1.5 Union–Find

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

See precept
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
- 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, ..., $n - 1$. 

---

Example:

8 elements, 3 disjoint sets

leader is 4

\[
\begin{align*}
\text{find}(1) &= 4 \\
\text{find}(4) &= 4 \\
\text{find}(5) &= 4
\end{align*}
\]

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

union(2, 5)

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

leader is 6

2 disjoint sets
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.
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.
- Union and find operations can be intermixed.

```java
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
    }
}
```
1.5 Union–Find

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

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

<table>
<thead>
<tr>
<th></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>leader[]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

- `leader[i] = 0` → `{ 0, 5, 6 }
- `leader[i] = 1` → `{ 1, 2, 7 }
- `leader[i] = 8` → `{ 3, 4, 8, 9 }

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[p]` is the leader of the set containing element `p`.

```
union(6, 1)

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

Problem: many values can change

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

**A.** Change all array entries with `leader[p]` to `leader[q]`. 

or vice versa
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;
    }
}

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 coefficient)

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.

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

- **find(i)** = `9`

- **6 disjoint sets (6 trees)**

```
{ 0 } { 1 } { 2, 3, 4, 9 } { 5, 6 } { 7 } { 8 }
```

**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)**

<table>
<thead>
<tr>
<th></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 i in tree.

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

<table>
<thead>
<tr>
<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>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

only one value changes
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
- 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.

![Diagram](attachment:image.png)

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

**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 coefficient)

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

quick-union

weighted
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 $T_3$.
- 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?
  - $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \cdots \rightarrow n$
  - $\log_2 n$
**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 coefficients)*
**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 \times n )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( m \times n )</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>( m \log 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>

**Ex.** [10⁹ union–find operations on 10⁹ elements]

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