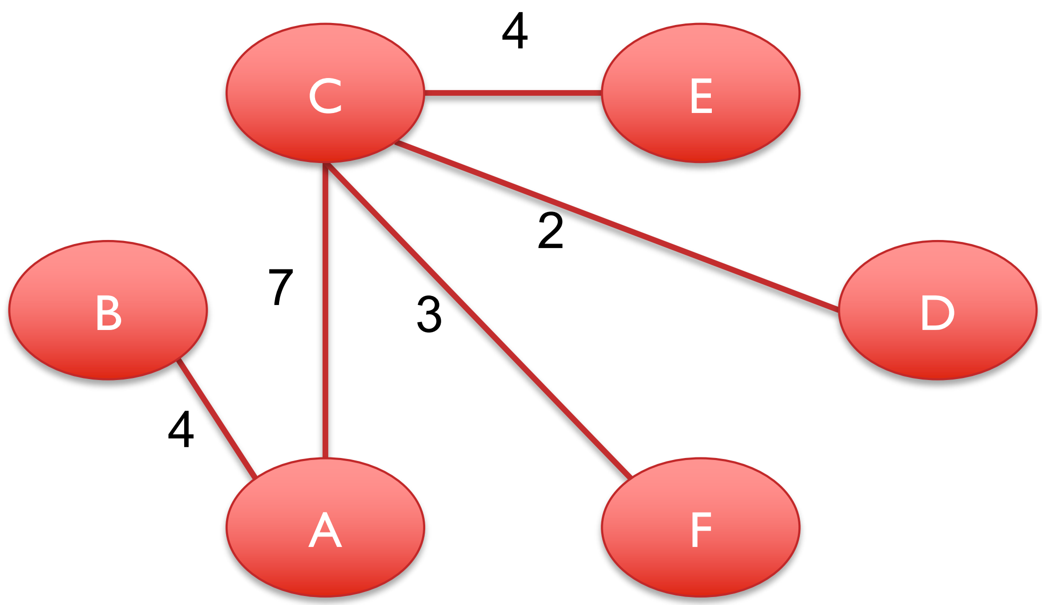
# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

# Kruskal's Algorithm




Running time:

***Kruskal's Algorithm Sketch***

1. Create a forest F (a set of trees), where each vertex in the grap separate tree

Easy: O(|V|)

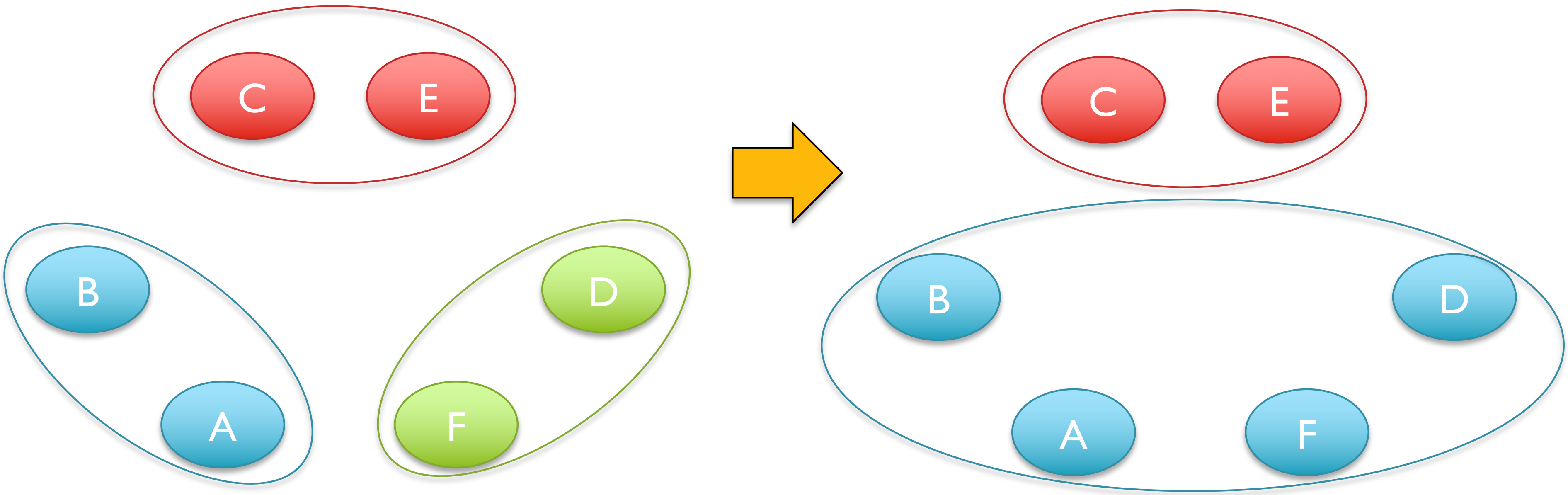
2. Create a set S containing all the edges in the graph

Easy: O(|E|lg|E|)

3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

Hmm???

# Disjoint Sets




***Disjoint Subset: every element is assigned to a single subset***

***Problem: Determine which elements are part of the same subset as sets are dynamically joined together***

Two main methods:
Find: Are elements x and y part of the same set?
Union: Merge together sets containing elements x and y

# Quick Find

**Data structure.**

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 8 | 9 |

5 and 6 are connected
2, 3, 4, and 9 are connected

**Find.** Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

**Union.** To merge components containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 6 | 6 | 6 | 6 | 6 | 7 | 8 | 6 |

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf

# Implementation

```
public class QuickFind
{
   private int[] id;

   public QuickFind(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean find(int p, int q)
   {
      return id[p] == id[q];
   }

   public void unite(int p, int q)
   {
      int pid = id[p];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = id[q];
   }
}
```

set id of each object to itself

1 operation

N operations

# Implementation

```
public class QuickFind
{
   private int[] id;

   public QuickFind(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean find(int p, int q)
   {
      return id[p] == id[q];
   }

   public void unite(int p, int q)
   {
      int pid = id[p];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = id[q];
   }
}
```

set id of each object to itself

1 operation

N operations

Find is quick O(1), but each union is O(N) time
Kruskal's ultimately merges all N items: $O(N^2)$
Can we do any faster?

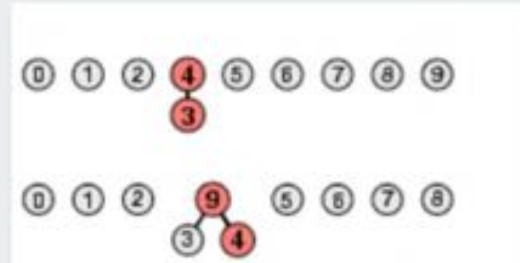
# Quick Find Forest

3-4 0 1 2 4 4 5 6 7 8 9

# Quick Find Forest

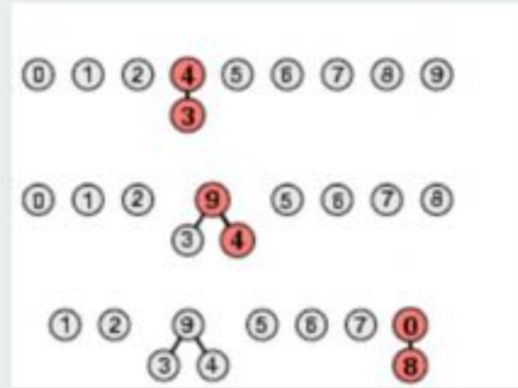
**3-4** 0 1 2 4 4 5 6 7 8 9

**4-9** 0 1 2 9 9 5 6 7 8 9



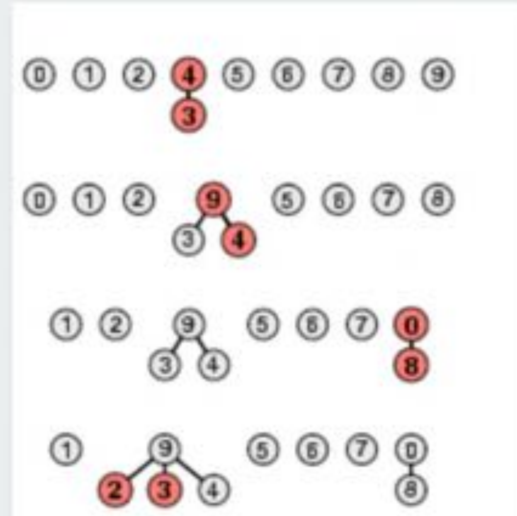
# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |



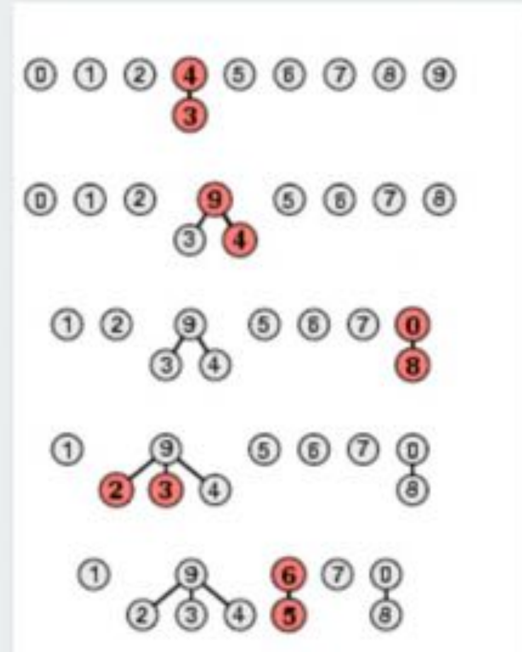
# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |



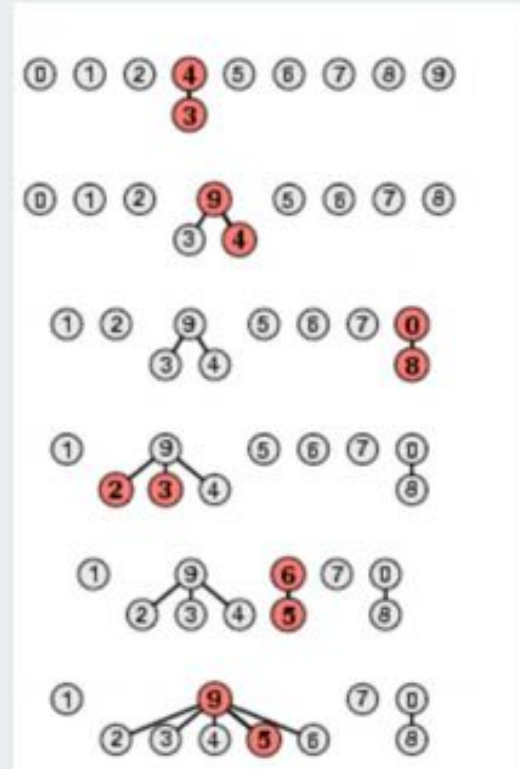
# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 0 | 9 |



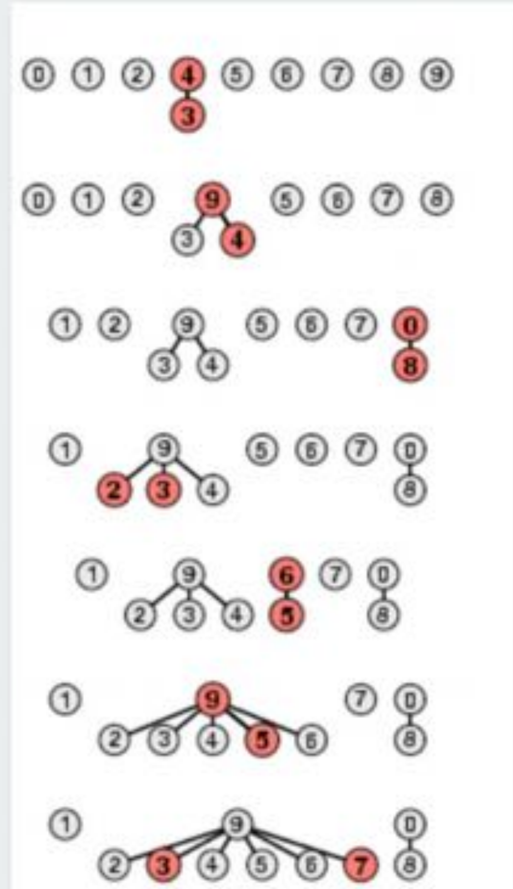
# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 0 | 9 |
| 5-9 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 7 | 0 | 9 |



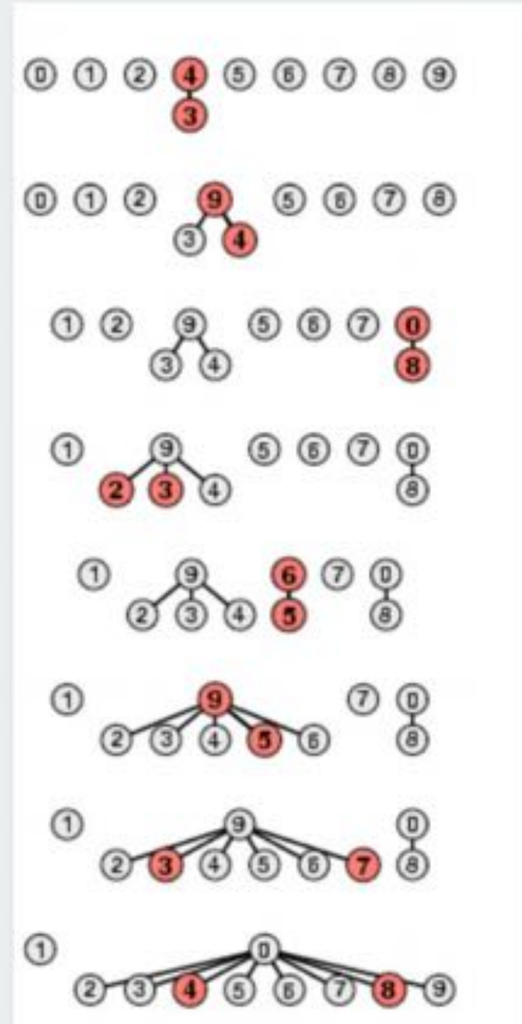
# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 0 | 9 |
| 5-9 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 7 | 0 | 9 |
| 7-3 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 9 | 0 | 9 |



# Quick Find Forest

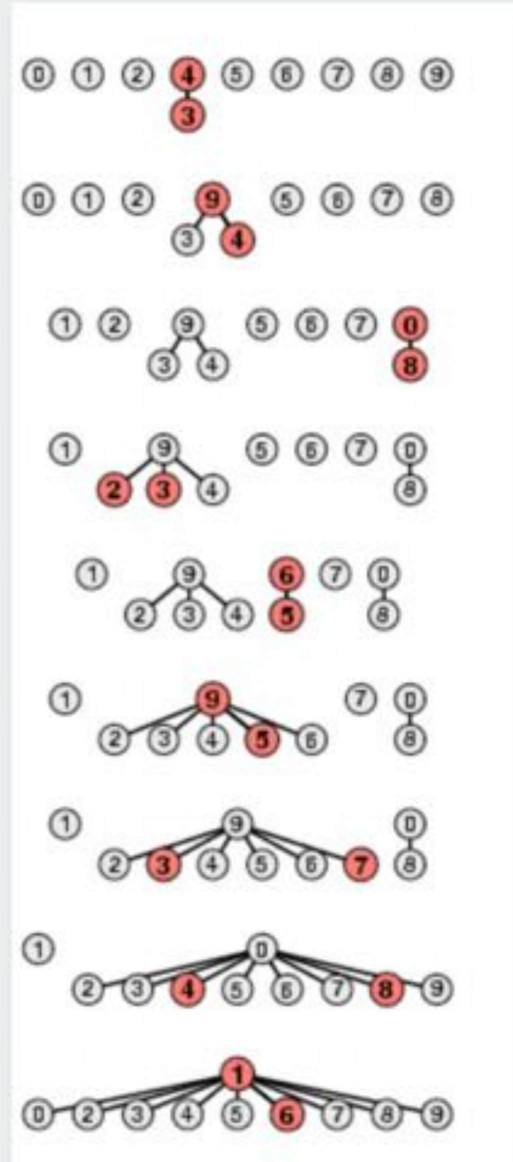
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 0 | 9 |
| 5-9 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 7 | 0 | 9 |
| 7-3 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 9 | 0 | 9 |
| 4-8 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



# Quick Find Forest

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 9 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 0 | 9 |
| 5-9 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 7 | 0 | 9 |
| 7-3 | 0 | 1 | 9 | 9 | 9 | 9 | 9 | 9 | 0 | 9 |
| 4-8 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6-1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

problem: many values can change



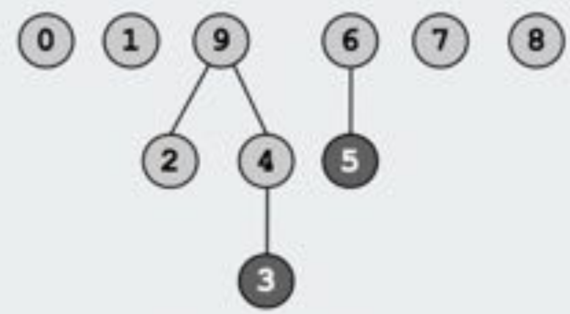
# Quick Union

**Data structure.**

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |



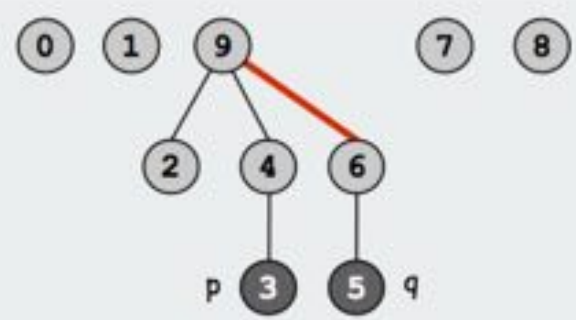

3's root is 9; 5's root is 6

**Find.** Check if p and q have the same root.

**Union.** Set the id of q's root to the id of p's root.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 4 | 9 | 6 | 9 | 7 | 8 | 9 |

only one value changes



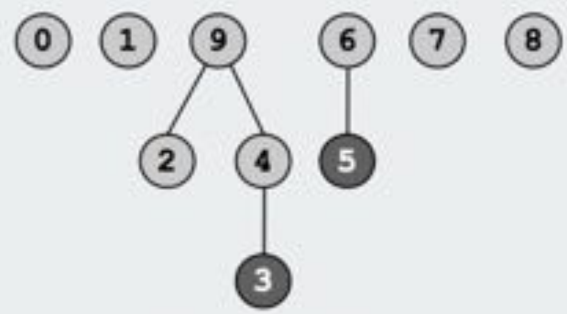
# Quick Union

**Data structure.**

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |



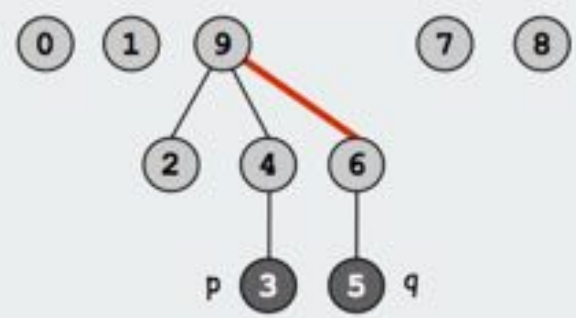

3's root is 9; 5's root is 6

**Find.** Check if p and q have the same root.

**Union.** Set the id of q's root to the id of p's root.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 4 | 9 | 6 | 9 | 7 | 8 | 9 |

only one value changes




Why not just change id[q]?

Why set it to p's root instead of p?

# Quick Union Implementation

```
public class QuickUnion
{
   private int[] id;

   public QuickUnion(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   private int root(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }


   public boolean find(int p, int q)
   {
      return root(p) == root(q);
   }

   public void unite(int p, int q)
   {
      int i = root(p);
      int j = root(q);
      id[i] = j;
   }
}
```

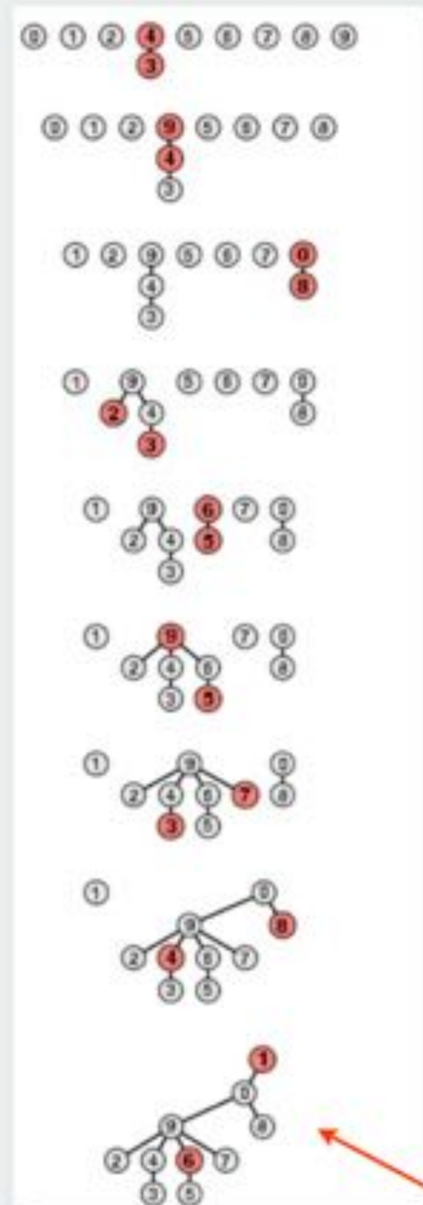
time proportional to depth of i

time proportional to depth of p and q

time proportional to depth of p and q

# Quick Union Example

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 4 | 9 | 5 | 6 | 7 | 8 | 9 |
| 8-0 | 0 | 1 | 2 | 4 | 9 | 5 | 6 | 7 | 0 | 9 |
| 2-3 | 0 | 1 | 9 | 4 | 9 | 5 | 6 | 7 | 0 | 9 |
| 5-6 | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 0 | 9 |
| 5-9 | 0 | 1 | 9 | 4 | 9 | 6 | 9 | 7 | 0 | 9 |
| 7-3 | 0 | 1 | 9 | 4 | 9 | 6 | 9 | 9 | 0 | 9 |
| 4-8 | 0 | 1 | 9 | 4 | 9 | 6 | 9 | 9 | 0 | 0 |
| 6-1 | 1 | 1 | 9 | 4 | 9 | 6 | 9 | 9 | 0 | 0 |



problem: trees can get tall

What is the worst case for Quick Union?

# Quick-Union Analysis

## Quick-find defect.

- Union too expensive (N steps).
- Trees are flat, but too expensive to keep them flat.

## Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N steps)
- Need to do find to do union

| algorithm | union | find |
|---|---|---|
| Quick-find | N | 1 |
| Quick-union | N* | N ← worst case |

* includes cost of find

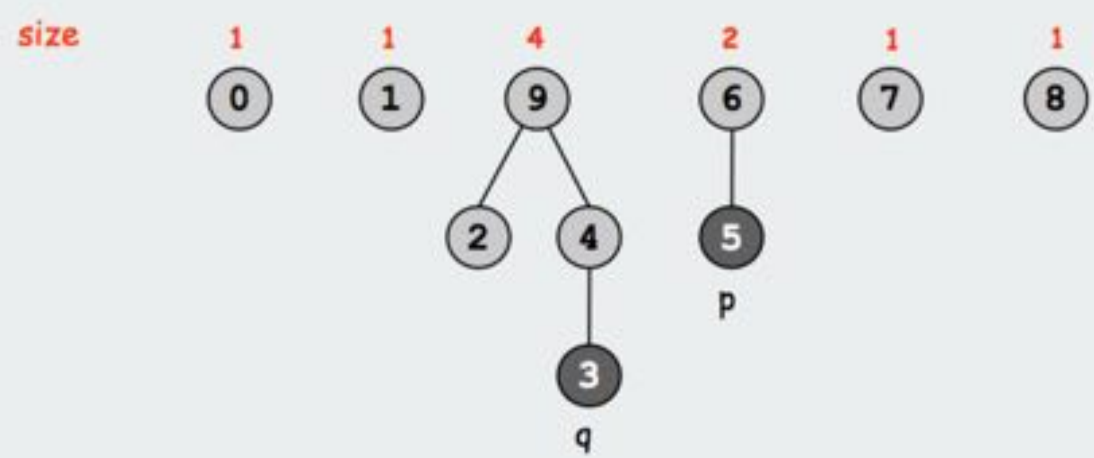
# Improved Quick-Union: Weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

Ex. Union of 5 and 3.

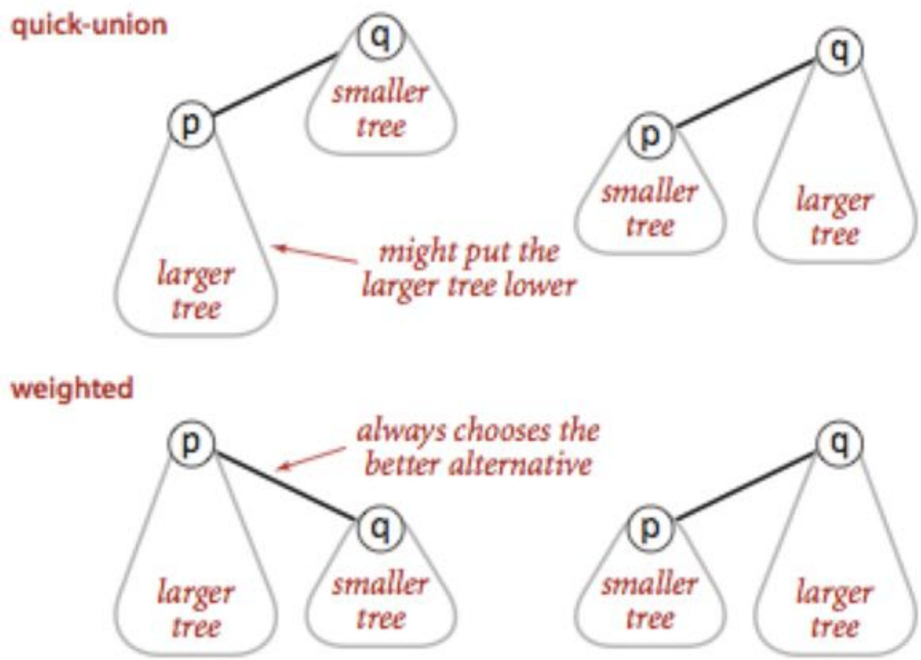
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.



# Improved Quick-Union: Weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.




Weighted quick-union

# Weighted Quick-Union Implementation

Java implementation.

- Almost identical to quick-union.
- Maintain extra array `sz[]` to count number of elements in the tree rooted at i.

Find. Identical to quick-union.

Union. Modify quick-union to

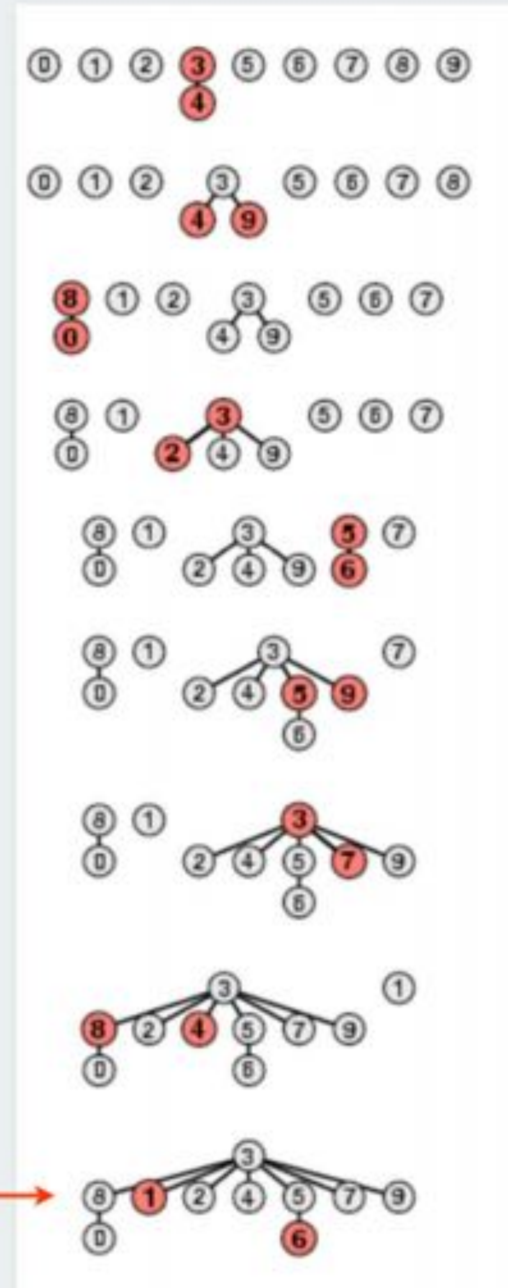
- merge smaller tree into larger tree
- update the `sz[]` array.

```
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else sz[i] < sz[j] { id[j] = i; sz[i] += sz[j]; }
```

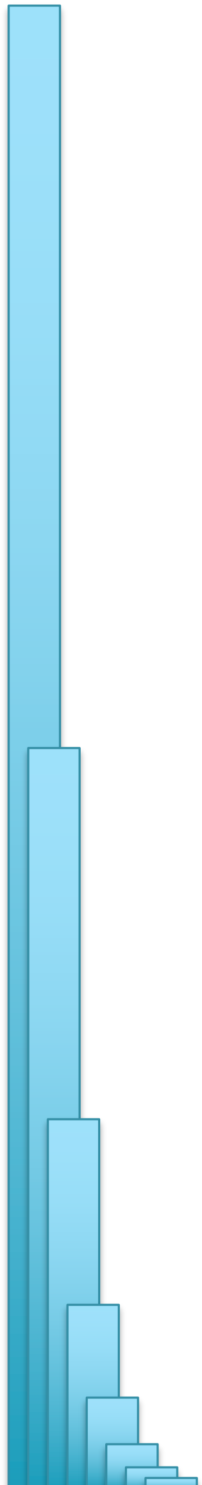
# Weighted Quick-Union

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 8-0 | 8 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 2-3 | 8 | 1 | 3 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 5-6 | 8 | 1 | 3 | 3 | 3 | 5 | 5 | 7 | 8 | 3 |
| 5-9 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 7 | 8 | 3 |
| 7-3 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 3 | 8 | 3 |
| 4-8 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 3 | 3 | 3 |
| 6-1 | 8 | 3 | 3 | 3 | 3 | 3 | 5 | 3 | 3 | 3 |

no problem: trees stay flat



# Weighted Quick-Union



Hooray! Tree stays pretty flat!

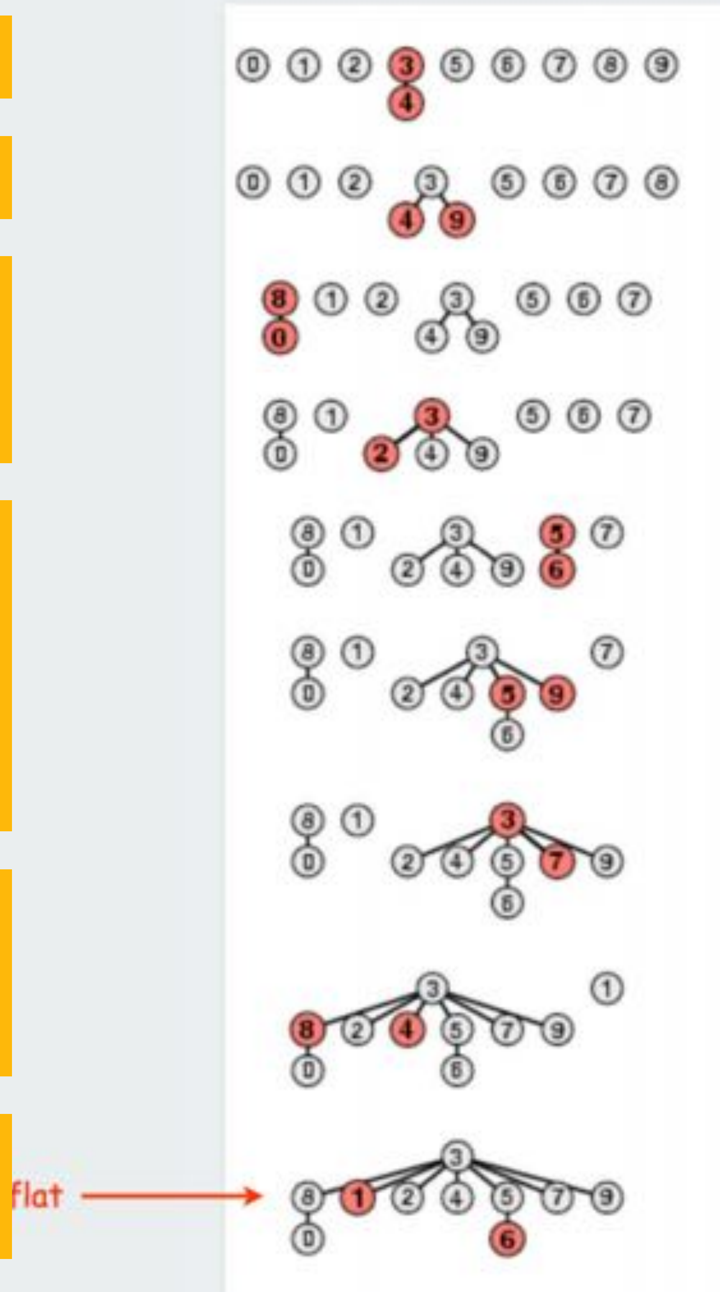
How tall can the tree grow?

Uniting a short tree and a tall tree doesn't increase the height of the tall tree

Height of tree only grows when ***two trees of equal height are united***, thereby doubling the number of elements

How many rounds of doubling can we do until we include all N elements?

lg N rounds
=> O(lg N) max height ☺



# Weighted Quick-Union Analysis

**Analysis.**

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.
- Fact: depth is at most lg N.

| Data Structure | Union | Find |
|---|---|---|
| Quick-find | N | 1 |
| Quick-union | N * | N |
| Weighted QU | lg N * | lg N |

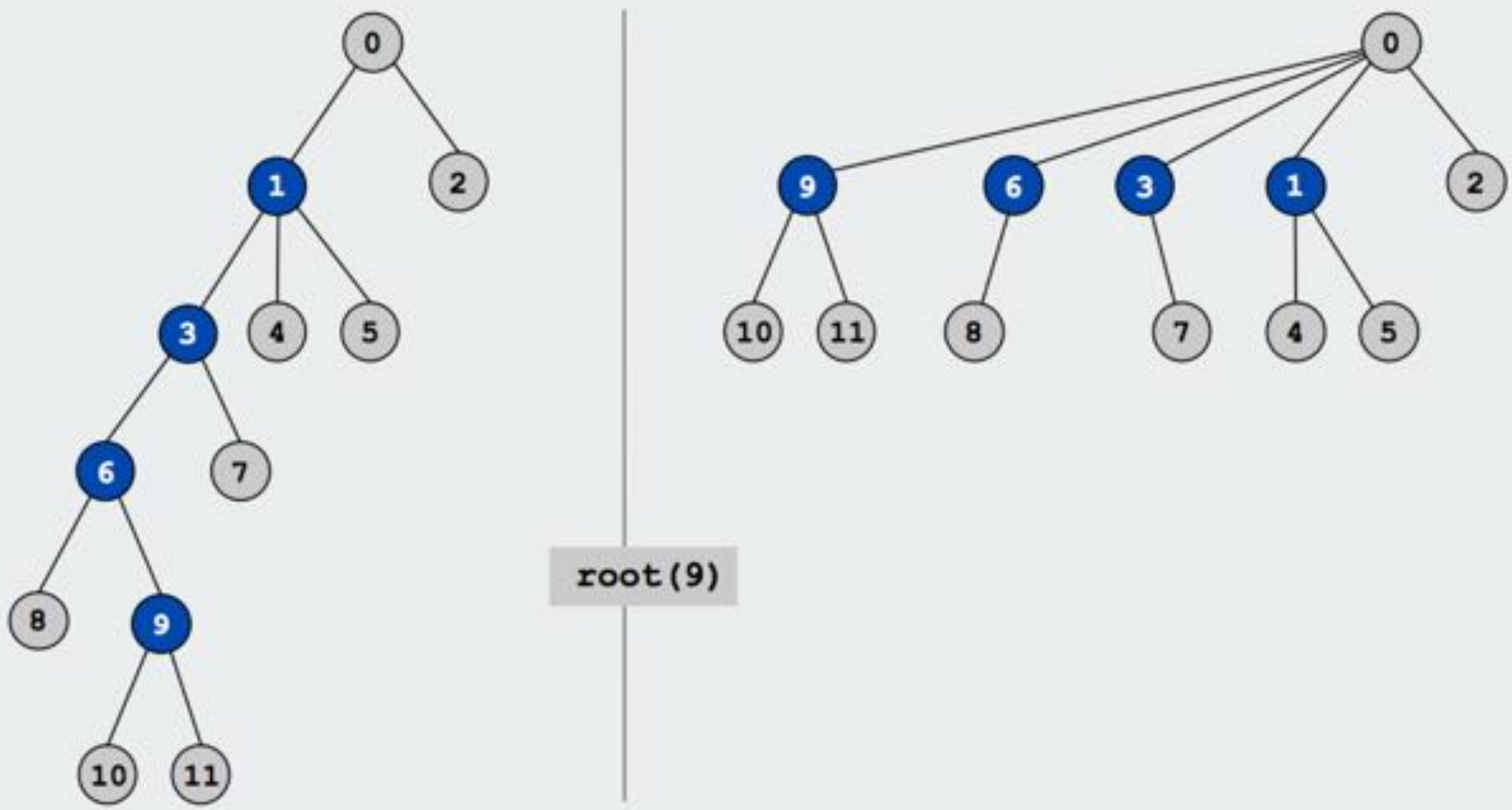
* includes cost of find

Should we stop trying here?

Usually very happy with lg(N), but here we can do better!

# Improvement 2: Path Compression

Path compression. Just after computing the root of `i`, set the `id` of each examined node to `root(i)`.

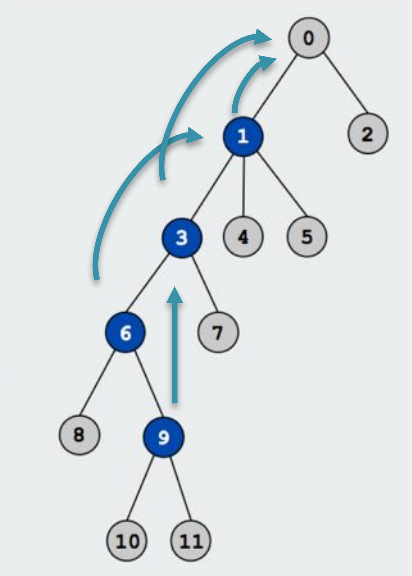


# Path Compression Implementation

Path compression.

- Standard implementation: add second loop to `root()` to set the id of each examined node to the root.
- Simpler one-pass variant: make every other node in path point to its grandparent.

```
public int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

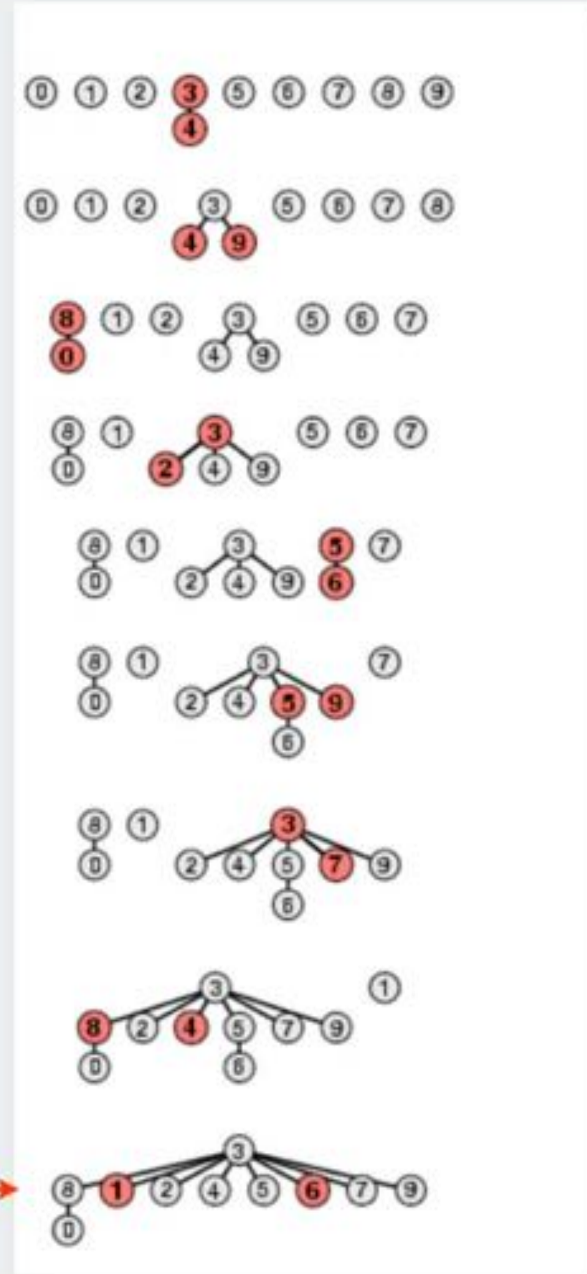



In practice. No reason not to! Keeps tree almost completely flat.

# Weighted Path Compression Example

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3-4 | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |
| 4-9 | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 8-0 | 8 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 2-3 | 8 | 1 | 3 | 3 | 3 | 5 | 6 | 7 | 8 | 3 |
| 5-6 | 8 | 1 | 3 | 3 | 3 | 5 | 5 | 7 | 8 | 3 |
| 5-9 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 7 | 8 | 3 |
| 7-3 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 3 | 8 | 3 |
| 4-8 | 8 | 1 | 3 | 3 | 3 | 3 | 5 | 3 | 3 | 3 |
| 6-1 | 8 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

no problem: trees stay VERY flat



# Weighted Path Compression Analysis

**Theorem.** Starting from an empty data structure, any sequence of M union and find operations on N objects takes $O(N + M \lg^* N)$ time.

- Proof is very difficult.
- But the algorithm is still simple!

number of times needed to take the lg of a number until reaching 1

**Linear algorithm?**

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

because $\lg^* N$ is a constant in this universe

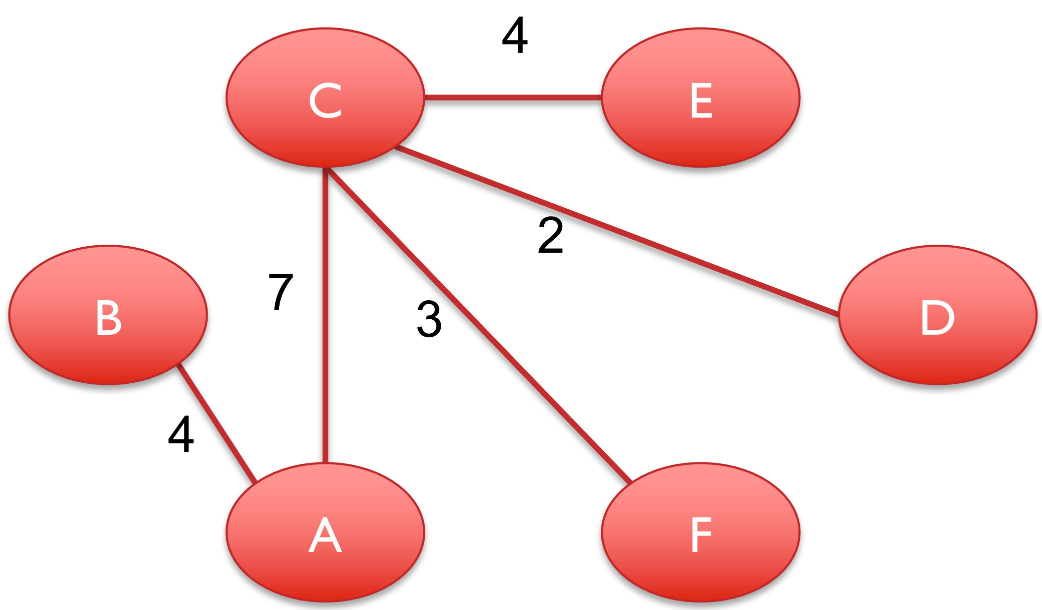
| $x$ | $\lg^* x$ |
|---|---|
| $(-\infty, 1]$ | 0 |
| $(1, 2]$ | 1 |
| $(2, 4]$ | 2 |
| $(4, 16]$ | 3 |
| $(16, 65536]$ | 4 |
| $(65536, 2^{65536}]$ | 5 |

Much bigger than #atoms in the universe

**Amazing fact:**

- In theory, no linear linking strategy exists

# Kruskal's Algorithm




***Kruskal's Algorithm Sketch***

Running time:

Using Weighted-Path Compression Quick Union (aka Union-Find) each find or union in O(lg*|V|) time. Total time is O(|E|lg|E| + Elg*|V|) => O(|E|lg|E|)

in the graph is a

Easy: O(|V|)

3. while S is nonempty and F is not yet spanning

Easy: O(|E|lg|E|)

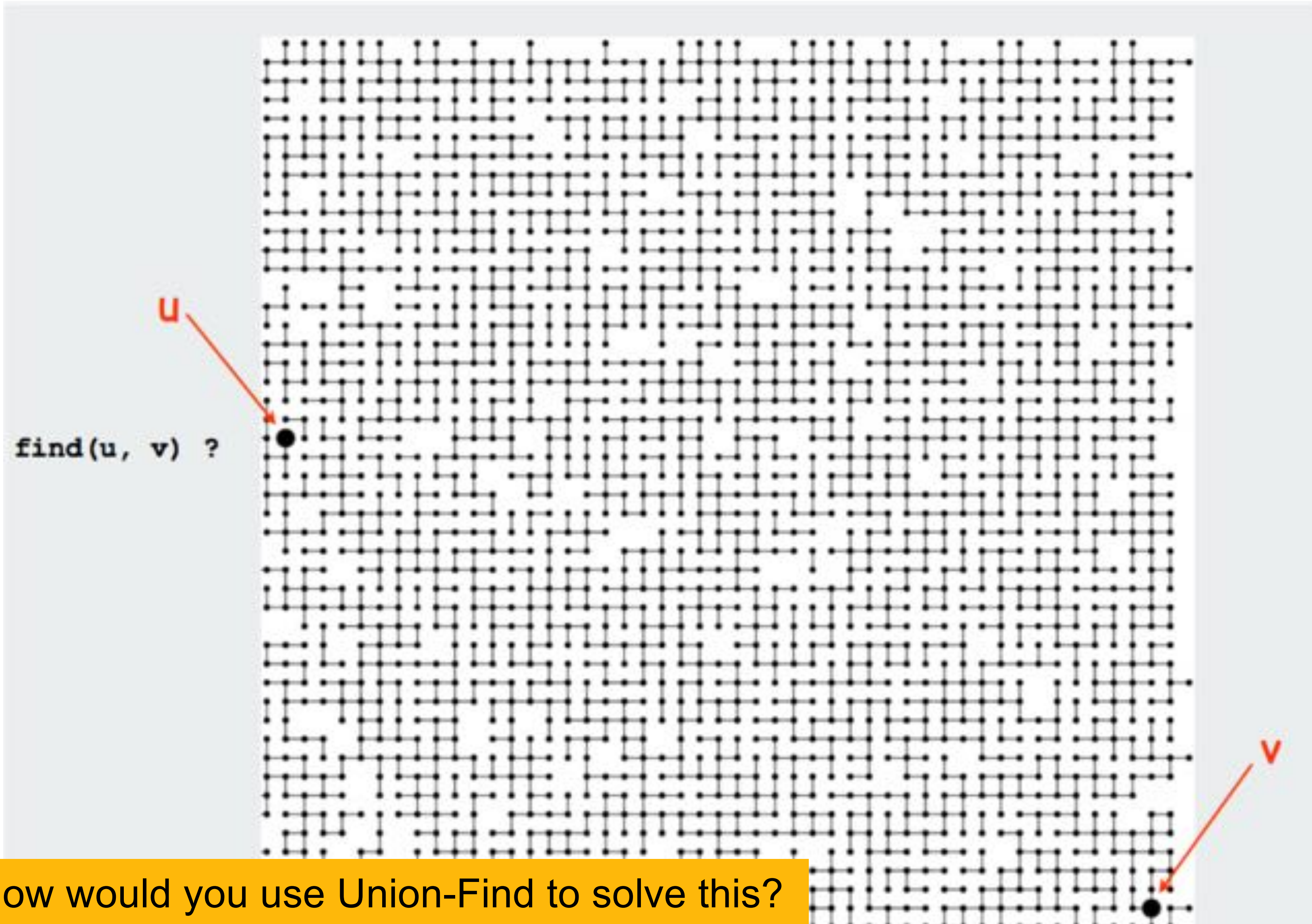
If the edges are already in sorted order (or use a linear time sorting algorithm such as counting sort), reduce total time to O(|E|lg*|V|)

s then add it to the

Hmm???

# Other Applications: Connected Components

# Other Applications: Connected Components

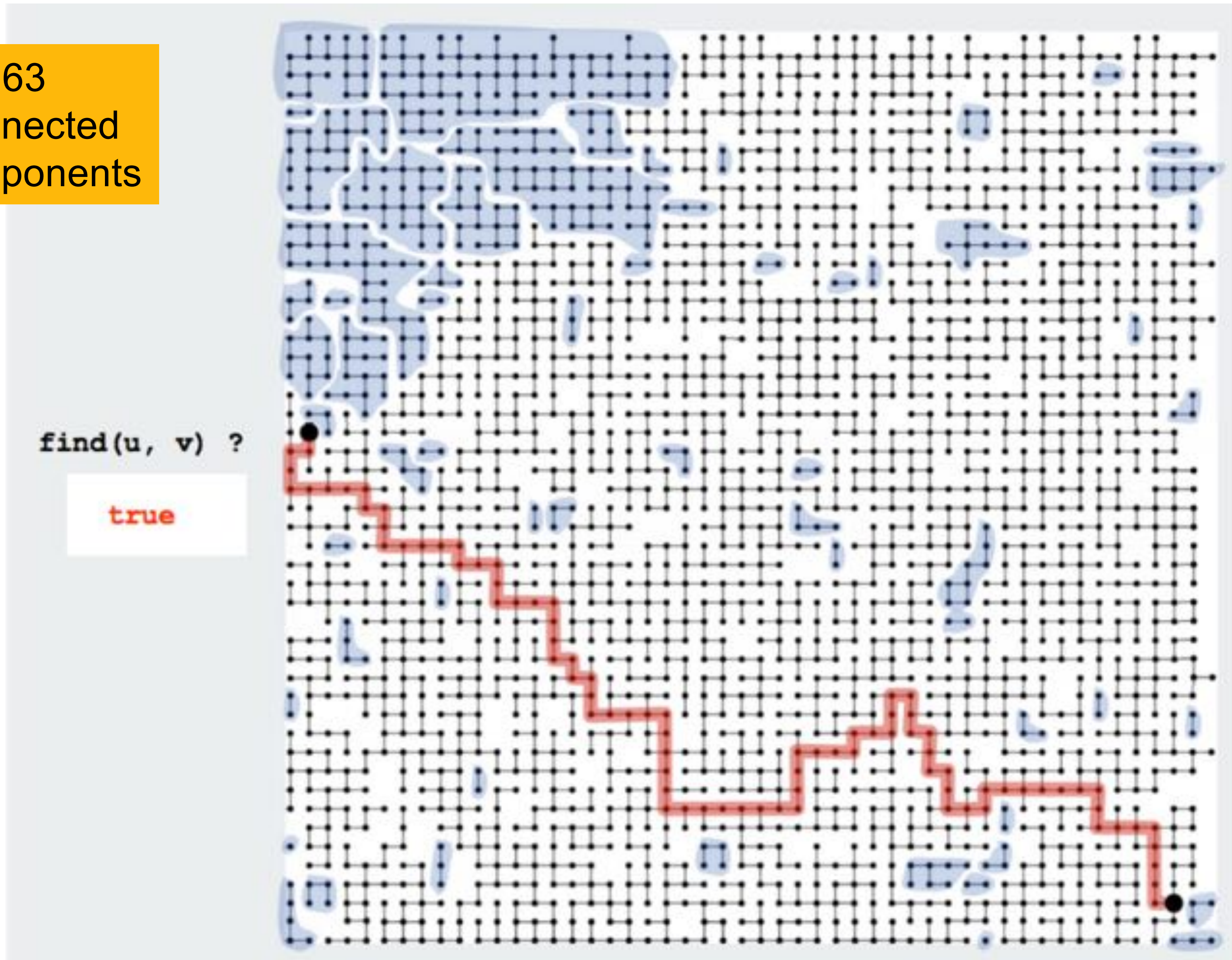



How would you use Union-Find to solve this?

Assign every dot to a set, then merge sets that have an edge between them

# Other Applications: Connected Components

63 Connected Components



*Never underestimate the power of the LOG STAR!!!!*

# Next Steps

1. Reflect on the magic and power of the logstar!

2. HW 10 due Friday @ 10pm