A set $S$ of $n$ elements.

An equivalence relation $\sim$ over $S$ is a subset of pairs $(a, b) \in S \times S$ so that:

- Reflexive: $(a, a) \in \sim \ orall a \in S$
- Symmetric: $(a, b) \in \sim \iff (b, a) \in \sim$
- Transitive: $(a, b) \in \sim \land (b, c) \in \sim \land (a, c) \in \sim$
Then \((a, c) \in \sim\)

\(a \neq b\) are equivalent if \((a, b) \in \sim\)

Partitions \(S\) into equivalence classes.

Will write \(a \sim b\) if \((a, b) \in \sim\).
Data structure maintains a equiv. relation to that changes over time (dynamic equiv. relation)

Initially, no pair are equivalent.

Will claim elements are in different classes.
unless we prove otherwise.

Two operations

union \((a, b)\): Declare \(a \rightarrow b\).
Implies other equivalences by symmetry and transitivity.

find \((a)\): Return some ID for \(a\)'s class.

\[
\text{find } (a) = \text{find } (b) \iff 
\]
a \rightarrow b \ (\text{until we do a union})

- in other words -

maintain a collection of disjoint (sub)sets of S. Initially, we have \{a\} for each \( a \in S \). Find (a) IDs a's subset.
union(a, b) merges a to b's subsets

often called union-find

data structure instead of disjoint sets

You cannot break apart subsets!
Typical use: maintain connected components of graph vertices as you add edges.

Assume \( S = \{0, 1, \ldots, n-1\} \)

Just arbitrarily number the objects you really care about in advance.
Idea 1: Fast find, slow union

keep an array of length \( n \).

\text{find}(a) \text{ returns value at position } a

\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}

\text{initially union}\,(a, 6): \text{ replace all}
labels of a's subset with the label for b

\text{\text{union}}(3, 3), \text{\text{union}}(2, 2), \text{\text{union}}(5, 7)

Need to visit everything to figure out updates:

\(\Theta(n)\) time per union
Idea 16:

Also store a linked list per disjoint set naming its elements. Now can union in time proportional to \# updates.

Update labels for smaller of a \& 6's sets.
Worst-case is still $\Theta(n)$, but...

Each time you update a's label, its set at least doubles in size.

$\Rightarrow \Theta(\log n)$ updates for a

$\Rightarrow m$ finds + $n$-1 unions takes $O(m + n \log n)$ time.
Idea 2: Fast union using "up-trees".
Assume with any call to union (a, b), we know their IDs.

Conceptually: A collection of rooted trees, one per disjoint set. Nodes hold elements for that set.
Find (a): Return element at root of a's tree.

union (a, b): make root of one tree the child of the other

union (1, 3)
only need references to parent nodes
But really, use an array of length \( n \).

\[
\text{arr}[i] : \text{index of parent of } -1 \text{ if } i \text{ is a root}
\]
public class DisjSets {
    public DisjSets(int numElements) {
        s = new int[numElements];
        for (int i = 0; i < s.length; i++) {
            s[i] = -1;
        }
    }

    public void union(int rootA, int rootB) {
        s[rootB] = rootA;
    }

    public int find(int a) {
        if (s[a] < 0) {
            return a;
        } else {
            return find(s[a]);
        }
    }

    private int[] s;
}

union: \( O(1) \) time

\( \text{find}() \): depth of \( a \) = \( \Theta(n) \)
Worst-case

Speeding up find:

1) union-by-size;
   make smaller tree the child

A nodes in tree doubles when a node sees depth increase

$\Rightarrow \mathcal{O}(\log n)$ depth

$\Rightarrow \mathcal{O}(\log n \log \text{find})$
Write \(-(\text{size of tree})\) in root positions of array.

Union-by-height:
smallest height tree is child

Inductively, tree of height \(h\) has \(\geq 2^h\) nodes.
=) height is \( O(\log n) \)

write\((- (height) - 1)\) at root positions

(in practise union (a, b)
finds() the roots for a + b itself)
path compression:
speed up future finds by updating parent
of all nodes on search path to be the root.

union-by-height becomes

union-by-rank is

smaller rank becomes child

rank of now root increases if old ranks where equal
always

\text{height} \leq \text{rank}

\text{rank} r \Rightarrow r \geq 2^r \text{ nodes}

so worst-case

\Theta(\log n) \text{ find}

\Theta(1) \text{ time}
public void union(int rootA, int rootB) {
    if (s[rootB] < s[rootA]) {  // rootB has larger rank
        s[rootA] = rootB;
    } else {
        if (s[rootB] == s[rootA]) {
            s[rootA]--;  // Update rank if they were the same
        }
        s[rootB] = rootA;
    }
}

public int find(int a) {
    if (s[a] < 0) {
        return a;
    } else {
        s[x] = find(s[x]);  // compress the path
        return s[x];
    }
}
Amortized analysis:

(an alarm interrupted as here, so I'm writing the below outside of class)

Amortized time for find is in...

\[ O(\log n) \]
\[ O(\log \log n) \]
\[ O(\log \log \log n) \]

\[ O(\log^4 n) \]

iterative logarithm: # of logs before you reach 1

\[ \log^5 (2^{65536}) = 5 \]

Not in O(1), but may as well be.
am, time for find in

\[ O(\log^* n) \leq \text{iterative iterative logarithm} \]

\[ O(\log^{**} n) \]

\[ O(\alpha(n)) \text{ where} \]

\[ \alpha(n) = \min \left\{ c \geq 1 : \log^* n \leq 3^c \right\} \]

inverse Ackermann function

More precisely, worst-case amortized time for find in

\[ \Theta(\alpha(m, n)) \text{ where} \]

\[ \alpha(m, n) = \min \left\{ c \geq 1 : \log^* (\log n) \leq m/\sqrt{n} \right\} \]
If \( m = \Theta(n) \), \( \alpha(mjn) = \Theta(\alpha(n)) \).

If \( m = n \log n \),
\[ \alpha(mjn) = \Theta(1). \]

Worst-case for any pointer-based structure does amortized \( \Omega(\alpha(mjn)) \) time finds.

We can't do better!