1.5 **Union–Find**

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *applications*  
  
  ![see precept](https://algs4.cs.princeton.edu)
Subtext of today’s lecture (and this course)

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

1. Model the problem
2. Design an algorithm
3. Efficient?
   - Yes: Solve the problem
   - No: Understand why not
     - Try again
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; each element in exactly one set.

**Find.** Return a “canonical” element in the set containing \( p \).

**Union.** Merge the set containing \( p \) with the set containing \( q \).

\[
\text{find(1) = find(4) = find(5) = 4}
\]

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

8 elements, 3 disjoint sets

\[
\text{union(2, 5)}
\]

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

2 disjoint sets

**Simplifying assumption.** The \( n \) elements are named 0, 1, \( \ldots \), \( n - 1 \).
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
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 canonical element in set containing p */
    }
}
```
1.5 UNION–FIND

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

https://algs4.cs.princeton.edu
Quick-find

Data structure.

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

```
id[] = 0 1 1 8 8 0 0 1 8 8
```

```
id[i] = 0  
    { 0, 5, 6 }  

id[i] = 1  
    { 1, 2, 7 }  

id[i] = 8  
    { 3, 4, 8, 9 }  
```

3 disjoint sets

Q. How to implement `find(p)`?
A. Easy, just return `id[p]`.  

Quick-find

Data structure.
- Integer array \( \text{id}[] \) of length \( n \).
- Interpretation: \( \text{id}[p] \) is canonical element in the set containing \( p \).

```
union(6, 1)
```

```
id[] = [1 1 1 8 8 1 1 1 8 8]
```

problem: many values can change

Q. How to implement \( \text{union}(p, q) \)?
A. Change all entries whose identifier equals \( \text{id}[p] \) to \( \text{id}[q] \).
Quick-find: Java implementation

```java
public class QuickFindUF
{
    private int[] id;

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

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

    public void union(int p, int q)
    {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = qid;
    }
}
```

- Set id of each element to itself \((n\) array accesses)
- Return the id of \(p\) \((1\) array access)
- Change all entries with \(id[p]\) to \(id[q]\) \(\geq n\) 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!
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>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>9</td>
</tr>
</tbody>
</table>

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

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

**A.** Use tree root as canonical element ⇒ return root of tree containing `p`.

```
6 disjoint sets (6 trees)
```

```plaintext
find(i) = 9
```

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

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

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.

![Diagram of a forest-of-trees]

**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.

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

**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

```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 r1 = find(p);
        int r2 = find(q);
        parent[r1] = r2;
    }
}
```

- 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 $q$
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 \textit{smaller} tree to the root of the \textit{larger} tree.

B. Link the root of the \textit{larger} tree to the root of the \textit{smaller} tree.

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

D. Doesn’t matter.
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 demo

parent[] =

0 1 2 3 4 5 6 7 8 9
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 r1 = find(p);
    int r2 = find(q);
    if (r1 == r2) return;

    if (size[r1] >= size[r2]) {
        int temp = r1; r1 = r2; r2 = temp;
    }

    parent[r1] = r2;
    size[r2] += size[r1];
}
```

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 $x$ in tree is at most $\log_2 n$.

$$n = 10$$

$$\text{depth}(x) = 3 \leq \log_2 n$$
Weighted quick-union analysis

Proposition. Depth of any node \( x \) in tree is at most \( \log_2 n \).

Pf.

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

\[
\begin{array}{c}
1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \ldots \rightarrow n \\
\end{array}
\]

\( \text{can happen at most } \log_2 n \text{ times. Why?} \)
Weighted quick-union analysis

**Proposition.** Depth of any node \( x \) in tree is at most \( \log_2 n \).

**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 \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>

order of growth for \( m \geq n \) union–find operations on a set of \( n \) elements

Ex. [10^9 unions and finds with 10^9 elements]

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

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

- **Percolation.** first programming assignment
- Terrain analysis.
- **Dynamic-connectivity problem.** see textbook
- Least common ancestors in trees.
- Games (Go, Hex, maze generation).
- Minimum spanning tree algorithms.
- Equivalence of finite state automata.
- Hoshen–Kopelman algorithm in physics.
- Hindley–Milner polymorphic type inference.
- Compiling equivalence statements in Fortran.
- Matlab’s `bwlabel()` function in image processing.
An abstract model for many physical systems:

- An $n$-by-$n$ grid of sites.
- Each site is open with probability $p$ (and blocked with probability $1 - p$).
- System percolates iff top and bottom are connected by open sites.

If and only if:

- There exists an open site connected to the top.
- No open site connected to the top.
An abstract model for many physical systems:

- \( n \)-by-\( n \) grid of sites.
- Each site is open with probability \( p \) (and blocked with probability \( 1 - p \)).
- System **percolates** iff top and bottom are connected by open sites.

<table>
<thead>
<tr>
<th>model</th>
<th>system</th>
<th>vacant site</th>
<th>occupied site</th>
<th>percolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>electricity</td>
<td>material</td>
<td>conductor</td>
<td>insulated</td>
<td>conducts</td>
</tr>
<tr>
<td>fluid flow</td>
<td>material</td>
<td>empty</td>
<td>blocked</td>
<td>porous</td>
</tr>
<tr>
<td>social interaction</td>
<td>population</td>
<td>person</td>
<td>empty</td>
<td>communicates</td>
</tr>
</tbody>
</table>
Likelihood of percolation

Depends on grid size \( n \) and site vacancy probability \( p \).

- \( p \) low (0.4) does not percolate
- \( p \) medium (0.6) percolates?
- \( p \) high (0.8) percolates

Legend:
- Empty open site (not connected to top)
- Full open site (connected to top)
- Blocked site
Percolation phase transition

When $n$ is large, theory guarantees a sharp threshold $p^*$.

- $p > p^*$: almost certainly percolates.
- $p < p^*$: almost certainly does not percolate.

Q. What is the value of $p^*$?
Monte Carlo simulation

Barrier. Determining the exact threshold $p^*$ is beyond mathematical reach.

Computational approach.
- Conduct many random experiments.
- Compute statistics.
- Obtain estimate of $p^*$. 
Monte Carlo simulation

- Initialize all sites in an $n$-by-$n$ grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates $p^*$. 
- Repeat many times to get more accurate estimate.

\[
\hat{p} = \frac{204}{400} = 0.51
\]

$n = 20$

full open site (connected to top)
empty open site (not connected to top)
blocked site

135 open sites
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an $n$-by-$n$ system percolates?
A. Model as a dynamic-connectivity problem problem and use union–find.

$n = 5$

open site

blocked site
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an $n$-by-$n$ system percolates?
• Create an element for each site, named $0$ to $n^2 - 1$.

$n = 5$

[Diagram of a 5x5 grid with open and blocked sites labeled from 0 to 24]
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an $n$-by-$n$ system percolates?
   - Create an element for each site, named 0 to $n^2 - 1$.
   - Add edge between two adjacent sites if they both open.

4 possible neighbors: left, right, top, bottom

$n = 5$

open site

blocked site
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an $n$-by-$n$ system percolates?
   • Create an element for each site, named 0 to $n^2 - 1$.
   • Add edge between two adjacent sites if they both open.
   • Percolates iff any site on bottom row is connected to any site on top row.

![Diagram](image)

$n = 5$

- open site
- blocked site

brute-force algorithm: $n^2$ connected queries
Clever trick. Introduce 2 virtual sites (and edges to top and bottom).
- Percolates iff virtual top site is connected to virtual bottom site.

Dynamic-connectivity solution to estimate percolation threshold

$n = 5$

- open site
- blocked site
Dynamic-connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

\[ n = 5 \]

![Diagram showing open and blocked sites](image)
Q. How to model opening a new site?

A. Mark new site as open; add edge to any adjacent site that is open.

Dynamic-connectivity solution to estimate percolation threshold

\[ n = 5 \]

open site

blocked site

open this site

adds up to 4 edges
Q. What is percolation threshold $p^*$?
A. About 0.592746 for large square lattices.

Fast algorithm enables accurate answer to scientific question.
Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.