Recall: Kruskal's algorithm
Disjoint sets using Arrays
Disjoint sets using Trees

- Kruskal
- Two operations
  1. Union: merging two sets
     (connected components)
  2. Find: check which set an element is in.
     \[ \text{find}(u) = \text{find}(v) \text{ iff } u \text{ and } v \text{ are in same set.} \]
- Union-Find using arrays
  Idea: for each set, maintain an array for its elements.
  Initially: have \( n \) sets, \( n \) arrays
  \[ [1], [2], [3], \ldots, [n] \]
  \[ \text{union}(3,5) = [3,5] \]
  \[ [1,3,5], [2,6,7] \]
  \[ \text{union}(3,6) \]
  \[ [1,3,5,2,6,7] \]
  For each element, maintain a pointer to the array it is in
- Running time: find: \( O(1) \) time
  union: \( O(n) \) time
  \[ \text{union}(1,2) \text{ union}(2,3) \text{ union}(3,4) \ldots \text{ union}(n-1,n) \]
  Total running time = \[ \sum_{k=1}^{n-1} (k+1) = \frac{n(n+1)}{2} - 1 \]
  \[ [1,2,\ldots,k] \quad [k+1] \]
  Improve: should just append \( k+1 \) to the end of first array.
- New Union algorithm
  \[
  \text{union} \ (a, b) \\
  \quad \text{if size(find}(a) > \text{size(find}(b)) \\
  \quad \quad \text{append find}(b) \text{ to find}(a)
  \]
else append find(a) to find(b)

\[ [2, 3, 5] \quad [6, 8, 9, 11] \]
size 3 \quad size = 4
\[ [6, 8, 9, 11, 2, 3, 5] \]

- analyze running time for new algorithm

\[ \text{find : } O(1) \text{ time} \]
\[ \text{union : time } = \text{ size of the smaller set} \]
\[ \Theta(n) \text{ in the worst case } [1, 2, \ldots, \frac{n}{2}, \frac{n+1}{2}, \ldots, n] \]

- amortized analysis: not all union operations can take \( \Theta(n) \text{ time.} \)

Claim: If element i has moved k times, then find(i) is a set of size at least \( 2^k \).

Proof: by induction. Clearly this is true when \( k = 0 \).

if this is true for \( k \), then at \( k+1 \)-th time when i is moving

\[ [i] \quad [j] \]
\[ [j, i] \quad [\geq 2] \]
\[ [i] \quad [\geq 4] \]

(by induction hypothesis) \quad (by algorithm) \quad \Box

Claim: No element can move more than \( \lceil \log_2 n \rceil \) times.

\[ \sum \text{union cost } \leq \sum_{i=1}^{\lceil \log_2 n \rceil} \# \text{ times that i moved} \]
\[ \leq \sum_{i=1}^{\lceil \log_2 n \rceil} \lceil \log_2 n \rceil = O(n \log n) \]

- union find by trees

- sets \(|=\) trees

- store parent for each element

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\text{Parent} & 1 & 5 & 4 & 1 & 1 & 2 & 6 & 5
\end{array}
\]

- root of tree : index for the set

\[ \text{find}(u) : \text{return root of the tree} \]
\[ \text{if } \text{parent}(u) = u \]
\[ \text{return } u \]
\[ \text{else return } \text{find(parent}(u)) \]

\[ \text{time } = \text{ height of the tree.} \]
- `time = height of the tree. {1, 2, 3, 5} {6, 8, 9}
  - union (5, 9)
    find(5): 1
    find(9): 8
    parent(8) = 1

- running time (worst case)
  find(n) takes \( O(h) \) time

- idea 1: want to minimize depth of tree when merging
  "union-by-rank"
  maintain rank for each tree when merging, always connect root with lower rank to root with higher rank.
  rank of new tree = max rank if ranks were different
  rank + 1 if ranks were the same

  intuition: rank = depth of the tree

  Claim: A tree of rank \( k \) has at least \( 2^k \) nodes.

- idea 2: path compression

  find(6)
  find(5)
  find(3)
\[ \text{find}(1) \]
\[ \text{find}(3) \]
\[ \text{find}(1) \]

\text{find}(u) : \text{return root of the tree}
\begin{align*}
\text{if} \quad \text{parent}(u) = u \\
\text{return} \quad u \\
\text{else} \\
\text{parent}(u) = \text{find}(	ext{parent}(u)) \\
\text{return} \quad \text{parent}(u) \\
\end{align*}

- running time (union-by-rank + path compression)
- for \( t \) union-find operations
  \[ \text{running time} = \mathcal{O}(t \alpha(n)) \]

- What is \( \alpha(n) \)? \( \alpha \): inverse Ackerman function

\[ \alpha(n) = \begin{cases} 
\log \log \cdots \log n & \text{for } k \text{ layers} \\
2 & \text{for any reasonable number } \alpha(n) \leq 5 
\end{cases} \]