1.5 **Union–Find**

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

**Applications**

For more information, see [precept](https://algs4.cs.princeton.edu)
Subtext of today’s lecture (and this course)

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

- model the problem
- design an algorithm
- efficient?
  - yes: solve the problem
  - no: try again
- understand why not
1.5 Union-Find

- union-find data type
- quick-find
- quick-union
- weighted quick-union
- applications
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 if its elements as “leader” to uniquely identify the set.

**Find.** Return the leader of the set containing element $p$.

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

Simplifying assumption. The $n$ elements are named $0, 1, ..., n - 1$. 
Union–find data type: applications

Disjoint sets can represent:

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

see Assignment 1
Goal. Design an efficient union–find data type.

- Number of elements \( n \) can be huge.
- Number of operations \( m \) can be huge.
- Union and find operations can be intermixed.

Public class `UF`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>UF(int n)</code></td>
<td>Initialize with ( n ) singleton sets (0 to ( n - 1 ))</td>
</tr>
<tr>
<td><code>void union(int p, int q)</code></td>
<td>Merge sets containing elements ( p ) and ( q )</td>
</tr>
<tr>
<td><code>int find(int p)</code></td>
<td>Return the leader of set containing element ( p )</td>
</tr>
</tbody>
</table>
1.5 Union–Find

- union–find data type
- quick-find
- quick-union
- weighted quick-union
- applications
Quick-find

Data structure.

- Integer array \( \text{leader}[] \) of length \( n \).
- Interpretation: \( \text{leader}[p] \) is the leader of the set containing element \( p \).

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{leader}[] & 0 & 1 & 1 & 8 & 8 & 0 & 0 & 1 & 8 & 8
\end{array}
\]

\[
\begin{align*}
\text{leader}[i] = 0 & \quad \Rightarrow \quad \{ 0, 5, 6 \} \\
\text{leader}[i] = 1 & \quad \Rightarrow \quad \{ 1, 2, 7 \} \\
\text{leader}[i] = 8 & \quad \Rightarrow \quad \{ 3, 4, 8, 9 \}
\end{align*}
\]

3 disjoint sets

Q. How to implement \( \text{find}(p) \)?
A. Easy, just return \( \text{leader}[p] \).
Quick-find

Data structure.

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

Q. How to implement `union(p, q)`?
A. Change all entries whose identifier equals `leader[p]` to `leader[q]`.`
Quick-find: Java implementation

```java
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 entries with $\text{leader}[p]$ to $\text{leader}[q]$ ($\geq n$ 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).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$n$</td>
<td>$n$</td>
<td>1</td>
</tr>
</tbody>
</table>

number of array accesses (ignoring leading constant)

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

quadratic in input size!
1.5 Union-Find

- union-find data type
- quick-find
- quick-union
- weighted quick-union
- applications
**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 `i` in tree.

![Diagram of a forest-of-trees with `parent[]` array and `find(p)` operation]

```
parent[] = [0, 1, 9, 4, 9, 6, 6, 7, 8, 9]
```

```
{ 0 } { 1 } { 2, 3, 4, 9 } { 5, 6 } { 7 } { 8 }
6 disjoint sets (6 trees)
```

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

**A.** Use tree roots as leaders \(\Rightarrow\) return root of tree containing `p`. 
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 i in tree.

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


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 `i` in tree.

```
union(3, 5)
```

Q. How to implement `union(p, q)`?
A. Set parent of `p`'s root to `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 i in tree.

<table>
<thead>
<tr>
<th>union(3, 5)</th>
<th>0 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 9 4 9 6 6 7 8 6</td>
</tr>
</tbody>
</table>

Q. How to implement union(p, q)?
A. Set parent of p’s root to 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

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.

\[
\text{depth}(x) = 3
\]

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.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$n$</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

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

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

- union-find data type
- quick-find
- quick-union
- weighted quick-union
- applications
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.
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.

reasonable 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[].

```java
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];
}
```

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

afterwards, root1 is root of smaller tree
link root of smaller tree to root of larger tree
update size
Quick-union vs. weighted quick-union: larger example

quick-union

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

$n = 10$

$\text{depth}(x) = 3 \leq \log_2 n$
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 $T_3$.
- In this case:
  - depth of $x$ increases by exactly 1
  - size of tree containing $x$ at least doubles because $\text{size}(T_3) = \text{size}(T_1) + \text{size}(T_2)$
    \[ \geq 2 \times \text{size}(T_1). \]

![Diagram](image)

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

\[ \begin{align*}
1 & \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \cdots \rightarrow n
\end{align*} \]

$\log_2 n$
Weighted quick-union analysis

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

**Running time.**

- Union: takes constant time, given two roots.
- Find: takes time proportional to depth of node in tree.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$n$</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

worst-case number of array accesses (ignoring leading constant)
Summary

**Key point.** Weighted quick-union makes it possible to solve problems that could not otherwise be addressed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case time</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$m \log n$</td>
</tr>
<tr>
<td>quick-union</td>
<td>$m \log n$</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>$m \alpha(n)$</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>$m \log n$</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>$m \alpha(n)$</td>
</tr>
</tbody>
</table>

Order of growth for $m \geq n$ union–find operations on a set of $n$ elements

**Ex.** $[10^9$ union–find operations on $10^9$ elements]

- Weighted quick-union reduces run time from 30 years to 6 seconds.
- Supercomputer won’t help much; good algorithm enables solution.