### Dynamic Connectivity

Given a set of objects
- **Union**: connect two objects.
- **Find**: is there a path connecting the two objects?

```plaintext
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
find(0, 2)   no
find(2, 4)   yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
find(0, 2)   yes
find(2, 4)   yes
```
Network connectivity: larger example

Q. Is there a path from p to q?

A. Yes.

but finding the path is more difficult: stay tuned (Chapter 4)

Dynamic connectivity applications involve manipulating objects of all types.
• Variable name aliases.
• Pixels in a digital photo.
• Computers in a network.
• Web pages on the Internet.
• Transistors in a computer chip.
• Metallic sites in a composite system.

When programming, convenient to name objects 0 to N-1.
• Use integers as array index.
• Suppress details not relevant to union-find.

Modeling the objects

Transitivity. If p is connected to q and q is connected to r, then p is connected to r.

Connected components. Maximal set of objects that are mutually connected.

Find query. Check if two objects are in the same set.

Union command. Replace sets containing two objects with their union.
Union-find data type (API)

**Goal.** Design efficient data structure for union-find.
- Number of objects $N$ can be huge.
- Number of operations $M$ can be huge.
- Find queries and union commands may be intermixed.

```java
public class UnionFind

  // create union-find data structure with N objects and no connections
  UnionFind(int N);

  // check if p and q are in the same set
  boolean find(int p, int q);

  // replace sets containing p and q with their union
  void union(int p, int q);
```

Quick-find  [eager approach]

**Data structure.**
- Integer array $id[]$ of size $N$.
- Interpretation: $p$ and $q$ are connected if they have the same id.

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

- 5 and 6 are connected
- 2, 3, 4, and 9 are connected

Find. Check if $p$ and $q$ have the same id.

```
  5 and 6 are connected
  2, 3, 4, and 9 are connected

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

- 3 and 6 not connected
Quick-find [eager approach]

Data structure.
- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

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

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

<table>
<thead>
<tr>
<th>i</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>id[i]</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

Problem: many values can change

Quick-find example

3-4 0 1 2 4 5 6 7 8 9
4-9 0 1 2 9 9 5 6 7 8 9
8-0 0 1 2 9 9 5 6 7 0 9
2-3 0 1 9 9 9 5 6 7 0 9
5-6 0 1 9 9 9 6 6 7 0 9
5-9 0 1 9 9 9 9 9 7 0 9
7-3 0 1 9 9 9 9 9 0 9
4-8 0 1 0 0 0 0 0 0 0 0
6-1 1 1 1 1 1 1 1 1 1 1

Problem: many values can change

Quick-find: Java implementation

```java
class QuickFind {
    private int[] id;
    public QuickFind(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }
    private int find(int p) { return id[p]; }
    public boolean find(int p, int q) { return find(p) == find(q); }
    public void union(int p, int q) {
        int pid = find(p);
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = id[q];
    }
}
```

Quick-find is too slow

Quick-find defect.
- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

Ex. Takes N^2 array accesses to process sequence of N union commands on N objects.
Quadratic algorithms do not scale

Rough standard (for now).
- $10^9$ operations per second.
- $10^9$ words of main memory.
- Touch all words in approximately 1 second.

Ex. Huge problem for quick-find.
- $10^9$ union commands on $10^9$ objects.
- Quick-find takes more than $10^{18}$ operations.
- 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.
- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

Quick-union [lazy approach]

Data structure.
- Integer array $id[]$ of size $N$.
- Interpretation: $id[i]$ is parent of $i$.
- Root of $i$ is $id[id[id[...id[i]...]]]$.

Quick-union [lazy approach]

Data structure.
- Integer array $id[]$ of size $N$.
- Interpretation: $id[i]$ is parent of $i$.
- Root of $i$ is $id[id[id[...id[i]...]]]$.

Find. Check if $p$ and $q$ have the same root.
Data structure.

- Integer array id[] of size N.
- Interpretation: id[i] is parent of i.
- Root of i is id[id[id[...id[i]...]]].

Find. Check if p and q have the same root.

Union. To merge sets containing p and q, set the id of p’s root to the id of q’s root.

Quick-union: Java implementation

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

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

    private int find(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    public boolean find(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int i = root(p), j = root(q);
        id[i] = j;
    }
}
```

Quick-find defect.

- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N array accesses).

- Quick-union is also too slow

<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N†</td>
<td>N</td>
</tr>
</tbody>
</table>

† includes cost of finding root
**Weighted quick-union.**

- Modify quick-union to avoid tall trees.
- Keep track of size of each set.
- Balance by linking small tree below large one.

**Ex.** Union of 3 and 5.
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.

**Improvement 1: weighting**

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each set.
- Balance by linking small tree below large one.

**Data structure.** Same as quick-union, but maintain extra array sz[] to count number of objects in the tree rooted at i.

**Find.** Identical to quick-union.

```java
return find(p) == find(q);
```

**Union.** Modify quick-union to:
- Merge smaller tree into larger tree.
- Update the sz[] array.

```java
int i = find(p);
int j = find(q);
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```
Weighted quick-union analysis

Analysis.
• Find: takes time proportional to depth of p and q.
• Union: takes constant time, given roots.

Proposition. Depth of any node x is at most \( \lg N \).

\[
\text{Algorithm} \quad \text{union} \quad \text{find}
\begin{array}{|c|c|c|}
\hline
\text{quick-find} & N & 1 \\
\text{quick-union} & N^* & N \\
\text{weighted QU} & \lg N^* & \lg N \\
\hline
\end{array}
\]

† includes cost of finding root

Q. Stop at guaranteed acceptable performance?
A. No, easy to improve further.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Weighted quick-union analysis

Analysis.
• Find: takes time proportional to depth of p and q.
• Union: takes constant time, given roots.

Proposition. Depth of any node x is at most \( \lg N \).

Pf. When does depth of x increase?
Increases by 1 when tree \( T_1 \) containing x is merged into another tree \( T_2 \).
• The size of the tree containing x at least doubles since \( |T_2| \geq |T_1| \).
• Size of tree containing x can double at most \( \lg N \) times. Why?
Path compression: Java implementation

**Standard implementation:** add second loop to `find()` to set the `id[]` of each examined node to the root.

**Simpler one-pass variant:** halve the path length by making every other node in path point to its grandparent.

```java
public int find(int i) {
    while (i != id[i]) {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

In practice. No reason not to! Keeps tree almost completely flat.

**WQUPC performance**

**Proposition.** [Tarjan 1975] Starting from an empty data structure, any sequence of `M` union and find ops on `N` objects takes `O(N + M \lg^* N)` time.

- Proof is very difficult.
- But the algorithm is still simple!

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

```
N  \lg^* N
1  0
2  1
4  2
16 3
65536 4
268435456 5
```

Amazing fact. No linear-time algorithm exists.

**Summary**

**Bottom line.** WQUPC makes it possible to solve problems that could not otherwise be addressed.

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
Percolation

A model for many physical systems:
- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if top and bottom are connected by open sites.

Union-find applications

- Percolation.
- Games (Go, Hex).
- Network connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal’s minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab’s `bwlabel()` function in image processing.

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>fluid flow</td>
<td>material</td>
<td>empty</td>
<td>blocked</td>
<td>porous</td>
<td></td>
</tr>
<tr>
<td>social interaction</td>
<td>population</td>
<td>person</td>
<td>empty</td>
<td>communicates</td>
<td></td>
</tr>
</tbody>
</table>
Likelihood of percolation

Depends on site vacancy probability $p$.

- $p_{\text{low}}$: does not percolate
- $p_{\text{medium}}$: percolates?
- $p_{\text{high}}$: percolates

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

- Initialize $N$-by-$N$ whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates $p^*$.

Percolation phase transition

UF solution to find percolation threshold

How to check whether system percolates?
- Create an object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row.

Brute force algorithm needs to check $N^2$ pairs.
Q. How to declare a new site open?

A. Take union of new site and all adjacent open sites.

UF solution to find percolation threshold

Q. How to avoid checking all pairs of top and bottom sites?

A. Create a virtual top and bottom objects; system percolates when virtual top and bottom objects are in same set.

UF solution to find percolation threshold

Q. How to avoid checking all pairs of top and bottom sites?
Q. What is percolation threshold $p^*$?
A. About 0.592746 for large square lattices.

Fast algorithm enables accurate answer to scientific question.

Percolation threshold

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.