1.5 Union-Find

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

https://algs4.cs.princeton.edu
Steps to develop a usable algorithm to solve a computational problem.

1. Model the problem
2. Design an algorithm
3. Efficient?
   - Yes: Solve the problem
   - No: Try again
4. Understand why not
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.

\[ \text{leader is } 0 \quad \text{leader is } 4 \quad \text{leader is } 6 \]

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

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 \( q \).
Union–find data type: API

**Goal.** Design an **efficient** union–find data type.

- Simplifying assumption: the $n$ elements are named 0, 1, ..., $n – 1$.
- The `union()` and `find()` operations can be intermixed.
- Number of elements $n$ can be huge.
- Number of operations $m$ can be huge.

```java
public class UF {
    // description
    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
}
```
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.  
  see Kruskal's algorithm (MST lecture)
- Interlinked friends in a social network.
- Interconnected devices in a mobile network.
- Equivalent variable names in a Fortran program.
- Adjoining stones of the same color in the game of Hex.
- Contiguous pixels corresponding to same feature in a digital image.
1.5 Union-Find

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

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

**performance issue:**
many array entries can change

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

```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;
    }
}
```

- `initialize 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] (≥ 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>

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

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

Ex. Performing $10^9 \text{ union()}$ operations on $10^9$ elements might take 30 years.
1.5 Union-Find

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

https://algs4.cs.princeton.edu
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.

<table>
<thead>
<tr>
<th>parent[]</th>
<th>0</th>
<th>1</th>
<th>9</th>
<th>4</th>
<th>9</th>
<th>6</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th><code>0</code></th>
<th><code>1</code></th>
<th><code>2</code></th>
<th><code>3</code></th>
<th><code>4</code></th>
<th><code>5</code></th>
<th><code>6</code></th>
<th><code>7</code></th>
<th><code>8</code></th>
<th><code>9</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0</code></td>
<td><code>1</code></td>
<td><code>9</code></td>
<td><code>4</code></td>
<td><code>9</code></td>
<td><code>6</code></td>
<td><code>6</code></td>
<td><code>7</code></td>
<td><code>8</code></td>
<td><code>9</code></td>
</tr>
</tbody>
</table>

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 element $i$ in tree.

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

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.

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

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

```java
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.
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.
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 coefficient)

Union and find are 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!*
1.5 Union-Find

- union-find data type
- quick-find
- quick-union
- weighted quick-union
When linking two trees, which of these strategies 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)**

**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.

---

**quick-union**

```
              r_2
             /   \
           /     \
         r_1   smaller
tree
```

```
              r_2
             /   \
           /     \
         r_1   smaller
tree
```

- **might put the larger tree lower**

---

**weighted**

```
              r_2
             /   \
           /     \
         r_1   smaller
tree
```

```
              r_2
             /   \
           /     \
         r_1   smaller
tree
```

- **always puts the smaller tree lower**

---

**fine alternative: link-by-height**

(minimize worst-case depth vs. average depth)
Weighted quick-union: Java implementation

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

- \texttt{find()}: identical to quick-union.
- \texttt{union()}: link root of smaller tree to root of larger tree; update \texttt{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
Quick-union vs. weighted quick-union: larger example
Weighted quick-union analysis

**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 $T_3$.
- When this happens:
  - depth of $x$ increases by exactly 1
  - size of tree containing $x$ at least doubles because $\text{size}(T_3) = \text{size}(T_1) + \text{size}(T_2)$ $\geq 2 \times \text{size}(T_1)$.

\[1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \cdots \rightarrow n\]

\[\log_2 n\]
Weighted quick-union analysis

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

**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 coefficient)
Summary

Key point. Weighted quick-union empowers us 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 n)</td>
</tr>
<tr>
<td>quick-union</td>
<td>(m n)</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>(m \log n)</td>
</tr>
<tr>
<td>quick-union + path compression</td>
<td>(m \log n)</td>
</tr>
<tr>
<td>weighted quick-union + path compression</td>
<td>(m \alpha(m, 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 ]

- Efficient algorithm reduces time from 30 years to 6 seconds.
- Supercomputer won’t help much.
<table>
<thead>
<tr>
<th><strong>image</strong></th>
<th><strong>source</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Game of Hex</em></td>
<td>Wolfram MathWorld</td>
</tr>
<tr>
<td><em>Cluster Labeling</em></td>
<td>Tiberiu Marita</td>
</tr>
<tr>
<td><em>Bob Tarjan</em></td>
<td>Princeton University</td>
</tr>
<tr>
<td><em>Computer and Supercomputer</em></td>
<td>New York Times</td>
</tr>
</tbody>
</table>
“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.” — Robert Tarjan