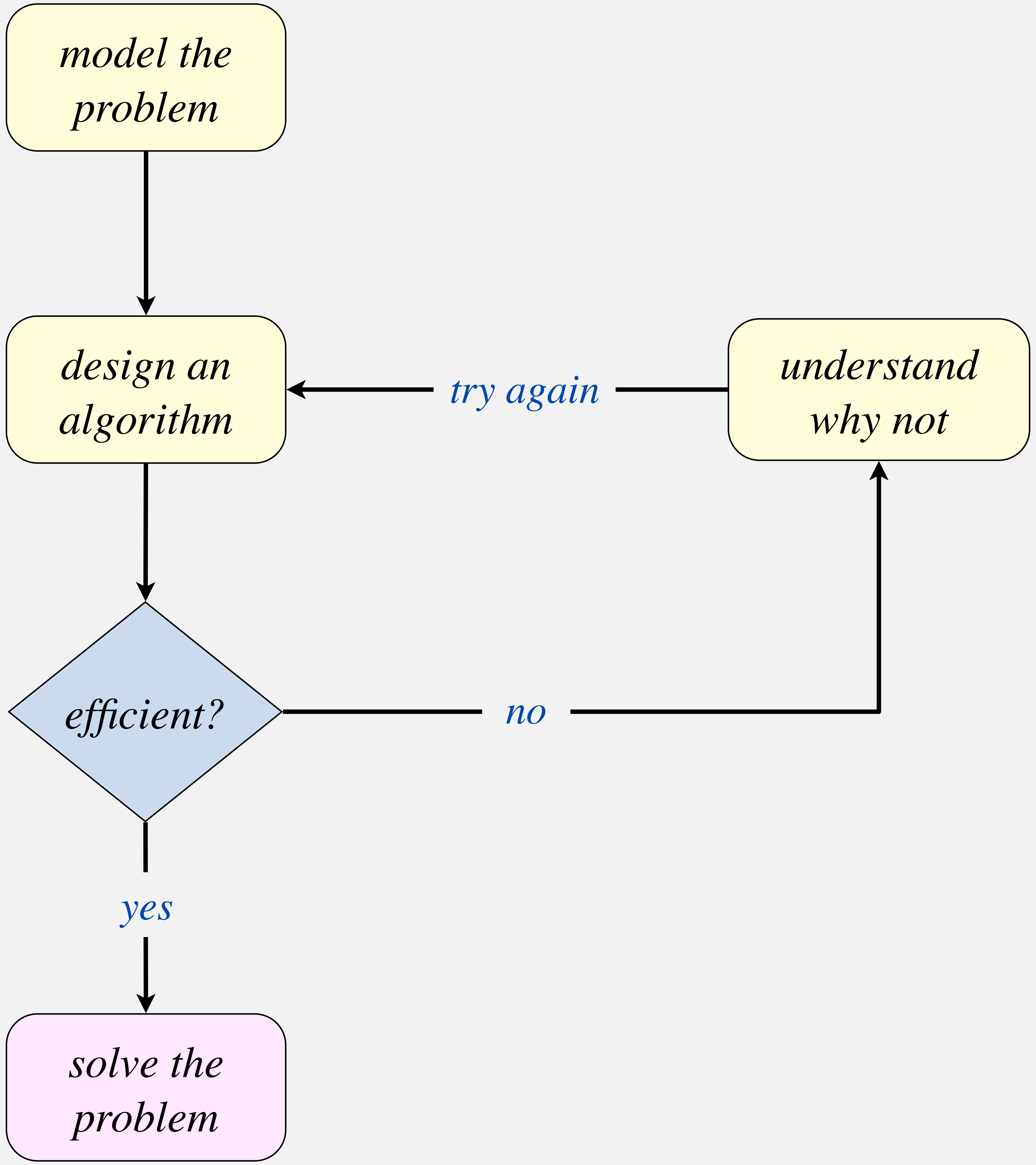
https://algs4.cs.princeton.edu

1.5 UNION–FIND

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *percolation* ← see precept

Subtext of today's lecture (and this course)

Steps to develop a usable algorithm to solve a computational problem.



https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *percolation*

Union–find data type

Disjoint sets. A collection of sets containing n elements, with each element in exactly one set.

Leader. Each set designates one of its elements as leader to uniquely identify the set.

no restriction on which element
(but leader of set can't change unless the set changes)

Find. Return the leader of the set containing element p. ← typical use case: are two elements in the same set?

Union. Merge the set containing element p with the set containing element q.

find(1) = 4
find(4) = 4
find(5) = 4

leader is 0 leader is 4 leader is 6

{ 0 } { 1, 4, 5 } { 2, 3, 6, 7 }

8 elements, 3 disjoint sets

union(2, 5)

leader is 6

{ 0 } { 1, 2, 3, 4, 5, 6, 7 }

2 disjoint sets

Union-find data type: API

Goal. Design an efficient union-find data type.

- Number of elements n can be huge.
- Number of operations m can be huge.
- The `union()` and `find()` operations can be intermixed.

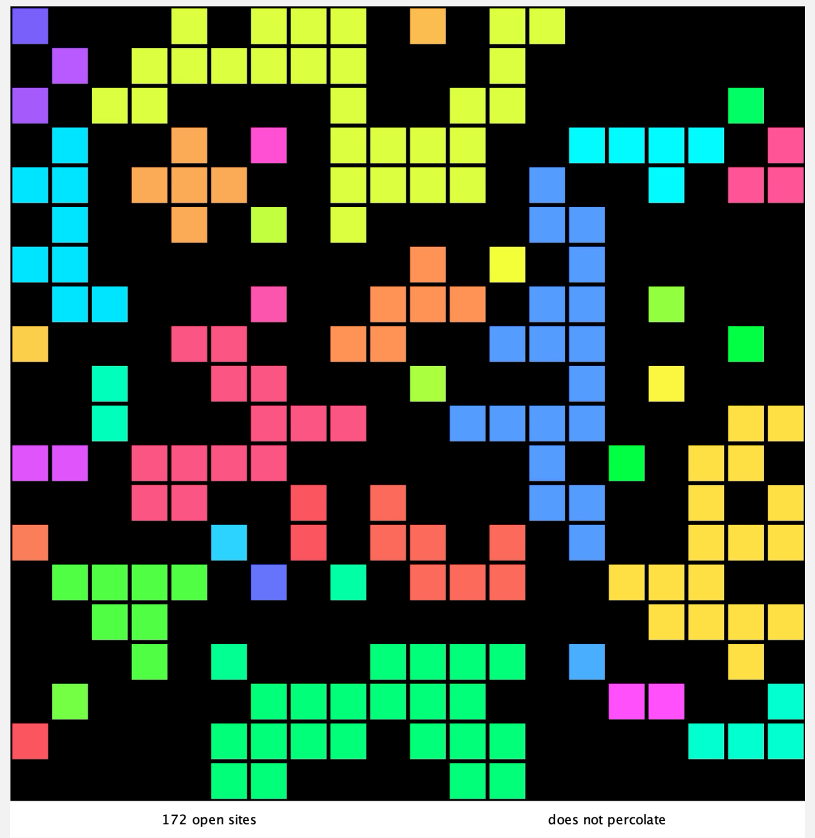
public class UF	
`UF(int n)`	*initialize with n singleton sets* $(0$ *to* $n-1)$
`void union(int p, int q)`	*merge sets containing elements p and q*
`int find(int p)`	*return the leader of set containing element p*

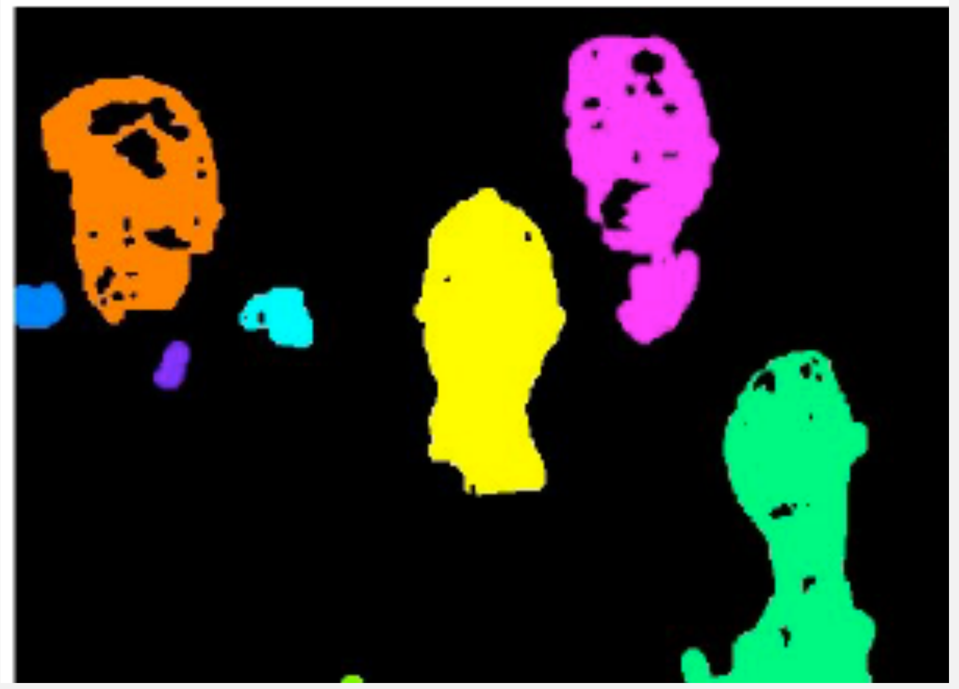
Simplifying assumption. The n elements are named $0, 1, \ldots, n-1$.

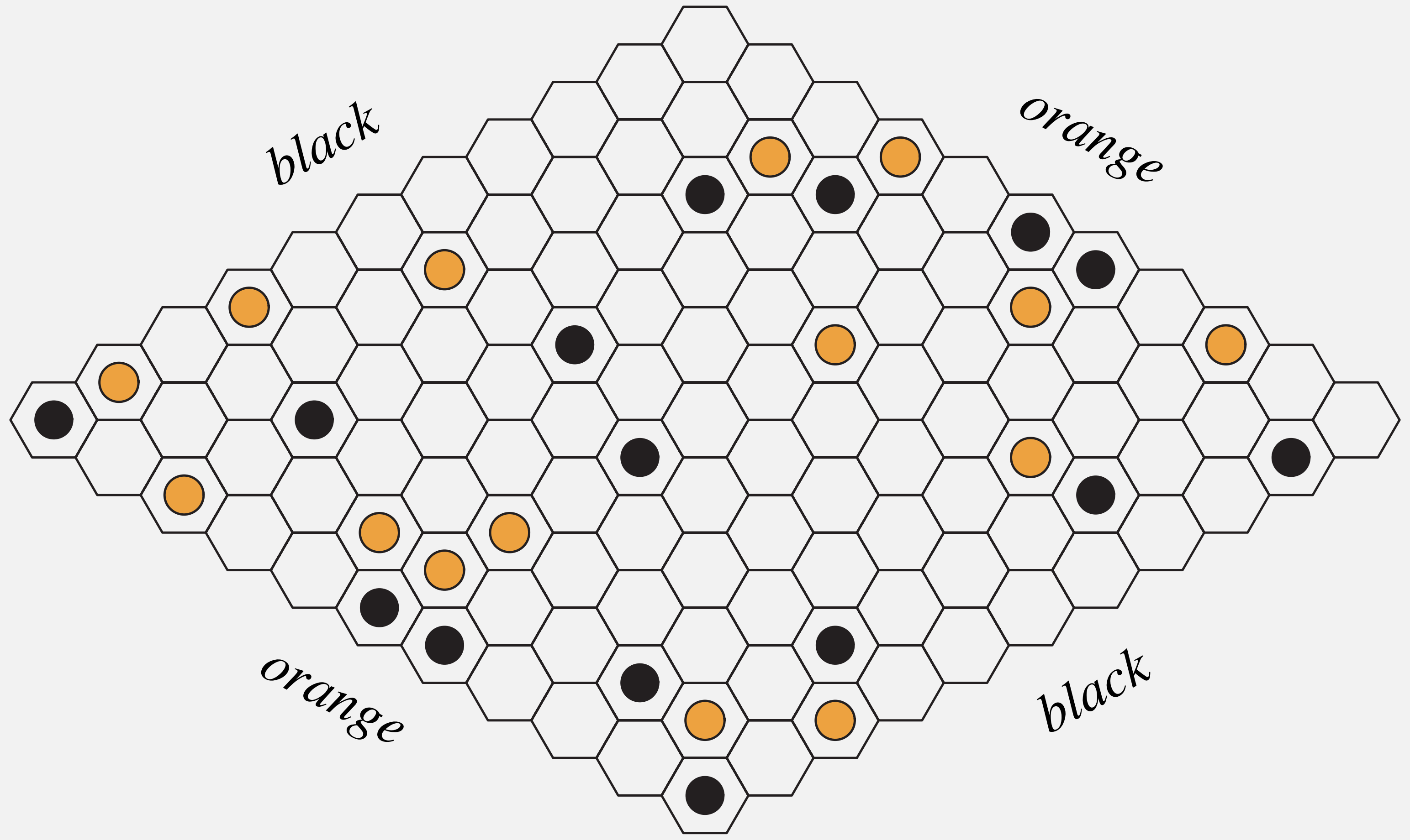
Union-find data type: applications

Disjoint sets can represent:

- Clusters of conducting sites in a composite system. ⟵ see Assignment 1 (Percolation)
- Connected components in a graph.
- Interlinked friends in a social network.
- Interconnected devices in a mobile network.
- Equivalent variable names in a Fortran program.
- Contiguous pixels of the same color in a digital image.
- Adjoining stones of the same color in the game of Hex.







https://algs4.cs.princeton.edu

1.5 Union–Find

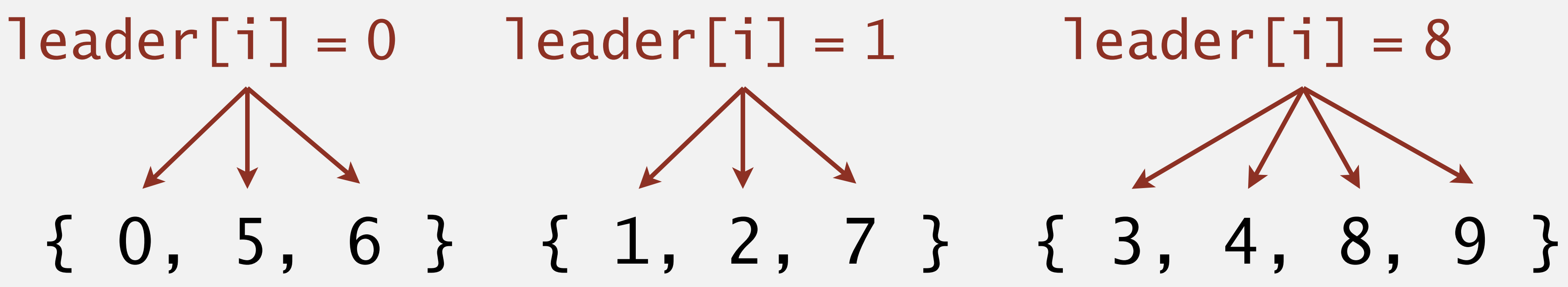
- union–find data type
- ***quick-find***
- quick-union
- weighted quick-union
- path compression
- percolation

Quick-find

Data structure.

- Integer array `leader[]` of length `n`.
- Interpretation: `leader[i]` is the leader of the set containing element `i`.

	0	1	2	3	4	5	6	7	8	9
leader[]	0	1	1	8	8	0	0	1	8	8



10 elements, 3 disjoint sets

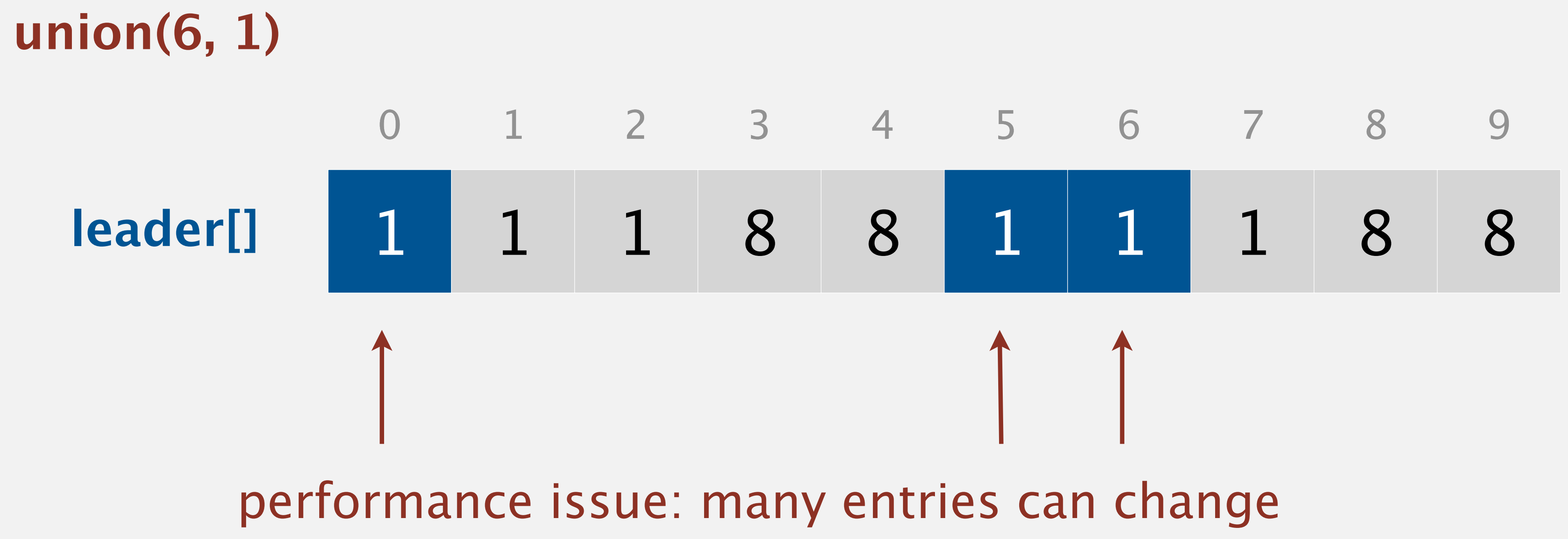
Q. How to implement `find(p)`?

A. Easy, just return `leader[p]`.

Quick-find

Data structure.

- Integer array `leader[]` of length `n`.
- Interpretation: `leader[i]` is the leader of the set containing element `i`.



Q. How to implement `union(p, q)`?

A. Change all array entries whose value is `leader[p]` to `leader[q]`. ← or vice versa

Quick-find: Java implementation

```
public class QuickFindUF
{
   private int[] leader;

   public QuickFindUF(int n)
   {
      leader = new int[n];
      for (int i = 0; i < n; i++)
         leader[i] = i;
   }

   public int find(int p)
   {  return leader[p];  }

   public void union(int p, int q)
   {
      int pLeader = leader[p];
      int qLeader = leader[q];
      for (int i = 0; i < leader.length; i++)
         if (leader[i] == pLeader)
            leader[i] = qLeader;
   }

}
```

set leader of each element to itself (n array accesses)

return the leader of p (1 array access)

change all array entries whose value is `leader[p]` to `leader[q]` ($\geq n$ array accesses)

https://algs4.cs.princeton.edu/15uf/QuickFindUF.java.html

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	n	n	1

worst-case number of array accesses (ignoring leading coefficient)

Union is too expensive. Processing any sequence of m `union()` operations on n elements takes $\geq mn$ array accesses.

quadratic in input size!

Ex. Performing 10^9 `union()` operations on 10^9 elements might take 30 years.

https://algs4.cs.princeton.edu

1.5 Union–Find

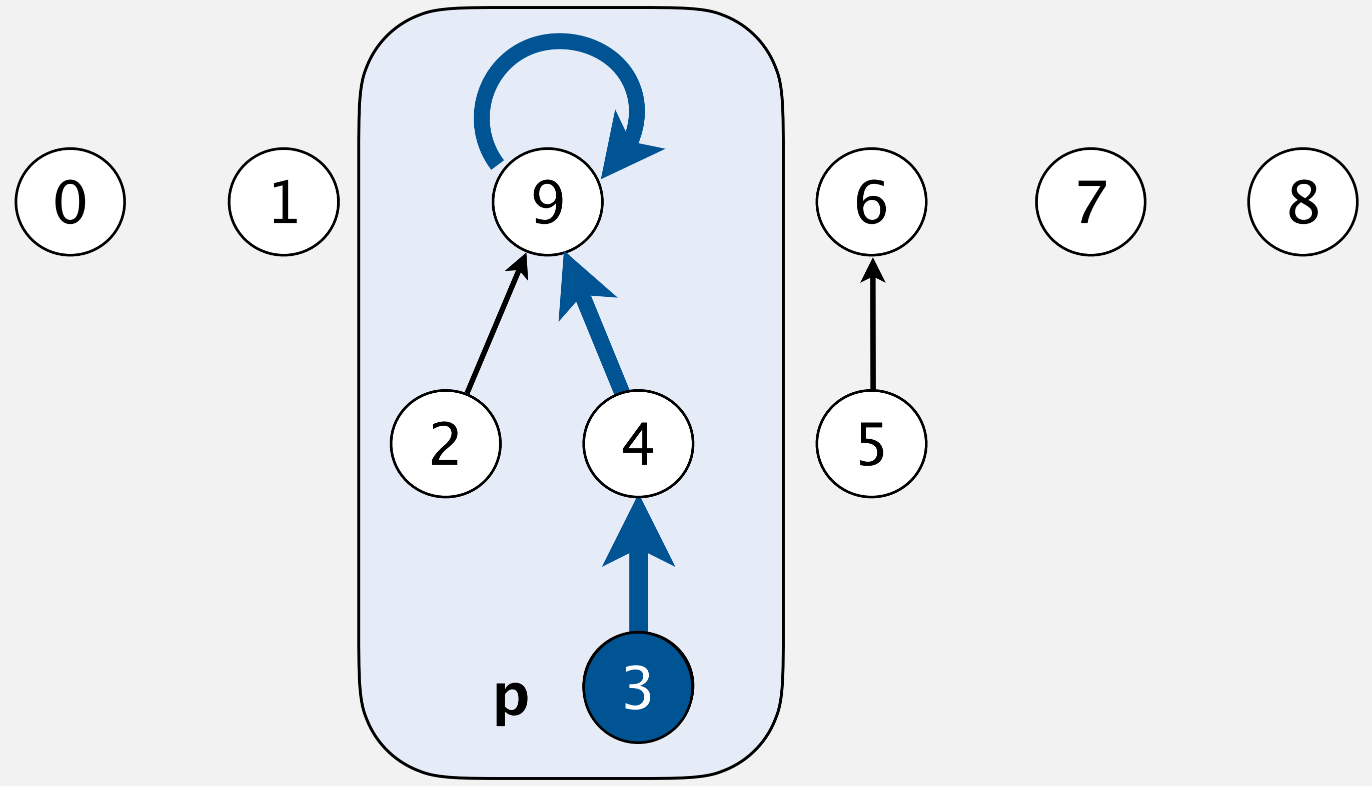
- *union–find data type*
- *quick-find*
- ***quick-union***
- *weighted quick-union*
- *percolation*

Quick-union

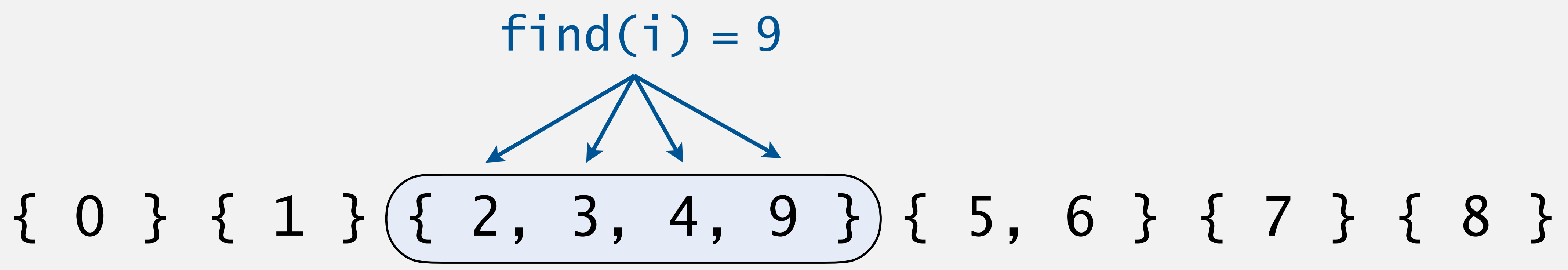
Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	9	4	9	6	6	7	8	9



parent of 3 is 4
root of 3 is 9



10 elements, 6 disjoint sets (6 trees)

Q. How to implement `find(p)`?

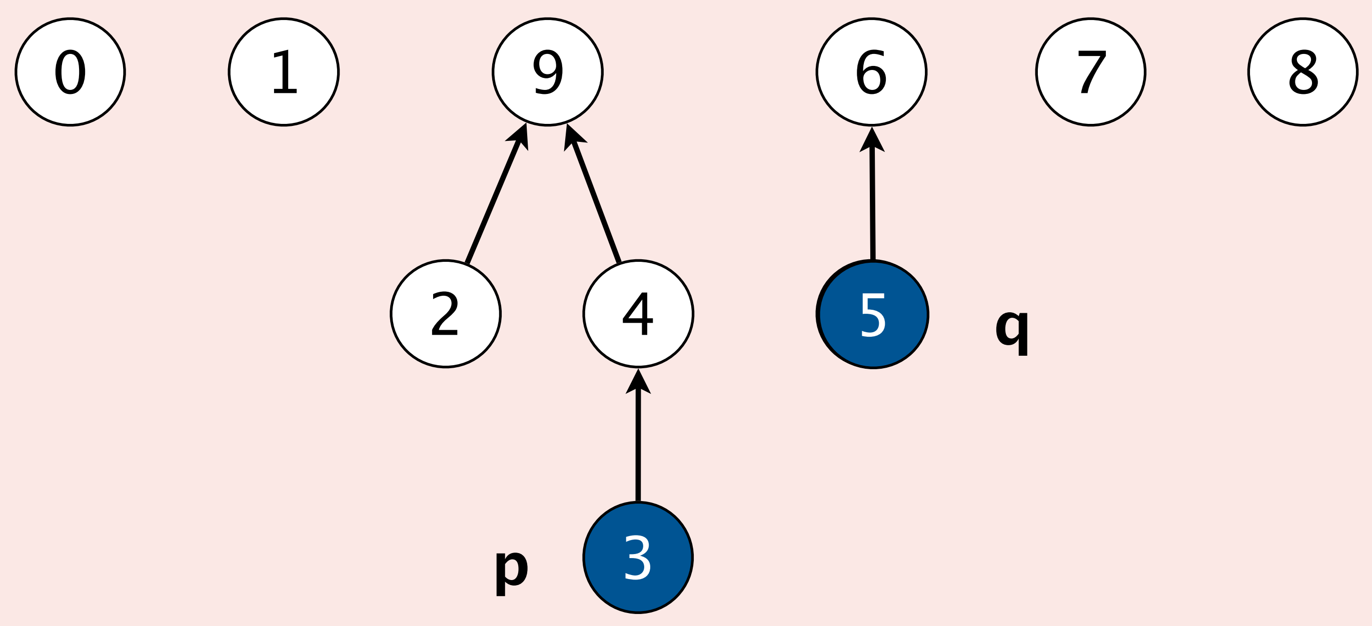
A. Use tree roots as leaders $\Rightarrow$ return root of tree containing `p`.

Union-find quiz 1

Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	9	4	9	6	6	7	8	9



Which is not a valid way to implement `union(3, 5)` ?

A. Set `parent[6] = 9`.

B. Set `parent[9] = 6`.

C. Set `parent[3] = 5`.

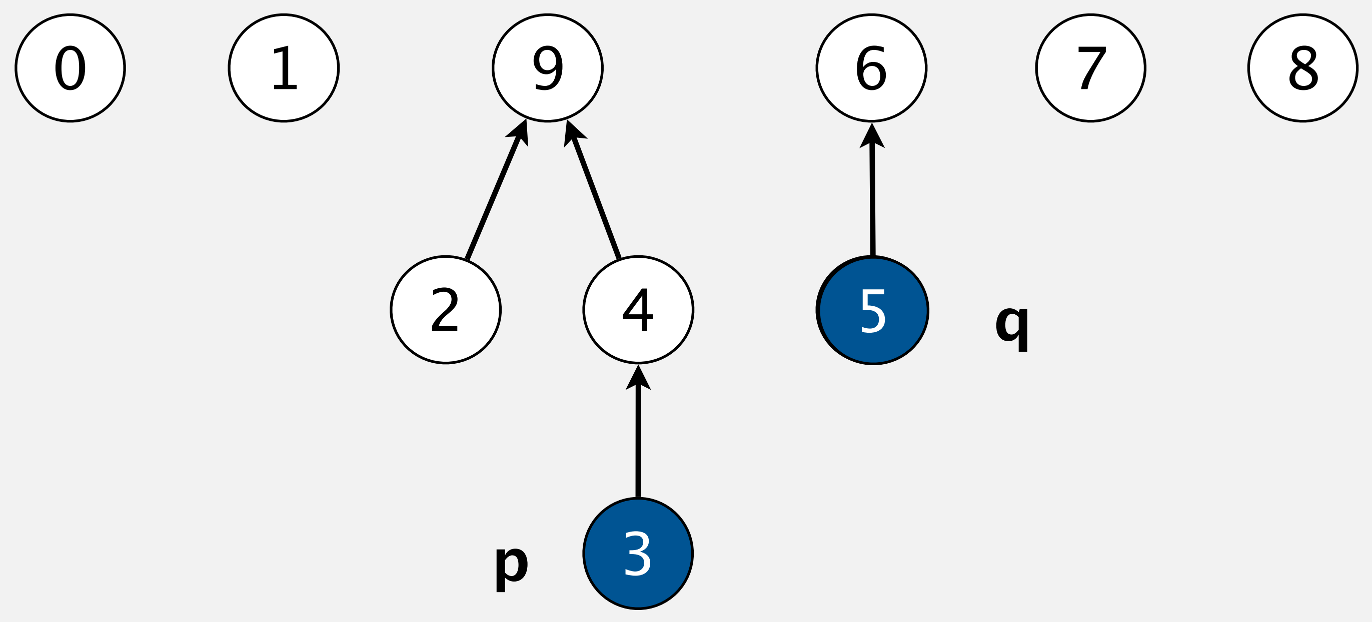
D. Set `parent[2] = parent[3] = parent[4] = parent[9] = 6`.

Quick-union

Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	9



Q. How to implement `union(p, q)`?

A. Set parent[`p`'s root] = `q`'s root. ← or vice versa

Quick-union

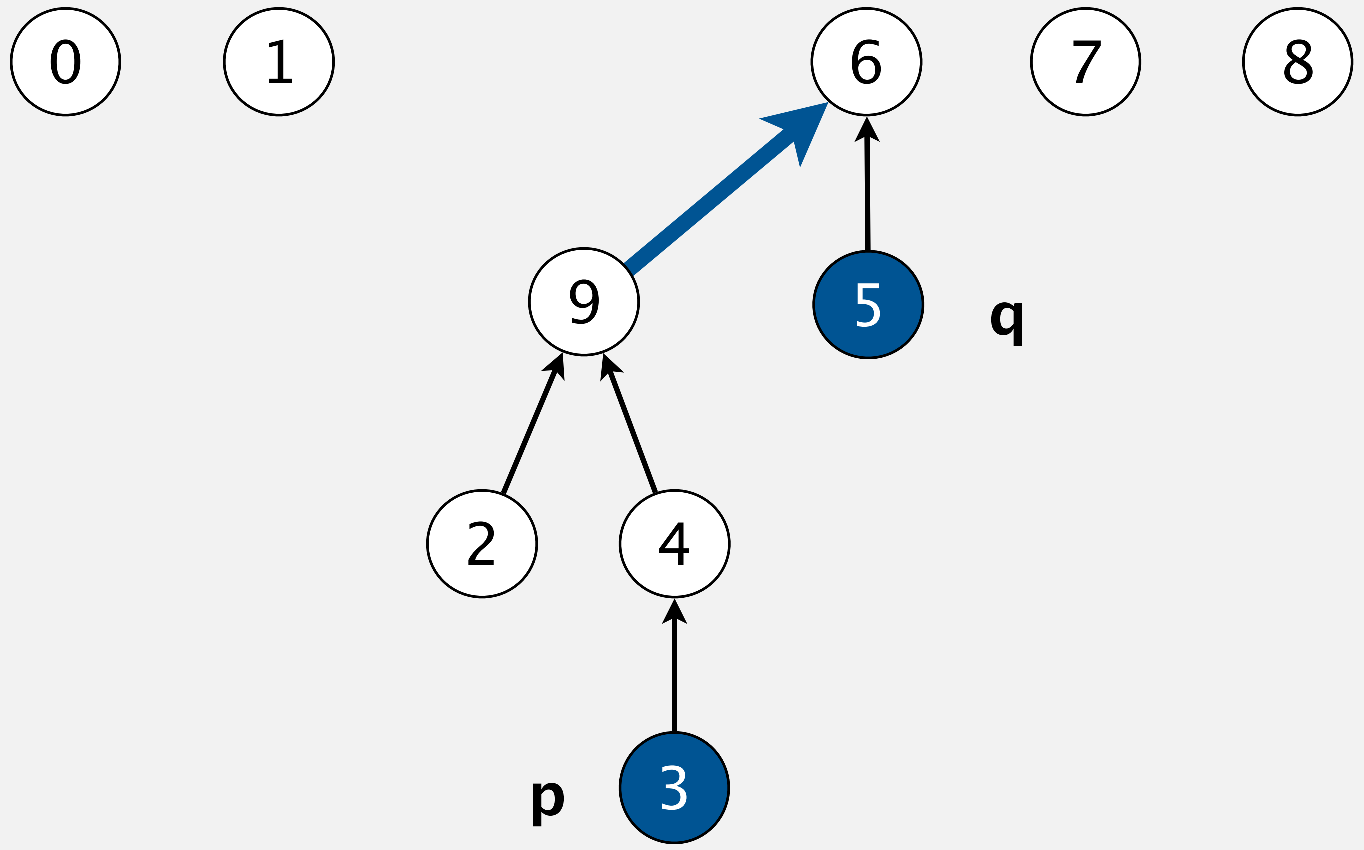
Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

union(3, 5)

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	6

only one entry changes



Q. How to implement `union(p, q)`?

A. Set parent[`p`'s root] = `q`'s root. ← or vice versa

Quick-union demo

Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] parent;

   public QuickUnionUF(int n)
   {
      parent = new int[n];
      for (int i = 0; i < n; i++)
         parent[i] = i;
   }

   public int find(int p)
   {
      while (p != parent[p])
         p = parent[p];
      return p;
   }

   public void union(int p, int q)
   {
      int root1 = find(p);
      int root2 = find(q);
      parent[root1] = root2;
   }
}
```

set parent of each element to itself (to create forest of n singleton trees)

follow parent pointers until reach root; return resulting root

link root of p to root of q

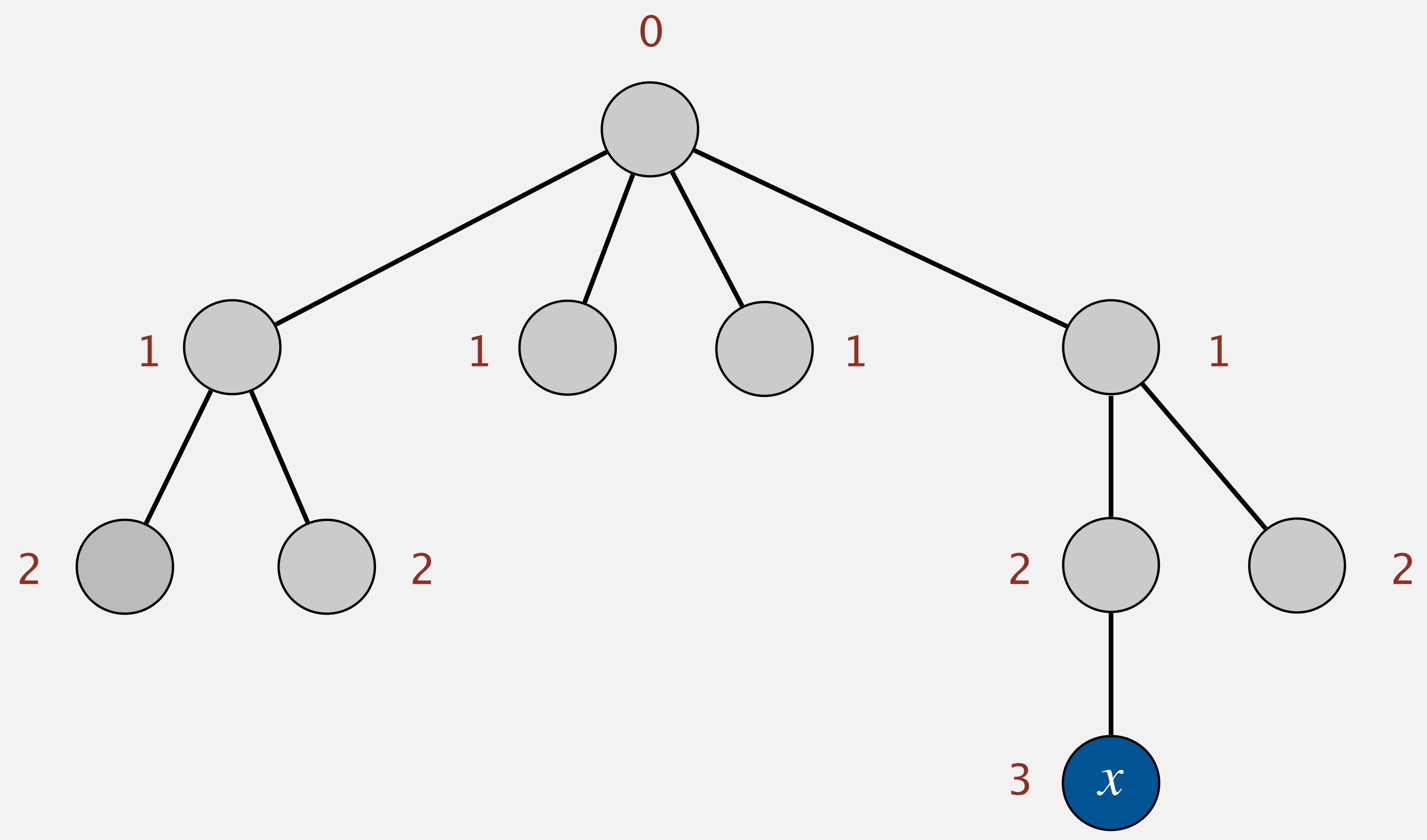
https://algs4.cs.princeton.edu/15uf/QuickUnionUF.java.html

Quick-union analysis

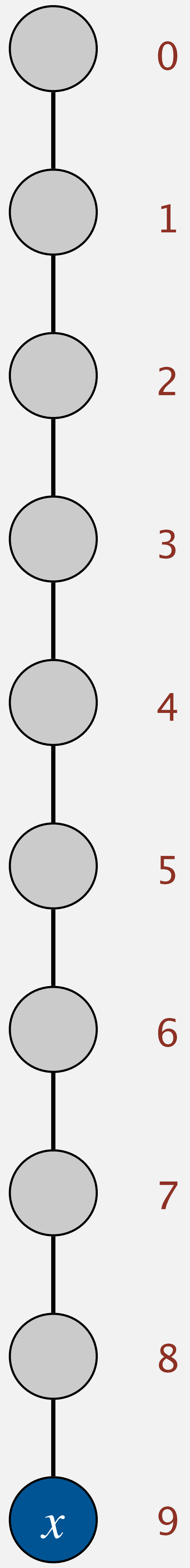
Cost model. Number of array accesses (for read or write).

Running time.

- `union()` takes constant time, given two roots.
- `find()` takes time proportional to depth of node in tree.



depth(x) = 3



worst-case depth = n–1

Quick-union analysis

Cost model. Number of array accesses (for read or write).

Running time.

- `union()` takes constant time, given two roots.
- `find()` takes time proportional to depth of node in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n

worst-case number of array accesses (ignoring leading coefficient)

Too expensive (if trees get tall). Processing some sequences of m `union()` and `find()` operations on n elements takes $\geq mn$ array accesses.

quadratic in input size!

https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *percolation*

Union-find quiz 2

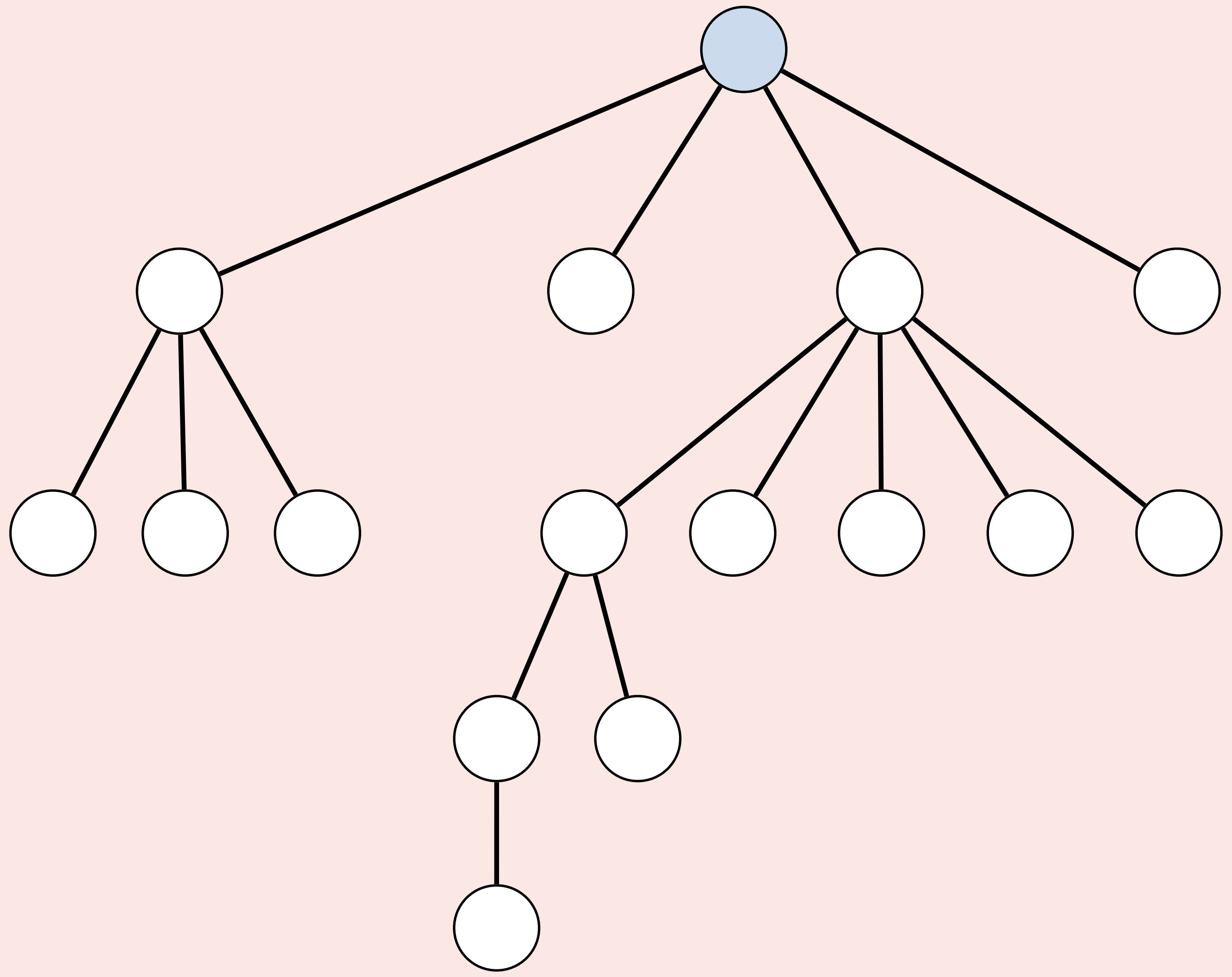
When linking two trees, which strategy is most effective?

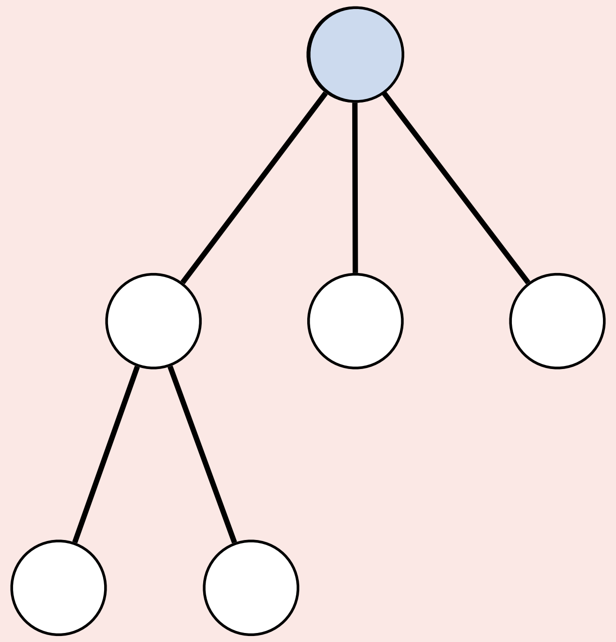
A. Link the root of the *smaller* tree to the root of the *larger* tree.

B. Link the root of the *larger* tree to the root of the *smaller* tree.

C. Flip a coin; randomly choose between A and B.

D. All of the above.

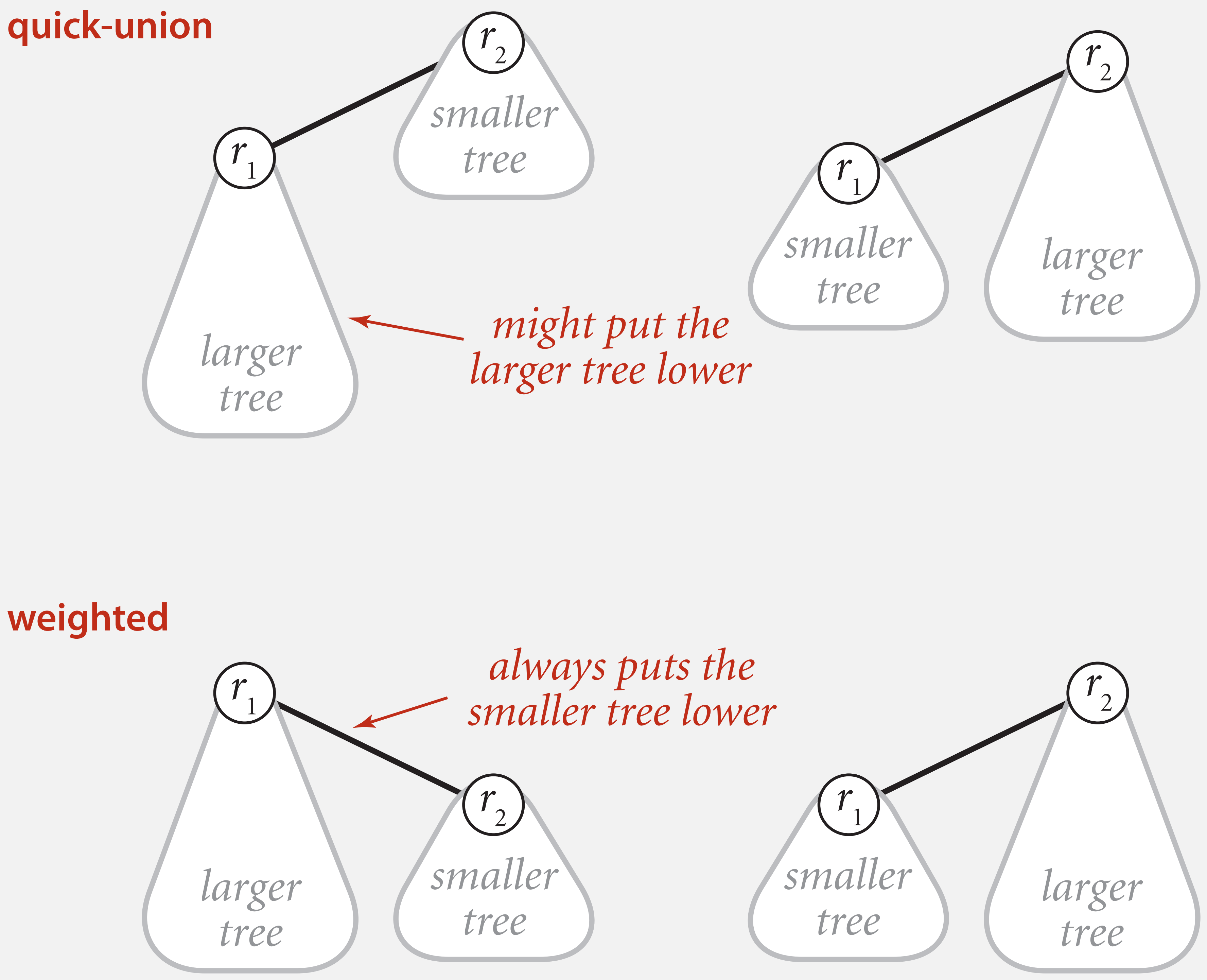




Weighted quick-union (link-by-size)

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree = number of elements.
- Always link root of smaller tree to root of larger tree.

fine alternative: link-by-height



Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `size[i]` to count number of elements in the tree rooted at `i`, initially `1`.

- `find()`: identical to quick-union.
- `union()`: link root of smaller tree to root of larger tree; update `size[]`.

```
public void union(int p, int q)
{
   int root1 = find(p);
   int root2 = find(q);
   if (root1 == root2) return;

   if (size[root1] >= size[root2])
   { int temp = root1; root1 = root2; root2 = temp; }

   parent[root1] = root2;
   size[root2] += size[root1];

}
```

afterwards, `root1` is root of smaller tree

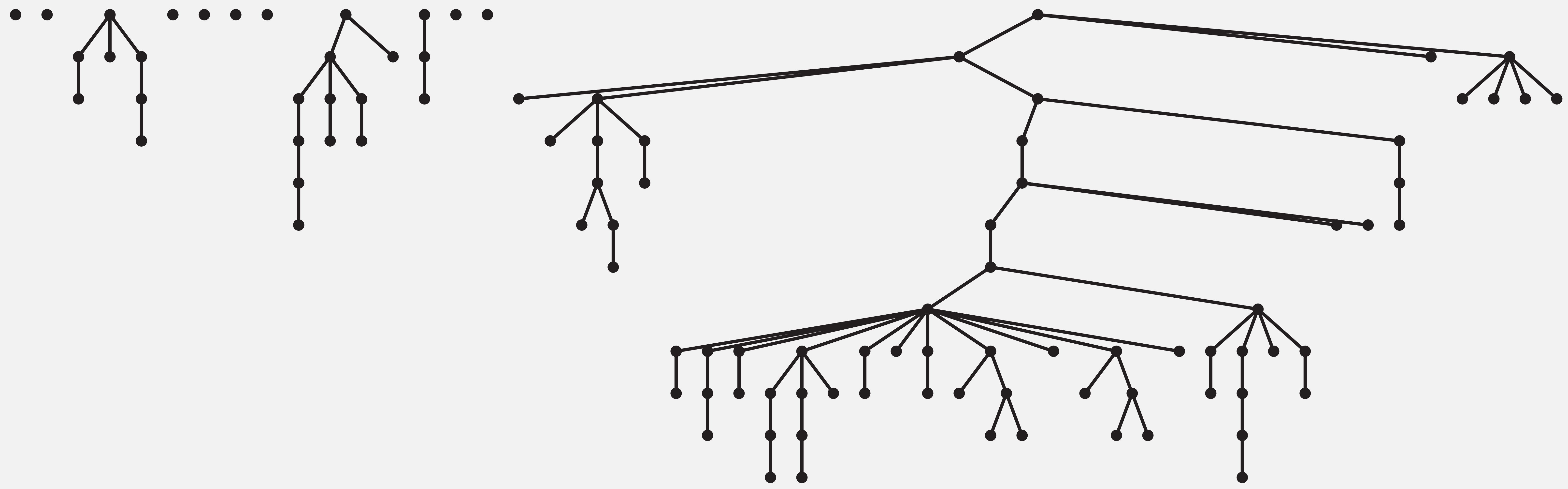
link root of smaller tree to root of larger tree

update size

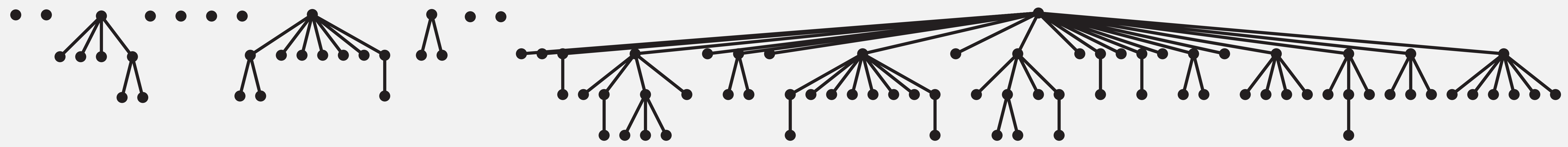
https://algs4.cs.princeton.edu/15uf/WeightedQuickUnionUF.java.html

Quick-union vs. weighted quick-union: larger example

quick-union

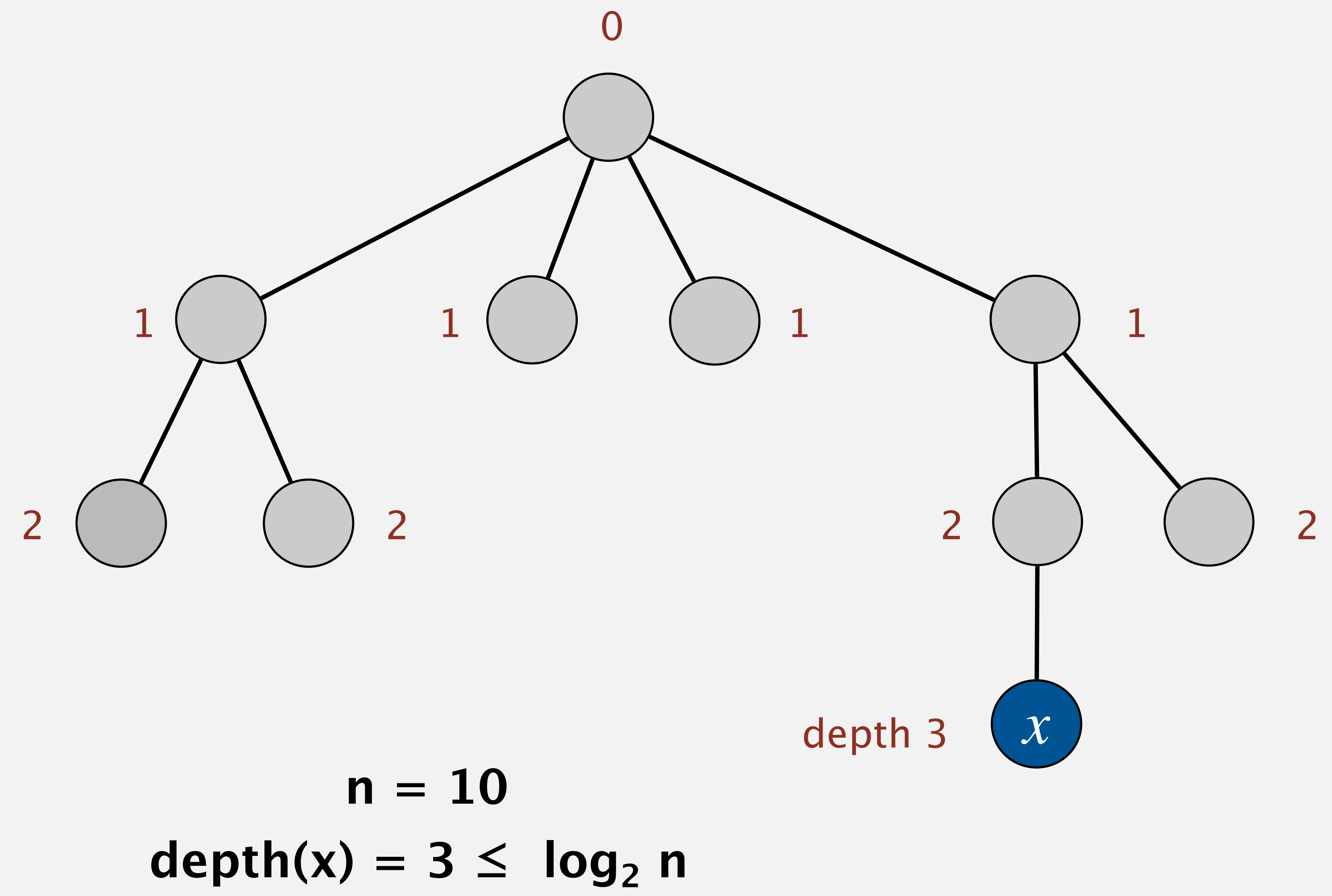


weighted



Weighted quick-union analysis

Proposition. Depth of any node $x \le \log_2 n$.



Weighted quick-union analysis

Proposition. Depth of any node $x \leq \log_2 n$.

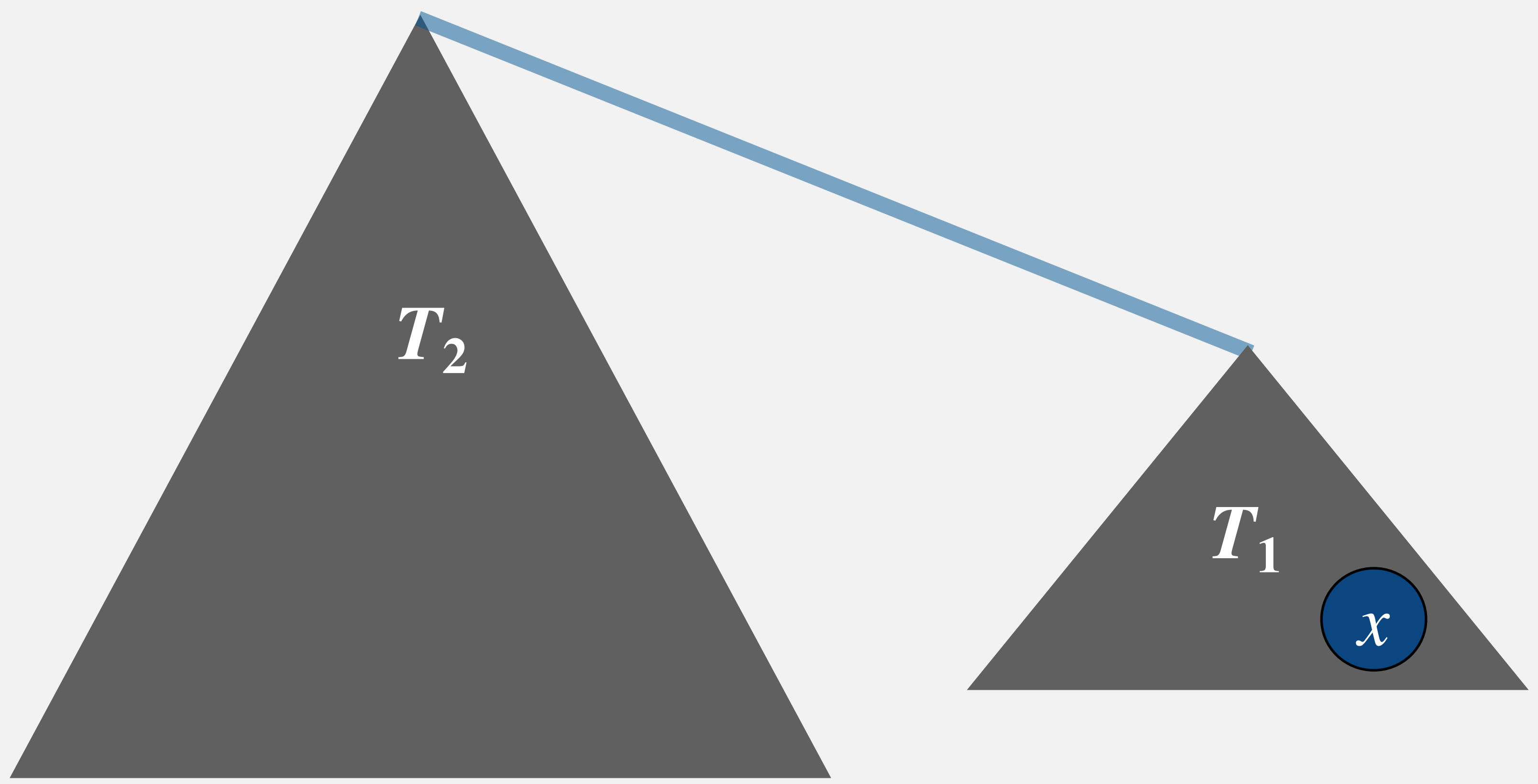
Pf.

- Depth of x does not change unless root of tree T_1 containing x is linked to the root of a larger tree T_2, forming a new tree T_3.
- when this happens:
  - depth of x increases by exactly 1
  - size of tree containing x at least doubles because $\text{size}(T_3) = \text{size}(T_1) + \text{size}(T_2) \geq 2 \times \text{size}(T_1)$.

can happen at most $\log_2 n$ times. Why?

$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \cdots \rightarrow n$

$\log_2 n$



Weighted quick-union analysis

Proposition. Depth of any node $x \le \log_2 n$.

Running time.

- `union()` takes constant time, given two roots.
- `find()` takes time proportional to depth of node in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n
weighted quick-union	n	$\log n$	$\log n$

worst-case number of array accesses (ignoring leading coefficient)

← in this course, $\log$ mean logarithm for some constant base

Summary

Key point. Weighted quick-union empowers us to solve problems that could not otherwise be addressed.

algorithm	worst-case time	
quick-find	$m\ n$	
quick-union	$m\ n$	
weighted quick-union	$m \log n$	
quick-union + path compression	$m \log n$	← fastest for percolation?
weighted quick-union + path compression	$m\ \alpha(m, n)$	← inverse Ackermann function (see COS 423)

order of growth for m ≥ n union-find operations on a set of n elements

Ex. [10^9 union-find operations on 10^9 elements]

- Efficient algorithm reduces time from 30 years to 6 seconds.
- Supercomputer won't help much.

" The goal is to come up with algorithms that you can apply in practice that run fast, as well as being simple, beautiful, and analyzable." — *Bob Tarjan*