COMP_SCI 214: Data Structures and Algorithms

Minimum Spanning Tree

PROF. SRUTI BHAGAVATULA

Announcements

- Project was released!
- See schedule on Canvas homepage for schedule
- See @386 on Piazza:
  - Read the whole document
  - How to get help
  - Advice
- You'll only get feedback on functions you genuinely attempted

Problem of the day

- You work for a cable company and are trying to determine cheapest way to lay cables in a town.
- To determine this, you are using a graph representing all the roads (edges), where they intersect (vertices), and the length of roads.
- From this graph, you want to decide the smallest subset of edges in the graph that would allow all vertices to still be connected but have the lowest total edge weights (corresponding to cable cost)

Spanning Trees

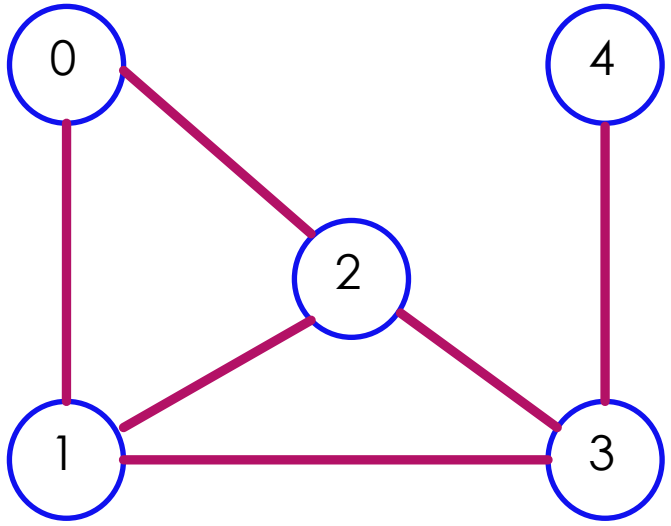
Spanning Tree

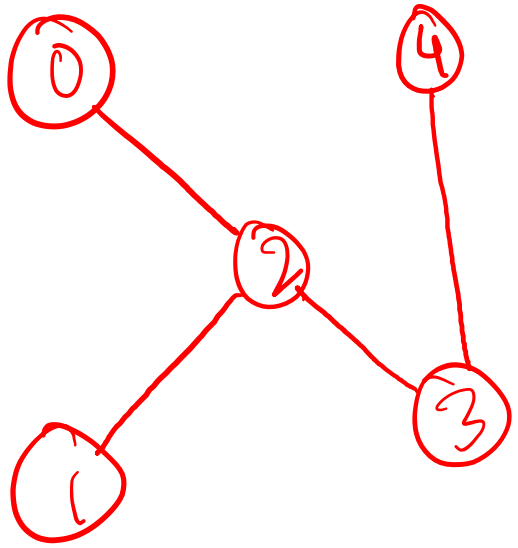
- A spanning tree of a graph `G` is a subgraph that contains all the vertices in `G` and a subset of edges in `G`
- Same connectivity among vertices

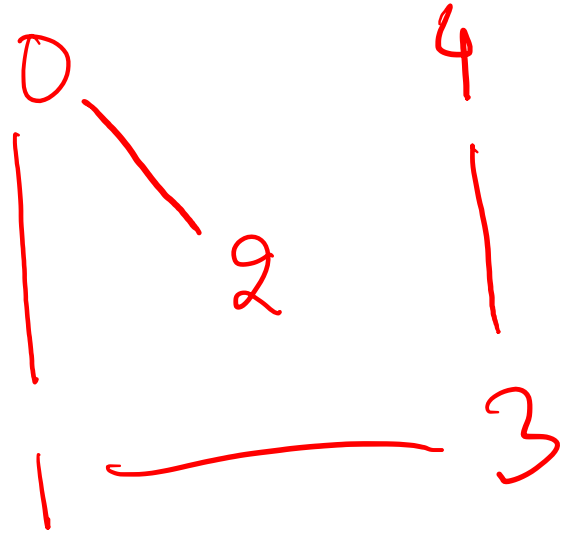
- A connected graph has a spanning tree
- A disconnected graph has a spanning forest (a bunch of spanning trees)

- A spanning tree has `v-1` edges and no cycles

Graphs to spanning trees







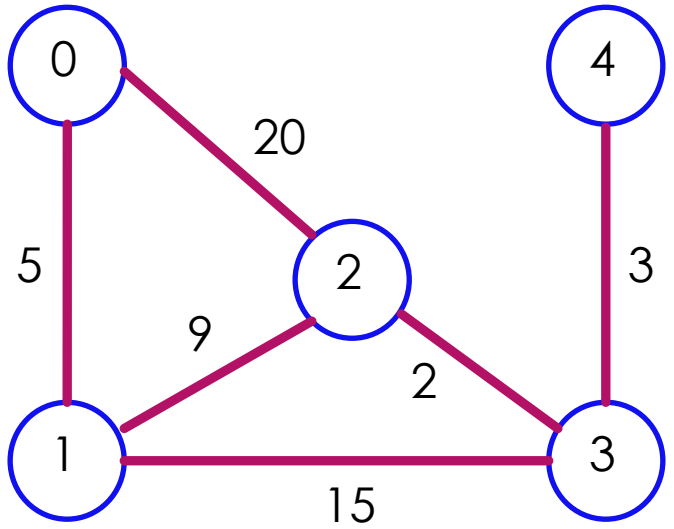
Spanning trees

- Recall our problem: "From this graph, you want to decide the smallest subset of edges in the graph that would allow all vertices to still be connected but have the lowest total edge weights (corresponding to cable cost)"

- Since spanning trees give us minimal #edges with same connectivity, can we just build a spanning tree of the original road graph and then lay cables where the spanning tree tells us?
  - Maybe? Minimal #edges is great, but what about the costs of edges?

When graphs have weights?

- Each spanning tree is not created equally when weights are involved
  - Compute the total weight of all the edges in all possible spanning trees. There will be differences.



- Instead, we construct a **Minimum Spanning Tree**

Minimum Spanning Tree (MST)

- A spanning tree with the minimal total edge weight is a minimum spanning tree

- **If unweighted graph:** spanning tree is enough to give you "lowest cost" of edges
- **If weighted graph:** you need an MST to give you the "lowest cost" of edges

Algorithms for finding MSTs

Two approaches

- Edge-centric algorithm (Kruskal's algorithm)
- Vertex-centric algorithm (Prim's algorithm)

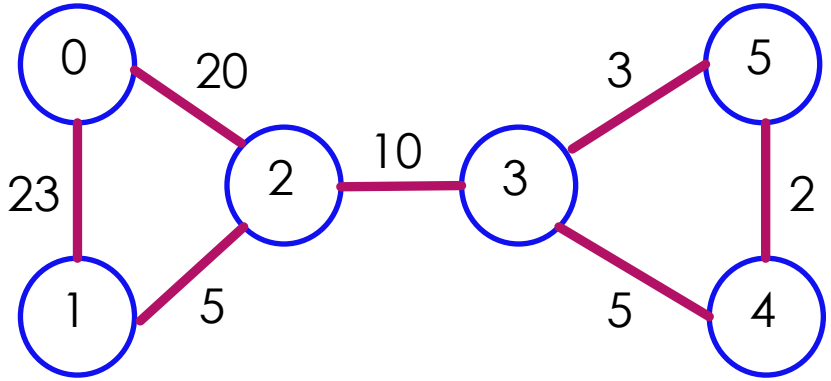
Kruskal's algorithm (edge-centric)

1. Sort edges in increasing weight order
2. Start with a graph `T` of singleton nodes (isolated vertices of `G`)
3. For each edge `(u, w)` of `G` in sorted order
   a. Are `u` and `w` already connected in `T`?
      I. If yes, discard edge
      II. If not, add `(u, w)` to `T`
   b. Stop once `T` has `v-1` edges

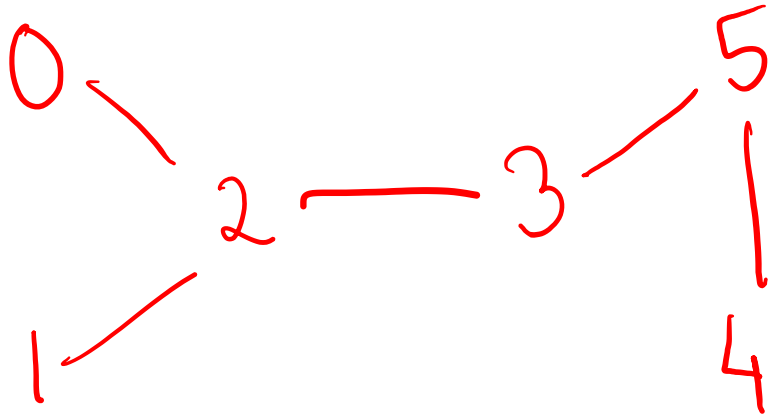
Finding an MST with Kruskal's algorithm

- Edges to look at:
  - (4, 5) ✓
  - (3, 5) ✓
  - (1, 2) ✓
  - (3, 4) ✗
  - (2, 3) ✓
  - (0, 2) ✓
  - (0, 1) ✗

Graph G

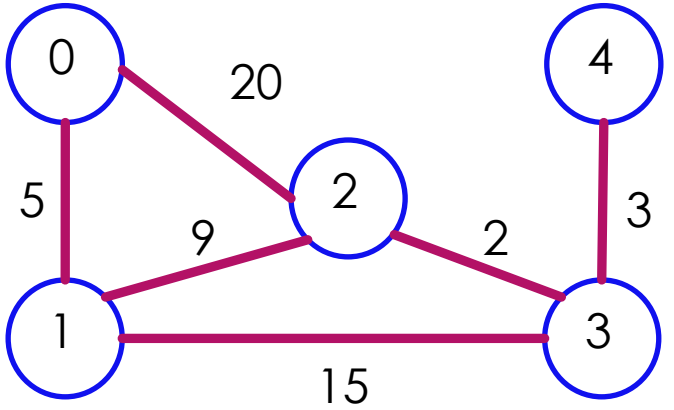


Building T



Prim's algorithm (vertex-centric)

1. Pick an arbitrary vertex `u`
   a. In a `seen` array, mark `u` as seen
   b. Add all edges `(u, v)` to a PQ
2. Repeat until the PQ is empty
   a. Pick an edge `(u, v)` from PQ
   b. If `v` is marked, discard edge and go back to 2
   c. Add `(u, v)` to `T`
   d. Mark `v`
   e. Add all `(v, w)` to PQ where `w` is unmarked
   f. Stop once T has v-1 edges



$O(e \log e)$ **algorithm**

Greedy algorithms

- Kruskal's and Prim's algorithm are greedy
- Algorithms where, at each step, we pick the best choice right now to form the optimal solution. No need to think ahead or backtrack
- DFS/BFS and Dijkstra's algorithm involve backtracking and trying other paths to arrive at the optimal solution

Kruskal's vs. Prim's

- Both greedy algorithms

- Kruskal's algorithm seems to have higher time complexity but simpler code
- Prim's algorithm has lower time complexity but more complex code

- Simpler code is usually better!
  - But does Kruskal's complexity justify this?

Pause

- Any questions?
- Anything unclear?

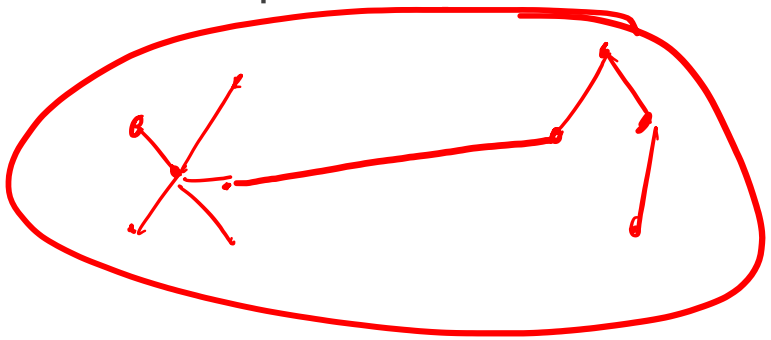
Kruskal's algorithm revisited

1. Sort edges in increasing weight order
2. Start with a graph `T` of singleton trees (isolated vertices of `G`)
3. For each edge `(u, w)` of `G`
   a. Are `u` and `w` already connected in `T`?
      I. If yes, discard edge
      II. If not, add `(u, w)` to `T`
   b. Stop once `T` has `v-1` edges

Most expensive operation in $O(v^2)$; can we improve on this?

"Are `u` and `w` already connected in `T`?"

- Reduce this to: "Are `u` and `w` in the same connected component in `T`"?
  - Treat each connected component as a set
  - Just check if two vertices are in the same set



- Operations we would need to do on these sets:
  - Check if two vertices are in the same set
  - Merge two sets if we connect two vertices in them with an edge
- Need to maintain multiple disjoint sets
  - We could use simple set ADTs
  - Keeping track of multiple disjoint sets and then merging doesn't seem ideal
  - What if there was a special data structure just for handling these disjoint sets?

Disjoint-Set (Union-Find) ADT

A new ADT: Disjoint-Set (aka Union-Find)

- **Universe:** set containing all the things we're interested in (e.g., all the nodes in our graphs)
- Disjoint-set ADT allows partitioning the universe into disjoint sets and merging of disjoint sets into larger sets
- Each set has one canonical representative: **{0} {1, 2, 5} {3, 7} {4} {6}**
- Interface in DSSL2

```
interface UNION_FIND:
  def len(self) -> nat?
  def union(self, p: nat?, q: nat?) -> NoneC
  def find(self, p: nat?) -> nat?
```

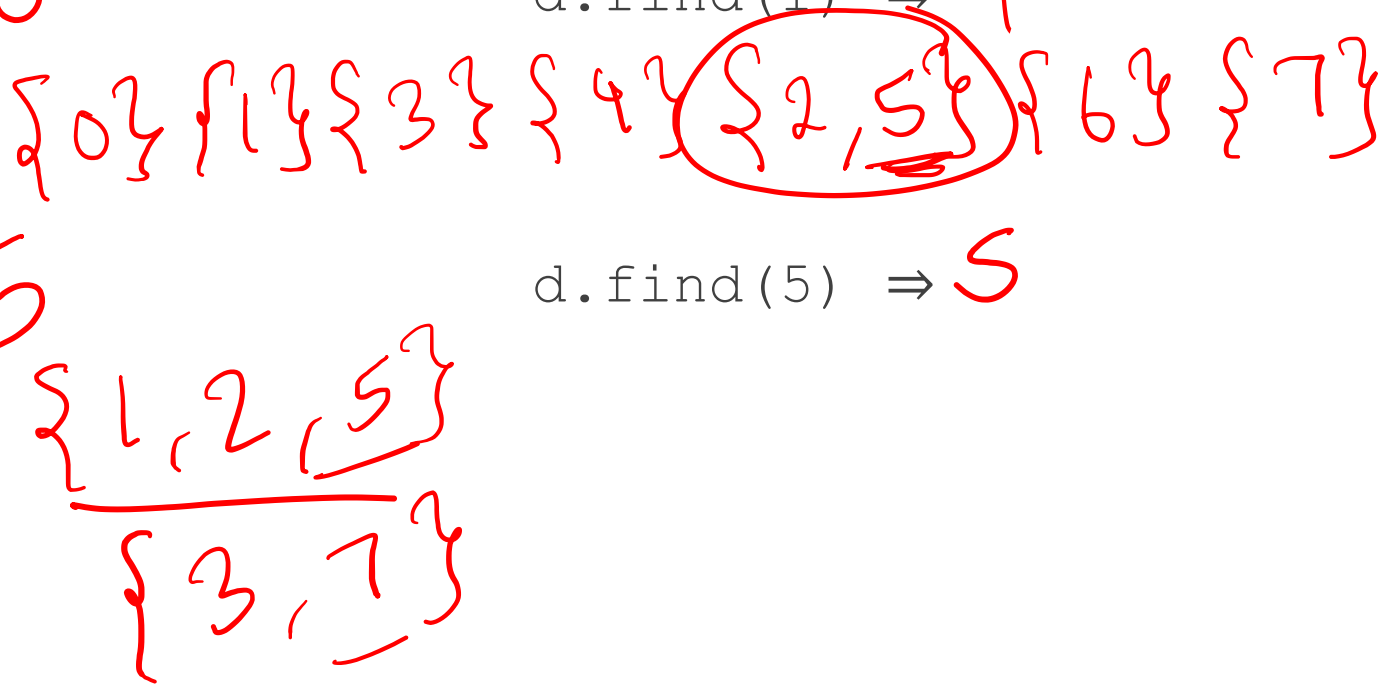
Disjoint-Set ADT

```
interface UNION_FIND:
  def len(self) -> nat?
  def union(self, p: nat?, q: nat?) -> NoneC
  def find(self, p: nat?) -> nat?
```

- Each set has one canonical representative; abstract data looks like:
  - **d = {0} {1, 2, 5} {3, 7} {4} {6}**
- `d.union(p, q)` joins `p` and `q`'s sets
- `d.find(p)` returns `p`'s canonical rep (i.e., which set it's in)
- `same_set?(p, q)` returns true if `p` and `q` have the same canonical representative (i.e., they're in the same set)

Disjoint-Set ADT in action

- Abstract data first looks like: **d = {0} {1} {2} {3} {4} {5} {6} {7}**
- `d.find(0)` ⇒ 0
- `d.find(1)` ⇒ 1
- `d.union(2, 5)`
- `d.find(2)` ⇒ 5
- `d.find(5)` ⇒ 5
- `d.union(1, 2)`
- `d.union(3, 7)`



Implementing Union-Find ADT

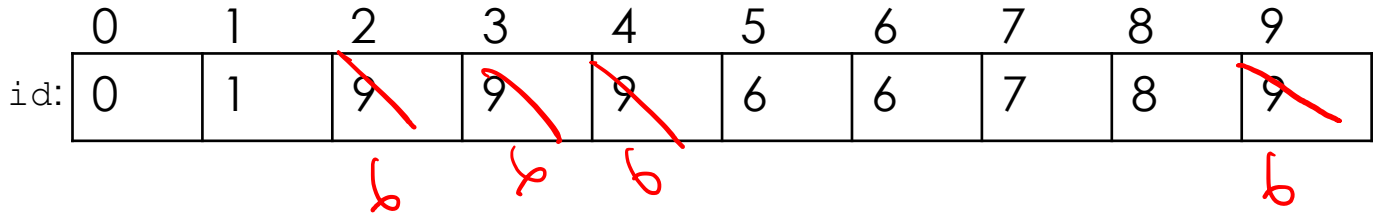
- **Attempt #1:** Quick-Find data structure
- Use a simple array
  - Each index corresponds to an element
  - The value at the index is the id of the element, which is the canonical representation of the set the element is in

d = {0} {1} {2, 3, 4, 9} {5, 6} {7} {8}

	0	1	2	3	4	5	6	7	8	9
id:	0	1	9	9	9	6	6	7	8	9

Quick-Find in action

d = {0} {1} {2, 3, 4, 9} {5, 6} {7} {8}



- `d.union(2, 5)`

- merge 2's set into 5's set (always merge `p` into `q`'s set)
- All elements in 2's set now have canonical rep. of 6

Implementing Union-Find ADT

- **Attempt #1:** Quick-Find data structure
- Use a simple array
  - Each index corresponds to an element
  - The value at the index is the id of the element: canonical representation of the set the element is in

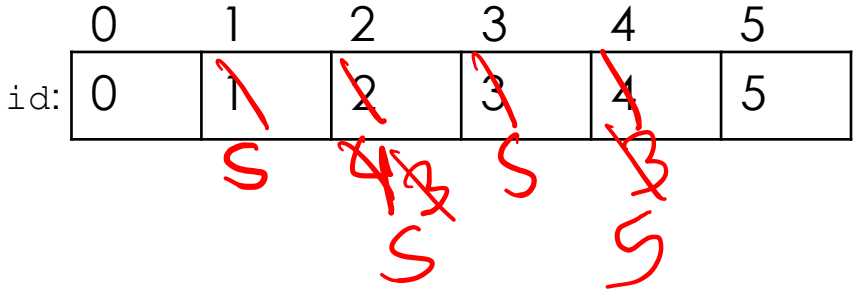
d = {0} {1} {2, 3, 4, 9} {5, 6} {7} {8}

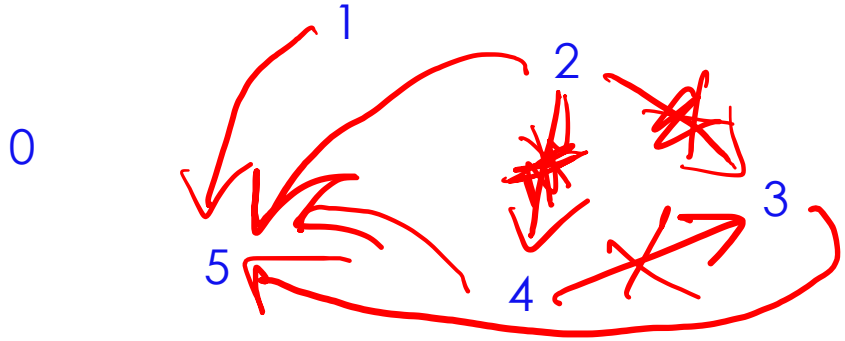
	0	1	2	3	4	5	6	7	8	9
id:	0	1	9	9	9	6	6	7	8	9

- Quick-Find is too slow
  - `find` is $O(1)$
  - But `union` loops over the array to find what needs to change: $O(n)$

Visualize disjoint sets as forest

- Think of sets as backward trees (arrows towards the root), reps as roots, universe as forest
- In a tree, find the root → find the canonical rep.
- Visualize Quick-Find DS as a forest
  - `d.union(2, 4)`
  - `d.union(4, 3)`
  - `d.union(3, 5)`
  - `d.union(1, 4)`





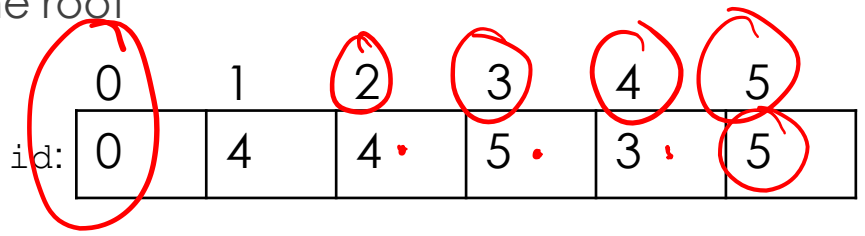
Visualize disjoint sets as forest

- What if instead of moving around trees to new roots as their reps., we just attach one tree to another?
  - Since we can just follow the root to get the canon. rep
  - Much less work to do a union?

Implementing Union-Find ADT

- **Attempt #2:** Quick-Union data structure
- Use a simple array
  - Each index corresponds to an element
  - The id leads to the canonical representation of the set the element is in
    - The ids map to the path in the tree up to the root
      - id of an element is its parent
      - when element == id, that's the root

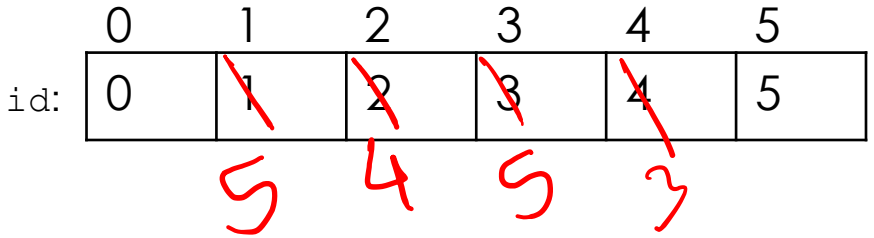
d = {0} {1, 2, 3, 4, 5}

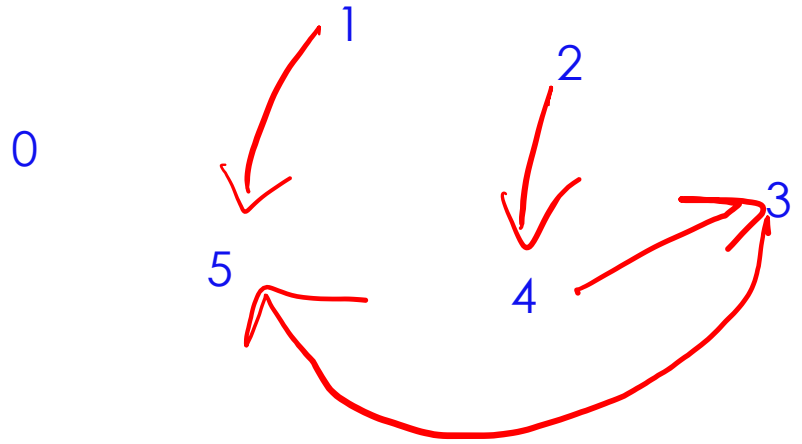


Quick-Union in action

- Find the root of `p` and `q`
- Set `p`'s root's ID to be `q`'s root
- `p`'s tree is now a subtree of `q`'s root

  (merge `p`'s tree into `q`'s tree)
  - `d.union(2, 4)`
  - `d.union(4, 3)`
  - `d.union(3, 5)`
  - `d.union(1, 4)`
  - `d.find(2)`





In-class exercise (5 minutes)

1. What would be the worst-case complexity of union with the Quick-Union data structure? Why?
2. What would be the worst-case complexity of find with the Quick-Union data structure? Why?

Implementing Union-Find ADT

- **Attempt #2:** Quick-Union data structure
- Use a simple array
  - Each index corresponds to an element
  - The id leads to the canonical representation of the set the element is in
    - The ids map to the path in the tree up to the root

d = {0} {1, 2, 3, 4, 5}

	0	1	2	3	4	5
id:	0	4	4	5	3	5

- Seems slower but this approach is the foundation for what's standard
  - `find` is now $O(n)$ since a tree can be one long path to traverse
  - `union` is also $O(n)$ since need to find the roots of both elements and then attach

Pause

- Any questions?
- Anything unclear?

Can we do better?

- Quick-Find has flat trees but it's expensive to keep them flat
- Quick-Union can have non-optimal tall trees → but getting closer

- The order of merging `p` into `q` may not be optimal
- If `p` is often taller than `q`, we'll keep building taller and taller trees
  - Gives us the worst case
- Instead, merge smaller into larger trees so height is roughly bounded by the larger tree

Size tracking

- When merging two trees `T1` and `T2` with sizes (# nodes) `s1` and `s2`:
  - If `s1 < s2`, merge `T1` into `T2`
  - If `s2 < s1`, merge `T2` into `T1`
  - If `s1 == s2`, merge either into the other

Implementing Union-Find ADT

- **Attempt #3:** Weighted Quick-Union (WQU) data structure
- Use a simple array
  - Each index corresponds to an element
  - The id leads to the canonical representation of the set the element is in
    - The ids map to the path in the tree up to the root
    - Keep track of size (# of nodes) of each tree as its weight to avoid tall trees

	0	1	2	3	4	5	6	7	8	9
id:	0	1	9	4	9	6	6	7	8	9

	0	1	2	3	4	5	6	7	8	9
wt:	1	1	1	1	2	1	2	1	1	4

Complexity of Attempt #3

- By linking smaller trees below larger ones, balance guaranteed
  - Longest path has length $O(\log n)$
  - Proof is complicated; take it on faith

- **Find operation:** get to root in $O(\log n)$
- **Union `p` and `q`:** find two roots and change one in $O(\log n)$

Kruskal's algorithm revisited

1. Sort edges in increasing weight order — O(e log e)
2. Start with a graph `T` of singleton trees (isolated vertices of `G`) — O(1)
3. For each edge `(u, w)` of `G` — [e times]
   a. Are `u` and `w` already connected in `T`? — O(log v)
      I. If yes, discard edge — O(1)
      II. If not, add `(u, w)` to `T` — O(1)
   b. Stop once `T` has `v-1` edges

$e \in O(v^2)$ $O(e \log e + e \times \log v) = O(e \log e)$

Put it all together: WQU in Kruskal's algorithm

- Cross-country road-map example (see Canvas materials)

Kruskal's algorithm revisited

1. Sort edges in increasing weight order
2. Start with a gra tices of G)
3. For each edge
   a. Are u and w
      I. If yes, discard edge
      II. If not, add (u, w) to T
   b. Stop once T has v-1 edges

Can we do even better?!

Implementing Union-Find ADT

- **Attempt #4:** Weighted Quick-Union w/ Path Compression (WQUPC) data structure
  - Everything included in WQU with one addition…
  - When doing a find: after we find the root of a tree, set the value for all the elements we saw along the way to be the root

WQUPC is crazy fast

- Trees stay **really shallow**
- $O(A^{-1}(v))$ amortized
  - A^{-1} is the inverse of the Ackermann function (grows extremely fast)
- Nearly constant amortized time (Thursday)

Kruskal's vs. Prim's revisited

- Both greedy algorithms
- Same complexity

- Kruskal's algorithm has more complex data but simpler code
  - Better for sparse graphs
- Prim's algorithm has simpler data but more complex data
  - Better for dense graphs